Fast Cone-beam CT reconstruction using GPU

Giovanni Di Domenico[1]

[1]Dipartimento di Fisica e Scienza della Terra, via Saragat 1, 44122 Ferrara, Italy
[2]INFN - Sezione di Ferrara, via Saragat 1, 44122 Ferrara, Italy

Fast 3D cone-beam CT reconstruction is required by many application fields like medical CT imaging or industrial non-destructive testing. We have used GPU hardware and CUDA platform to speed-up the Feldkamp Davis Kress (FDK) algorithm, which permits the reconstruction of cone-beam CT data. In this work, we present our implementation of the most time-consuming step of FDK algorithm: back-projection. We also explain the required transformations to parallelize the algorithm for the CUDA architecture. Our FDK algorithm implementation in addition allows to do a rapid reconstruction, which means that the reconstruction is completed just after the end of data acquisition.

1 Introduction

There are a lot of application areas that benefit of the GPU computing, among them we find the image reconstruction in medical physics applications [1]. In the last years, the computational complexity of image reconstruction has increased with the progress in imaging systems and reconstruction algorithms. Moreover, fast image reconstruction is required in an imaging diagnostic department to allow the technologist to review the images while the patient is waiting or to obtain the output in real-time imaging applications. Many researchers have worked to accelerate Cone-Beam CT (CBCT) reconstruction using various accelerators, such as GPU [3, 4, 5, 6], Cell Broadband Engine (CBE) [7], and field programmable gate array (FPGA) [8]. The Common Unified Device Architecture (CUDA) [2] is a software development platform, invented by nVIDIA, that allows us to write and execute general-purpose application on graphics processing unit (GPU). Scherl et al. [4] have developed one of the first GPU-accelereted CT reconstruction using CUDA and they have compared the performance results of CUDA implementation with the CBE ones concluding that the CUDA-enable GPU is well suited for high-speed CBCT recontruction. In this paper, we propose a CUDA based method capable of accelerating CBCT reconstruction based on FDK algorithm, showing some optimization techniques for the backprojection step.

2 Theory

The ideal geometry for cone-beam CT, see Fig.1, is made by an x-ray source that moves on a circle through the x-y plane around z, the axis of rotation. The (u,v,w) denotes the rotated reference frame, where β is the angle of rotation, R_s is the radius of the source circle, while R_d is the source to detector distance. The line from the x-ray source through the origin O hits the

Chapter 1

GPU in High-Level Triggers

Convenors:

Ivan Kisel

Daniel Hugo Campora Perez
Andrea Messina

The GAP project: GPU applications for High Level Trigger and Medical Imaging

Matteo Bauce[1,2], *Andrea Messina*[1,2,3], *Marco Rescigno*[3], *Stefano Giagu*[1,3], *Gianluca Lamanna*[4,6], *Massimiliano Fiorini*[5]

[1]Sapienza Università di Roma, Italy
[2]INFN - Sezione di Roma 1, Italy
[3]CERN, Geneve, Switzerland
[4]INFN - Sezione di Frascati, Italy
[5]Università di Ferrara, Italy
[6]INFN - Sezione di Pisa, Italy

The aim of the GAP project is the deployment of Graphic Processing Units in real-time applications, ranging from the online event selection (trigger) in High-Energy Physics to medical imaging reconstruction. The final goal of the project is to demonstrate that GPUs can have a positive impact in sectors different for rate, bandwidth, and computational intensity. Most crucial aspects currently under study are the analysis of the total latency of the system, the algorithms optimisations, and the integration with the data acquisition systems. In this paper we focus on the application of GPUs in asynchronous trigger systems, employed for the high level trigger of LHC experiments. The benefit obtained from the GPU deployement is particularly relevant for the foreseen LHC luminosity upgrade where highly selective algorithms will be crucial to maintain a sustainable trigger rates with very high pileup. As a study case, we will consider the ATLAS experimental environment and propose a GPU implementation for a typical muon selection in a high-level trigger system.

1 Introduction

The Graphic Processing Units (GPU) are commercial devices optimized for parallel computation, which, given their rapidly increasing performances, are being deployed in many general purpose application. The GAP project is investigating GPU applications in real-time environments, with particular interest in High Energy Physics (HEP) trigger systems, which will be discussed in this paper, but also Medical Imaging reconstruction (CT, PET, NMR) discussed in [1]. The different areas of interest span several orders of magnitude in terms of data processing, bandwidth and computational intensity of the executed algorithms, but can all benefit from the implementation of the massively parallel architecture of GPUs, optimizing different aspects, in terms of execution speed and complexity of the analyzed events. The trigger system of a typical HEP experiment has a crucial role deciding, based on limited and partial information, whether a particular event observed in a detector is interesting enough to be recorded. Every

experiment is characterised by a limited Data Acquisition (DAQ) bandwidth and disk space for storage hence needs real-time selections to reduce data throughput selectively. The rejection of uninteresting events only, is crucial to make an experiment affordable, preserving at the same time its discovery potential. In this paper we report some results obtained from the inclusion of GPU in the ATLAS High Level Trigger (HLT); we will focus on the main challenges to deploy such parallel computing devices in a typical HLT environment and on possible improvements.

2   ATLAS trigger system

The LHC proton-proton accelerator provides collisions at a rate of 40 MHz, which corresponds, for events of a typical size of 1-2 MByte, to an input data rate of the order of tens of TB/s. The reduction of this input rate to a sustainable rate to be stored on disk, of the order of ∼100 kHz, is achieved through a hybrid multi-level event selection system. Lower selection stages (Lower Level Triggers) are usually implemented on customized electronics, while HLT are nowadays implemented as software algorithms executed on farms of commodity PCs. HLT systems, in particular those of LHC experiments, offer a very challenging environment to test cutting-edge technology for realtime event selection. The LHC upgrade with the consequent increase of instantaneous luminosity and collision pile-up, poses new challenges for the HLT systems in terms of rates, bandwidth and signal selectivity. To exploit more complex algorithms aimed at better performances, higher computing capabilities and new strategies are required. Moreover, given the tendency of the computing industry to move away from the current CPU model towards architectures with high numbers of small cores well suited for vectorial computation, it is becoming urgent to investigate the possibility to implement a higher level of parallelism in the HLT software.

The GAP project is investigating the deployment of GPUs for the HLT in LHC experiments, using as a study case the ATLAS muon HLT. The ATLAS trigger system is organized in 3 levels [2], as shown in Figure 1. The first trigger level is built on custom electronics, while the second level (L2) and the event filter (EF) are implemented in software algorithms executed by a farm of about 1600 PCs with different Xeon processors each with 8 to 12 cores. During the Run II of the LHC (ex-
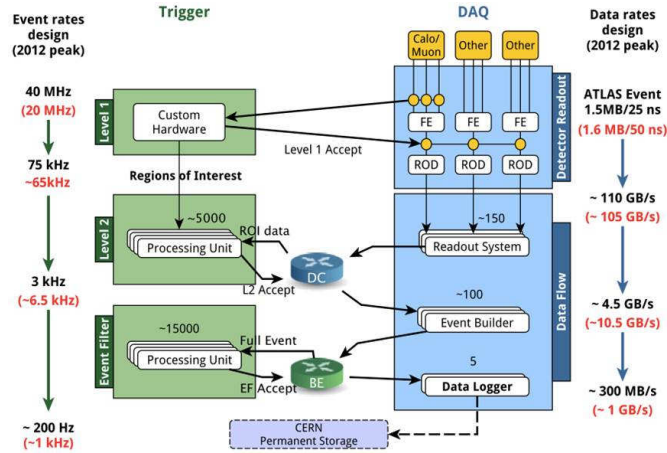

Figure 1: A scheme of the ATLAS trigger system; values in red are indicating the data-taking conditions during the first run of the LHC ('10-'12) while values in black represents the design conditions, expected to be reached during the upcoming run (starting in 2015).

pected to start in 2015) the L2 and EF will be merged in a single software trigger level. Currently, a first upgrade is foreseen in 2018 [3], when real-time tracking capabilities will also be available, followed by a complete renovation of the trigger and detector systems in 2022. We intend to explore the potential improvements attainable in the near future by deploying GPUs in the ATLAS LVL2 muon trigger algorithms. Such algorithms are now implemented as simplified versions and are based on the execution for a large number of times of the same algorithms that reconstruct and match segments of particle trajectories in the detector. The high computing capabilities of GPUs would allow the use of refined algorithms with higher selection efficiency, and thus to maintain the sensitivity to interesting physics signals even at higher luminosity.

In the current ATLAS data acquisition framework it is not possible to include directly a parallel computing device; the integration of the GPU in this environment is done through a server-client structure (Accelerator Process Experiment - APE [4]) that can manage different tasks and their execution on an external coprocessor, such as the GPU. This implementation is flexible, able to deal with different devices having optimized architecture, with a reduced overhead. With the help of this structure it is possible to isolate any trigger algorithm and optimize it for the execution on a GPU (or other parallel architecture device).

This will imply the translation into a parallel computing programming language (CUDA[1] [7]) of the original algorithm and the optimization of the different tasks that can be naturally parallelized. In such a way the dependency of the execution time on the complexity of the processed events will be reduced. A similar approach has been investigated in the past for the deployment of GPUs in different ATLAS algorithms with promising results [5]. The evolution of the foreseen ATLAS trigger system, that will merge the higher level trigger layers in a unique software processing stage, can take even more advantage from the use of a GPU since a more complex algorithm, with offline-like resolution can be implemented on a thousand-core device with significant speedup factors. The timing comparison between the serial and the parallel implementation of the trigger algorithm is done on the data collected in the past year.

3   Muon reconstruction isolation algorithm

The benchmark measurements that has been carried out has focused on one of the algorithms developed for muon reconstruction in ATLAS. In the L2 trigger a candidate muon particle is reconstructed combining three different and sequential algorithms: the first one reconstructs a charged particle track segment in the muon spectrometer [6], a second algorithm matches in space such track segment to a charged particle track reconstructed in the ATLAS Inner Detector (ID) [6], the third evaluates the energy deposits in the electromagnetic and hadronic calorimeters (ECAL, HCAL) [6], as well as the track density in the detector region around the candidate muon trajectory, to check the consistency with a muon crossing the whole ATLAS detector. This third step of the muon reconstruction has been considered for our first test deploying a GPU in the ATLAS trigger system.
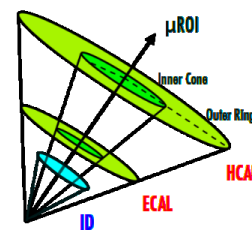

Figure 2: Scheme of the cone used in the muon isolation algorithm.

---

[1]We perform the tests described in this article on a server set up for this purpose including an NVIDIA graphic accelerator, hence the GPU code has been developed in CUDA.

The muon isolation algorithm starts from the spatial coordinates of the candidate muon trajectory and consider a cone of fixed width $\Delta R = \sqrt{\Delta\phi^2 + \Delta\eta^2}$ in the (η,ϕ) ([6]) space around such trajectory. In order to pass the muon track isolation requirement there should be only a limited number of tracks in the considered cone, and these must be consistent with background in the inner ATLAS tracking system. The calorimeter isolation is applied summing the energy deposits in the electromagnetic and hadronic calorimeter cells lying within the considered cone, and requiring this is only a small fraction of the estimated candidate muon energy[2]. Figure 2 shows the definition of the cone used to evaluate the muon isolation in the calorimeter and in the inner detector.

The integration of the GPU and the execution of such algorithm within the ATLAS trigger framework can be summarized in the following steps:

1. retrieve information from the detector: access to the calorimeter cells information;

2. format information needed by the algorithm, namely the cell content, data-taking conditions, and calorimeter configuration information, into a memory buffer;

3. transfer the prepared buffer to the server (APE) which handles the algorithm execution and transfer the buffer to the GPU;

4. algorithm execution on the GPU (or on a CPU in the serial version of the algorithm);

5. transfer of the algorithm results through the APE server back into the ATLAS trigger framework;

Step 1 is the same also in the current implementation of the muon isolation algorithm; in the standard ATLAS trigger implementation (serial) this step is followed directly by the algorithm execution (step 4) on the CPU. Step 2 is needed to optimize the data-transfer toward the GPU; it is important at this stage to convert the object-oriented structures to a *plain* buffer containing the minimum amount of information needed by the algorithm to minimize the CPU→GPU communication latency. Step 3 is implemented through libraries dedicated to the client-server communication, as a part of the APE server. Such server manages the assignement of the task to the GPU for the execution and waits to retrieve the results. To accomplish step 4, the simple algorithm which evaluates the calorimeter isolation has been translated into the CUDA language optimizing the management of the GPU resources in terms of computing CUDA cores and memory usage. Step
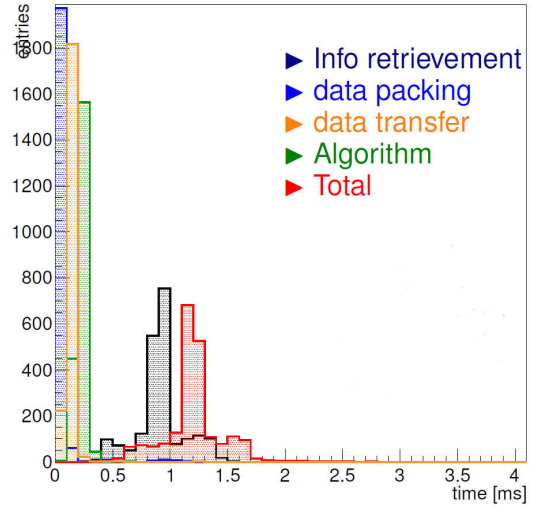

Figure 3: Measurement of the muon isolation algorithm execution time using a Nvidia GTX Titan GPU.

---

[2]The requirement on this energy fraction varies depending on the region of the detector and the desired quality of the reconstructed muon, but this aspect is not relevant for the purpose of these tests.

5 is implemented within the APE server, similarly to step 3, and completes the CPU-GPU communication, reporting the algorithm results in the original framework.

The measurement of the trigger algorithm execution latency has been performed using a server machine containing an Intel Xeon E5-2620 CPU and a Nvidia GTX Titan GPU, re-processing a sample of ATLAS recorded data with no dedicated optimization of the machine. Figure 3 shows the execution latency measured for the several steps and their sum. The overall execution time resulted in being $\Delta t_{GPU}^{tot} \approx$1.2$\pm$0.2 ms when using the GPU, with respect to $\Delta t_{CPU}^{tot} \approx$0.95$\pm$0.15 ms, obtained with the standard execution on CPU. As it is shown in Figure 3 the largest fraction of the time (~ 900 μs) is spent to extract the detector data and convert them from the object-oriented structure to a *flat* format which is suitable for the transfer to the GPU. This contribution to the latency is independent from the the serial or parallel implementation of the algorithm, since it's related to data structure decoding; the current version of the ATLAS framework heavily relies on object-oriented structures, which are not the ideal input for GPUs. The contribution due to the CPU-GPU communication through the client-server structure is found to be $\Delta t_{GPU}^{trans.} \sim$250 μs, which is within the typical time budget of the ATLAS HLT ($\mathcal{O}(10$ ms)). This result confirms the possibility to include GPUs into a HLT system for the execution of parallel algorithms, hence motivates further studies in this direction.

4   Conclusions and Perspectives

From this first study we succesfully deployed a GPU within the ATLAS pre-existing trigger and DAQ framework through a flexible client-server scheme. We observed that the CPU-GPU communication does not introduce a dramatic overhead, which is indeed well within the typical execution latencies of software trigger algorithms, $\mathcal{O}(10$ ms). This result shows that is feasible to consider the GPUs as extremely powerful computing devices for the evolution of the current asynchronous trigger systems. Most of the execution time is devoted to the data extraction from object-oriented structures, which is currently an *external* constraint from the ATLAS trigger framework. This observation, confirmed by similar studies in the ATLAS experiment, focused the attention on this topic; a common effort is ongoing to overcome this problem and benefit at most from the GPU computing power. At the moment two viable approaches are being considered: on one hand it's interesting to consider more complex algorithms that can reduce the time spent for data structure handling to a negligible fraction; on the other hand it is possible to handle *simpler* data structures (raw byte streams from the detector), bypassing most of the currently existing framework. As an example of the first approach the possibility to execute at the trigger level, a refined evaluation of
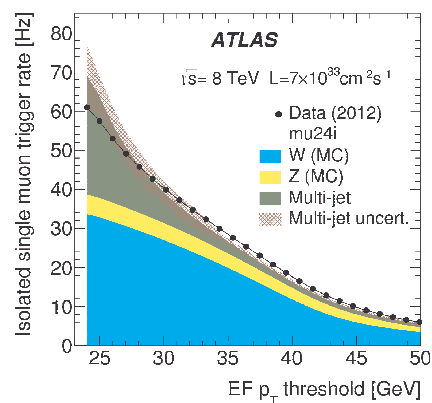

Figure 4: Isolated single muon trigger observed rate as a function of the muon transverse momentum (p_T), compared to simulations for the different expected contributions.

the muon isolation, which would be not sustainable in single-threaded execution on a CPU, is currently under investigation. The muon isolation evaluated in the offline data reprocessing takes into account also environmental corrections due to multiple interactions leading to additional noise in the calorimeter and tracking system. Figure 4 shows the trigger rate as a function of the muon transverse momentum, p_T, for the isolated muon trigger in a data-taking period during 2012. One can see that a relevant fraction of the trigger rate is due to spurious events (multi-jet production). A refined calculation of the muon isolation would reduce such trigger rate, maintaining the efficiency for prompt muon reconstruction high .

As a typical scenario for the second approach, it is possible to decode and process simple data from the muon spectrometer (position and time information) [6] to reconstruct the muon track segment. A similar strategy has been developed in a standalone test considering track reconstruction in the ATLAS Inner Detector, with interesting results [5].

5 Acknowledgements

References

[1] Proceeding of *Estimated speed-up from GPU computation for application to realtime medical imaging* from this same workshop (2014).

[2] ATLAS Collaboration, JINST **3** P08003 (2008).

[3] ATLAS Collaboration, -CERN-LHCC-2011-012 (2012).

[4] The client-server structure is obtained using APE, an ATLAS tool developed independently from this project.

[5] D. Emeliyanov, J. Howard, J. Phys.: Conf. Ser. **396** 012018 (2012).

[6] ATLAS Collaboration, JINST **3** S08003 (2008).

[7] http://docs.nvidia.com/cuda/cuda-c-programming-guide

GPU-based quasi-real-time Track Recognition in Imaging Devices: from raw Data to Particle Tracks

Cristiano Bozza[1], Umut Kose[2], Chiara De Sio[1], Simona Maria Stellacci[1]

[1]Department of Physics University of Salerno, Via Giovanni Paolo II 132, 84084 Fisciano, Italy
[2]CERN, 1211 Geneve 23, Switzerland

Nuclear emulsions as tracking devices have been used by recent experiments thanks to fast automatic microscopes for emulsion readout. Automatic systems are evolving towards GPU-based solutions. Real-time imaging is needed to drive the motion of the microscope axes and 3D track recognition occurs quasi-online in local GPU clusters. The algorithms implemented in the Quick Scanning System are sketched. Most of them are very general and might turn out useful for other detectors.

1 Nuclear emulsions as tracking detectors

Nuclear emulsions have a long history in high-energy physics and recently experienced revived interest in the CHORUS[1], DONUT, PEANUT[2] and OPERA[3] experiments. They provide the best spatial resolution currently available, of the order of 0.1 μm. On the other hand, they have no time resolution, recording all charged tracks since the time of production until photographic development. In common setups, a detector is made up of one or more stacks of films, placed approximately orthogonally to the direction of the incoming particles. Each film has two layers of emulsion coating a transparent plastic base (Fig. 1). Typical dimensions are 50 μm for the thickness of emulsion layers and 200 μm for the base. A nuclear emulsion contains AgBr crystals in a gel matrix. Charged particles sensitise the crystals by ionisation, producing a *latent image*. Development of the film produces metallic Ag grains in place of the latent image, resulting in aligned sequences of grains (*microtracks*), typically 0.3~1 μm in diameter (Fig. 2). In an optical transmission micro-

Figure 1: Nuclear emulsion film.

Figure 2: Nuclear emulsion image. High-energy tracks are orthogonal to plane.

scope, grains appear as dark spots on light background. In white light, the average alignment residuals of grains with respect to the straight line fit is of the order of 50 nm. The depth of
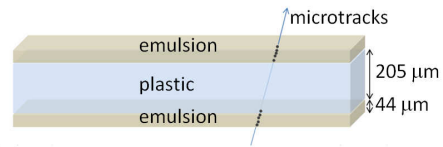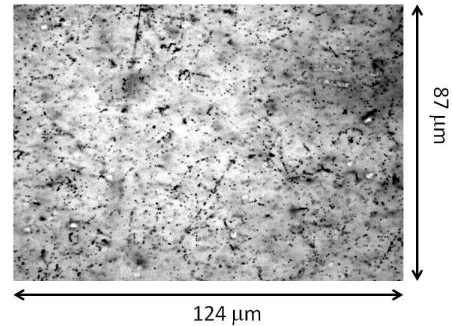
field of the optics is usually below 5 μm. The full thickness of the emulsion layer is spanned by moving the optical axis, thus producing optical tomographies with a set of equally spaced planes (15~31 usually). The image of a particle track in a focal plane is a single grain, not necessarily present in each plane (ionisation is a stochastic process). Because the chemical process of development is just faster for sensitised grains, but it occurs in general for every crystal in a random fashion, many grains appear without having been touched by any particle. Such so-called *fog* grains are overwhelming in number: as a reference, the ratio of *fog* grains to microtrack grains ranges from 10^3 through 10^5. Only 3D alignment is able to single out the few microtrack grains, but also many fake microtracks due to random alignments survive in a single film. Stacking several films allows using coincidence methods to improve background rejection.

It is worth noticing that this resembles the situation of an electronic detector in which background hits due to noise or pile-up overwhelm track hits. Normally, electronic detectors use a time trigger to reduce combinatorial complexity, but in emulsion this is not possible. It is reasonable to think that a tracking algorithm working in emulsion finds an even easier environment if fed with data from other detectors, such as cloud chambers or planes of silicon pads.

2 Data from nuclear emulsion

A nuclear emulsion film has typically a rectangular shape, spanning several tens of cm in both directions. The whole surface is scanned by acquiring optical tomographies in several fields of view with a repetitive motion. An electronic shutter ensures that images are only negligibly affected by motion blur. In the ESS (*European Scanning System* [4], [5], [6], [7], [8]), developed in the first decade of the 21$^{\text{st}}$ century, the XY microscope axes hold steady while the Z axis (optical axis) moves and images are grabbed on the fly and sent to vision processors. Its evolution, named QSS (*Quick Scanning System*), moves the X axis continuously. Hence, each *view* (an optical tomography) of ESS is a cuboid, whereas those of QSS are skewed prisms. Images are acquired by fast monochromatic sensors (CMOS) mounted on the optical axis, capable of 200~400 frames per second, each frame being 1~4 MPixel (or more for future sensors) using 8 bits/pixel. The resulting data rate ranges from 0.5 GB/s to 2 GB/s. The linear size of the image stored in a pixel, with common optics, is of the order of 0.3 μm. The full size of the field of view is 400×320 μm^2 for ESS, 770×550 μm^2 for QSS.

2D image processing used to be shared in the case of ESS by an industrial vision processor (Matrox Odyssey) based on an FPGA and by the host workstation CPU. It consists of several substeps:

1. grey level histogram computation and stretching – used to compensate for varying light yield of the lamp;

2. FIR (*Finite Impulse Response*) processing with a 5×5 kernel and comparison of filter output to a threshold – selects dark pixels on light background, producing a binary image;

3. reduction of sequences of pixels on horizontal scan lines to segments;

4. reduction of segments in contact on consecutive scan lines to clusters of dark pixels – produces grains to be used for tracking.

In the ESS, steps 1 and 2 are performed on the FPGA-based device. Steps 3 and 4 are executed by the host workstation CPU. For the same task, QSS uses a GPU board hosted in

the workstation that controls the microscope: common choices are double-GPU boards such as NVidia GeForce GTX 590 or GTX 690. A single board can do everything without intervention of the host workstation CPU. The first 3 steps are quite natural applications for GPU's. One single GPU can be up to 7 times faster than the Odyssey, reducing the price by an order of magnitude. Steps 4 and 5 require special attention in code development, because they are reduction operations with an *a priori* unknown number of output objects. In step 4 each thread takes care of a single scan line of the image. In step 5 a recursive implementation has been chosen: at iteration n, the segments on scan line $i \times 2^n$ are compared to those on line $i \times 2^n - 1$ and the related dark clusters are merged together. Indeed, steps 4 and 5 are the slowest, and they define the total processing speed, which is 400 MB/s for GTX 590. The system is scalable, and more GPU boards or more powerful ones can be added if needed. The output of this step is, for each view, a set of clusters of dark pixels on light background, each one being encoded with its X, Y, Z coordinates and size in pixels. Automatic readout uses the distribution of clusters to continuously adjust the Z axis drive of the tomography. 60∼124 MB of image data are encoded to 8∼16 MB cluster block, ready for local storage or to be transmitted over the network for further processing. In the latter case, a RAM disk area is used as a buffer to distribute data to a cluster of GPU's. Tracking servers and the workload manager provide a command-line interface for administration and have an embedded lightweight Web server, originally developed for the SySal project ([7]), that provides a graphical interface for human access and is the backbone for an HTTP-based network protocol for control messages needed for workload assignment.

Particle tracks can cross the boundary of a view, and tracking must be able to do the same. The alignment tolerance to recognise real tracks and discard background cannot exceed 0.5 μm and is usually smaller. The motion control electronics is capable of position feedback, but triaxial vibrations due to combined motion arise, well beyond 1 μm in a single image. Corrections to raw data are needed before they can be used to recognise microtracks. Some such corrections, sketched in Fig. 3, depend on optical aberrations and are systematic effects that can be computed off-line from small data-sets and then applied on each incoming cluster block (view):

1. Spherical aberrations: XY and Z curvature;

2. trapezium distortions: dependence of magnification factor on X and Y;

3. magnification-vs.-Z dependence;

4. camera tilt: rotation in the XY plane;

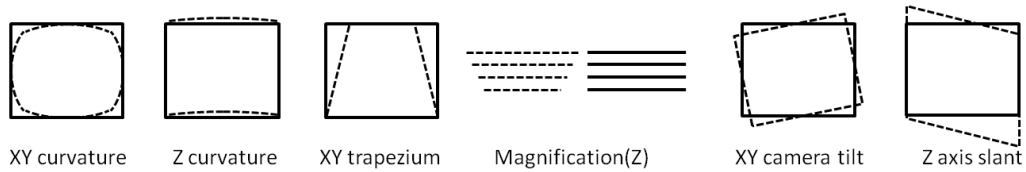5. optical axis slant: X-Z and Y-Z couplings due to inclined optical axis.

Figure 3: Optical distortion of emulsion image.

Most of the computing power needed for data correction is taken by other effects that are purely stochastic: vibrations due to motion, increased by mechanical wear and aging are unpredictable and spoil the alignment of images within the same view and between consecutive views. Because the depth-of-field usually exceeds the Z pitch between two images in the same sequence taken while the Z axis moves, a sizable fraction of the dark clusters in one image will appear also in the next. Pattern matching works out the relative shift between the images, usually within 1 μm. This procedure requires scanning a square region of possible values of the plane shift vector. Combinatorial complexity is reduced by arranging clusters in a 2D grid of cells and considering only pair matching within each cell. The shift vector that maximizes the number of combinations is the best approximation of the misalignment between the images. Likewise, despite position feedback of all axes, a whole tomography is misaligned with respect to the next. Film scanning is performed so as to leave 30∼40 μm overlap between adjacent views. The dark clusters found in the overlap region are used to perform 3D pattern matching, in the same way as for 2D pattern matching. The standard deviation of the distribution of residuals is of the order of 150 nm (Fig. 4) for X and Y, and 2.6 μm for Z.
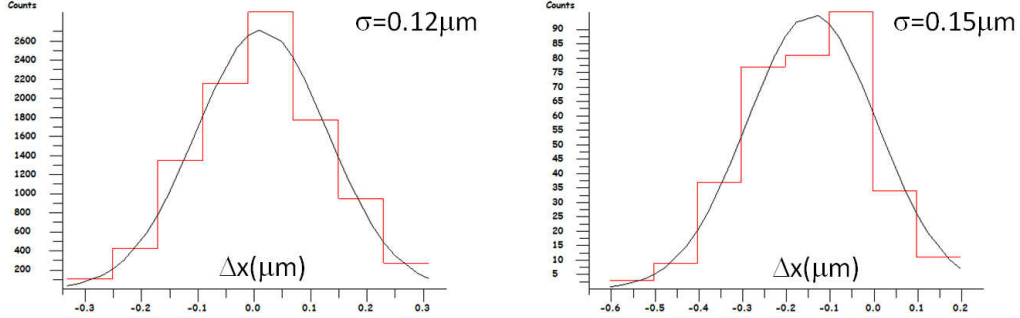

Figure 4: Left: precision of image-to-image alignment in the same tomographic sequence. Right: precision of relative alignment of two tomographic sequences.

3   Track reconstruction

Reconstructing 3D tracks from a set of primitive objects such as emulsion grain images or electronic detector hits is a common task in high-energy physics. The method depicted in the following would work in all cases of straight tracks, i.e. absent or weak magnetic field and scattering effects negligible with respect to the granularity of the detector. Because ionisation is a stochastic process, the algorithm does not require that a microtrack has a dark cluster in every image; furthermore, the notion of sampling planes is deliberately discarded to keep the algorithm general. It just needs to work on 3D pointlike objects, each having a weight, which corresponds to the number of pixels of the dark cluster in this case (e.g. it may be the deposited energy in a silicon counter). Each pair of dark clusters defines a *seed*, i.e. a straight line in 3D space; other clusters are expected to be aligned with it within proper tolerance (Fig. 5-left). In thin emulsion layers, microtracks are formed with 6∼40 clusters, depending on the local sensitivity, on statistical fluctuations and on track angle (the wider the inclination with respect to the

thickness dimension, the longer the path length). Furthermore, high-energy particles ionise less than slow ones. Such reasons suggest not to filter out too many possible pairs of clusters as track seeds, considering all those that are within geometrical acceptance. Physics or experimental conditions constrain the angular acceptance to a region of interest. Optimum thread allocation to computing cores demands that constraints be enforced constructively instead of generating all seeds and discarding invalid ones. Dark clusters are arranged in a 2D grid of prisms, each one spanning the whole emulsion thickness (Fig. 5-right). The angular region of interest is scanned in angular steps. At each step the prisms are generated skewed with the central slope of the step. This ensures that seeds that are very far from the acceptance window will not even be built and followed.
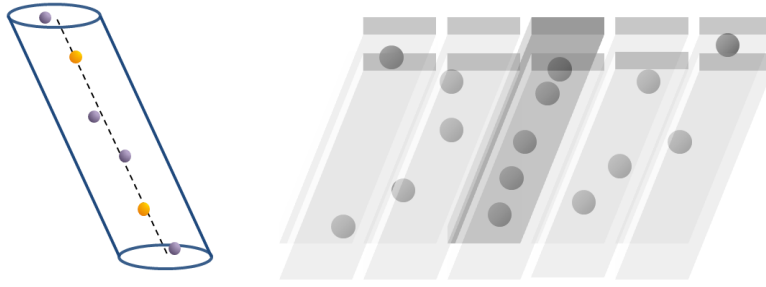

Figure 5: Left: microtrack seeds and grains. Right: 2D grid of prisms to build seeds. A prism containing a track is shown darker.

With common operational parameters, the number of useful combinations ranges within 10^7 and 10^9 per tomographic sequence, depending on the amount of random alignments and *fog*. For each seed, one thread scans the containing prism to find aligned clusters and build the microtrack. This procedure naturally produces clones of the same track, normally in the range 2∼4:1. They are discarded by comparing the geometrical parameters of all neighbor microtracks, neighborhood being defined by means of a 2D spatial grid.

4 Performances and outlook

Performances in terms of processing time vs. grain or microtrack density has been estimated using several NVidia GPU's. The results are shown in Figures 6, 7, 8 and 9.

Denoting the grain density (grains per tomography) with N, the number of seed combinations is of order N^2, and the search for clusters to attach to the seed is of order N^3. Results show that processing steps of high computational complexity are not overwhelming for typical operational conditions.

While one would expect the processing time to scale inversely with the number of available cores, more recent GPU's perform proportionally worse than older ones. The reason for that is to be sought mostly in branch divergence, which affects more the multiprocessors with larger number of cores. In some cases the divergence is due to threads exiting while others keep running. This could be eliminated, but it's not clear whether the additional complications of code would pay off. In other cases the divergence is due to the specific coding style used in
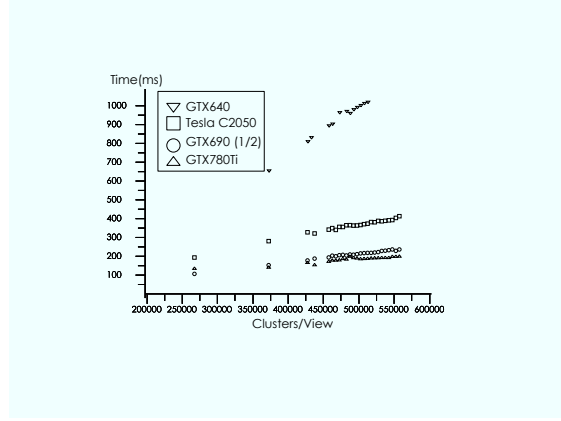
Figure 6: Absolute time (ms) for cluster correction and alignment. For GTX690, only one GPU is considered.
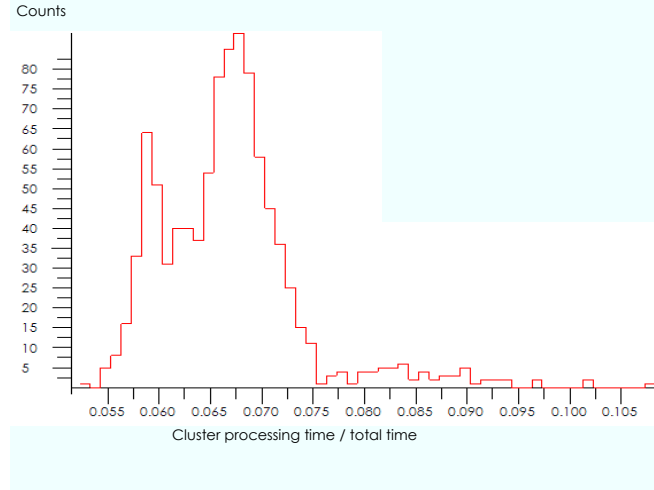
Figure 7: Fraction of total time used for cluster correction and alignment.

the implementation, and could be reduced by fine-tuning the logic: e.g, the kernel that merges track clones takes about $1/3$ of the total time, and removing or taming its divergence offers good chances of overall speed-up. The relative improvements of a GPU-based system over traditional technologies can effectively be estimated by the cost of hardware. For QSS, taking data at $40{\sim}90$ cm^2/h with 1 GTX 590 for cluster recognition + 6 GTX 690 for alignment and tracking costs about 5.5 times less than the hardware of ESS taking data at 20 cm^2/h on the same emulsion films. The power consumption is similar, although it is worth noticing the data taking speed increase. The GPU-based system is also more modular and scalable.
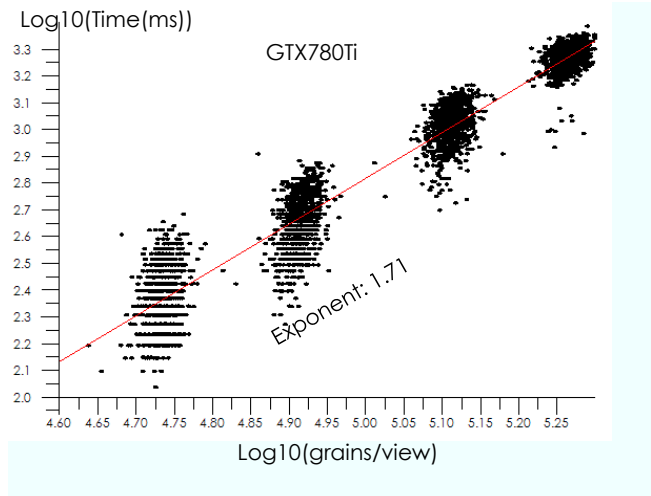
Figure 8: Dependency of tracking time (GTX780Ti) on the number of grain clusters/view (dark clusters with minimum size constraint).
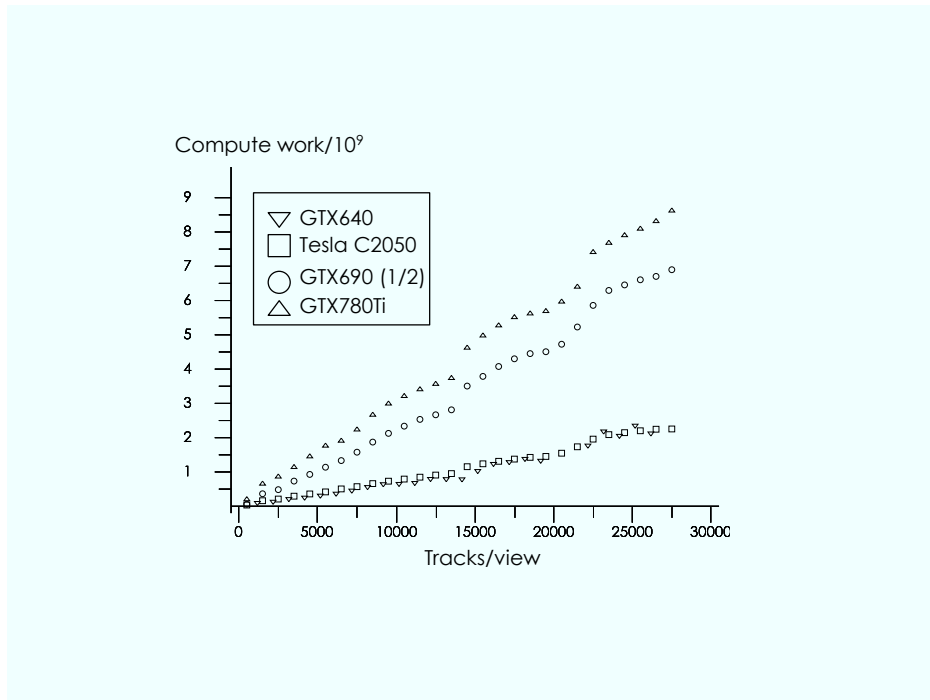
Figure 9: Compute work ($Cores \times Clock \times Time$, arbitrary units) for several boards. For GTX690, only one GPU is considered.

References

[1] E. Eskut *et al.*, Nucl. Phys. B **793** 326 (2008).

[2] A. Aoki *et al.*, New. J. Phys. **12** 113028 (2010).

[3] R. Acquafredda *et al.*, J. Inst. **4** P04018 (2009).

[4] N. Armenise *et al.*, Nucl. Inst. Meth. A **552** 261 (2005).

[5] L. Arrabito *et al.*, Nucl. Inst. Meth. A **568** 578 (2007).

[6] L. Arrabito *et al.*, J. Inst. **2** P05004 (2005).

[7] C. Bozza *et al.*, Nucl. Inst. Meth. A. **703** 204 (2013).

[8] M. De Serio *et al.*, Nucl. Inst. Meth. A **554** 247 (2005).

GPGPU for track finding in High Energy Physics

L. Rinaldi[1], *M. Belgiovine*[1], *R. Di Sipio*[1], *A. Gabrielli*[1], *M. Negrini*[2], *F. Semeria*[2], *A. Sidoti*[2], *S. A. Tupputi*[3], *M. Villa*[1]

[1]Bologna University and INFN, via Irnerio 46, 40127 Bologna, Italy
[2]INFN-Bologna,v.le Berti Pichat 6/2, 40127 Bologna, Italy
[3]INFN-CNAF,v.le Berti Pichat 6/2, 40127 Bologna, Italy

The LHC experiments are designed to detect large amount of physics events produced with a very high rate. Considering the future upgrades, the data acquisition rate will become even higher and new computing paradigms must be adopted for fast data-processing: General Purpose Graphics Processing Units (GPGPU) is a novel approach based on massive parallel computing. The intense computation power provided by Graphics Processing Units (GPU) is expected to reduce the computation time and to speed-up the low-latency applications used for fast decision taking. In particular, this approach could be hence used for high-level triggering in very complex environments, like the typical inner tracking systems of the multi-purpose experiments at LHC, where a large number of charged particle tracks will be produced with the luminosity upgrade. In this article we discuss a track pattern recognition algorithm based on the Hough Transform, where a parallel approach is expected to reduce dramatically the execution time.

1   Introduction

Modern High Energy Physics (HEP) experiments are designed to detect large amount of data with very high rate. In addition to that weak signatures of new physics must be searched in complex background condition. In order to reach these achievements, new computing paradigms must be adopted. A novel approach is based on the use of high parallel computing devices, like Graphics Processing Units (GPU), which delivers such high performance solutions to be used in HEP. In particular, a massive parallel computation based on General Purpose Graphics Processing Units (GPGPU) [1] could dramatically speed up the algorithms for charged particle tracking and fitting, allowing their use for fast decision taking and triggering. In this paper we describe a tracking recognition algorithm based on the Hough Transform [2, 3, 4] and its implementation on Graphics Processing Units (GPU).

2   Tracking with the Hough Transform

The Hough Transform (HT) is a pattern recognition technique for features extraction in image processing, and in our case we will use a HT based algorithm to extract the tracks parameters from the hits left by charged particles in the detector. A preliminary result on this study has been already presented in [5]. Our model is based on a cylindrical multi-layer silicon detector

installed around the interaction point of a particle collider, with the detector axis on the beam-line. The algorithm works in two serial steps. In the first part, for each hit having coordinates (x_H, y_H, z_H) the algorithm computes all the circles in the $x - y$ transverse plane passing through that hit and the interaction point, where the circle equation is $x^2 + y^2 - 2Ax - 2By = 0$, and A and B are the two parameters corresponding to the coordinates of the circle centre. The circle detection is performed taking into account also the longitudinal (θ) and polar (ϕ) angles. For all the θ, ϕ, A, B, satisfying the circle equation associated to a given hit, the corresponding $M_H(A, B, \theta, \phi)$ Hough Matrix (or Vote Matrix) elements are incremented by one. After computing all the hits, all the M_H elements above a given threshold would correspond to real tracks. Thus, the second step is a local maxima search among the M_H elements.

In our test, we used a dataset of 100 simulated events (pp collisions at LHC energy, Minimum Bias sample with tracks having transverse momentum $p_T > 500$ MeV), each event containing up to 5000 particle hits on a cylindrical 12-layer silicon detector centred on the nominal collision point. The four hyper-dimensions of the Hough space have been binned in $4 \times 16 \times 1024 \times 1024$ along the corresponding A, B, θ, ϕ parameters.

The algorithm performance compared to a χ^2 fit method is shown in Fig. 1: the $\rho = \sqrt{A^2 + B^2}$ and $\varphi = \tan^{-1}(B/A)$ are shown together with the corresponding resolutions.
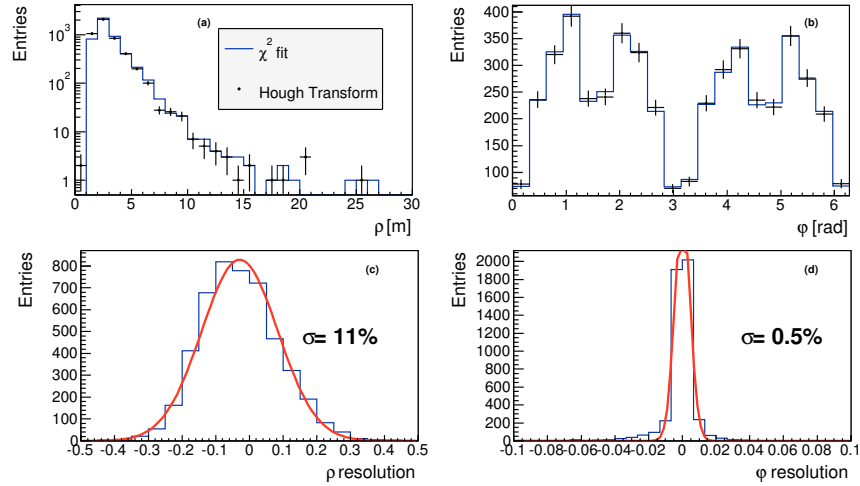
Figure 1: Hough Transform algorithm compared to χ^2 fit. (a) ρ distribution; (b) φ distribution; (c) ρ resolution; (d) φ resolution.

3   SINGLE-GPU implementation

The HT tracking algorithm has been implemented in GPGPU splitting the code in two kernels, for Hough Matrix filling and searching local maxima on it. Implementation has been performed both in CUDA [1] and OpenCL [6]. GPGPU implementation schema is shown in Fig. 2.

Concerning the CUDA implementation, for the M_H filling kernel, we set a 1-D grid over all the hits, the grid size being equal to the number of hits of the event. Fixed the (θ, ϕ) values, a thread-block has been assigned to the A values, and for each A, the corresponding

B is evaluated. The $M_H(A, B, \theta, \phi)$ matrix element is then incremented by a unity with an `atomicAdd` operation. The M_H initialisation is done once at first iteration with `cudaMallocHost` (pinned memory) and initialised on device with `cudaMemset`. In the second kernel, the local maxima search is carried out using a 2-D grid over the θ, ϕ parameters, the grid dimension being the product of all the parameters number over the maximum number of threads per block $(N_\phi \times N_\theta \times N_A \times N_B)$/`maxThreadsPerBlock`, and 2-D threadblocks, with `dimXBlock`=N_A and `dimYBlock=MaxThreadPerBlock`/N_A. Each thread compares one $M_H(A, B, \theta, \phi)$ element to its neighbours and, if the biggest, it is stored in the GPU shared memory and eventually transferred back. With such big arrays the actual challenge lies in optimizing array allocation and access and indeed for this kernel a significant speed up has been achieved by tuning matrix access in a coalesced fashion, thus allowing to gain a crucial computational speed-up. The OpenCL
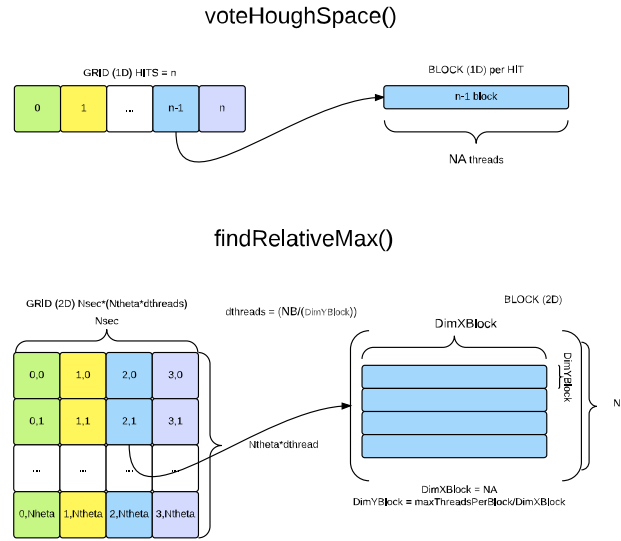
Figure 2: GPGPU implementation schema of the two Hough Transform algorithm kernels.

implementation has been done using a similar structure used for CUDA. Since in OpenCL there is no direct pinning memory, a device buffer is mapped to an already existing *memallocated* host buffer (`clEnqueueMapBuffer`) and dedicated kernels are used for matrices initialisation in the device memory. The memory host-to-device buffer allocation is performed concurrently and asynchronously, saving overall transferring time.

3.1   SINGLE-GPU results

The test has been performed using the NVIDIA [1] GPU boards listed in table 1. The GTX770 board is mounted locally on a desktop PC, the Tesla K20 and K40 are installed in the INFN-CNAF HPC cluster.

The measurement of the execution time of all the algorithm components has been carried out as a function of the number of hits to be processed, and averaging the results over 100 independent runs. The result of the test is summarised in Fig. 3. The total execution time comparison between GPUs and CPU is shown in Fig. 3a, while in Fig. 3b the details about

| Device | NVIDIA | NVIDIA | NVIDIA |
specification	GeForce GTX770	Tesla K20m	Tesla K40m
Performance (Gflops)	3213	3542	4291
Mem. Bandwidth (GB/s)	224.2	208	288
Bus Connection	PCIe3	PCIe3	PCIe3
Mem. Size (MB)	2048	5120	12228
Number of Cores	1536	2496	2880
Clock Speed (MHz)	1046	706	1502

Table 1: Computing resources setup.

the execution on different GPUs are shown. The GPU execution is up to 15 times faster with respect to the CPU implementation, and the best result is obtained for the CUDA algorithm version on the GTX770 device. The GPUs timing are less dependent on the number of the hits with respect to CPU timing.

The kernels execution on GPUs is even faster with respect to CPU timing, with two orders of magnitude GPU-CPU speed up, as shown in Figs. 3c and 3e. When comparing the kernel execution on different GPUs (Figs. 3d) and 3f), CUDA is observed to perform slightly better than OpenCL. Figure 3g shows the GPU-to-CPU data transfer timings for all devices together with the CPU I/O timing, giving a clear idea of the dominant part of the execution time.

4 MULTI-GPU implementation

Assuming that the detector model we considered could have multiple readout boards working independently, it is interesting to split the workload on multiple GPUs. We have done this by splitting the transverse plane in four sectors to be processed separately, since the data across sectors are assumed to be read-out independently. Hence, a single HT is executed for each sector, assigned to a single GPU, and eventually the results are merged when each GPU finishes its own process. The main advantage is to reduce the load on a single GPU by using lightweight Hough Matrices and output structures. Only CUDA implementation has been tested, using the same workload schema discussed in Sec. 3, but using four $M_H(A, B, \theta)$, each matrix processing the data of a single ϕ sector.

4.1 MULTI-GPU results

The multi-GPU results are shown in Fig. 4. The test has been carried out in double configuration, separately, with two NVIDIA Tesla K20 and two NVIDIA Tesla K40. The overall execution time is faster with double GPUs in both cases, even if timing does not scale with the number of GPUs. An approximate half timing is instead observed when comparing kernels execution times. On the other hand, the transferring time is almost independent on the number of GPUs, this leading the overall time execution.
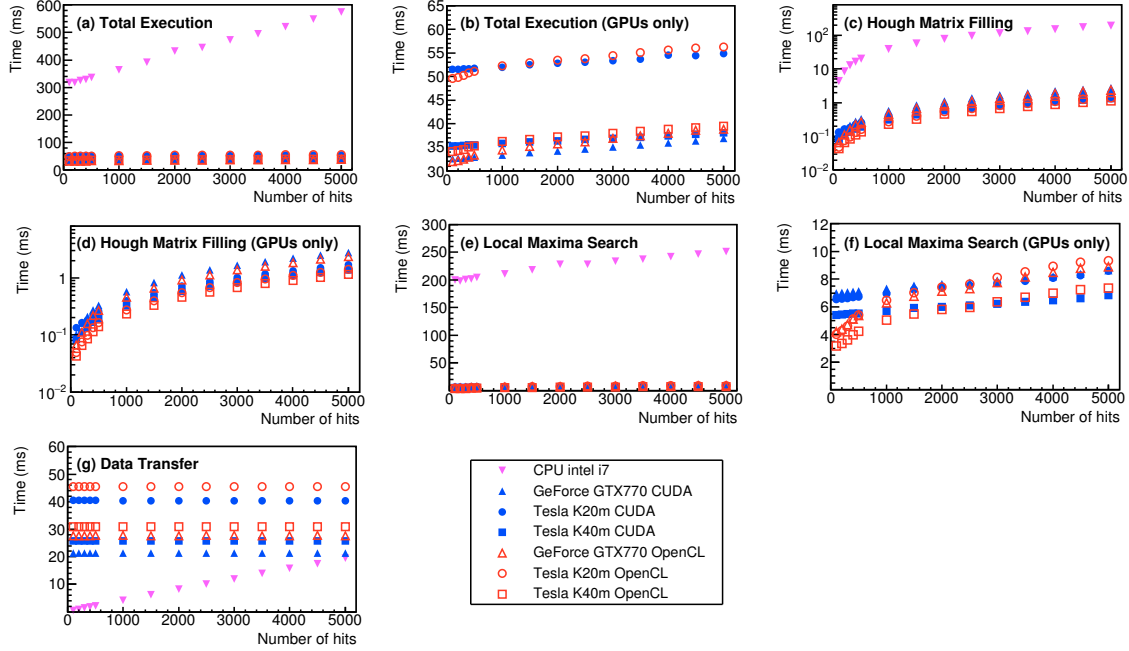
Figure 3: Execution timing as a function of the number of analysed hits. (a) Total execution time for all devices; (b) Total execution time for GPU devices only; (c) M_H filling time for all devices; (d) M_H filling timing for GPU devices only; (e) local maxima search timing for all devices; (f) local maxima search timing for GPU devices only; (g) device-to-host transfer time (GPUS) and I/O time (CPU).

5 Conclusions

A pattern recognition algorithm based on the Hough Transform has been successfully implemented on CUDA and OpenCL, also using multiple devices. The results presented in this paper show that the employment of GPUs in situations where time is critical for HEP, like triggering at hadron colliders, can lead to significant and encouraging speed-up. Indeed the problem by itself offers wide room for a parallel approach to computation: this is reflected in the results shown where the speed-up is around 15 times better than what achieved with a normal CPU. There are still many handles for optimising the performance, also taking into account the GPU architecture and board specifications. Next steps of this work go towards an interface to actual experimental frameworks, including the management of the experimental data structures and testing with more graphics accelerators and coprocessor.

References

[1] NVidia Corporation URL http://www.nvidia.com/
    NVidia Corporation URL http://www.nvidia.com/object/gpu.html
    NVidia Corporation URL http://www.nvidia.com/object/cuda_home_new.html

[2] P. Hough; 1959 *Proc. Int. Conf. High Energy Accelerators and Instrumentation* **C590914** 554558
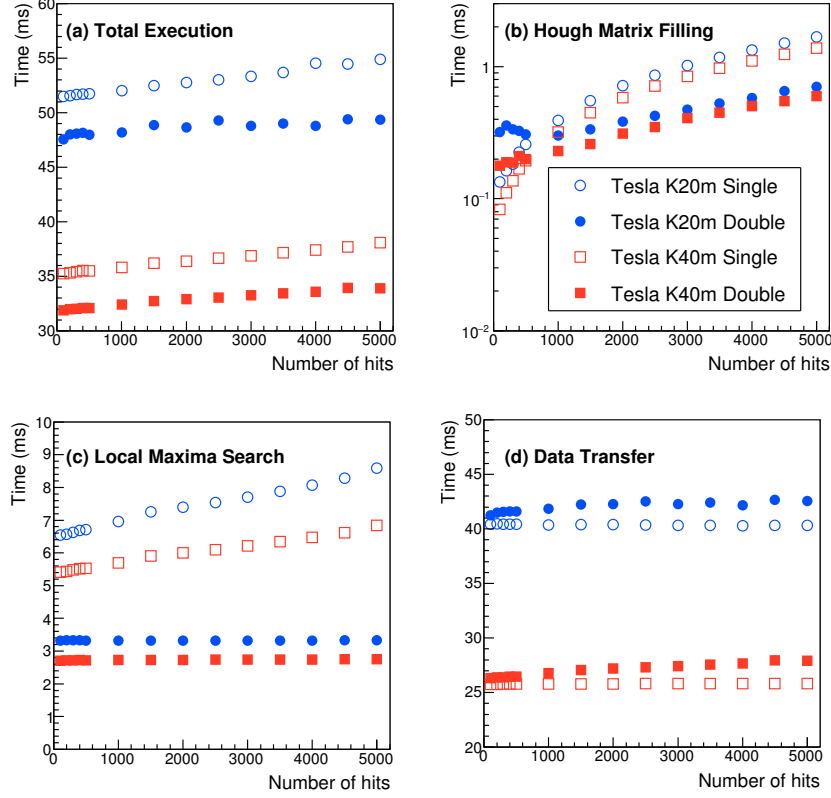
Figure 4: Execution timing as a function of the number of the hits for multi-GPU configuration. (a) Total execution time; (b) M_H filling timing; (c) local maxima search timing; (d) device-to-host transfer time.

[3] V Halyo et al; 2014 *JINST* **9** P04005

[4] M R Buckley, V Halyo, P Lujan; 2014 http://arxiv.org/abs/1405.2082
     P. Hough, 1962, United States Patent 3069654.

[5] S Amerio et al; PoS (TIPP2014) 408

[6] OpenCL, Khronos Group URL https://www.khronos.org/opencl/

FLES – First Level Event Selection Package for the CBM Experiment

Valentina Akishina[1,2], *Ivan Kisel*[1,2,3], *Igor Kulakov*[1,3], *Maksym Zyzak*[1,3]

[1]Goethe University, Grueneburgplatz 1, 60323 Frankfurt, Germany
[2]FIAS Frankfurt Institute for Advanced Studies, Ruth-Moufang-Str. 1, 60438 Frankfurt, Germany
[3]GSI Helmholtz Center for Heavy Ion Research, Planckstr. 1, 64291 Darmstadt, Germany

The CBM (Compressed Baryonic Matter) experiment is a future heavy-ion experiment at the future Facility for Anti-Proton and Ion Research (FAIR, Darmstadt, Germany). First Level Event Selection (FLES) in the CBM experiment will be performed on-line on a dedicated processor farm. An overview of the on-line FLES processor farm concept, different levels of parallel data processing down to the vectorization, implementation of the algorithms in single precision, memory optimization, scalability with respect to number of cores, efficiency, precision and speed of the FLES algorithms are presented and discussed.

1 Introduction

The CBM (Compressed Baryonic Matter) experiment [1] is an experiment being prepared to operate at the future Facility for Anti-Proton and Ion Research (FAIR, Darmstadt, Germany). Its main focus is the measurement of very rare probes, that requires interaction rates of up to 10 MHz. Together with the high multiplicity of charged particles produced in heavy-ion collisions, this leads to huge data rates of up to 1 TB/s. Most trigger signatures are complex (short-lived particles, e.g. open charm decays) and require information from several detector sub-systems.

The First Level Event Selection (FLES) package [2] of the CBM experiment is intended to reconstruct the full event topology including tracks of charged particles and short-lived particles. The FLES package consists of several modules: track finder, track fitter, particle finder and physics selection. As an input the FLES package receives a simplified geometry of the tracking detectors and the hits, which are created by the charged particles crossing the detectors. Tracks of the charged particles are reconstructed by the Cellular Automaton (CA) track finder [3] using to the registered hits. The Kalman filter (KF) based track fit [4] is used for precise estimation of the track parameters. The short-lived particles, which decay before the tracking detectors, can be reconstructed via their decay products only. The KF Particle Finder, which is based on the KFParticle package [2], is used in order to find and reconstruct the parameters of short-lived particles by combining already found tracks of the long-lived charged particles. The KF Particle Finder also selects particle-candidates from a large number of random combinations. In addition, a module for quality assurance is implemented, that allows to control the quality of the reconstruction at all stages. It produces an output in a simple ASCII format, that can be

interpreted later as efficiencies and histograms using the ROOT framework. The FLES package is platform and operating system independent.

The FLES package in the CBM experiment will be performed on-line on a dedicated many-core CPU/GPU cluster. The FLES algorithms have to be therefore intrinsically local and parallel and thus require a fundamental redesign of the traditional approaches to event data processing in order to use the full potential of modern and future many-core CPU/GPU architectures. Massive hardware parallelization has to be adequately reflected in mathematical and computational optimization of the algorithms.

2   Kalman Filter (KF) track fit library

Searching for rare interesting physics events, most of modern high energy physics experiments have to work under conditions of still growing input rates and regularly increasing track multiplicities and densities. High precision of the track parameters and their covariance matrices is a prerequisite for finding rare signal events among hundreds of thousands of background events. Such high precision is usually obtained by using the estimation algorithms based on the Kalman filter (KF) method. In our particular case, the KF method is a linear recursive method for finding the optimum estimation of the track parameters, grouped as components into the so-called state vector, and their covariance matrix according to the detector measurements.

The Kalman filter based library for track fitting includes following tracking algorithms: track fit based on the conventional Kalman filter; track fit based on the square root Kalman filter; track fit based on the Kalman filter with UD factorization of the covariance matrix; track smoother based on the listed above approaches and deterministic annealing filter based on the listed above track smoothers.

High speed of the reconstruction algorithms on modern many-core computer architectures can be accomplished by: optimizing with respect to the computer memory, in particular declaring all variables in single precision, vectorizing in order to use the SIMD (Single Instruction, Multiple Data) instruction set and parallelizing between cores within a compute node.

Several formulations of the Kalman filter method, such as the square root KF and the UD KF, increase its numerical stability in single precision. All algorithms, therefore, can be used either in double or in single precision.

The vectorization and parallelization of the algorithms are done by using of: header files, Vc vector classes, Intel TBB, OpenMP, Intel ArBB and OpenCL.

The KF library has been developed and tested within the simulation and reconstruction framework of the CBM experiment, where precision and speed of the reconstruction algorithms are extremely important.

When running on CPU the scalability with respect to the number of cores is one of the most important parameters of the algorithm. Figure 1 shows the scalability of the vectorized KF algorithm. The strong linear behavior shows, that with further increase of the number of cores on newer CPUs the performance of the algorithm will not degrade and the maximum speed will be reached. The stair-like dependence appears because of the Intel Hyper-Threading technology, which allows to run two threads per core and gives about 30% of performance advantage. The scalability on the Intel Xeon Phi coprocessor is similar to CPU with four threads per core running simultaneously.

In case of the graphic cards the set of tasks is divided into working groups and distributed among compute units (or streaming multiprocessors) by OpenCL and the load of each compute
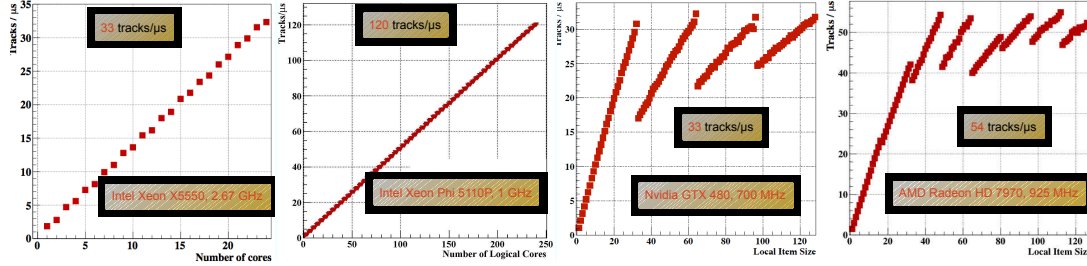
Figure 1: Portability of the Kalman filter track fit library on different many-core CPU/Phi/GPU architectures.

unit is of the particular importance. Each working group is assigned to one compute unit and should scale within it with respect to the number of tasks in the group. Figure 1 shows that the algorithm scales linearly on the graphic cards up to the number of cores in one compute unit (for Nvidia GTX480 — 32, for AMD Radeon HD 7970 — 16). Then the drop appears, because when first 32 (for Nvidia) or 16 (for AMD) tasks are processed, only one task is left and all other cores of the compute unit are idle. Increasing the number of tasks in the group further the speed reaches the maximum with the number of tasks dividable by the number of cores in the compute unit. Due to the overhead in tasks distribution the maximum performance is reached when the number of tasks in the group is two-three times more than the number of cores.

3 Cellular Automaton (CA) track finder

Every track finder must handle a very specific and complicated combinatorial optimization process, grouping together one- or two-dimensional measurements into five-dimensional tracks.

In the Cellular Automaton (CA) method first (1) short track segments, so-called cells, are created. After that the method does not work with the hits any more but instead with the created track segments. It puts neighbor relations between the segments according to the track model here and then (2) one estimates for each segment its possible position on a track, introducing in such a way position counters for all segments. After this process a set of tree connections of possible track candidates appears. Then one starts with the segments with the largest position counters (3) and follows the continuous connection tree of neighbors to collect the track segments into track candidates. In the last step (4) one sorts the track candidates according to their length and χ^2-values and then selects among them the best tracks.

The majority of signal tracks (decay products of D-mesons, charmonium, light vector mesons) are particles with momentum higher than 1 GeV/c originating from the region very close to the collision point. Their reconstruction efficiency is, therefore, similar to the efficiency of high-momentum primary tracks that is equal to 97.1%. The high-momentum secondary particles, e.g. in decays of K_s^0 and Λ particles and cascade decays of Ξ and Ω, are created far from the primary vertex, therefore their reconstruction efficiency is lower – 81.2%. Significant multiple scattering of low-momentum tracks in the material of the detector system and large curvature of their trajectories lead to lower reconstruction efficiencies of 90.4% for primary tracks and of 51.1% for secondary low-momentum tracks. The total efficiency for all tracks is

88.5% with a large fraction of low-momentum secondary tracks. The levels of clones (double found tracks) and of ghost (wrong) tracks are 0.2% and 0.7% respectively. The reconstruction efficiency of the CA track finder is stable with respect to the track multiplicity.

The high track finding efficiency and the track fit quality are crucial, especially for reconstruction of the short-lived particles, which are of the particular interest for the CBM experiment. The reconstruction efficiency of short-lived particles depends quadratically on the daughter track reconstruction efficiency in case of two-particle decays. The situation becomes more sensitive for decays with three daughters and for decay chains. The level of a combinatorial background for short-lived particles depends strongly on the track fit quality. The correct estimation of the errors on the track parameters improves distinguishing between the signal and the background particle candidates, and thus to suppress the background. The ghost (wrong) tracks usually have large errors on the track parameters and therefore are easily combined with other tracks into short-lived particle candidates, thus a low level of ghost tracks is also important to keep the combinatorial background low. As a result, the high track reconstruction efficiency and the low level of the combinatorial background improve significantly the event reconstruction and selection by the FLES package.

4   4-Dimensional time-based event building

Since resolving different events is a non-trivial task in the CBM experiment, the standard reconstruction routine will include an event building, the process of defining exact borders of events within a time-slice and grouping tracks into even-corresponding clusters, which they originate from. For this task an efficient time-based tracking is essential. Since the CA track finder is proved to be fast and stable with respect to the track multiplicity, the next step towards the time-slice based reconstruction would be the implementation of time measurements.
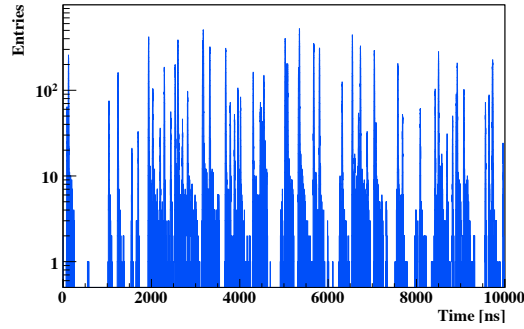
Figure 2: Part of a time-slice with 100 minimum bias events. With blue color the distribution of hit time measurements in a time-slice is shown.
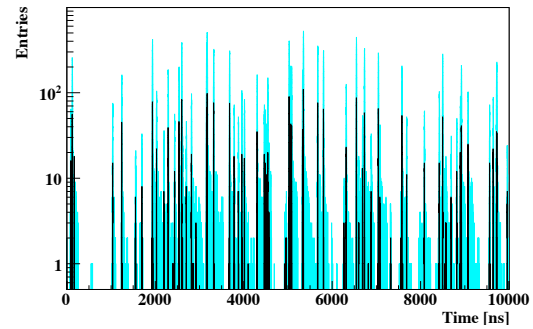
Figure 3: Part of a time-slice with 100 minimum bias events. With light blue color the initial distribution of hit measurements is reproduced, black color shows time measurements of reconstructed tracks.

In order to introduce a time measurement into the reconstruction procedure to each minimum bias event in a 100 events group an event start time was assigned during simulation phase. The start time was obtained with the Poisson distribution, assuming the interaction

rate of 10^7 Hz. A time stamp we assign to a certain hit consists of this event start time plus the time shift due to the time of flight, which is different for all hits. In order to obtain a time measurement for a hit we then smear a time stamp according to a Gaussian distribution with a sigma value of the detector resolution of 5 ns.

After introducing the time measurement we can use the time information in the CA track finder. We do not allow to build triplets out of hits, which time difference is greater than 3σ of the detector time resolution. It is a very good approximation, since the time of flight between the detector planes is negligible in comparison to the detection precision. Apart from that, we perform the reconstruction procedure in a regular way. After the reconstruction we assign to each track a time measurement, which is calculated as an average of its hits measurements.

The initial distribution of hits measurements representing the complexity of defining event borders in a time-slice at interaction rate of 10^7 Hz is shown in Fig. 2 with blue color. The resulting distribution of reconstructed track measurements (black color), as well as the distribution of initial hit measurements (light blue color), one can see in Fig. 3. The reconstructed tracks clearly represent groups, which correspond to events, which they originate from.

5 KF Particle Finder – a package for reconstruction of short-lived particles

Today the most interesting physics is hidden in the properties of short-lived particles, which are not registered, but can be reconstructed only from their decay products. A fast and efficient KF Particle Finder package, based on the Kalman filter (hence KF) method, for reconstruction and selection of short-lived particles is developed to solve this task. A search of more than 50 decay channels has been currently implemented.

In the package all registered particle trajectories are divided into groups of secondary and primary tracks for further processing. Primary tracks are those, which are produced directly in the collision point. Tracks from decays of resonances (strange, multi-strange and charmed resonances, light vector mesons, charmonium) are also considered as primaries, since they are produced directly at the point of the primary collision. Secondary tracks are produced by the short-lived particles, which decay not in the point of the primary collision and can be clearly separated. These particles include strange particles (K_s^0 and Λ), multi-strange hyperons (Ξ and Ω) and charmed particles (D_0, $D^\pm$, $D_s^\pm$ and Λ_c). The package estimates the particle parameters, such as decay point, momentum, energy, mass, decay length and lifetime, together with their errors. The package has a rich functionality, including particle transport, calculation of a distance to a point or another particle, calculation of a deviation from a point or another particle, constraints on mass, decay length and production point. All particles produced in the collision are reconstructed at once, that makes the algorithm local with respect to the data and therefore extremely fast.

KF Particle Finder shows a high efficiency of particle reconstruction. For example, for the CBM experiment efficiencies of about 15% for Λ and 5% for Ξ^- with AuAu collisions at 35 AGeV are achieved together with high signal-to-background ratios (1.3 and 5.9 respectively).

6 A standalone FLES package for the CBM experiment

The First Level Event Selection (FLES) package of the CBM experiment is intended to re-construct on-line the full event topology including tracks of charged particles and short-lived particles. The FLES package consists of several modules: CA track finder, KF track fitter, KF Particle Finder and physics selection. In addition, a quality check module is implemented, that allows to monitor and control the reconstruction process at all stages. The FLES package is platform and operating system independent.

The FLES package is portable to different many-core CPU architectures. The package is vectorized using SIMD instructions and parallelized between CPU cores. All algorithms are optimized with respect to the memory usage and the speed.
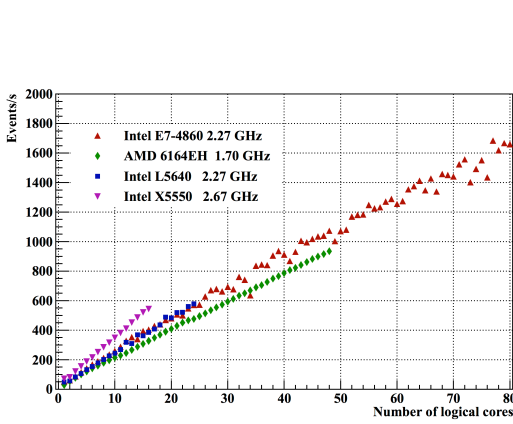

Figure 4: Scalability of the FLES package on many-core servers with 16, 24, 48 and 80 logical cores.
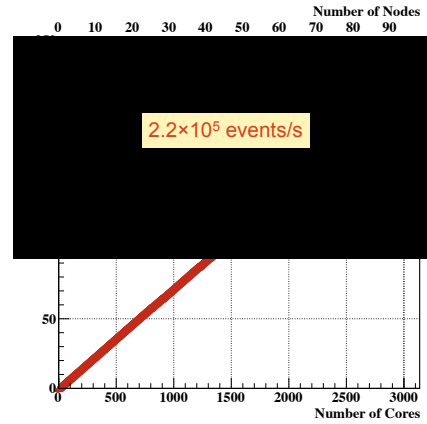
Figure 5: Scalability of the FLES package on 3 200 cores of the FAIR-Russia HPC cluster (ITEP, Moscow).

Four servers with Intel Xeon X5550, L5640 and E7-4860 processors and with AMD 6164EH processor have been used for the scalability tests. The AMD server has 4 processors with 12 physical cores each, in total 48 cores. All Intel processors have the hyper-threading tech-nology, therefore each physical core has two logical cores. The Intel X5550 and L5640 servers with two CPUs each have in total 16 and 24 logical cores respectively. The most powerful Intel server has 4 processors E7-4860 with 10 physical cores each, that gives 80 logical cores in total.

The FLES package has been parallelized with ITBB (Intel Threading Building Blocks) imple-menting the event-level parallelism by executing one thread per one logical core. Reconstruction of 1000 minimum bias Au-Au UrQMD (Ultrarelativistic Quantum Molecular Dynamics model) events at 25 AGeV has been processed per each thread. In order to minimize the effect of the operating system each thread is fixed to a certain core using the pthread functionality pro-vided by the C++ standard library. Fig. 4 shows a strong scalability for all many-core systems achieving the reconstruction speed of 1700 events per second on the 80-cores server.

The FLES package in the CBM experiment will be performed for the on-line selection and the off-line analysis on a dedicated many-core CPU/GPU farm. The farm is currently estimated to have a compute power equivalent to 60 000 modern CPU cores. Fig. 5 shows the scalability of the FLES package on a many-core computer farm with 3 200 cores of the FAIR-Russia HPC cluster (ITEP, Moscow).

7 Summary

The standalone FLES package has been developed for the CBM experiment. It contains track finding, track fitting, short-lived particles finding and physics selection. The Cellular Automaton and the Kalman filter algorithms are used for finding and fitting tracks, that allows to achieve a high track reconstruction efficiency. The event-based CA track finder was adapted for the time-based reconstruction, which is a requirement in the CBM experiment for the event building. The 4D CA track finder allows to resolve hits from different events overlapping in time into event-corresponding clusters of tracks. Reconstruction of about 50 decay channels of short-lived particles is currently implemented in the KF Particle Finder. The package shows a high reconstruction efficiency with an optimal signal to background ratio.

The FLES package is portable to different many-core CPU architectures. The package is vectorized and parallelized. All algorithms are optimized with respect to the memory usage and the processing speed. The FLES package shows a strong scalability on the many-core CPU systems and the processing and selection speed of 1700 events per second on a server with 80 Intel cores. On a computer cluster with 3 200 AMD cores it processes up to $2.2 \cdot 10^5$ events per second.

References

[1] The CBM Collaboration, *Compressed Baryonic Matter Experiment*, Tech. Stat. Rep., GSI, Darmstadt, 2005; 2006 update

[2] I. Kisel, I. Kulakov and M. Zyzak, IEEE Trans. Nucl. Sci. **60**, 3703–3708 (2013)

[3] I. Kisel, Nucl. Instr. and Meth. **A566** 85–88 (2006)

[4] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth, and W.F.J. Müller, Comp. Phys. Comm., **178**, 374–383 (2008)

Tree Contraction, Connected Components, Minimum Spanning Trees: a GPU Path to Vertex Fitting

Raul H. C. Lopes[1], Ivan D. Reid[1], Peter R. Hobson[1]

[1]Department of Electronic & Computer Engineering, Brunel University London, Kingston Lane, Uxbridge, UB8 3PH, United Kingdom

Standard parallel computing operations are considered in the context of algorithms for solving 3D graph problems which have applications, e.g., in vertex finding in HEP. Exploiting GPUs for tree-accumulation and graph algorithms is challenging: GPUs offer extreme computational power and high memory-access bandwidth, combined with a model of fine-grained parallelism perhaps not suiting the irregular distribution of linked representations of graph data structures. Achieving data-race free computations may demand serialization through atomic transactions, inevitably producing poor parallel performance. A Minimum Spanning Tree algorithm for GPUs is presented, its implementation discussed, and its efficiency evaluated on GPU and multicore architectures.

1   Introduction

We are concerned with the problem of finding parallel algorithms to compute a Minimum Spanning Tree for a weighted undirected graph. We introduce a parallel algorithm for computing minimum spanning trees over weighted undirected graphs.

An **undirected graph** G is a pair (V, E), where V is a finite nonempty set of vertices and E is a set of unordered pairs of vertices in V, called the set of edges of G. A **path** in G is a sequence of edges $e_0, e_1, \ldots, e_n$, with e_i and e_{i+1}, for $i \in \{0..n-1\}$, sharing exactly one vertex. A **cycle** is path $e_0, e_1, \ldots, e_n$, where e_0 and e_n share one vertex. A graph G is **connected** if there is a path between any pair of its vertices. A **tree** is connected graph containing no cycles. A **spanning tree** for a graph G is a tree containing all vertices in G. A **spanning forest** for a graph G that is not connected is a set of spanning trees, one for each connected component of G. A graph G is weighted when a function w assigns weights to each edge of G. A graph's weight is the sum of weights of its edges. A Minimum Spanning Tree (MST) of a graph G is a tree of minimal weight among all spanning trees of G. The concept extends to that of **Minimum Spanning Forest** (MSF) of G as a spanning forest of minimum weight.

Throughout this paper we will use n to denote number of vertices in a graph G, and m to denote its number of edges. An MST in $\mathcal{R}^2$ can be computed in $O(n \lg n)$ using Delaunay triangulations. However, the work demanded to compute an MST in $\mathcal{R}^d$ is limited from below by $\Omega(m)$ when $d > 2$, see [1]. Even the representation in memory of the edges of a 2^{20} vertices graph could be a challenge.

Minimum spanning trees have applications, for example, in computer networks, water supply networks, and electrical grids. The computation of MSTs is considered a fundamental challenge in Combinatorial Optimization [2] and is of interest for the HEP community.

In the next sections, we discuss first sequential algorithms for computing connected and spanning trees, and the difficulties involved in parallelizing them. Then we present an algorithm for computing minimum spanning trees on GPU architectures and discuss its performance on both GPU and multi-core architectures.

2 Sequential Algorithms

Minimum spanning tree computations make use of the following property to choose edges from the underlying graph to add to its MST. See [3] for a proof of its soundness.

Cut property: The lightest edge connecting any nonempty subset X of $V(G)$ to $V(G) - X$ belongs to $MSF(G)$.

The Borŭvka algorithm is historically the first and possibly most general MSF algorithm. It uses the cut property to maintain the following invariant: each tree in the forest is an MST for the vertices in it, and each vertex in the initial graph belongs to exactly one tree. It starts with a set of forests, each containing exactly one vertex from the given graph G. Borŭvka's is essentially a non-deterministic algorithm in which, at any time, all lightest edges that can expand any of its trees can be added. It's important to notice, however, that each edge addition joins two trees and this must be reflected in the data structures used in its implementation. Also, its non-deterministic nature appeals to parallel implementations, but introduces possible data-race conditions.

Most theoretically efficient MST algorithms use Borŭvka functions, as for example [4], and the deterministic algorithm given by [1] and [5]. These algorithms, however, seem to lack efficient practical implementations or parallel versions. The most successful implementations of sequential MST algorithms, the **Prim-Dijkstra** and **Kruskal** algorithms evaluated in [6], can be seen as specializations of the Borŭvka algorithm and they also maintain an invariant, see [7], where at any moment in the computation the edges already selected by the algorithm form an MST of the subgraph in question and any new edge added satisfies the *cut property*. In particular the Kruskal algorithm [3] is a sequential version of the Borŭvka algorithm where edges are processed in increasing order of weights.

In the next section, we discuss the challenges found in parallelizing the Borŭvka algorithm and the fundamental requirements for parallelization of a minimum spanning tree computation.

3 Parallel Algorithms

The nondeterministic nature of Borŭvka's algorithm clearly invites the introduction of asynchronous parallelism in its implementation. We would have an algorithm that performs a sequence of parallel steps, where each parallel step joins all possible pairs of trees, using the lightest edges connecting them. However, care must be taken in that: more than two trees may be joined in the same parallel step; given two trees t_0 and t_1 joined in a parallel step, all edges connecting vertices in t_0 to vertices in t_1 must be discarded.

Those problems result from the fact that many trees may be expanded simultaneously. The Prim-Dijkstra algorithm expands exactly one tree and thus avoids complications of performing simultaneous unions of vertices and edges by being strictly sequential. The **Kruskal** algorithm

excludes the possibility of joining more than two trees simultaneously by processing the edges in increasing order of weight. It still keeps track of more than one tree being expanded at any time, but each expansion and join is serialized. Its first obvious disadvantage is in processing the edges in increasing weight order, which demands ordering all edges before the real construction of the tree starts, which can be very expensive or even prohibitive when processing, for example, Euclidean graphs where the number of edges can approach n^2. A graph with 2^{20} vertices might demand memory to maintain close to 2^{40} edges, maybe two to four terabytes just to store edge weights.

The Kruskal algorithm, or any other derived from the Borůvka algorithm, must still take care of the joining of trees and the union of sets of respective vertices and edges. In practice this is performed using efficient implementations of the disjoint-set data structure, see [8]. A disjoint-set data structure offers fast union of two sets by labeling the elements of two sets with a common identifier that defines that new common set that they belong to. It offers fast set membership tests by performing a fast search from the vertex to the label of the structure representing the set it belongs to. In a sequential implementation this is achieving by making vertices the leaves of trees where each node points to its parent, with the root of the tree pointing to itself and being taken as the label of the tree. Set union is trivially performed by making the root of one tree to point to the root of another tree. However, this is an operation that must be synchronized. Synchronization may also be needed if the search for a tree's root is performed concurrently with a join operation.

Nondeterminism has been forcefully defended by authors like Dijkstra [9] and Lamport [10] as a powerful tool in the design of computer algorithms and concurrent systems. More recently, Steele [11] has made a strong point for the integration of asynchronous computations in synchronous parallel algorithms, given that asynchronous programs tend to be more flexible and efficient when processing conditionals. In addition many authors have pointed out asynchronous and non-deterministic parallelism as the root of success of many programs based on the MapReduce model [12].

An ideal implementation of Borůvka's algorithm would start as many asynchronous threads as possible, each expanding a different minimum spanning tree. Synchronization, however, would be needed, for example, if a tree t_0 must be joined to a tree t_1 while, simultaneously, t_1 is being joined to t_2 and t_2 is being joined with t_0. That sort of synchronism is a fundamental requirement of any parallel implementation of Borůvka's algorithm.

The objective of this paper is to present a GPU algorithm. Its design must take into account that GPU architectures perform at their peak when running in SIMD (Single-Instruction-Multiple-Data) fashion. Performance is lost when atomic transactions, or conditions that send threads into different paths, are present.

Correctness in parallel and concurrent computation can only be achieved by ensuring non-interference between threads. Blelloch et ali [13], see also [14], have argued that being data-race free is the minimum requirement in order to achieve correctness in the presence of concurrency. However, being data-race free at the cost of atomic operations may be too costly for SIMD computing. Internal determinism is the solution they introduce to obtain a fast multi-core algorithms by transforming the code to ensure that internal steps are always deterministic. Internal determinism, however, should not be introduced through the introduction of expensive atomic transactions or nesting of conditional structures and code. Possibly the fundamental mandate for SIMD (and GPU) architectures must be functional determinism, where the absence of side effects between threads is a guarantee of non-interference and absence of the need for atomic constructions as semaphores or even transactional memory. This should be achieved by

program transformations to perform parallel computation through the composition of efficient GPU parallel constructions.

In the next section we present a parallelization of the Kruskal algorithm and discuss its performance on both GPU and multi-core architectures.

4 A Parallelization of the Kruskal Algorithm and its Performance Evaluation

We introduce a parallel version of the Kruskal algorithm. The algorithm is composed of two phases: parallel sorting of the edges of the given graph, followed by the construction of the minimum spanning tree. The sorting phase is performed by a standard parallel sorting function and is not described here. We describe the construction phase by introducing first its objects and the morphisms that transform them. The objects used by the algorithm follow.

- A set of weighted edges. Represented by a vector of triples (v, v, w), where u and v are the edge vertices and w is its associated weight, it is ordered by a parallel sorting function in the first phase of the algorithm.

- An ordered sequence of prioritized edges. Represented by a vector of triples (i, u, v), u and v being vertices and i the order of the edge (u, v) in the sorted set, it is generated by a parallel zip of the index i and an edge from the sequence of ordered edges fused with a parallel map to form triple.

- A set of vertex reservations. Represented by a vector that at any moment assigns a vertex to an edge, where $reserved[u] = i$, states the fact that edge i owns the vertex u.

- A set of delayed edges. A vector of booleans indexed by edges, which determines if, in a given parallel step, the addition of an edge to the tree will be suspended and delayed to be performed in a future step.

- A disjoint set of trees. Represented by a vector SV of vertices to trees, it assigns to a vertex v the partial minimum spanning tree it belongs, in the sense that if $SV[u]$ equals to v then v if the parent of u in disjoint-set structure, and they belong to same tree. We assume a function $root(u)$ which returns the root of u in SV.

The edges are processed in blocks of p edges. An edge (u, v) is considered for processing and consequent addition to a tree only if its vertices do not belong to the same tree, i.e., $root(u)$ and $root(v)$ are different. Each block is processed by the sequence of parallel transformations described below.

- Parallel reservation. $reserve(i, u, v)$. It is implemented by a parallel loop, where each parallel step takes a triple (i, u, v) from the sequence of prioritized edges described above and tries to mark u and v as reserved by i. It is important to notice that a data-race condition arises when two parallel steps try to acquire the same vertex for different edges.

- Parallel delay. A parallel loop where each step takes a prioritized edge (i, u, v) and the reserved set and marks the edge i as delayed if neither u nor v is reserved for i.

- Parallel commit. Another parallel loop where each step commits a prioritized edge by linking v to u when v is reserved by i, or linking u to v, when u is reserved by i. Linking u to v consists in assigning v to $SV[u]$.

- Parallel disjoint set compression. Using a pointer jumping technique as described in the rake and compress algorithm of Shiloach and Vishkin [15] applied to the disjoint set structure implemented by SV, it compresses the structure to make each leaf to point directly to its respective root.

Table 1 shows times for tests with an implementation of the algorithm on a machine with an *Intel Xeon E5620* and an *Nvidia Tesla C2070*. *Ubuntu 14.04* was the operating system and *nvcc*, the compiler. Each line shows the time to solve a problem using 1, 4, 8 threads, and the time on the GPU. The first line shows the problem of sorting 2^{25} single-precision floating-point numbers from a uniform distribution. It is important to notice

Problem	1 HT	4 HT	8 HT	GPU
sort	1.34	1.11	0.91	0.078
Uniform	6.31	4.05	4.12	0.74
PLaw	7.90	5.91	6.05	1.24

Table 1: Time (seconds) comparing sorting floats and MST construction

that the *Intel Xeon E5620* has four cores, that can be used to run 2 hyperthreads (HT) each. The GPU sort produces close to 11.7 times acceleration compared to sorting on eight threads. In both cases, the tests used the standard sort function provided by the NVidia *thrust library*.

The second line shows the times to build a minimum spanning tree with 2^{20} uniformly distributed vertices, and 2^{23} edges on an Euclidean three dimensional space. The GPU gives around 5.56 times acceleration compared with running on 8 threads. The last line shows tests for random graphs with 2^{20} nodes and 2^{24} edges with a power law distribution, based on [16].

5   Conclusions and Further Work

The technical literature generally presents the construction of minimum spanning forests as an outstanding combinatorial optimization problem, that is challenging enough to be used in several benchmarks as shown in the recent [13]. Even if many theoretical parallel algorithms can be found, only a few seem to show effective implementations. Badder et ali [17] have studied parallelizations both based on the Borůvka algorithm and on the Prim-Dijkstra algorithm. The paper [17] shows some gain from the range of two to four processors, but the absence of numbers for one core tests doesn't allow for a clear evaluation of the acceleration obtained. Chong [18] presents a multi-core asynchronous parallel algorithm relying on atomic transactions, that would hardly be realizable on a GPU architecture. The algorithm in [19] seems to be an exception in targeting a GPU architecture. It is based on the Prim-Dijkstra algorithm and achieves less than 3 times acceleration when compared with a single core algorithm. Belloch [13] presents an algorithm for multi-core architectures that introduces the idea of vertex reservation used in this paper. It seems, however, to depend on atomic transactions to access a disjoint-set data structure. The algorithm presented in this paper borrows from Blelloch and from Steele's ideas on combining asynchronous and synchronous parallelism.

A clear limitation of any algorithm derived from Kruskal is the need to use an explicit representation of the edges. We have previously presented a parallel algorithm for k-d-tree construction [20] that could be used to avoid the huge allocation of space when a quadratic number of edges is present in an Euclidean graph.

6 Acknowledgments

References

[1] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackerman type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[2] William J. Cook, William H. Cunningham, and William R. Pulleybank. *Combinatorial Optimization*. Willey-Blackwell, 1997.

[3] Thomas Cormen, C. Leiserson, Ron Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[4] D. R. Karger, Phillip N. Klein, and Robert E. Tarjan. A randomized liner-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[5] Seth Petit and Vijaya Ramachandran. An optimal minimum spanning algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[6] Bernard M. E. Moret and Henry D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs 15*, pages 99–117, 1994.

[7] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Mathematics*. Springer, 2010.

[8] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[9] Edsger E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002.

[11] Guy L. Steele. Making asynchronous parallelism safe for the world. *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231, 1990.

[12] Derek G. Murray and Steven Hand. Non-deterministic-parallelism considered useful. *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 19–19, 2011.

[13] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[14] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[15] Y. Shiloach and Uzi Vishkin. An $o(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[16] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446, 2004.

[17] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66(11):1366–1378, November 2006.

[18] Ka Wong Chong, Yijie Han, and Tak Wah Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *Journal of the ACM*, 48(2):297–323, March 2001.

[19] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[20] Raul H C Lopes, Ivan D Reid, and Peter R Hobson. A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures. *Journal of Physics: Conference Series*, 513(052011), 2014.

Manycore feasibility studies at the LHCb trigger

D.H. Campora Perez[1]

[1] CERN, Geneve, Switzerland

The LHCb trigger is a real time system with high computation requirements, where incoming data from the LHCb detector is analyzed and selected by applying a chain of algorithms. The infrastructure that sustains the current trigger consists of Intel Xeon based servers, and is designed for sequential execution. We have extended the current software infrastructure to include support for offloaded execution on manycore platforms like graphics cards or the Intel Xeon/Phi. In this paper, we present the latest developments of our offloading mechanism, and we also show feasability studies over subdetector specific problems which may benefit from a manycore approach.

Contribution not received.

Track pattern-recognition on GPGPUs in the LHCb experiment

Stefano Gallorini[1,2*]
[1]University and INFN Padova, Via Marzolo 8, 35131, Padova, Italy
[2]CERN, 1211 Geneve 23, Switzerland

The LHCb experiment is entering in its upgrading phase, with its detector and read-out system re-designed to cope with the increased LHC energy after the long shutdown of 2018. In this upgrade, a trigger-less data acquisition is being developed to read-out the full detector at the bunch-crossing rate of 40 MHz. In particular, the High Level Trigger (HLT) system has to be heavily revised. Since the small LHCb event size (about 100 kB), many-core architectures such as General Purpose Graphics Processing Units (GPGPUs) and multi-core CPUs can be used to process many events in parallel for real-time selection, and may offer a solution for reducing the cost of the HLT farm. Track reconstruction and vertex finding are the more time-consuming applications running in HLT and therefore are the first to be ported on many-core. In this talk we present our implementation of the existing tracking algorithms on GPGPU, discussing in detail the case of the VErtex LOcator detector (VELO), and we show the achieved performances. We discuss also other tracking algorithms that can be used in view of the LHCb upgrade.

1 Introduction

One of the most stringent restrictions upon reconstruction algorithms for the software trigger is their throughput. Data rates require fast execution times, which eventually limit the type of algorithms to be used. One may not count anymore on the fast development of processors to expect a given algorithm to become faster just because the CPU clock frequency increases: clock frequencies are frozen for more than ten years now. The trend has moved towards having several cores, ranging from two to, likely, many-cores in the near future. For this reason, we might expect improvements in execution times coming from a clever use of the multicore structure and parallelization. Therefore, sensibility advises to build up a program to study and exploit the possibilities of parallelization of the algorithms involved in the reconstruction and also in the trigger. Among the candidate architectures to support these algorithms we find General Purpose Graphics Processing Units (GPGPUs), specialized for compute-intensive, highly parallel computation. GPGPUs may offer a solution for reducing the cost of the HLT farm for the LHCb upgrade and R&D studies have started to evaluate the possible role of this architecture in the new trigger system.

In the following section we discuss our preliminary attempt to port the existing tracking algorithm of the VErtex LOcator (VELO) detector on GPGPU, and we show the achieved

---

*On behalf of the GPU@LHCbTrigger working group

performances.

1.1 FastVelo

The VELO [1] is a silicon strip detector that provides precise tracking very close to the interaction point. It is used to locate the position of any primary vertex within LHCb, as well as secondary vertices due to decay of any long lived particles produced in the collisions. The VELO detector is formed by 21 stations, each consisting of two halves of silicon-strip sensors, which measure R and ϕ coordinates. Each half is made of two type of sensors: R sensors with strips at constant radius covering 45° in a so called sector or zone (four zones/sensor), and ϕ sensors with nearly radial strips.

"FastVelo" [2] is the algorithm developed for tracking of the current VELO and was written to run online in the High Level Trigger (HLT) tracking sequence. For this reason, the code was optimized to be extremely fast and efficient in order to cope with the high rate and hit occupancy present during the 2011-2012 data collection. FastVelo is highly sequential, with several conditions and checks introduced throughout the code to speed up execution and reduce clone and ghost rates.

The algorithm can be divided into two well-defined parts. In the first part (RZ tracking), all tracks in the RZ plane are found by looking at four neighbouring R-hits along the Z axis ("quadruplet"). The quadruplets are searched for starting from the last four sensors, where tracks are most separated. Then the quadruplets are extended towards the lower Z region as much as possible, allowing for some inefficiency. In the second part of the algorithm (space tracking), 3D tracks are built by adding the information of the ϕ hits to the RZ track. A first processing step is to define the first and last ϕ sensor to use, then the algorithm starts from the first station with hits searching for a triplet of nearby ϕ hits. The triplet is then added to the RZ track to form a 3D tracklet, so that the track parameters can be estimated. These 3D segments are then extrapolated towards the interaction region by adding hits in the next stations compatible with the tracklet. The final 3D track is re-fitted using the information of R and ϕ hits, while hits with the worst χ^2 are removed from the track. Hits already used in a track are marked as used and not further considered for following iterations ("hit tagging"); this is done to reduce the number of clones produced by the algorithm, avoiding encountering the same track several times. The full FastVelo tracking includes additional algorithms for searching R-hit triplets and unused ϕ hits; these algorithms ran only at HLT2 during 2012. However, the GPU implementation of FastVelo reported in this work refers only to the VELO tracking running on HLT1 during the RUN1.

2 GPU implementation

The strategy used for porting FastVelo to GPU architectures takes advantage of the small size of the LHCb events ($\approx$ 60kB per event, $\approx$ 100 kB after the upgrade) implementing two level of parallelization: "of the algorithm" and "on the events". With many events running concurrently, it can be possible, in principle, to gain more in terms of time performances with respect to the only parallelization of the algorithm. The GPU algorithm was adapted to run on GPU using the NVIDIA Compute Unified Device Architecture (CUDA) framework [3].

One of the main problems encountered in the parallelization of FastVelo concerns hit tagging, which explicitly spoils data independence between different concurrent tasks (or "threads"

in CUDA language). In this respect, any implementation of a parallel version of a tracking algorithm relying on hit tagging implies a departure from the sequential code, so that the removal of tagging on used hits is almost unavoidable. The main drawback of this choice is that the number of combinations of hits to be processed diverges and additional "clone killing" algorithms (intrinsically sequential and not easy to parallelize) have to be introduced to mitigate the increase of ghost and clone rates. Another issue encountered in the development of the parallel version of FastVelo is due to the R-ϕ geometry of the current VELO that impose a splitting of the tracking algorithm in two sequential steps (RZ tracking plus 3D tracking).

The approach described in this note follows closely the sequential algorithm; therefore, also the tracking algorithm implemented on GPU is based on a local search ("local" method): first seeds are formed by looking only to a restricted set of sensors (quadruplets), then the remaining hits on the other sensors are added to build the full tracks.
The outline of the implementation chosen to parallelize FastVelo can be summarized as follows:

- The algorithm searches for long tracks first, using only the last five sensors downstream the VELO (upstream for backward tracks). Four threads (one for each sensor zone) find all possible quadruplets in these sensors. Then, each quadruplet is extended independently as much as possible by adding R-hits of other sensors. The R-hits of each RZ track are marked as used; potential race-conditions are not an issue in this case, because the aim is to flag an hit as used for the next algorithms.

- Then the remaining sensors are processed: each thread works on a set of five contiguous R-sensors and find all quadruplets on a zone of these sensors. A check is done on the hits in order to avoid hits already used for the long tracks. In a sense, the algorithm gives more priority to the long tracks with respect to the short ones.

    At this stage the number of quadruplets belonging to the same tracks is huge and a first "clone killer" algorithm is needed to protect against finding the same track several times. All pairs of quadruplets are checked in parallel: each thread of the clone killer algorithm takes a quadruplet and computes the number of hits in common with the others; if two quadruplets have more than two hits in common, the one with worst χ^2 is discarded (here, the χ^2 is defined as the sum of residual of the position of the R-hits of the RZ track with respect to the predicted position given by fitted track).

- Next, each quadruplet is extended independently as much as possible by adding R-hits of other sensors on both directions. After this step, all possible RZ tracks are built. The number of clones generated by the algorithm is still huge, and another clone killer algorithm similar to the one implemented in the previous step is used to reduce the fraction of clone tracks to a tolerable value.

    It should be noted that this procedure of cleaning clones follows the same lines of the one implemented in the original FastVelo algorithm, the only difference being that in FastVelo the clone killer algorithm is applied only to the full 3D tracks (almost at the end of the tracking), while in the parallel implementation, without hit tagging, we are forced to introduce it well before in the tracking sequence in order to reduce the number of tracks in input to the next steps.

- Next step is to perform full 3D tracking by adding ϕ hits information. Each RZ track is processed concurrently by assigning a space-tracking algorithm to each thread.

This part is almost a re-writing in CUDA language of the original space-tracking algorithm, with the notable exception of the removal of tag on the used ϕ hits. When all 3D tracks have been found, a final cleanup is done on the tracks to kill the remaining clones and ghosts; the clone killer algorithm is the same of the one used in the previous steps, with the exception that now the χ^2 is based on the information of both R and ϕ hits of the track.

3   Preliminary results

In this section the timing performances and tracking efficiencies obtained with FastVelo on GPU will be compared to the sequential algorithm running on 1 CPU core. Other studies will be presented using a multi-core CPU. These preliminary results refer only to the tracking time without including data transfer time from CPU to GPU, and vice-versa. The reason for this choice was dictated in part by the approach of exploiting the parallelization over the events, where each thread is assigned to an event. This strategy cannot be easily implemented using the standard software framework, originally developed to process sequentially one event at time. A simple offloading mechanism has been developed which is able to load data on GPU memory decoding the information of the VELO hits from raw data. After the tracking on GPU, the tracks are sent back to the original sequential framework.

The measurements of the tracking time for GPU have been taken using the standard CUDA timer ("CUDA event"), while the timing for CPU has been taken from the detailed FastVelo profile given by the LHCb reconstruction program. Tracking efficiencies for both GPU and CPU have been obtained from the standard tools provided by Brunel. The GPU model used for these tests is an NVidia GTX Titan (14 Streaming multiprocessors, each equipped with 192 single-precision CUDA cores), while the CPU is an Intel(R) Core(TM) i7-3770 3.40 GHz. A MonteCarlo (MC) sample of $B_s \to \phi\phi$ events generated with 2012 conditions (with pile-up of $\nu = 2.5$[1]) has been used to evaluate the tracking and timing performances. Timing performances have been compared also with real data using a NoBias sample collected during 2012 ($\mu = 1.6$). In the $B_s \to \phi\phi$ MC sample, the average number of hits per sensor is ≈ 17, while the average number of reconstructed VELO tracks per event is ≈ 80.

The comparison of tracking efficiencies between the GPU implementation and the original FastVelo algorithm for different categories of tracks is shown in Tab. 1 [2]. The efficiencies obtained by FastVelo on GPU are quite in agreement with the sequential FastVelo; in particular, clones and ghosts are at the same level of the original code. Fig. 1 shows the tracking efficiency as a function of the true track momentum P_{true} as obtained by the two algorithms; the overall agreement is good, showing that the GPU implementation does not introduce any distortion on the resolution of the track parameters.

The speed-up obtained by the GPU algorithm with respect to FastVelo running on a single CPU core as a function of the number of processed events is shown in Figs. 2. The maximum speed-up obtained by the GPU algorithm with respect to the sequential FastVelo is $\approx 3\times$ for the 2012 datasets. The speedup as a function of the number of events can be explained by the fact that the GPU computing resources are more efficiently used as the number of events

---

[1]ν is the number of total elastic and inelastic proton-proton interactions per bunch crossing, while μ represents the number of visible interactions per bunch-crossing. LHCb labels simulated event samples according to ν.

[2]Only the VELO tracking running on HLT1 has been implemented on GPU, so that the quoted efficiencies and timings refer to FastVelo in the HLT1 configuration.

Track category	FastVelo on GPU		FastVelo	
	Efficiency	Clones	Efficiency	Clones
VELO, all long	86.6%	0.2%	88.8%	0.5%
VELO, long, p > 5 GeV	89.5%	0.1%	91.5%	0.4%
VELO, all long B daughters	87.2%	0.1%	89.4%	0.7%
VELO, long B daughters, p > 5 GeV	89.3%	0.1%	91.8%	0.6%
VELO, ghosts	7.8%		7.3%	

Table 1: Tracking efficiencies obtained with FastVelo on GPU, compared with the results obtained by original FastVelo code (only the VELO tracking running on HLT1). The efficiencies are computed using 1000 $B_s \to \phi\phi$ MC events, generated with 2012 conditions.

increases (there are more threads running at the same time).

The comparison of the timing performance has been done using also a multi-core CPU (Intel Xeon E5-2600, 12 cores with hyper-threading and 32 GB of memory). A instance (job) of FastVelo was sent to each core at the same time, with each job processing the same number of events (for this study the number of events/job was set to 1000): the throughput of a single core goes down the more instances are running in parallel (this is due to memory IO pressure, the CPU scaling down its frequency when a lot of cores are running to stay within its power budget). In the case of $B_s \to \phi\phi$ MC events, the rate of processed events on the multi-core CPU, using all the 24 logical cores, is ≈ 5000 events/sec, while on GPU the rate decrease down to ≈ 2600 events/sec. However, the number of processed events per second is not a real measure for performances, because it has no meaning when comparing different computing platforms or even computing architectures. A better estimator for these performance studies is the rate of events normalized to the cost of the hardware (events/sec/cost): the GPU gaming-card cost a small fraction of the server used in the HLT farm, so also a moderate speed-up (e.g. 2×) compared to a Xeon CPU can bring a real saving to the experiment (provided the GPU is reasonably well used).

Next steps of this work will include a development of the full FastVelo tracking on GPU (the part running on HLT2) and the remaining tracking algorithms, such as the Forward tracking [4]. In addition, we plan to test FastVelo on GPU in parasitic mode during the RUNII in 2015 in order to assess the impact and the feasibility of the many-core solution on the HLT infrastructure.

References

[1] P R Barbosa-Marihno et al. *LHCb VELO (VErtex LOcator): Technical Design Report*, CERN-LHCC-2001-011 (2001).

[2] O. Callot, *FastVelo, a fast and efficient pattern recognition package for the Velo*, LHCb-PUB-2011-001 (2011)

[3] NVIDIA Corp. *CUDA C programming guide* PG-02829-001 v6.0, February 2014

[4] O. Callot, S. Hansmann-Menzemer, *The Forward Tracking: Algorithm and Performance Studies* LHCb-015-2007 (2007)
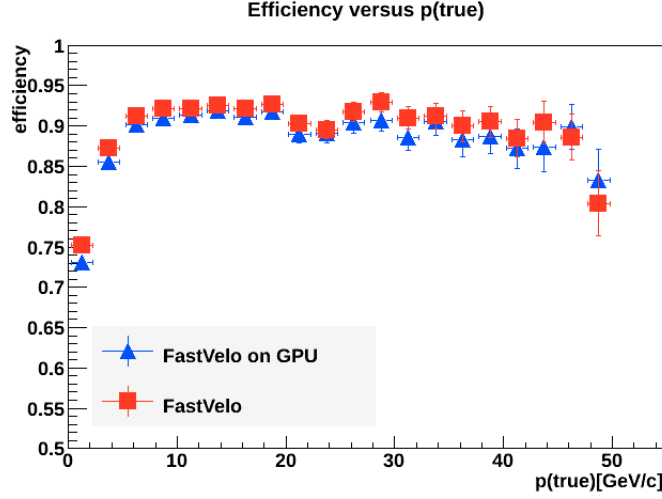
Figure 1: Tracking performance comparisons between the sequential FastVelo and FastVelo on GPU. Tracking efficiency as a function of the true track momentum P_{true}.
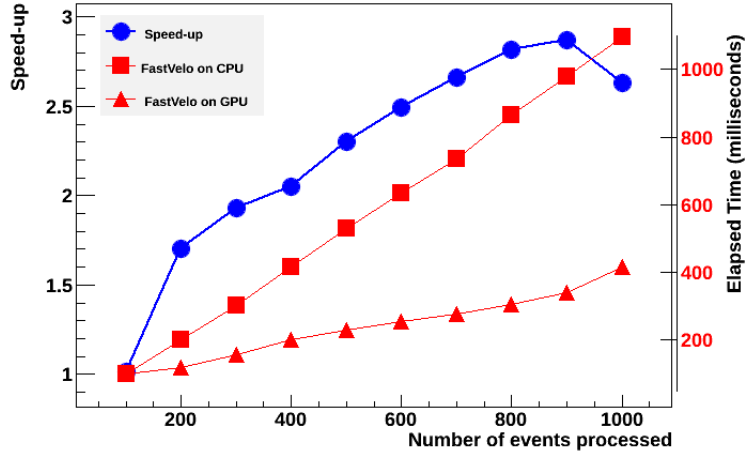

Figure 2: Tracking execution time and speedup versus number of events using a 2012 MC sample of $B_s \to \phi\phi$ decays ($\nu = 2.5$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz).

A GPU-based track reconstruction in the core of high p_T jets in CMS

F. Pantaleo[1]

[1] CERN, Geneve, Switzerland

The Large Hadron Collider is presently undergoing work to increase the centre-of-mass energy to 13 TeV and to reach much higher beam luminosity. It is scheduled to return to operation in early 2015. With the increasing amount of data delivered by the LHC, the experiments are facing enormous challenges to adapt their computing resources, also in terms of CPU usage. This trend will continue with the planned future upgrade to the High-Luminosity LHC. Of particular interest is the full reconstruction of the decay products of 3rd generation-quarks in high pT jets that have a crucial role in searches of new physics at the energy frontier. At high pT, tracks from B-decays become more collimated, hence reducing the track-finding efficiency of generic tracking algorithms in the core of the jet. The problem of reconstructing high pT tracks in the core of the jet, once a narrow eta-phi region around the jet is defined, was found to be especially beneficial for the application of GPU programming techniques due to the combinatorial complexity of the algorithm. Our approach to the problem will be described, and particular focus will be given to the partitioning of the problem to map the GPU architecture and improve load balancing. To conclude, measurements are described, which show the execution speedups achieved via multi-threaded and CUDA code in the context of the object-oriented C++ software framework (CMSSW) used to process data acquired by the CMS detector at the LHC.

Contribution not received.

An evaluation of the potential of GPUs to accelerate tracking algorithms for the ATLAS trigger.

J. Howard[1]

[1] University of Oxford, UK

The potential of GPUs has been evaluated as a possible way to accelerate trigger algorithms for the ATLAS experiment located at the Large Hadron Collider (LHC). During LHC Run-1 ATLAS employed a three-level trigger system to progressively reduce the LHC collision rate of 20 MHz to a storage rate of about 600 Hz for offline processing. Reconstruction of charged particles trajectories through the Inner Detector (ID) was performed at the second (L2) and third (EF) trigger levels. The ID contains pixel, silicon strip (SCT) and straw-tube technologies. Prior to tracking, data-preparation algorithms processed the ID raw data producing measurements of the track position at each detector layer. The data-preparation and tracking consumed almost three-quarters of the total L2 CPU resources during 2012 data-taking. Detailed performance studies of a CUDATM implementation of the L2 pixel and SCT data-preparation and tracking algorithms running on a Nvidia Tesla C2050 GPU have shown a speed-up by a factor of 12 for the tracking code and by up to a factor of 26 for the data preparation code compared to the equivalent C++ code running on a CPU. A client-server technology has been used to interface the CUDATM code to the CPU-based software, allowing a sharing of the GPU resource between several CPU tasks. A re-implementation of the pixel data-preparation code in openCL has also been performed, offering the advantage of portability between various GPU and multi-core CPU architectures.

Contribution not received.

Use of hardware accelerators for ATLAS computing

Matteo Bauce [2], *Rene Boeing* [3] [4], *Maik Dankel* [3] [4], *Jacob Howard* [1], *Sami Kama* [5] *for the ATLAS Collaboration*

[1]Department of Physics Oxford University, Oxford, United Kingdom
[2]Dipartimento di Fisica Sapienza Universit di Roma and INFN Sezione di Roma, Roma, Italy
[3]Fachbereich C Physik, Bergische Universitaet Wuppertal, Wuppertal, Germany
[4]Fachhochschule Muenster - University of Applied Sciences, Muenster, Germany
[5]Southern Methodist University, Dallas TX, US

Modern HEP experiments produce tremendous amounts of data. This data is processed by in-house built software frameworks which have lifetimes longer than the detector itself. Such frameworks were traditionally based on serial code and relied on advances in CPU technologies, mainly clock frequency, to cope with increasing data volumes. With the advent of many-core architectures and GPGPUs this paradigm has to shift to parallel processing and has to include the use of co-processors. However, since the design of most of the existing frameworks is based on the assumption of frequency scaling and predate co-processors, parallelisation and integration of co-processors are not an easy task. The ATLAS experiment is an example of such a big experiment with a big software framework called Athena. In this proceedings we will present studies on parallelisation and co-processor (GPGPU) use in data preparation and tracking for trigger and offline reconstruction as well as their integration into the multiple process based Athena framework using the Accelerator Process Extension APE.

1 Introduction

The Large Hadron Collider (LHC) is a 27km long circular particle accelerator near Geneva, situated about 100m below the Swiss and French border [2]. It is designed to collide proton bunches with a center-of-mass energy of 14TeV every 25ns. It is equipped with 4 detectors namely ALICE and LHCb, two relatively small special purpose detectors, and CMS and ATLAS two larger general purpose detectors. The AT-LAS detector is the biggest of them and composed of concentric cylindrical detectors with end-caps [3]. The inner Detec-
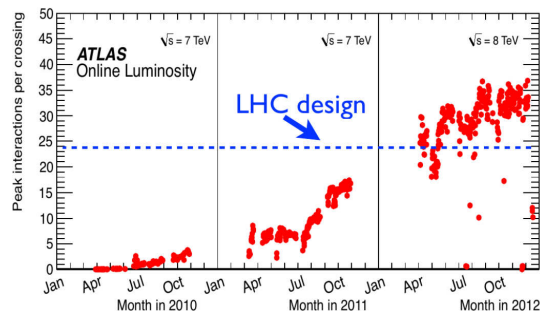

Figure 1: Peak interactions per bunch-crossing per time. LHC exceeded design value in 2012. [1]

tor(ID) is the closest to the beam and it is composed of Pixel, SCT and TRT trackers. Calorimeters are located around the ID. Calorimeters have electromagnetic and hadronic components comprised of Liquid Argon and Tile calorimeters. Muon detectors form the outermost shells of the detector. Toroidal and solenoid magnets provide the magnetic field for momentum and charge determination.

During the Run-1 period, LHC operated below its design energy, at 7TeV and 8TeV with a bunch spacing of 50ns. However the single bunch luminosity was increased which led to a higher number of collisions at each bunch crossing(pile-up) than design expectations as shown in Figure 1. Towards the end of Run-1, average pile-up at ATLAS exceeded 40 interactions per crossing, creating more than 1200 tracks. Since the end of 2012, LHC is being upgraded and will operate at full design energy and bunch crossing period in the Run-2 phase, starting in 2015. Pile-up is expected to increase up to 80 interactions per bunch crossing leading to many more tracks. Predictions for Run-3 with a peak luminosity of $10^{35}cm^{-2}s^{-1}$ and a pile-up of up to 140 interactions per bunch crossing are even higher. Since track finding is a combinatorial problem, total processing time will also increase exponentially. The estimated pile-up dependency of average reconstruction time is given in Figure 2.
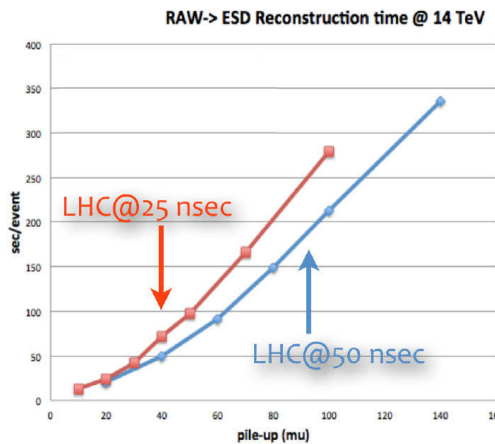

Figure 2: Average event reconstruction time versus pile-up at different bunch crossing rates. [4]

Up until a few years ago, the computing capacity of CPUs increased mostly due to increases in clock frequency. This increase usually compensated the increase in data rates with no changes in the code. However due to physical constraints, clock frequency of the CPUs has plateaued and the increase in computing capacity is provided by adding more cores and vector units to processors or in the form of co-processors. Unlike CPUs, co-processors tend to work effectively on certain types of highly parallel problems. However, they have a higher computing capacity per watt at a lower cost than CPUs. These properties make them attractive solutions for highly parallelizable workloads such as tracking.

2 ATLAS Software Framework

The ATLAS software framework, Athena, is composed of more than 4 million lines of C++ and about 1 million lines of python code written by hundreds of developers [5, 6]. The code is spread over more than 4000 shared libraries in about 2000 packages. Its design and implementation predates multi-core CPUs and thus was designed to run serially in a single process. Multi-core CPUs are exploited by running multiple independent processes at the expense of increased memory usage. Because of the design and complexity of the existing code, porting it to co-processors is not feasible if not impossible. However it is still possible to utilize co-processors

such as GPUs for some self contained parts of the software, yet it is still a challenge due to the multi-process nature of the framework.

2.1 Accelerator Process Extension (APE)

The Accelerator Process Extension (APE) is a framework designed for managing and sharing co-processors between different processes. It is based on a Server-Client model and its working diagram is given in Figure 3. Algorithms that are running in Athena have to prepare and serialize the input data and issue a processing request to the Offload Service. The Offload Service manages the communication between the APE Server and algorithms. Processing requests are forwarded to the APE Server via Yampl, an inter-process communication (IPC) abstraction layer. It abstracts various IPC technologies, such as shared memory, pipes and ZeroMQ [7] based layer for network communication. This enables running the APE server on a local host or a dedicated server host. The APE server does bookkeeping of requests and relays them to appropriate module which is a dynamically loaded plug-in that manages resources and contain algorithm implementations for a given hardware like GPUs or Intel MICs. They execute the requested operations on input data and return the results to the APE server. The APE server passes the results back to the Offload Service which in turn returns the results to algorithms that made the request.
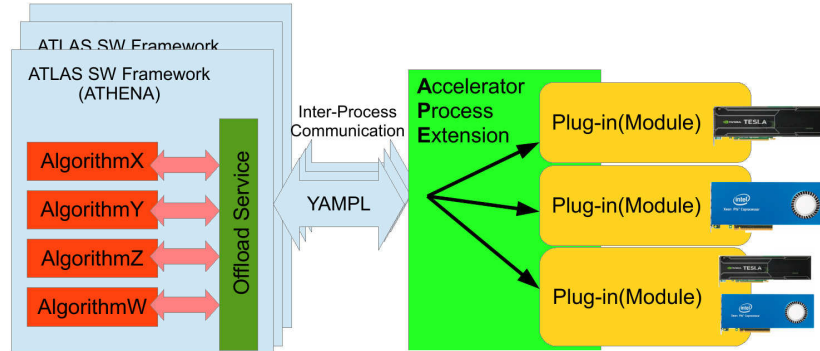
Figure 3: APE flow chart. Processing requests from algorithms together with data are sent to the server. Data are processed in appropriate module and results are sent back.

The APE framework also simplifies the development cycle and enables use of different languages and compilers which would not be possible inside the Athena framework. Since the client is hidden from algorithms and only input data is important, it is possible to use a standalone application to send a request with some predefined data sets without running the full reconstruction framework. IPC time between server and client is small compared to time budgets of the offloaded algorithms. In spite of that, it introduces a serial section and thus reduces the scaling as per Amdahl's Law. On the other hand, having multiple requests from different processes increases the throughput and utilization of GPU as per Gustafson's Law thus reducing the effect of the overhead.

3 GPU studies in ATLAS

There are several ongoing studies in ATLAS to use GPU resources. The Tracking for Muon and ID in Trigger and Reference Kalman-Filter for offline reconstruction are briefly described below.

3.1 GPUs for track finding in ATLAS Triggers

The ATLAS Trigger systems filter out more than 99.9% (in particular about 100 over 1 billion events are retained) of the events and select only interesting events. In order to decide whether an event is interesting or not, an incremental reconstruction is performed and a decision is produced in a few seconds. Data preparation and track reconstruction typically consume 50%-70% of the processing time budget. These processes contain some parallelization possibilities and they are implemented as GPU algorithms composed of various steps.

Figure 4: Bytestream decoding and clustering show a 26x speed-up on NVIDIA C2050 GPU vs single-threaded Intel E5620 CPU. [8]

Data preparation starts with decoding of detector output, the Bytestream. In this step, the compactly-encoded pixel and SCT hit information from the detector's readout buffers are retrieved, and decoded into hits within individual pixel and SCT modules. The Bytestream itself is divided into 16/32-bit words. While the CPU implementation of decoding iterates over the words sequentially, the GPU implementation maps each word to a GPU thread and does context detection and value decoding in GPU threads.

After the Bytestream is decoded, clusterization of neighboring hits in each module is done in order to take the activation of multiple adjacent detector cells by a single traversing particle. In the CPU, the pixel clustering is done in two nested loops and the SCT clustering is done in a single loop. In the GPU, a special cellular automaton-based algorithm is used to parallelize the comparisons done between hits. This approach allows parallelization in a module as well as across modules by assigning modules to different thread blocks. These clusters then converted to Space Points by using their locations in space and calculating their centers. A comparison of timings for CPU and GPU based data preparation implementations for different Bytestream sizes is given in Figure 4.

After data preparation is completed, track formation starts. The first step in track formation is the creation of track candidates. In this step,
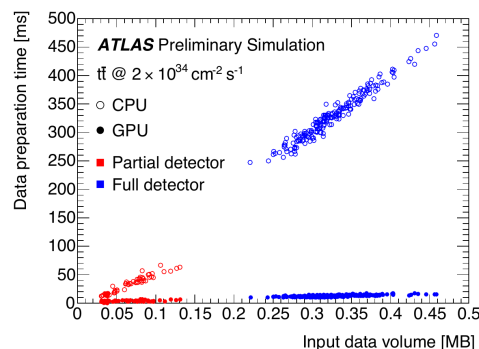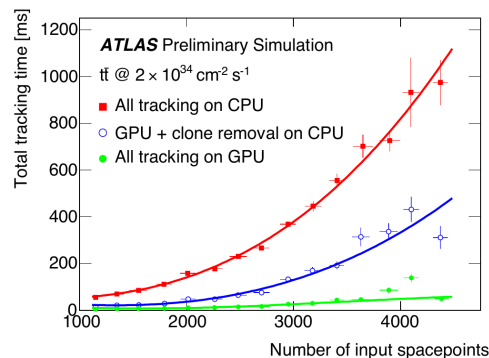

Figure 5: Track formation and clone removal show a 12x speed-up on NVIDIA C2050 GPU vs single-threaded Intel E5620 CPU. [8]

space points in different layers are paired to create seeds. These seeds are then extended into outer silicon layers by looking for tertiary hits. This is a combinatoric algorithm which does not scale well with hit multiplicity. The CPU version is implemented as a series of nested loops while the GPU version essentially takes the Cartesian product of different detector layers by using a 2-dimensional block of GPU threads.

Once track candidates are formed, clone removal is done. During the clone removal step, track candidates which are formed by the same outer layer hits but different inner layer seeds are merged and copies are removed. Due to nature of the task, the GPU implementation splits the task in to identification and merging/removal while on the CPU it is done in a single step. Timing results of the track formation steps are shown in Figure 5.

Another place in the Trigger where GPUs are being studied is the Muon triggers. The ATLAS Muon trigger tries to select online events with relevant physics properties, based on the presence of a muon in the event. A muon is expected to leave hits in the inner tracking systems of ATLAS, as well as in the outer Muon Spectrometers while traversing the detector. On the other hand it is expected to leave a very small amount of energy in the calorimeter. To achieve best-possible muon reconstruction, all these have to be taken in account within allowed time budget. Initial implementation is aimed to calculate energy deposition in calorimeter cells around expected muon trajectory, which involves looping over several hundreds of calorimeter cells. However, flattening data structures for GPU utilization was found to be a limiting factor.
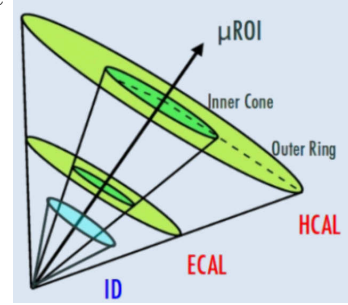
Figure 6: Muon regions of interest. Data around expected muon track at different detectors is taken into account.

3.2 GPU Based Reference Kalman-Filter for Offline Reconstruction

To allow a fair performance comparison between different systems, a reference Kalman- Filter[9] was implemented in four independent versions. A serial C++ version of the Kalman-Filter is used to generate a base performance index for further comparison. Then we implemented three different parallelized Kalman-Filter algorithms using OpenMP, OpenCL[10] and CUDA[11]. Each of these uses the same flat data structures and produces the same results. To measure the performance of each implementation a set of test data
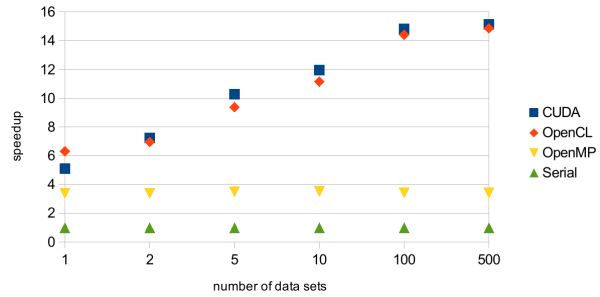
Figure 7: relative speedup of OpenMP, CUDA and OpenCL implementations compared to serial C++ code

is used which consists of 96 events containing 19500 tracks and a total of 220000 hits. The test system uses an Intel XEON E5-1620 processor with 8 GB RAM and a nVidia GeForce GTX Titan 6 GB graphics card.

To achieve the results shown in Figure 7, we improved the code base of both the OpenCL and CUDA implementation so that the whole Kalman-Filter chain is processed on the GPU. This includes forward and backward filtering and the smoothing step. All tracks per event are

processed in parallel using 5x5 GPU threads (corresponding to co-variance matrix entries). For an even higher performance, we use GPU-side matrix inversion which is done completely in Shared Memory of the GPU. With the used hardware we were able to invert up to 224 matrices in parallel on the GPU.

To simulate a higher load we ran the implementation in a benchmark fashion by calculating the same dataset several times in a loop (up to 500 times) as shown in Figure 7. The GPU implementations (CUDA and OpenCL) then achieved a relative speedup of about 15x compared to the serial C++ implementation.

4   Conclusion

ATLAS is continuously producing a tremendous amount of data that has to be processed. The possibility of compensating this rising amount of data just by the increase of CPU clock frequency will be no longer an option due to physical constraints. CPU clock frequencies are nearly at their maximum and therefore simply more cores and vector units are added so that one now has to make use of parallel programming. Co-processors such as GPUs have a relatively higher computing power per watt at a lower cost compared to CPUs, such that the use of co-processors looks like a promising solution.

We introduced the APE framework, an approach for integrating co-processors into the existing ATLAS software framework Athena. This study allows us to use GPUs within Athena with as little changes in the existing code as possible. We therefore have a working solution for short- to medium-term software framework integration.

We also have first implementations for online computing tasks in the ATLAS Trigger as well as a Kalman-Filter approach for offline computing. We are still evaluating other possible parallelizable problems from which we could gain reasonable speedups in computation time. Especially because of its combinatorial nature, track reconstruction seems to be promising.

We already achieved several encouraging results. Performing the data preparation steps of the ATLAS trigger on a GPU reached a relative speedup of up to 26x compared to a serial version run on a CPU. Parallelizing a Reference Kalman-Filter implementation achieved a speedup of 16x also compared to a single threaded CPU version.

References

[1] ATLAS Collaboration. Atlas experiment luminosity public results. `https://twiki.cern.ch/twiki/bin/view/AtlasPublic/LuminosityPublicResults`.

[2] Lyndon Evans and Philip Bryant. LHC Machine. *JINST*, 3:S08001, 2008.

[3] The Atlas Collaboration. The ATLAS Experiment at the CERN Large Hadron Collider. *JINST*, 3:S08003, 2008.

[4] M Elsing. Online and Offline Tracking for High Pileup. (ATL-SOFT-SLIDE-2014-513), Aug 2014.

[5] Marco Clemencic, Hubert Degaudenzi, Pere Mato, Sebastian Binet, Wim Lavrijsen, et al. Recent developments in the LHCb software framework Gaudi. *J.Phys.Conf.Ser.*, 219:042006, 2010.

[6] S Binet, P Calafiura, M K Jha, W Lavrijsen, C Leggett, D Lesny, H Severini, D Smith, S Snyder, M Tatarkhanov, V Tsulaia, P VanGemmeren, and A Washbrook. Multicore in production: advantages and limits of the multiprocess approach in the atlas experiment. *Journal of Physics: Conference Series*, 368(1):012018, 2012.

[7] iMatix Corporation. ZeroMQ web page. `http://zeromq.org/`.

[8] JTM Baines, TM Bristow, D Emeliyanov, JR Howard, S Kama, AJ Washbrook, and BM Wynne. An evaluation of the potential of GPUs to accelerate tracking algorithms for the ATLAS trigger. Technical Report ATL-COM-DAQ-2014-094, CERN, Geneva, Sep 2014.

[9] R. Fruehwirth, M. Regler, R.K. Bock, H. Grote, and D. Notz. *Data Analysis Techniques for High-Energy Physics*, chapter 3.2.5, pages 244 – 252. Cambridge Univerity Press, 2nd edition, 2000.

[10] Maik Dankel. Implementierung eines GPU-beschleunigten Kalman-Filters mittels OpenCL. Master's thesis, Fachhochschule Muenster, 2013.

[11] Rene Böing. Implementation eines CUDA basierten Kalman-Filters zur Spurrekonstruktion des ATLAS-Detektors am LHC. Master's thesis, Fachhochschule Muenster, 2013.

Chapter 2

GPU in Low-Level Trigger

Convenors:

Massimiliano Fiorini
Silvia Arezzini

GPU-based Online Tracking for the $\overline{\text{P}}$ANDA Experiment

Andreas Herten[1] on behalf of the $\overline{\text{P}}$ANDA collaboration

[1]Forschungszentrum Jülich, Wilhelm-Johnen-Straße, 52428 Jülich

The $\overline{\text{P}}$ANDA experiment is a new hadron physics experiment currently being built at FAIR, Darmstadt (Germany). $\overline{\text{P}}$ANDA will study fixed-target collisions of antiprotons of $1.5\,\text{GeV}/c$ to $15\,\text{GeV}/c$ momentum with protons and nuclei at a rate of 20 million events per second. To distinguish between background and signal events, $\overline{\text{P}}$ANDA will utilize a novel data acquisition technique. The experiment uses a sophisticated software-based event filtering scheme involving the reconstruction of the whole incoming data stream in real-time to trigger its data taking. Algorithms for online track reconstruction are essential for this task. We investigate algorithms running on GPUs to solve $\overline{\text{P}}$ANDA's real-time computing challenges.

1 The $\overline{\text{P}}$ANDA experiment

$\overline{\text{P}}$ANDA, short for Anti**p**roton **An**nihilation at **Da**rmstadt, is a new hadron physics experiment currently being built for FAIR, the Facility for Antiproton and Ion Research in Darmstadt, Germany. $\overline{\text{P}}$ANDA will deliver unprecedented insight into current topics of hadron physics. The experimental program includes meson spectroscopy in the charm energy region (charmonium; open charm), baryon production, nucleon structure, charm in nuclei, and searches for exotic states.

$\overline{\text{P}}$ANDA is an internal experiment at the High Energy Storage Ring (HESR) of FAIR. Antiprotons of $1.5\,\text{GeV}/c$ to $15\,\text{GeV}/c$ momentum collide with protons inside of $\overline{\text{P}}$ANDA in a fixed-target fashion. The resulting physical events are detected by $\overline{\text{P}}$ANDA's target spectrometer, located around the interaction region, and the forward spectrometer, dedicated to precision measurements of the forward-boosted event structure. [1]

Tracking detectors The innermost sub-detector of $\overline{\text{P}}$ANDA is the Micro Vertex Detector (MVD), a silicon-based detector directly around the interaction point. 10 million pixel and $200\,000$ strip channels enable vertex reconstruction with a resolution of $< 100\,\mu\text{m}$. The Straw Tube Tracker (STT), $\overline{\text{P}}$ANDA's central tracking detector, surrounds the MVD. The detector consists of 4636 small drift tubes of $1\,\text{cm}$ diameter, aligned into 26 layers (18 parallel to the beam axis, 8 tilted by $\pm 1.5°$ with respect to the beam axis). Both the STT and MVD are located within the field of $\overline{\text{P}}$ANDA's $2\,\text{T}$ solenoid. A third tracking detector covers forward angles around the beam line with gas electron multiplying (GEM) foils. Three GEM stations with two tracking planes each will be installed.

Online trigger In the investigated energy region, background and signal events have similar signatures and can not be distinguished easily. A conventional hardware-based triggering mechanism would not be efficient for $\overline{\text{P}}$ANDA. Instead, the experiment will use a sophisticated software-based online event trigger for event selection in realtime [1]. This process needs to reduce the raw data rate of $200\,\text{GB/s}$ ($\approx 2 \times 10^7$ event/second) by three orders of magnitude to match the imposed limits by the storage facility ($\sim 3\,\text{PB/year}$) for further in-depth offline analysis.

Figure 1: $\overline{\text{P}}$ANDA's online triggering system.

The online trigger of $\overline{\text{P}}$ANDA is structured as in Figure 1. In particular, online track reconstruction is a challenging part in terms of performance and efficiency, as a large number of computations are performed on a combinatorial complex datastream.

In this work, different tracking algorithms are investigated for their suitability to match the performance needed for $\overline{\text{P}}$ANDA's online reconstruction. The algorithms are implemented on GPUs, exploiting the low-cost, high-performance computing infrastructure offered by these devices. In addition to GPU-based tracking, event reconstruction using FPGAs is also investigated for $\overline{\text{P}}$ANDA [2], but not part of this report.

2 Online tracking algorithms

We are currently investigating three different algorithms for GPU-based track reconstruction. They are presented in the following.

2.1 Hough Transform

Hough Transforming is a method used in image processing to detect edges in images. It was first used for digitalization of images from bubble chamber experiments at CERN in the 1960s [3]. Adapting it for track finding and fitting means, to stay in computer vision terminology, to find edges (*tracks*) in images with a small number of pixels (*hit points*).

Method The method centers around transforming a point into a set of values in a parameter space (Hough space) and then extracting the most frequently generated parameters.

For every hit point (x_i, y_i), the equation

$$r_{ij} = x_i \cos \alpha_j + y_i \sin \alpha_j \tag{1}$$

is solved for a number of $\alpha_j \in [0°, 180°)$. r_{ij} is an equation of a line going through (x_i, y_i) avoiding possible poles. Every parameter pair (r_{ij}, α_j) is filled into a histogram. As more and more hit points of the same track are evaluated, a bin with the highest multiplicity emerges. This is the parameter pair equivalent to the line best connecting all hit points – the desired track parameter. See Figure 2.

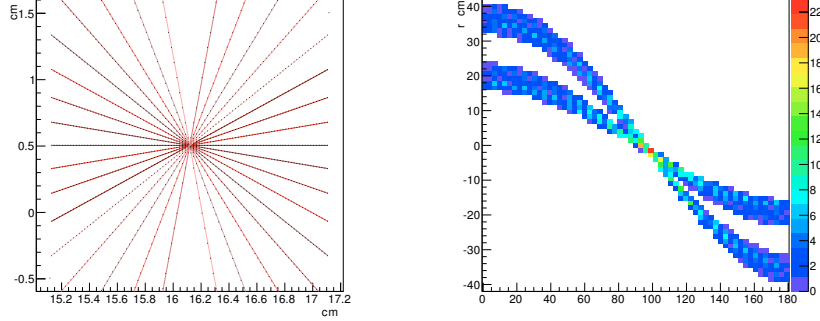Figure 2: Schematic visualizations of Hough Transforms with coarse step sizes α for illustration. *Left*: Different lines going through a hit point (center) are sampled in 10° steps with Equation 1. *Right*: Hough space for a set of STT hit points, the α granularity is 2°.

Conformal map pre-step Since $\overline{\mathrm{P}}$ANDA's central tracking detectors are enclosed in a solenoid magnet, the trajectories of particles are bent. A conformal mapping as a Hough Transform pre-step is used to convert curved tracks with hit points (x_i, y_i) into straight lines with hit points (x_i', y_i'):

$$\begin{pmatrix} x_i' \\ y_i' \end{pmatrix} = \frac{1}{x_i^2 + y_i^2} \begin{pmatrix} x_i \\ y_i \end{pmatrix}. \tag{2}$$

While conformal mapping involves only two computations per hit point, the actual Hough Transform calculates a large set of transformed points per hit – all independent from each other and hence very suitable to compute on a GPU. For example, an average event with 80 STT hits, evaluated every $0.2\,°$, leads to $72\,000$ computations, all independent from each other. The α granularity is here the main source for parallelism and subject to tuning. Its value is limited by detector measurement uncertainties on the one side and by available computing resources on the other.

Implementations Currently, two implementations are available for the Hough Transform. The first uses Thrust [6], the template library for CUDA, offering a big set of standard computing functions. No explicit CUDA kernel has been written for this implementation, only specialized operators were used. The benefit of this method is the ability to make use of the pre-programed sophisticated algorithms, optimized for high GPU performance. The drawback is an overhead of converting the data into Thrust-compatible data types, and the inflexibility when customizing the algorithms. With $3\,\mathrm{ms/event}$, this implementation is also not at its performance maximum.

The second implementation is a plain CUDA implementation, built completely for this task. Its performance is six times better ($0.5\,\mathrm{ms/event}$) than the Thrust version. It is fitted for $\overline{\mathrm{P}}$ANDA's exact task and is completely customizable, since it uses plain kernels throughout. Since this version is both faster and more flexible, we consider the plain CUDA implementation the better approach for Hough Transforms at $\overline{\mathrm{P}}$ANDA.

The Hough Transform implementations give the expected results and fill the Hough space reliably with the data of possible tracks. (Figure 3, left). A further challenge, though, is peak
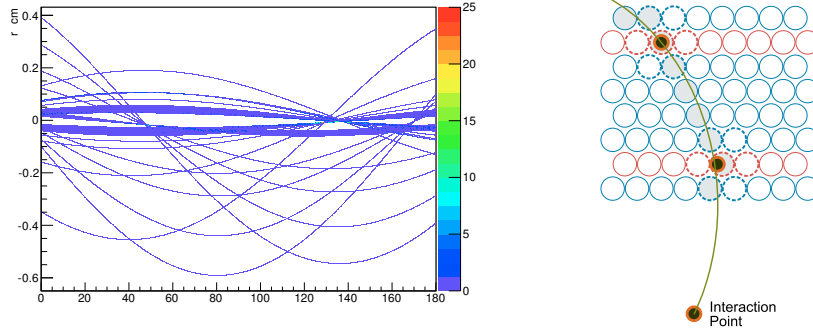
Figure 3: *Left*: Hough Transform of 68 hit points (STT, MVD). α is evaluated every $0.1°$. Thrust is used for computation, ROOT [4] for displaying, and CUSP [5] for providing fast, templated methods to interface between both using the GPU. *Right*: Sketch of Triplet generation of the Triplet Finder.

finding in the Hough space: Large sets of hits create overlapping bands of parameter values in the Hough space. A complicated picture emerges to find peaks in – a simple threshold cut is not feasible. This part is currently under study.

2.2   Riemann Track Finder

The Riemann Track Finder is a track reconstruction algorithm already in use in PandaRoot, $\overline{\text{P}}$ANDA's software framework, since several years [7]. The basic principle of the method is a projection of two-dimensional hit points onto a Riemann surface (paraboloid) in a three dimensional space. There, a plane going through the mapped hit points can easily be fitted. The parameters of the plane are re-mapped into the two-dimensional and, with this, converted into track parameters. The method is based on [8].

GPU optimization   As three points can always parameterize a circle, this is the minimum number of hits required for the Riemann Track Finder. A set of three hits (*seed*) is grown to a proper track candidate by subsequently testing additional hits against it and checking for passing certain quality cuts. The algorithm works on MVD hits. The seeding three-hit combinations are generated in the CPU version by three cascaded `for` loops, each one for a different layer of the MVD, implicitly assuming a track can not have two hits in the same layer. For running seed generation in parallel on the GPU, the serial loops are flattened out. Using explicit indexes, a $1:1$ relation between global kernel variables involving CUDA's `threadIdx.x` and a layer hit combination is formulated.

As this produces a considerable amount of track seeds, the rest of the core Riemann Track Finder code is directly ported to the GPU. No further optimization has initially been done – the parallelism based on seeds suffices for a $100\times$ performance increase when compared to the CPU version. Tracks of an event are found, on average, in $0.6\,\text{ms}$ using a NVIDIA Tesla K20X GPU. The overhead needed for copying data to and from the device comprises $7\,\%$ of this time, making the Riemann Track Finder a computing-intensive algorithm and a good candidate for running on GPUs.
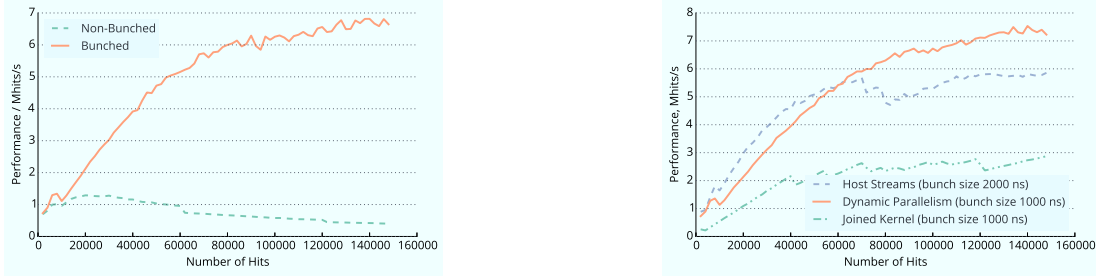
Figure 4: Performance for different optimization aspects of the Triplet Finder. Shown: Time needed to process a given number of hits, in million hits processed per seconds. *Left*: Comparison of Triplet Finder performance with and without bunching wrapper (optimal bunch size (1000 ns), dynamic parallelism approach). *Right*: Comparison of performance of different kernel launch strategies, invoked with respective ideal bunch sizes in bunching wrapper.

2.3 Triplet Finder

The Triplet Finder is an algorithm specifically designed for the $\overline{\text{P}}$ANDA STT [9]. It is ported to the GPU in collaboration with the NVIDIA Application Lab of the Jülich Supercomputing Centre.

Method   Instead of evaluating data of the whole STT all the time, the Triplet Finder initially takes into account only a subset. Certain rows of neighboring STT drift tubes (*straws*) are selected for initiating the algorithm. As soon as a hit is detected in such a so-called *pivot layer*, the algorithm searches for additional hits in the directly neighboring straws. A center-of-gravity point is formed from all the hits, called a *Triplet*. Combining a Triplet from a pivot layer on the inner side of the STT with a Triplet from a pivot layer from the outer side of the STT with the interaction point at $(0, 0)$, a circle as a track candidate is calculated. In the next step, hits close to the track candidate are associated to it to eventually form a proper track, as shown in Figure 3, right. [10]

The Triplet Finder is a robust algorithm with many algorithmic tuning possibilities. Track candidates are calculated without relying on the event start time t_0, a value usually needed by algorithms to generate an isochronous hit circle around a straw's anode wire. t_0 is not known a-priori, as $\overline{\text{P}}$ANDA is running without any trigger information and needs to be provided by dedicated algorithms.

GPU optimizations   A number of different GPU-specific optimizations are performed on the algorithm. Two of them are highlighted in the following, for a full list see [11].

Bunching wrapper   The Triplet Finder looks at a set of hits at once and computes all possible track candidates. For a certain amount of hits, the algorithm reaches its peak performance. On the tested graphics card (NVIDIA Tesla K20X), this point is roughly equivalent to 25 000 hits (or 1000 ns STT data). To always invoke the algorithm with the number of hits at which it is performing best, a *bunching wrapper* is introduced. This wrapper cuts the continuous hit stream into sets (*bunches*) of sizes that maximize the occupancy of the GPU.

The algorithmic complexity is reduced and a performance maximum of 7 Mhit/s is reached (Figure 4, left).

Kernel launch strategies   Different methods of launching the Triplet Finder processes are evaluated.

- **Dynamic Parallelism**: This method was used during the GPU development of the algorithm. Per bunch, one GPU-side process (*thread*) is called, launching itself the different Triplet Finder stages subsequently as individual, small kernels directly from the GPU.

- **Joined kernel**: One CUDA block per bunch is called on the GPU. Instead of individual kernels for the stages of the Triplet Finder (as in the previous approach), one fused kernel takes care of all stages.

- **Host streams**: Similarly to the first approach, the individual stages of the algorithm exist as individual kernels. But they are not launched by a GPU-side kernel, but by *stream* running CPU-sided. One stream is initiated per bunch.

Results in Figure 4, right, show the Dynamic Parallelism approach to be the fastest, as GPU occupancy is high and GPU-CPU communication reduced.

Optimization results   The best performance in terms of computing time needed to process an event, which was reached with the aforementioned and further optimizations, is 0.02 ms/event (see also [11]). Employing only the Triplet Finder as a tracking algorithm, a multi-GPU system consisting of $\mathcal{O}(100)$ GPUs seems sufficient for $\overline{\text{P}}$ANDA.

3   Conclusions

Different algorithms have been presented for online track reconstruction on GPUs in the $\overline{\text{P}}$ANDA experiment. Each algorithm is in a different stage of development and has distinct feature sets specializing on different objectives. The most optimized algorithm is the Triplet Finder, with performance results making GPU-based online tracking a promising technique for $\overline{\text{P}}$ANDA's online event filter.

We are continuing tailoring the algorithms for $\overline{\text{P}}$ANDA's needs – optimizing performance and modifying specific aspects. All presented algorithms still need to be validated with physics cases of $\overline{\text{P}}$ANDA and benchmarked in terms of reconstruction quality. Also, further research is needed beyond the STT when including more sub-detectors into the different algorithms. Recently, high-throughput data transfer to the GPU is also subject of our research to complement the promising algorithmic developments towards a integrated GPU online track reconstruction system for $\overline{\text{P}}$ANDA.

References

[1] PANDA Collaboration. Physics Performance Report for PANDA: Strong Interaction Studies with Antiprotons. *arXiv:0903.3905*, 2009.

[2] H. Xu, Z.-A. Liu, Q. Wang, J. Zhao, D. Jin, W. Kühn, S. Lang, and M. Liu. An atca-based high performance compute node for trigger and data acquisition in large experiments. *Physics Procedia*, 37(0):1849 – 1854, 2012. Proceedings of the 2nd International Conference on Technology and Instrumentation in Particle Physics (TIPP 2011).

[3] Paul VC Hough. Machine analysis of bubble chamber pictures. In *International Conference on High Energy Accelerators and Instrumentation*, volume 73, 1959.

[4] Rene Brun and Fons Rademakers. Root—an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997.

[5] Webpage: `http://cusplibrary.github.io/`.

[6] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems*, 7, 2011.

[7] Stefano Spataro and the PANDA Collaboration. The PandaRoot framework for simulation, reconstruction and analysis. *Journal of Physics: Conference Series*, 331(3):032031, 2011.

[8] R Frühwirth, A Strandlie, and W Waltenberger. Helix fitting by an extended riemann fit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 490(1):366–378, 2002.

[9] PANDA Collaboration. Technical design report for the panda (antiproton annihilations at darmstadt) straw tube tracker. *The European Physical Journal A*, 49(2):1–104, 2013.

[10] M. C. Mertens for the PANDA collaboration. Triplet based online track finding in the PANDA-STT. *Journal of Physics: Conference Series*, 503(1):012036, 2014.

[11] Andrew Adinetz, Andreas Herten, Jiri Kraus, Marius C Mertens, Dirk Pleiter, Tobias Stockmanns, and Peter Wintz. Triplet finder: On the way to triggerless online reconstruction with gpus for the panda experiment. In *Procedia Computer Science*, volume 29, pages 113–123. Elsevier, 2014.

Parallel Neutrino Triggers using GPUs for an Underwater Telescope

Bachir Bouhadef [1], *Mauro Morganti* [1,2], *Giuseppe Terreni* [1]
for the KM3Net-It Collaboration

[1] INFN, Sezione di Pisa, Polo Fibonacci, Largo B. Pontecorvo 3, 56127 Pisa,Italy
[2] Accademia Navale di Livorno, viale Italia 72, 57100 Livorno, Italy

Graphics Processing Units are high performance co-processors originally intended to improve the use and the acceleration of computer graphics applications. Because of their performance, researchers have extended their use beyond the computer graphics scope. We have investigated the possibility of implementing online neutrino trigger algorithms in the KM3Net-It experiment using a CPU-GPU system. The results of a neutrino trigger simulation on a NEMO Phase II tower and a KM3-It 14 floors tower are reported.

1 Introduction

A neutrino telescope is a tool to increase our knowledge and to answer fundamental questions about the universe. Following the success of the IceCube experiment [1], which is a km^3 size telescope in the ice at the south pole, and of the ANTARES experiment [2], an underwater telescope with a volume of 0.4 km^3. The European scientific community is going to construct a neutrino telescope similar to but larger than IceCube called Km3Net in the Mediterranean Sea. NEMO [3] and NESTOR [4] are R&D experiments for the same purpose. All these optical telescopes use a Photomultiplier Tube (PMT), or a group of it, as a Detection Unit (DU). The NEMO collaboration has already deployed a tower of 32 single PMT DUs. For the much larger Km3Net telescope, thousands of DUs will be used to detect the muon passage produced by undersea neutrino interactions. This large number of sensors will lead to a huge amount of data because of the presence of high rate background, and the data must be filtered by an efficient trigger algorithm to reduce the background rate and to keep all possible muon tracks. In the case of the ANTARES telescope, the amount of data acquired in a second is 0.3-0.5 GB, and a set of CPUs is used for such a task. The general strategy of data analysis for an online trigger is that a set of CPUs works in parallel on consecutive time slices within one second of the data coming from the underwater telescope [3]. In the present work, we describe the study of combining a Graphical Processor Unit (GPU) and CPU (GPU-CPU) to implement the muon trigger algorithms. In addition, the use of GPU-CPU leads to savings in power, hardware and time. The parallel version of the trigger algorithm is shown to be suitable for an online muon track selection and was tested on simulated data of the NEMO towers of 32 (NEMO Phase II) and 84 (KM3Net-It tower) single PMT DUs.

2 DAQ system of NEMO Phase II

During March 2013, the NEMO collaboration has deployed a tower (See Fig.2) at the Capo Passero site, south east of Sicily island [3]. Since then, it was taking data until August 2014, when it was shut-down for upgrades. The tower is 450 m high and it is composed of 8 floors, each floor is equipped with 4 DUs and two hydrophones. The distance between floors is 40 m, where the length of the floor is 6 m. The tower is kept straight by an appropriate buoyancy on its top.

The tower is hosted at a depth of 3400 m nearly 100 km off the Capo Passero harbor. Each floor is equipped with a Sea Floor Control Module (SFCM) that collects data streams from the four PMTs as well as the two hydrophones and sends it to its twin card on-shore called Earth FCM (EFCM). The communication between the SFCM and EFCM is guaranteed via an Electro-Optical-Cable (EOC) which permits 2.5 Gbs of bandwidth. During data acquisition and low bioluminescence activity, the measured rate by each PMT is 50-55 kHz. The PMT hit size is 28 bytes in average and 20% of the available bandwidth is used.

The data streams from all EFCMs are grouped into two streams and routed to the first stage of CPUs data processing called Hit Managers (HM) via Gbit link, which means the tower has two HMs as in Figure 3. Each HM merges together the 16 PMT data streams and divides them in consecutive time intervals of 200 ms called time-slices. All time slices from HMs within the same time interval are sent to one Trigger CPU (TCPU), where the trigger is applied. Successive time slices are sent to different TCPUs. A detailed description of the DAQ system can be found in [3]. Once trigger conditions are satisfied, a $6\mu s$ time window of the tower data (centered at the time of the trigger seed) is sent to the Event Manager for storage. And last, another off-line filter is applied on the saved events for muon track reconstruction.

3 Muon trigger strategy

A muon passing through water, faster than the speed of light in that medium, produces Cherenkov photons. These photons have an angle θ_C with respect to the trajectory of the muon and are detected by distributed DUs. The arrival times of Cherenkov photons at PMTs are correlated in time and space which is not the case for background photons (most photons are generated by ^{40}K decays). Hence, looking for coincidences of multiple PMTs within a certain time window reduces the background rate.

In fact, the time difference of the arrival time of photons generated by a muon are distributed in a $3\mu s$ (time needed for a muon to traverse the detector) interval or less. A time-space correlation between PMT hits within a time interval less than $3\mu s$ can be used to filter out the background hits. In addition to the time-space correlation, a hit charge over threshold trigger can be applied on all hits. These types of triggers can be parallelized as they are applied on data streams independently.

Even though these triggers are simple, the amount of background hits in the NEMO phase II (8 floors tower) is about 1.7 Mhits/s/tower (comparing to roughly a few hits/s of a muon track) and it will be 4.6 Mhits/s for the 14 floors tower. In addition to these standard triggers, we have studied a new level 0 trigger algorithm. It is easy to implement on a GPU and is based on N hits in a fixed Time Windows TW, we chose N=7 and TW=1000ns (N7TW1000). This trigger reduces the time-consuming of time-space correlation and charge over threshold triggers [5]. After this level 0 trigger, we apply the previously mentioned triggers, reducing the rate

drastically.

4 CPU-GPU

The aim of our work is to replace the TCPU by a TCPU-GPU system (TGPU). The work in the TGPU is done in three steps: first prepare 1s of data to be sent to GPU, then, send data to the GPU for trigger selection, and last, save the selected events.

Every PMT hit data point in the NEMO Phase II tower contains a header with the GPS time and geometry information, followed by a sampled charge waveform with a variable size. To apply triggers in the GPU, the raw hits are converted to a fixed size structure that contains all needed information for the trigger algorithms: charge, time, DU identification (DU_ID), trigger flags, and rates. We have optimized the work in CPU and GPU to minimize the trigger searching time as explained in next paragraphs.

4.1 Preparation in CPU

The main work of the CPU is to convert 5 consecutive time slices of 200 ms to a unique time slice of 1 s to be sent to the GPU (one fixed-size memory buffer). This is done by running 5 threads in the CPU; each thread is filling the dedicated memory zone, which means that the 5 CPU treads are filling the same memory zone but at different offsets. Each thread converts the 200 ms time slice to N time slices to be used by GPU threads. The number of threads N is chosen to have in average 100 hits per thread at a nominal rate of 50 kHz/PMT. Hence, the total number of threads (NTHRD) in a GPU is NTHRD=$5 \times N$.

In our simulation code we have also taken into account the edge effect between threads. When the trigger algorithm reaches the last hit of the current thread, it proceeds on first hits of the next time slice. The data sent to the GPU is a 1 second time interval and its size takes into account the maximum rate of each PMT (5 times the nominal rate). Hence, with the NEMO tower of 84 PMTs and in the presence of bioluminescence at two PMTs, for example at rate of 1 Mhits/s, and assuming a nominal hit number per thread of 100 (55 kHz/PMT), we expect an increase of 40% (140 hits/thread) of the number of total hits.

4.2 Sorting and triggering in GPU

The work on the GPU side is done in two steps. First we sort the hits in time using classical sorting methods (Shell, quick, and merge sort algorithms). Than we apply the needed trigger algorithms (see the list below). Figure 4 shows the performance of quicksort and Shellsort algorithms on uniformly distributed time values. To see the effect of the hit structure size on the sorting time, the sorting algorithms are tested on hits of different sizes (one float is used for the time value). Clearly the quicksort algorithm shows a good performance and the measured times for all cases remain below 100 ms for 100 elements per thread. In addition, Figure 4 shows how the size of the hit structure affects the performance of the sorting algorithms.

In the case of NEMO tower data, we have noticed, that the Shellsort algorithm shows a better performance over the others. The reason for this is that our data within a GPU thread is not completely random, but is time-ordered for single PMT within a GPU thread.

After time sorting, we apply the following trigger algorithms:

- Time-space correlation (for NEMO Phase II tower):

- N7TW1000: looking for 7 hits in TW = 1000 ns: `hit[i].time`- `hit[i+7].time`< TW

- DN7TW1000: looking for 7 hits from different PMTs within TW = 1000 ns

- Simple Coincidence (SC): a coincidence between two hits occurred in two adjacent PMTs within 20 ns

- Floor Coincidence (FC): a coincidence between two hits occurred in two PMTs in the same floor within 100 ns

- Adjacent Floor Coincidence (AFC): a coincidence between two hits occurred in two PMTs located at two consecutive floors within 250 ns

- Charge threshold (QTRIG) looking for charge over a threshold: `hit[i].Charge`> QTRIG

- Combination of the above trigger seeds: for example applying N7TW1000, than SC and FC

N7TW1000 and DN7TW1000 have the same efficiency for muon track selection, however DN7TW1000 is more efficient in the presence of bioluminescence activities for background reduction [5]. In addition, the DN7TW1000 is time consuming and for this reason we have used N7TW1000 in our simulation. To select muon events we combine these trigger seeds. Once the candidate hits satisfy the trigger, all hits within $\pm 3\mu s$ are tagged to be saved with respect to the time of one of the trigger seeds (in our simulation we have used the SC trigger). After that, the data is sent back to the CPU to save the tagged hits. Even though there is a difference in the rate of the selected events between applying N7TW1000 before or after time coincidences, we have chosen N7TW1000 first (as level 0 trigger) because the trigger time searching is less and the muon track selection efficiency remains the same [5].

5   Results and perspectives

For our simulation we have used a Tesla 20c50 (448 CUDA cores) and a GTX Titan (2688 CUDA cores) device. 2 PCs were used, one as a HM for time slice sending and the other PC was used with the GPU cards for the trigger algorithms. The measured time of the CPU to prepare 1 second of data ranges from 200 ms (for the 32 PMT tower) to 300 ms (for the 84 PMT tower), the CPU-GPU data memory transfer time is included. Our first aim was to see whether the TGPU can cope with data streaming and online triggering for muon track selection within the remaining time (less than 700 ms).

The first step was to simulate the actual NEMO Phase II tower with 32 DUs at a rate of 55kHz, applying the following triggers: charge trigger, SC, FC, AFC, N7TW1000, and a combination of them: applying N7TW1000, and if it is satisfied, we apply [SC and AFC] or [FC and AFC]. The last two triggers (N7TW1000 with [SC and AFC] or [FC and AFC]) are used to tag the muon track candidate. In the case of the KM3Net-It tower with 84 DUs, we use N7TW500 (the inter-floor distance is 20 m instead of 40 m).

Table (1) shows measured Tesla-GPU times for 1 second data from 32 and 84 PMTs using 8000, 20000, and 40000 threads, respectively, including the data preparation time on CPU (200-300 ms). We verify that the Tesla 20c50 GPU card is able to cope with the online

trigger algorithms of the NEMO Phase II tower, as well as of KM3Net-It tower (with number of GPU threads ≥ 20000).

We have also compared the performance of our triggers between a Tesla 20c50 and a GTX Titan devices. We have tested only the GPU work without including the 1 second time slice preparing as well as data memory transfer. The results are shown in Table (2). We see that the measured trigger times, for Tesla 20c50, are less than the results shown in Table (1), because the CPU is not busy by the 1 second time slice

NTHRDs	32 PMTs (t/ms)	84 PMTs (t/ms)
8000	250	950
20000	230	600
40000	190	500

Table 1: Measured trigger times (ms) using Tesla 2050c for 1 second time slice of NEMO towers data (time slice preparation on CPU is included).

preparation. For a sufficient number of GPU threads (>=2000), both GPU cards can handle the online trigger with data of 84 PMTs. The measured trigger times can be used to give a first estimate of the speeding up of the trigger algorithm in the GPU over the CPUs.

NTHRDs	32 PMTs (t/ms)	84 PMTs (t/ms)
8000	160 / 90	500 / 450
20000	100 / 50	200 / 140
40000	50 / 45	150 / 110

Table 2: Measured trigger times (ms) using Tesla20c50 / GTX TITAN for 1 second of NEMO towers data without including the time preparation.

The task of HMs can also be included in the CPU-GPU work, given that we were using only 40% of the CPU resources. HMs and TCPUs can be grouped in a TGPU system, with an adequate network structure. Our conclusion is that both GPU cards can be used in the TGPU system for the on-line muons trigger. We propose a new DAQ system based on TGPU structures (Figure 1) for the 8 towers of KM3Net-It, where each of the 8 TGPUs looks for all trigger seeds in a 1 second window of raw data of the corresponding tower and sends all trigger seeds to the TCPU for muon tracks selection. Once the TCPU selects the candidate trigger seeds for muon tracks, it sends back the corresponding times to all TGPUs to send their window of data to the Event Manger, and frees the corresponding memory buffer.

This new DAQ of the CPU-GPU is a huge simplification of the classical CPU DAQ system (using only CPUs) and can be used for both the online trigger and also the muon track reconstruction, by processing thousands of events at one step, and reducing further the fake events.

References

[1] A. Karle *et al.*, arXiv:hep-ex/14014496 (2014).

[2] S. Mangano *et al.*, arXiv:astro-ph/13054502 (2013).

[3] T. Chiarusi *et al.*, JINST 9 C03045 (2014).

[4] P.A. Rapidis, Nucl. Instr. and Meth. A, 602 (2009).

[5] B. Bouhadef *et al.*,
    Trigger Study for NEMO Phase 2, 6th Very Large Volume Neutrino Telescope Workshop, Stockholm, Sweden (2013).
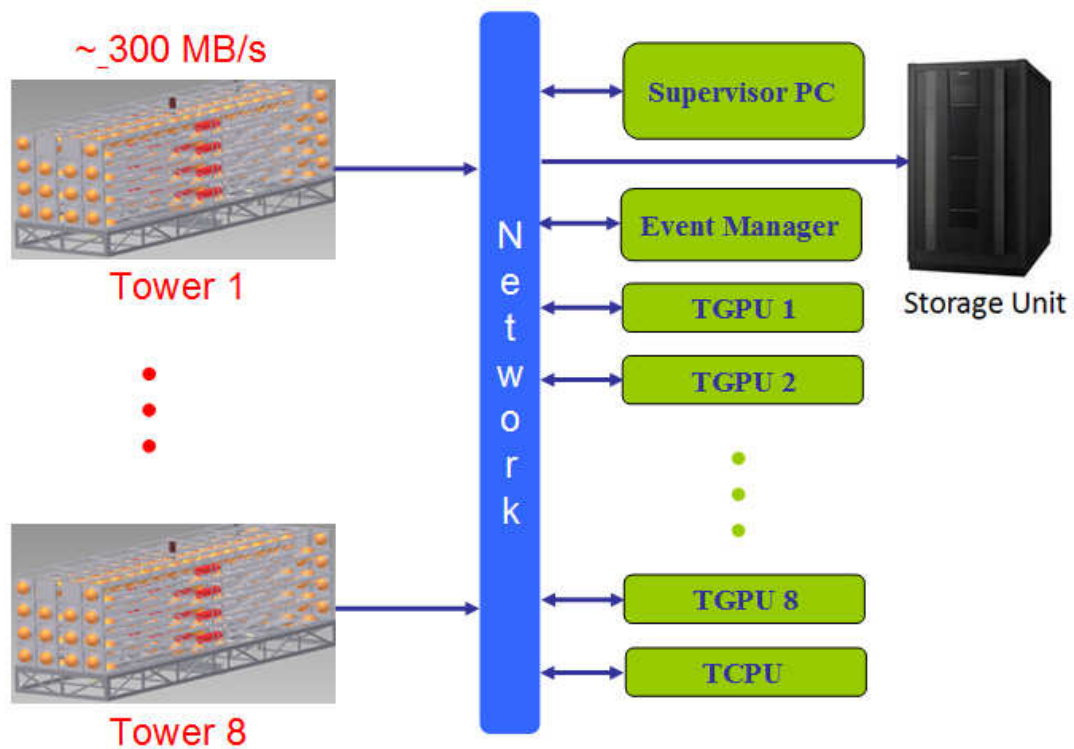
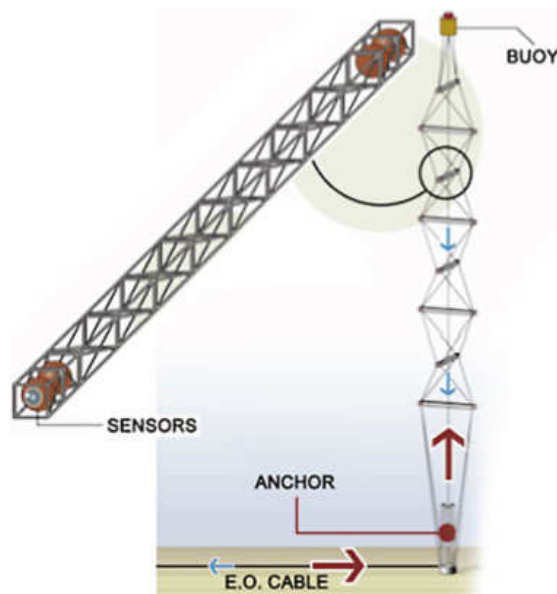Figure 1: Proposed DAQ based on CPU-GPU structure.
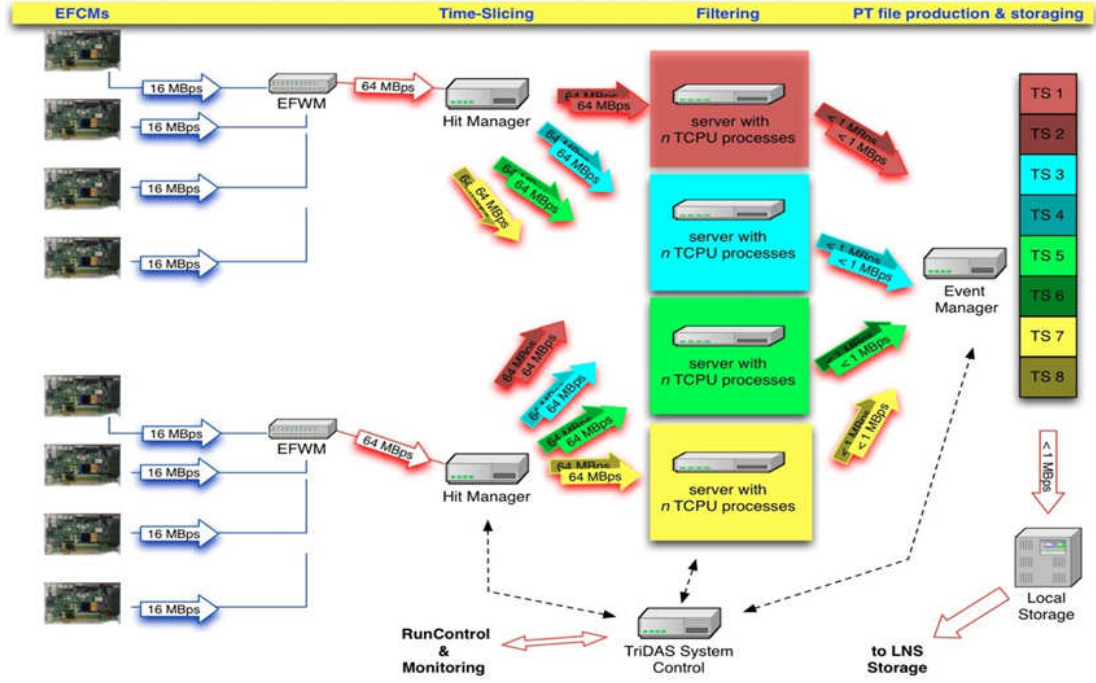

Figure 2: NEMO tower.

BACHIR BOUHADEF , MAURO MORGANTI, GIUSEPPE TERRENI

Figure 3: DAQ system for NEMO Phase II and the trigger scheme.
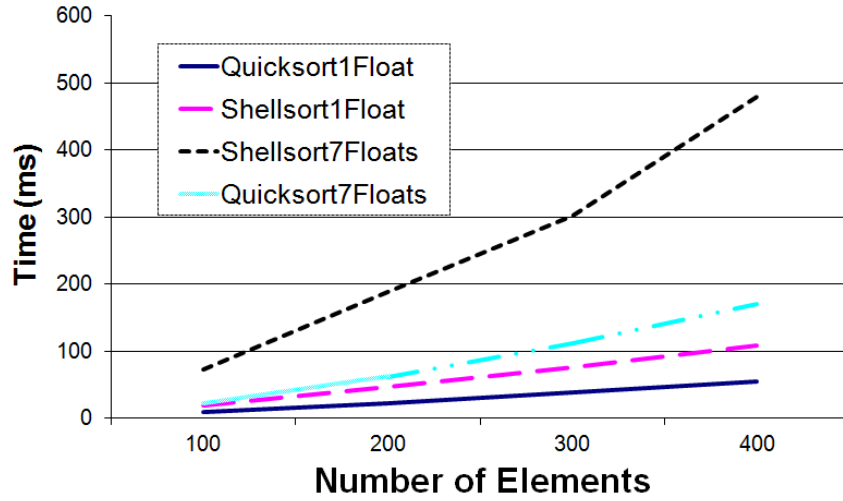

Figure 4: Measured sorting time (ms).

Track and Vertex Reconstruction on GPUs for the Mu3e Experiment

Dorothea vom Bruch[1] for the Mu3e collaboration

[1]Physikalisches Institut, Universität Heidelberg, Im Neuenheimer Feld 226, 69120 Heidelberg, Germany

The Mu3e experiment searches for the lepton flavour violating decay $\mu \to eee$, aiming at a branching ratio sensitivity of 10^{-16}. A high precision tracking detector combined with timing detectors will measure the momenta, vertices and timing of the decay products of more than 10^9 muons/s stopped in the target. The trigger-less readout system will deliver about 100 GB/s of data. The implementation of a 3D tracking algorithm on a GPU is presented for usage in the online event selection. Together with a vertex fit this will allow for a reduction of the output data rate to below 100 MB/s.

1 The Mu3e experiment

The Mu3e experiment [1] searches for the lepton flavour violating decay $\mu \to eee$. Within the standard model, this process is allowed via neutrino oscillations. It is however suppressed to a branching fraction below 10^{-54}. If lepton flavour violation is observed in the charged lepton sector, this is a clear indication for new physics. Many models beyond the standard model, such as supersymmetry, grand unified models or the extended Higgs sector predict lepton flavour violation at a level to which future detectors are sensitive. The current limit on the $\mu \to eee$ branching fraction was set by the SINDRUM experiment to 10^{-12} [2]. The Mu3e experiment aims to improve this limit by four orders of magnitude and to reach a sensitivity of 10^{-16} at 90 % CL.

To reach this sensitivity level, the distinction between signal and background events is crucial. In the Mu3e experiment, muons will be stopped in a target and decay at rest. A signal event consists of two positrons and one electron originating from one single vertex as shown in Figure 1a. They are coincident in time and the momentum sum is zero. The total energy of the event is equal to the rest mass of the muon.

One source of background is a radiative muon decay with internal conversion $\mu^+ \to e^+ e^- e^+ \bar{\nu}_\mu \nu_e$. This process is shown in Figure 1b. Here, the decay products are also coincident in time and have a single vertex, however the momenta do not add up to zero and the energy does not equal the muon rest mass. Combinatorial background stems from two ordinary muon decays $\mu^+ \to e^+ \nu_e \bar{\nu}_\mu$ taking place close to each other in space and time together with an additional electron from photon conversion, Bhabha scattering etc. (see Figure 1c). When both the e^+ and e^- from Bhabha scattering are detected, only one additional muon decay is required and the probability of misreconstruction as signal event is higher.

(a) Signal event   (b) Internal conversion   (c) Combinatorial background
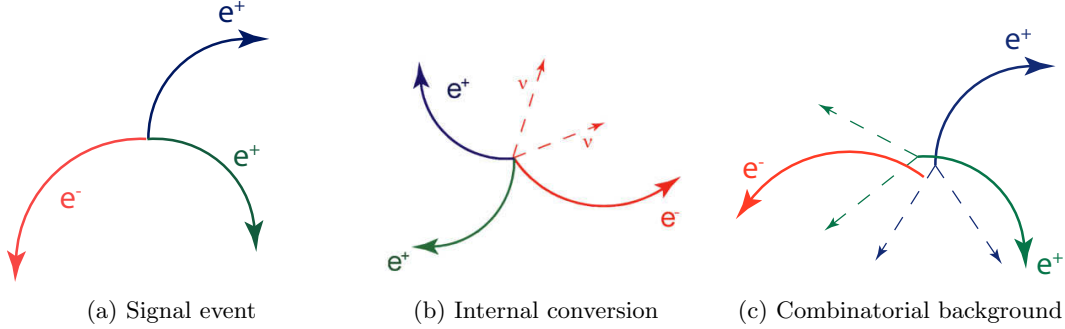
Figure 1: Comparison of signal and background events.

Momentum, timing and vertex resolution are therefore the key parameters for the distinction between signal and background events. To achieve the desired sensitivity, Mu3e aims for a momentum resolution < 0.5 MeV/c, a timing resolution of 100 ps and a vertex resolution of < 200 μm. These requirements guide the design of the Mu3e detector.

1.1 Detector design

Since the muons decay at rest in the target, the momentum of the decay electrons is at maximum half the muon mass (53 MeV/c). In this energy range, the momentum resolution is dominated by multiple Coulomb scattering whose variance is inversely proportional to the momentum. Therefore the design of the Mu3e experiment is aimed at minimizing the material budget. Ultralight mechanics will be used for construction and a pixel tracking detector is built from high voltage monolithic active pixels sensors [3, 4, 5] thinned to 50 μm, with a pixel size of $80{\times}80$ μm^2. In addition, scintillating tiles and fibres are included for precise timing information. The muons are stopped on the surface of a hollow double cone target, leading to a spread in the vertex locations. A magnetic field of 1 T is applied so that the tracks are bent and recurling tracks are detected by outer detector regions. This allows for a more precise momentum measurement. A schematic of the detector is shown in figure 2. To reach the sensitivity of 10^{-16} within a reasonable time, muons at high rates are desired which will be provided by the Paul Scherrer Institut. At the current beamlines, up to 10^8 μ/s are available. A future high intensity muon beam line (HiMB) could deliver in excess of $2 \cdot 10^9$ μ/s.

2 Readout scheme

A triggerless readout is foreseen for the detector. At the above mentioned rates this results in a data rate of about 100 GB/s. Consequently, an online selection is required to cope with this amount of data and reduce the rate by a factor of 1000. A schematic of the readout is shown in Figure 3. Zero-suppressed data are sent from the pixel sensors, the fibres and the tiles to front-end FPGAs via flex print cables. The FPGAs merge the data and sort it into time slices of 50 ns which are then transferred via optical links to the readout boards. Each readout board receives the information from each sub-component of the detector and then sends the complete detector information for one time slice to one computer of the filter farm via optical links. With
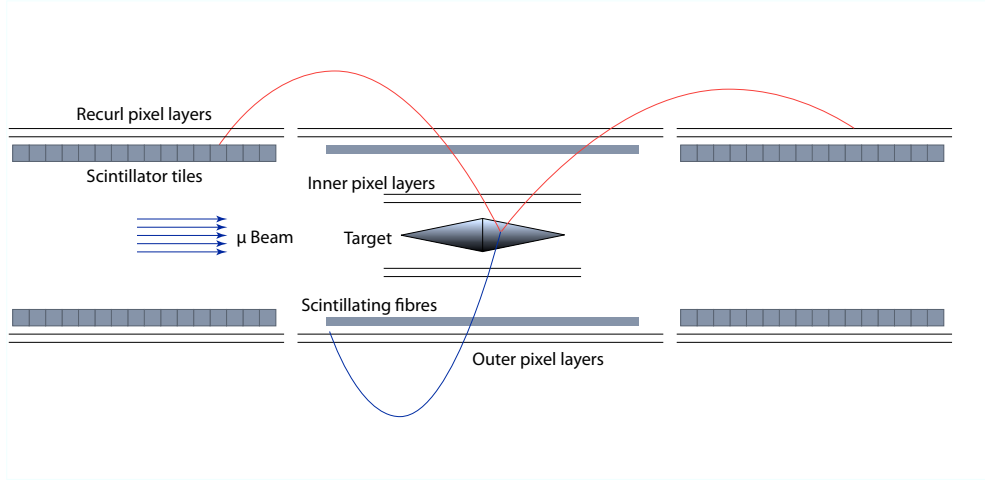
Figure 2: Schematic of the Mu3e detector design.

this procedure, each PC analyzes different time slices of full detector information. The GPUs of the filter farm PCs perform the online event selection by searching for 3 tracks originating from one single vertex.
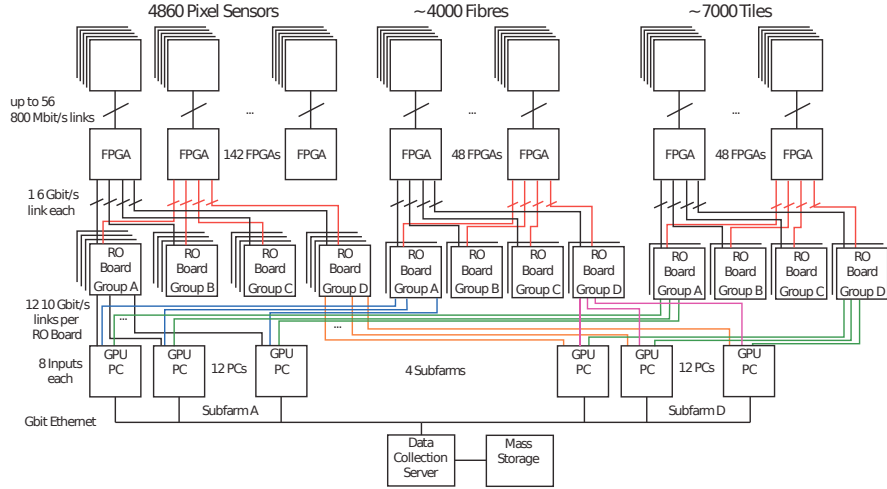
Figure 3: Readout scheme of the Mu3e detector design. [6]

3   Multiple scattering fit

Tracks are reconstructed based on the hits in the pixel detectors using a 3D tracking algorithm for multiple scattering dominated resolution [7, 8]. Triplets of subsequent hits in three layers are selected and multiple scattering is assumed at the middle hit of the triplet. In the momentum

region of interest the position resolution of the pixel detector ($\sigma_{\mathrm{pixel}} = 80/\sqrt{12}$ μm) is small compared to the effects of multiple scattering.

Figure 4 shows one triplet with the azimuthal scattering angle Φ_{MS} in the x-y plane on the left and the polar scattering angle Θ_{MS} in the s-z plane on the right, where s is the 3D path length. The combined $\chi^2 = \frac{\phi_{MS}^2}{\sigma_{MS}^2} + \frac{\theta_{MS}^2}{\sigma_{MS}^2}$ is minimized during the non-iterative fitting procedure. The variances of the scattering angles are obtained from multiple scattering theory. Several triplets grouped together form one track.
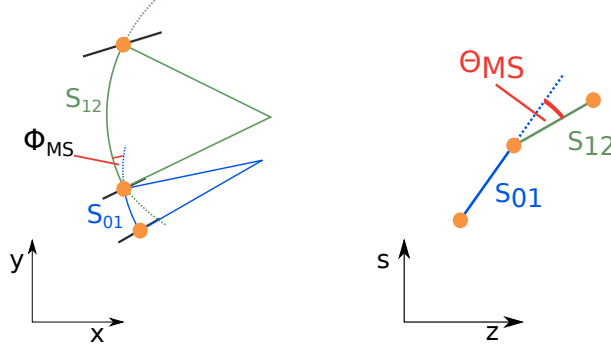
Figure 4: Sequence of three hits forming a triplet in the x-y plane on the left, and the s-z plane on the right with respective scattering angles.

4   Fit implementation on GPU

For online event selection, a basic version of the multiple scattering fit runs on the GPUs of the filter farm PCs. It has been implemented on Nvidia's GeForce GTX 680 using the CUDA environment. Triplets are constructed from hits in the first three detector layers. The number of possible hit combinations to form a triplet is proportional to $n_1 \times n_2 \times n_3$ where n_i is the number of hits in layer i. The main steps of the processing are:

- Sorting the hit array with respect to the z-coordinate

- Geometric filtering on ordered hits

- Triplet fitting and selection

Notice that in the final readout mechanism, the hit arrays will be sorted by the FPGAs in the PCs and geometrical selection cuts will be applied. The preselected arrays will then be copied via direct memory access to the GPUs to perform the fitting step. Currently the CPU sorts the hit arrays with respect to the z-coordinate, then the sorted hit arrays are copied to the GPU global memory.

A preliminary kernel implementation gathers the hit filtering and triplet fitting. The selection cuts require proximity in the x-y plane and in z for pairs of hits in layers one and two, and two and three respectively. If these cuts were passed, the fitting procedure is applied and hits of triplets are saved in global memory with an atomic function given a successful fit completion and a certain χ^2.

For the first kernel implementation, a 2D-grid is set up with $n_1 \times n_2$ grid dimensions, each kernel loops over the n_3 hits in layer three. Since the geometrical selection cuts and fit completion cuts introduce branch divergence, 87 % of the threads are idle in this configuration.

Consequently, an alternative implementation with two separate kernels was tested. Within the *geometric kernel*, only the geometry cuts are applied and triplets of hits passing these cuts are saved into an array in global memory. The grid dimensions for this kernel are the same as described for the first implementation. Then, the *fit kernel*, which fits and selects hit triplets, is launched on the number of selected triplets using a 1D-grid associated with 128-block size. Consequently, one fit is performed per thread without any divergence. In addition, since the block dimension is now a multiple of the warp size (32), no inherently idle threads are launched in the grid.

In terms of run time, there is no significant improvement with the two kernel version compared to one kernel. However, only tasks for which the GPU is optimized are now performed in the fitting kernel, so this is one step towards the final readout and selection chain of the filter farm.

5 Performance
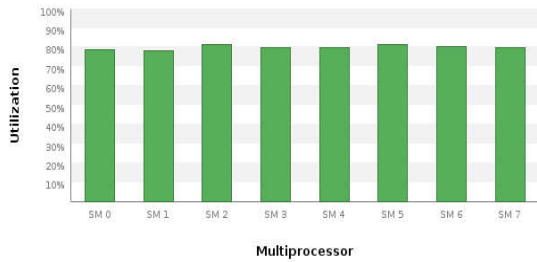
Figure 5: Compute utilization of the streaming multiprocessors on the GTX 680 for the fitting kernel of the two kernel fit implementation.

Next we will focus on the triplet fitting performance, since it will be the processing step deployed on GPUs.
Using Nvidia's Visual Profiler tool, the performance of the different versions was studied. The compute utilization of the streaming multiprocessors averages around 80 % for the fitting kernel of the two kernel version, as shown in Figure 5.

The block size has been chosen to optimize the number of warps per streaming multiprocessor: 128 threads per block take advantage of all 64 warps in a streaming multiprocessor (see Figure 6 on the left). Since the fitting kernel requires 29 registers per thread, the kernel implementation can make optimal use of these 64 warps per streaming multiprocessor (see Figure 6 on the right). Therefore, the compute efficiency is not limited by the block size or register count.

Currently, $1.4 \cdot 10^{10}$ triplets/s are processed. This measurement is based on the wall time of both CPU and GPU, so that all sorting and selection times are included. Most of the time is spent on the first kernel applying the selection cuts. Therefore, further improvement is expected when the pre-selection is outsourced to the FPGAs on the readout boards. Similarly, the ratio of copying data from CPU to GPU compared to computing time (40 %) will improve once more selection criteria are applied before transferring the data to the PC.

Figure 6: Warps per Streaming Multiprocessor (SM) as a function of the block size (left) and the register count (right).

6 Summary and outlook

To achieve a sensitivity of 10^{16} in the measurement of the $\mu \to eee$ branching ratio, high rates on the order of $10^9 \mu/s$ are required, resulting in a data rate on the order of 100 GB/s in the detector.

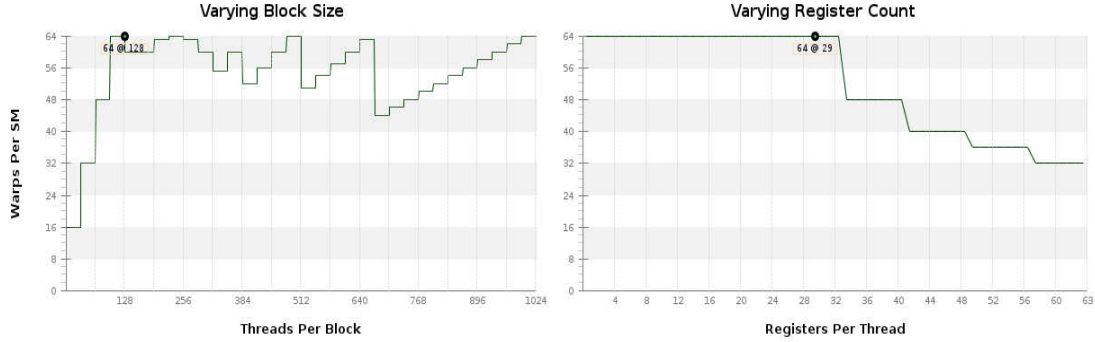The implementation of the triplet fit on the GTX 680 can currently process 1.4×10^{10} triplets/s. For a muon rate of 10^8, about 10^{12} hit combinations are expected per second in the first three layers, therefore requiring 10 - 100 GPU computers in the filter farm. Faster filtering is required for the higher rate of $10^9 \mu/s$. This can be achieved by sorting the data on FPGAs and further improving the performance of the fit, and as a result, reduce by a factor of 1000 the data rate. In addition, a new vertex fit [9] will be implemented on the GPU in order to apply the selection criteria for a signal of 3 tracks originating from one single vertex.

References

[1] A. Blondel et al. Research Proposal for an Experiment to Search for the Decay $\mu \to eee$. *ArXiv e-prints*, January 2013.

[2] W. Bertl et al. Search for the decay $\mu^+ \to e^+e^+e^-$. *Nucl. P*, B 260(1):1 – 31, 1985.

[3] N. Berger et al. A Tracker for the Mu3e Experiment based on High-Voltage Monolithic Active Pixel Sensors. *Nucl. Instr. Meth. A*, 732:61–65, 2013.

[4] I. Perić. A novel monolithic pixelated particle detector implemented in high-voltage CMOS technology. *Nucl.Instrum.Meth.*, A582:876, 2007.

[5] I. Perić et al. High-voltage pixel detectors in commercial CMOS technologies for ATLAS, CLIC and Mu3e experiments. *Nucl.Instrum.Meth.*, A731:131–136, 2013.

[6] S. Bachmann et al. The proposed trigger-less tbit/s readout for the mu3e experiment. *Journal of Instrumentation*, 9(01):C01011, 2014.

[7] A. Schöning et al. A multiple scattering triplet fit for pixel detectors. *publication in preparation*.

[8] M. Kiehn. Track fitting with broken lines for the mu3e experiment. Diploma thesis, Heidelberg University, 2012.

[9] S. Schenk. A Vertex Fit for Low Momentum Particles in a Solenoidal Magnetic Field with Multiple Scattering. Master's thesis, Heidelberg University, 2013.

INTEL HPC portfolio

E. Politano[1]

[1] INTEL®

Contribution not received.

GPUs for the realtime low-level trigger of the NA62 experiment at CERN

R. Ammendola[4], M. Bauce[3,7], A. Biagioni[3], S. Chiozzi[1,5], A. Cotta Ramusino[1,5], R. Fantechi[2], M. Fiorini[1,5,*], A. Gianoli[1,5], E. Graverini[2,6], G. Lamanna[2,8], A. Lonardo[3], A. Messina[3,7], I. Neri[1,5], F. Pantaleo[2,6], P. S. Paolucci[3], R. Piandani[2,6], L. Pontisso[2], F. Simula[3], M. Sozzi[2,6], P. Vicini[3]

[1]INFN Sezione di Ferrara, Via Saragat 1, 44122 Ferrara, Italy
[2]INFN Sezione di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
[3]INFN Sezione di Roma"La Sapienza", P.le A. Moro 2, 00185 Roma, Italy
[4]INFN Sezione di Roma "Tor Vergata", Via della Ricerca Scientifica 1, 00133 Roma, Italy
[5]University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy
[6]University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
[7]University of Rome "La Sapienza", P.le A.Moro 2, 00185 Roma, Italy
[8]INFN Sezione di Frascati, Via E.Fermi 40, 00044 Frascati (Roma), Italy
* Corresponding author. E-mail: fiorini@fe.infn.it

A pilot project for the use of GPUs (Graphics processing units) in online triggering applications for high energy physics experiments (HEP) is presented. GPUs offer a highly parallel architecture and the fact that most of the chip resources are devoted to computation. Moreover, they allow to achieve a large computing power using a limited amount of space and power. The application of online parallel computing on GPUs is shown for the synchronous low level trigger of NA62 experiment at CERN. Direct GPU communication using a FPGA-based board has been exploited to reduce the data transmission latency and results on a first field test at CERN will be highlighted. This work is part of a wider project named GAP (GPU application project), intended to study the use of GPUs in real-time applications in both HEP and medical imaging.

1 Introduction

The trigger system plays a fundamental role in any HEP experiment since it must decide whether a particular event observed in a detector should be recorded or not, based on limited information. Every experiment features a limited amount of DAQ bandwidth and disk space for data storage, the use of real-time selections is crucial to make the experiment affordable maintaining at the same time its discovery potential. Online selection of important events can be performed by arranging the trigger system in a cascaded set of computational levels. The low-level (hereinafter referred to as L0) is usually realized in hardware, often based on custom electronics devoted to the experiment and normally developed using programmable logic (FPGA) that offer flexibility and possibility of reconfiguration. The upper trigger levels (L1 and L2) are implemented in software on a commodity PC farm for further reconstruction

and event building. In the baseline implementation, the FPGAs on the readout boards compute simple trigger primitives on the fly, such as hit multiplicities and rough hit patterns, which are then timestamped and sent to a central trigger processor for matching and trigger decision.

This paper presents the idea of using GPUs for low-level triggering in HEP experiments. In fact, GPUs provide a huge computing power on a single device, thus allowing to take complex decisions with a significantly high speed. In particular, in the standard multi-level trigger architecture, GPUs can be easily exploited in the higher software levels, where the number of computing farm nodes can be reduced and the capability of the processing system can be improved without increasing the scale of the system itself. As an example, GPUs are currently under study in the software trigger level of ATLAS experiment at CERN [1] and are implemented with encouranging results in tracking of Pb-Pb Events in the ALICE experiment [2].

Low-level triggers can also take advantage from the usage of GPUs, but a careful assessment of their online performance is required especially in terms of computing power and latency. In fact, low level trigger systems are designed to perform very rough selection based on a sub-set of the available information, typically in a pipelined structure housed in custom electronics, in order to bring to a manageable level the high data rate that would otherwise reach the software stages behind them. Due to small buffers size in the read-out electronics, such systems usually require very low latency. A low total processing latency is usually not crucial in the applications for which GPUs have been originally developed. On the contrary, for a GPU-base trigger system, data transfer latency to the GPU and its stability in time become a very important issue.

In this paper we present recent results of the ongoing R&D on the use of GPUs in the lowest trigger level of the NA62 Experiment at CERN, within the framework of the GAP project [3]. Next section describes the idea of the adoption of GPUs in the NA62 L0 trigger while in the other subsection we address more specifically the latency control problem, and we show the results on the solution that we are currently developing. This is based on a custom "smart" NIC that allows copying data directly into the GPU.

2   GPUs in the low-level trigger of the NA62 experiment

The NA62 experiment at CERN aims at measuring the branching ratio of the ultra-rare decay of the charged kaon into a pion and a neutrino-antineutrino pair. The goal is to collect ~ 100 events with a 10:1 signal to background ratio, using a high-energy (75 GeV/c) unseparated hadron beam decaying in flight [4, 5]. In order to manage the 25 GB/s raw data stream due to a ~ 10 MHz rate of particle decays illuminating the detectors, the NA62 trigger consists of three levels as illustrated in the previous section: the L0 is based on FPGA boards which perform detector data readout [6], while the next two levels are developed on PCs, thus implemented in software. L0 must handle an input event rate of the order of 10 MHz and apply a rejection factor of around 10, in order to allow a maximum input rate of 1 MHz to the L1 trigger. The latter, together with the L2 trigger, must reduce the rate to about 10 kHz in order to permit permanent data storage for later offline analysis. The maximum total latency allowed by the NA62 experiment for the L0 trigger is 1 ms.

A pilot project within NA62 is investigating the possibility of using a GPU system as L0 trigger processor (GL0TP), exploiting the GPU computing power to process unfiltered data from the readout in order to implement more selective trigger algorithms. In the standard L0 implementation, trigger primitives contributing to the final trigger decision are computed on
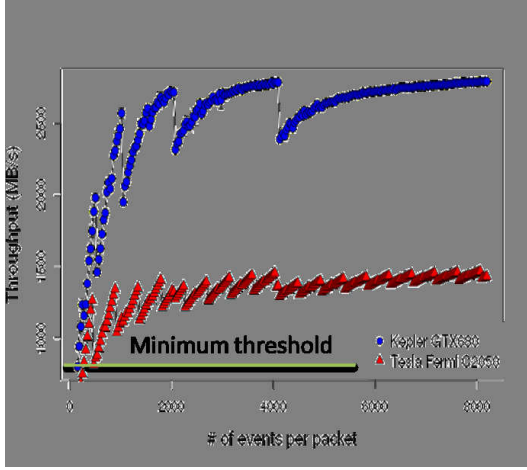
Figure 1: Throughput as a function of the number of events for last generation GPUs.
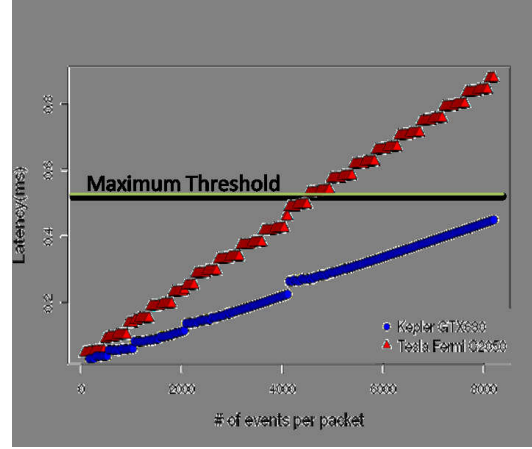

Figure 2: Total latency (including data transfer and computing). Here the maximum threshold does not take into account the data transfer from readout, but it only estimates the total latency contribution within the GPU.

the readout board FPGAs, and are mostly based on event hit patterns. The use of GPUs in this level would allow building more complex physics-related trigger primitives, such as energy or direction of the final state particles in the detectors, therefore leading to a net improvement of trigger conditions and data handling.

In particular, the reconstruction through GPUs of the ring-shaped hit patterns within the NA62 Ring Imaging Cherenkov (RICH) detector represents the first study case on the use of GPUs at low-level trigger in the GAP project. The proposed GL0TP will operate in parasitic mode with respect to the main L0 trigger processor by processing data coming only from the RICH detector. Such detector, described in [7], provides a measurement of the velocity and direction of the charged particles crossing its volume above the Cherenkov threshold. It can therefore contribute to the computation of other physical quantities, such as the decay vertex of the K^+ and the missing mass. On the basis of such information, highly selective trigger algorithms can be implemented for several interesting K^+ decay modes.

As highlighted in [8], several ring reconstruction algorithms have been studied in order to assess the best GPU-based implementation. In particular, the one based on a simple coordinate transformation of the hits which reduces the problem to a least square procedure was found to be the best ring-fitting algorithm in terms of computing throughput. This algorithm was developed and tested on different GPUs, such as the NVIDIA Tesla C1060, Tesla C2050 and GeForce GTX680. The computing performance of the C2050 and GTX680 proved to be a factor 4 and 8 higher than that of the C1060, respectively.

Figure 1 shows the computing throughput for these devices as a function of the number of events processed in one batch. The effective computing power is seen to increase with the number of events to be processed concurrently. The horizontal line shows the minimum throughput

requirement for an online trigger based on the RICH detector of the NA62 experiment.

Figure 2 points out the total computing latency that includes data transfer times to and from the GPU and the kernel execution time. Here NVIDIA Tesla C2050 and GeForce GTX680 devices have been used for the measurement. The significant reduction of the latency for the newer GTX680 GPU is due to the faster data transfer allowed by the 3rd generation PCIExpress bus. As can be seen, the maximum threshold (green horizontal line) seems to be attainable when a reasonable number of events is processed in one batch.

In a standard GPU computing approach, data from the detector reach the Network Interface Card (NIC) which copies them periodically on a dedicated area in the PC RAM, from where they are then copied to the user space memory where applications can process them. Here a sufficient data load of buffered events is usually prepared for the following stages, and they are copied to GPU memory through the PCI express bus. The host (the PC on which the GPU card is plugged) has the role of starting the GPU kernel, which operates on the data. Computation results can then be sent back to the host for further processing or distribution to the detectors, to ultimately trigger the read-out of the complete data event. In this system, the most important contribution to the total latency is due to the data transfer latency from the NIC to the GPU memory. Thus, in order to reduce the maximum total latency, an approach will be described in detail in the following, i.e., the use of a direct data transfer protocol from a custom FPGA-based NIC to the GPU.

2.1 NaNet

NaNet is a modular design of a low-latency NIC with GPUdirect capability developed at INFN Rome division, that is being integrated in the GPU-based low level trigger of the NA62 RICH detector [9, 10]. Its design comes from the APEnet+ PCIe Gen 2 x8 3D NIC [11] and the board supports a configurable number of different physical I/O links (see Figure 3). The Distributed Network Processor (DNP) is the APEnet+ core logic, behaving as an off-loading engine for the computing node in performing inter-node communications [12]. NaNet is able to exploit the GPUDirect peer-to-peer (P2P) capabilities of NVIDIA Fermi/Kepler GPUs enabling a host PC to directly inject into its memory an UDP input data stream from the detector front-end, with rates compatible with the low latency real-time requirements of the trigger system.

To measure the data transmission latency a readout board (TEL62) has been employed to send input data. In particular, data communication between the TEL62 readout boards and the L0 trigger processor (L0TP) happens over multiple GbE links using UDP streams. The main requisite for the communication system comes from the request for <1 ms and deterministic response latency of the L0TP, thus communication latency and its fluctuations are to be kept under control. The requisite on bandwidth is 400÷700 MB/s, depending on the final choice of the primitives data protocol which in turn depends on the amount of preprocessing to be actually implemented in the TEL62 FPGA. This means that, to extract primitives data from the readout board towards the L0TP in the final system, 4÷6 GbE links will be used.

NaNet latency was measured sending UDP packets from one of the host GbE ports to the NaNet GbE interface: using the x86 TSC register as a common reference time, a single-process test could measure the time difference between the moment before the first UDP packet of a bunch (needed to fill the receive buffer) is piped out through the host GbE port and when the signal of a filled receive buffer reaches the application. Within this measurement setup ("system loopback"), the latency of the `send` process is also taken into account.

Measurements in Figure 5 were thus taken; UDP packets with a payload size of 1168 Bytes
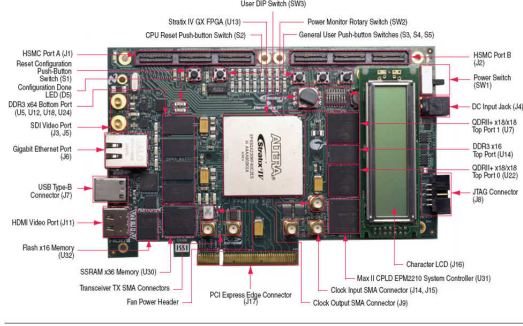
Figure 3: NaNet as a FPGA-based NIC implemented using an Altera development board (Stratix IVGX230 FPGA).
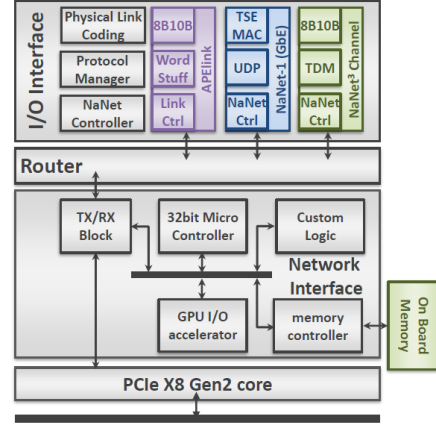
Figure 4: NaNet architecture schematic.

(16 events) were sent to a GPU memory receiving buffer of size variable between 1 and 64 times the UDP packet payload.

2.1.1 Synchronization of NaNet with the Timing Trigger and Control system at CERN

The Timing Trigger and Control (TTC) system distributes the system clock for the NA62 experiment as well as the first level triggers and synchronization commands, which are all distributed on a single optical fibre. In particular, a mezzanine card (TTCrq) [13], developed by the CERN microelectronics group, acts as an interface between the TTC system for NA62 detectors and its receiving end users. The card delivers the clock together with control and synchronization information to the front-end electronics controllers in the detector.

Synchronization of NaNet with the TTC system through the TTCrq, has been tested by firstly realizing an interface board between NaNet and the TTCrq. This has been produced at Physics Department at University of Ferrara (Italy) and connected to the High Speed Mezzanine Card (HSMC) port B of the FPGA board. The whole TTCrq-interface-NaNet system (see Figure 6) proved to correctly receive all the signals necessary to synchronise the detectors, i.e. clock, event counter reset and bunch counter reset (the last ones encoding start-of-burst and end-of-burst commands).

3   Conclusions

The GAP Project aims at studying the application of GPUs in real-time HEP trigger systems and in medical imaging. In this paper the application to a low-level trigger system for the NA62 experiment at CERN has been described: the critical point to be pursued is the reduction of
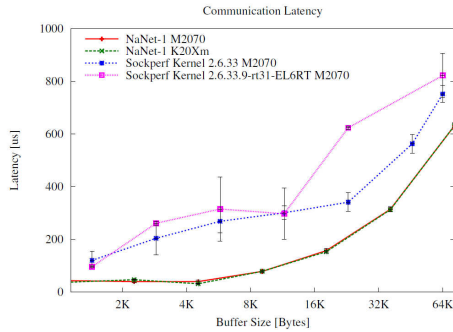
Figure 5: A comparison between NaNet communication latency and standard GbE interfaces results.
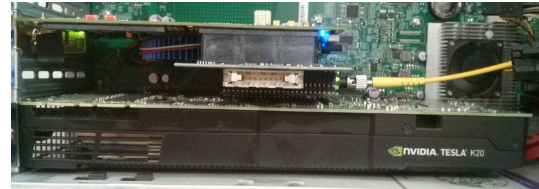
Figure 6: The TTCrq-interface-NaNet system together with a NVIDIA Tesla K20 GPU.

the contributions to the total latency due to data transfer from the detectors to the GPU. An approach based on a FPGA board to establish a peer-to-peer connection with the GPU is being developed. The reconstruction of photon rings in the RICH detector has been considered as first case of study on the use of GPUs at L0 in the NA62 experiment. Preliminary results show that current GPUs are suitable for sustaining the event rates, while at the same time minimizing the latency to an acceptable level.

Acknowledgment

References

[1] P.J. Clark, C. Jones. D Emeilyanov, M. Rovatsou, A. Washbrook, and the ATLAS collaboration, "Algorithm Acceleration from GPGPUs for the ATLAS Upgrade" Journal of Physics: Conference Series 331 (2011) 022031.

[2] D. Rohr, S. Gorbunov, A. Szostak, M. Kretz, T. Kollegger, T. Breitner and Torsten Alt, "ALICE HLT TPC Tracking of Pb-Pb Events on GPUs", Journal of Physics: Conference Series 396 (2012) 012044.

[3] *http://web2.infn.it/gap*

[4] M. Fiorini [NA62 Collaboration], "The NA62 experiment at CERN", PoS HQL **2012** (2012) 016.

[5] *http://cern.ch/NA62/*

[6] B. Angelucci, E. Pedreschi, M. Sozzi and F. Spinella, "TEL62: an integrated trigger and data acquisition board" Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2011 IEEE.

[7] B. Angelucci, G. Anzivino, C. Avanzini, C. Biino, A. Bizzeti, F. Bucci, A. Cassese and P. Cenci *et al.*, "Pion-muon separation with a RICH prototype for the NA62 experiment," Nucl. Instrum. Meth. A **621** (2010) 205.

[8] R. Ammendola, A. Biagioni, L. Deri, M. Fiorini, O. Frezza, G. Lamanna, F. Lo Cicero, A. Lonardo, A. Messina, M. Sozzi, F. Pantaleo, P.S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto and P. Vicini, "GPUs for Real Time processing in HEP trigger systems" Journal of Physics: Conference Series 523 (2014) 012007 doi: 10.1088/1742-6596/523/1/012007 and references therein.

[9] R. Ammendola, A. Biagioni, O. Frezza, G. Lamanna, A. Lonardo, F. Lo Cicero, P. S. Paolucci, F. Pantaleo, D. Rossetti, F. Simula, M. Sozzi, L. Tosoratto and P. Vicini, "NaNet: a flexible and configurable low-latency NIC for real-time trigger systems based on GPUs" JINST **9** (2014) C02023, doi:10.1088/1748-0221/9/02/C02023.

[10] R. Ammendola, A. Biagioni, R. Fantechi, O. Frezza, G. Lamanna, F. L. Cicero, A. Lonardo, P. S. Paolucci, F. Pantaleo, R. Piandani, L. Pontisso, D. Rossetti, F. Simula, M. Sozzi, L. Tosoratto and P. Vicini, "NaNet:a low-latency NIC enabling GPU-based, real-time low level trigger systems", Journal of Physics: Conference Series, 513, (2014) 012018.

[11] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti and F. Simula, L. Tosoratto and P. Vicini, "APEnet+: A 3D Torus network optimized for GPU-based HPC systems", J. Phys. Conf. Ser. **396** (2012) 042059.

[12] A. Biagioni, F. L. Cicero, A. Lonardo, P. S. Paolucci, M. Perra, D. Rossetti, C. Sidore, F. Simula, L. Tosoratto and P. Vicini, "The Distributed Network Processor: a novel off-chip and on-chip interconnection network architecture", arXiv:1203.1536.

[13] *http://proj-qpll.web.cern.ch/proj-qpll/ttcrq.htm*

A FPGA-based Network Interface Card with GPUDirect enabling real-time GPU computing in HEP experiments.

Alessandro Lonardo[a]*, Fabrizio Ameli[a], Roberto Ammendola[b], Andrea Biagioni[a], Angelo Cotta Ramusino[c], Massimiliano Fiorini[c], Ottorino Frezza[a], Gianluca Lamanna[de], Francesca Lo Cicero[a], Michele Martinelli[a], Ilaria Neri[c], Pier Stanislao Paolucci[a], Elena Pastorelli[a], Luca Pontisso[f], Davide Rossetti[g], Francesco Simeone[a], Francesco Simula[a], Marco Sozzi[fe], Laura Tosoratto[a], Piero Vicini[a]

[a]INFN Sezione di Roma - Sapienza, P.le Aldo Moro, 2 - 00185 Roma, Italy
[b]INFN Sezione di Roma - Tor Vergata, Via della Ricerca Scientifica, 1 - 00133 Roma, Italy
[c]Università degli Studi di Ferrara and INFN Sezione di Ferrara, Polo Scientifico e Tecnologico, Via Saragat 1 - 44122 Ferrara, Italy
[d]INFN Laboratori Nazionali di Frascati, Via E. Fermi,40 - 00044 Frascati (Roma), Italy
[e]CERN, CH-1211 Geneva 23, Switzerland
[f]INFN Sezione di Pisa, Via F. Buonarroti 2 - 56127 Pisa, Italy
[g]NVIDIA Corp, 2701 San Tomas Expressway, Santa Clara, CA 95050

The capability of processing high bandwidth data streams in real-time is a computational requirement common to many High Energy Physics experiments. Keeping the latency of the data transport tasks under control is essential in order to meet this requirement. We present NaNet, a FPGA-based PCIe Network Interface Card design featuring Remote Direct Memory Access towards CPU and GPU memories plus a transport protocol offload module characterized by cycle-accurate upper-bound handling. The combination of these two features allows to relieve almost entirely the OS and the application from data tranfer management, minimizing the unavoidable jitter effects associated to OS process scheduling. The design currently supports one GbE (1000Base-T) and three custom 34 Gbps APElink I/O channels, but four-channels 10GbE (10Base-R) and 2.5 Gbps deterministic latency KM3link versions are being implemented. Two use cases of NaNet will be discussed: the GPU-based low level trigger for the RICH detector in the NA62 experiment and the on-/off-shore data acquisition for the KM3Net-IT underwater neutrino telescope.

1 Introduction

In many fields of Experimental Physics ranging from Radio Astronomy to High Energy Physics, the GPU adoption effort that in High Performance Computing brings such outstanding results

---

*Corresponding author. E-mail: alessandro.lonardo@roma1.infn.it

is hampered by the real-time constraints that the processing of data streams outflowing from experimental apparatuses is often subject to. GPUs processing latency is mostly stable once data has landed in their own internal memories; a data transport mechanism with deterministic or at least bound latency is then crucial in building a GPGPU system honouring these constraints.

Our NaNet design is one such mechanism, reusing several IPs developed for the APEnet+ 3D torus NIC [1] targeted at HPC hybrid clusters; its real-time features were achieved by adding dedicated modules and support of multiple link technologies, both standard and custom. The resulting FPGA-based, low-latency PCIe NIC is highly configurable and modular, with RDMA and GPUDirect capabilities. It has been employed in widely varying configurations and physical device implementations in two different High Energy Physics experiments: the NA62 experiment at CERN [2] and the KM3NeT-IT underwater neutrino telescope [3].

2 NaNet design overview

NaNet is a low-latency PCIe NIC supporting standard — GbE (1000BASE-T) and 10GbE (10Base-R) — and custom — 34 Gbps APElink [4] and 2.5 Gbps deterministic latency optical KM3link [5] — network links. NaNet bridges the gap between real-time and GPGPU heterogeneous computing by inheriting the GPUDirect P2P/RDMA capability from its HPC-dedicated sibling — the APEnet+ 3D torus NIC — and enhancing it with real-time-enabling features, *e.g.* a network stack protocol offload engine for stable communication latency. NaNet design is partitioned into 4 main modules: *I/O Interface*, *Router*, *Network Interface* and *PCIe Core* (see Fig. 1). The *I/O Interface* module performs a 4-stages processing on the data stream: following the OSI Model, the Phys-

Figure 1: NaNet architecture schematic.

ical Link Coding stage implements the channel physical layer (*e.g.* 1000BASE-T) while the Protocol Manager stage handles, depending on the kind of channel, data/network/transport layers (*e.g.* Time Division Multiplexing or UDP); the Data Processing stage implements application-dependent reshuffling on data streams (*e.g.* performing de/compression) while the APEnet Protocol Encoder performs protocol adaptation, encapsulating inbound payload data in APElink packet protocol — used in the inner NaNet logic — and decapsulating outbound APElink packets before re-encapsulating their payload into output channel transport protocol (*e.g.* UDP). The *Router* module implements a parametric full crossbar switch responsible for data routing, sustaining multiple data flows @2.8 GB/s.

The *Network Interface* block acts on the trasmitting side by forwarding PCIe-originated data to the Router ports and on the receiving side by providing support for RDMA in communications involving both the host and the GPU (via a dedicated *GPU I/O Accelerator* module). A Nios II μcontroller handles configuration and runtime operations.

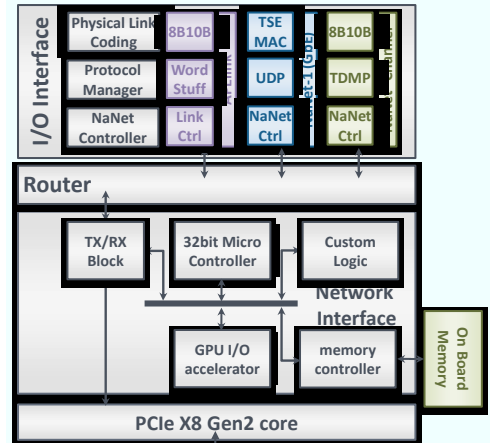Finally, the *PCIe Core* module is built upon a powerful commercial core from PLDA that

sports a simplified but efficient backend interface and multiple DMA engines.

This general architecture has been specialized up to now into three configurations, namely NaNet-1, NaNet[3] and NaNet-10, to match the requirements of different experimental setups: **NaNet-1** is implemented on the Altera Stratix IV FPGA Dev Kit; it sports a PCIe Gen2 x8 host interface, a GbE channel and three optional 34 Gbps APElink ones; **NaNet**[3] is implemented on the Terasic DE5-net Stratix V FPGA Dev Kit; it supports a PCIe Gen2 x8 host interface while the four SFP+ cages of the board are used for the KM3link channels which are 2.5 Gbps optical links with deterministic latency;**NaNet-10** is another implementation on the Terasic DE5-net board; the PCIe Gen2 x8 host interface is the same as NaNet[3] while the SFP+ ports support four 10GbE channels.

3 NaNet-1: a NIC for NA62 GPU-based low-level trigger

The NA62 experiment at CERN aims at measuring the Branching Ratio of the ultra-rare decay of the charged Kaon into a pion and a neutrino-antineutrino pair. The NA62 goal is to collect ~ 100 events with a 10:1 signal to background ratio, using a novel technique with a high-energy (75 GeV) unseparated hadron beam decaying in flight. In order to manage the 25 GB/s raw data stream due to a ~ 10 MHz rate of particle decays illuminating the detectors, the trigger system is designed as a set of three cascaded levels that decrease this rate by three orders of magnitude [6]. The low-level trigger (L0) is a synchronous real-time system implemented in hardware by means of FPGAs on the readout boards and reduces the stream bandwidth tenfold: whether the data on the readout board buffers is to be passed on to the higher levels has to be decided within 1 ms to avoid data loss. The upper trigger levels (L1 and L2) are implemented in software on a commodity PC farm for further reconstruction and event building. A pilot project within NA62 is investigating the possibility of using a GPGPU system as L0 trigger processor (GL0TP), exploiting the GPU computing power to process unfiltered data from the readout in order to implement more selective trigger algorithms. In order to satisfy the real-time requirements of the system, a fast, reliable and time-deterministic dedicated link must be employed. NaNet-1 has been designed and developed with the motivation of building a fully functional and network-integrated GL0TP prototype, albeit with a limited bandwidth with respect to the experiment requirements, in order to demonstrate as soon as possible the suitability of the approach and eventually evolve the design to incorporate better I/O channels. NaNet-1 real-time characterization has been carefully assessed on dedicated testbeds [7, 8]. The results of this activity have driven the latest developments towards a lower latency design. Allocation of time-consuming RDMA related tasks has been moved from the Nios II μcontroller to dedicated logic blocks. The Virtual Address Generator (VAG) included in the NaNet Controller module is in charge of generating memory addresses of the receiving buffers for incoming data, while a Translation Lookaside Buffer (TLB) module performs fast virtual-to-physical address translations.

A bandwidth-downscaled GL0TP setup was realized in a loopback configuration by a host system simulating TEL62 UDP traffic through one of its GbE ports towards a NaNet-1 NIC streaming the incoming data into a GPU memory circular list of receive buffers; once landed, buffers are consumed by a CUDA Kernel implementing the pattern matching algorithm. Communication and kernel processing tasks were serialized in order to perform the measure; in Fig. 2 (left) the results for the K20Xm system are shown. During normal operation, this serialization constraint can be relaxed, and kernel processing task overlaps with data communication. Actually this is what was done to measure system throughput, with the results in Fig. 2 (right).
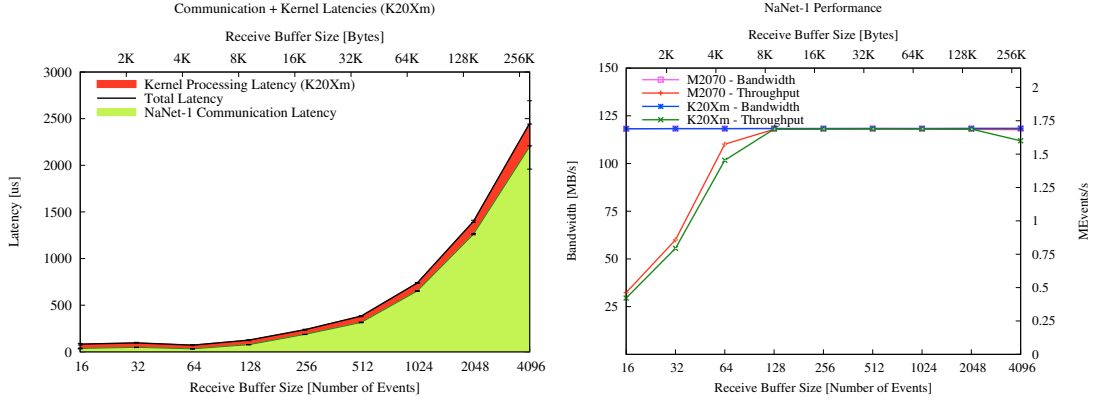
Figure 2: (left) NaNet-1 latency over GbE towards GPU memory for varying datagram payload sizes (nVIDIA Fermi K20Xm); (right) NaNet-1 GbE bandwidth towards GPU memory at varying datagram payload sizes.

We see that using GPU receive buffer sizes ranging from 128 to 1024 events allows the system to stay within the 1 ms time budget while keeping a ~ 1.7 MEvents/s throughput.

Besides optimizing performances, we undertook several developments to cope with the NaNet-1 integration within the NA62 experiment. A Timing Trigger and Control (TTC) HSMC daughtercard was designed to provide NaNet-1 with the capability of receiving either trigger and 40 MHz clock streams distributed from the experiment TTC system via optical cable: this common reference clock was used to perform latency measurements discussed above. A decompressor stage was added in the I/O interface to reformat events data in a GPU-friendly fashion on the fly. Finally, a timeout was implemented in the NaNet Controller module that triggers the DMA of received data towards CPU/GPU memory on a configurable deadline rather than on the filling of a buffer. The prototype to be deployed at the NA62 experiment site will integrate into NaNet-1 a TTC HSMC daughtercard and a nVIDIA Kepler K20 GPU.

4 NaNet^3: KM3NeT-IT neutrino telescope on-shore board

KM3NeT-IT is an underwater experimental apparatus for the detection of high energy neutrinos in the TeV÷PeV range by means of the Čerenkov technique. The KM3NeT-IT detection unit consists of 14 floors vertically spaced 20 meters apart, with $\sim$8 m long floor arms bearing 6 glass spheres each called Optical Modules (OM); each OM contains one 10 inches photomultipliers (PMT) and front-end electronics that digitizes, formats and emits the PMT signal. All data produced by OMs and auxiliary floor instrumentation is collected by an off-shore electronic board called *Floor Control Module* (FCM) contained in a vessel at the floor centre; the FCM manages the communication between the on-shore laboratory and the underwater devices, also distributing the timing information and signals. Due to the distance between apparatus and shoreland the connecting medium is optical fiber.

The spatially distributed DAQ architecture requires a common clock distributed all over the system to correlate signals, with respect to a fixed reference, coming from different parts of the apparatus. For this purpose data acquisition and transport electronics labels each signal with a "time stamp", *i.e.* hit arrival time. Time resolution is also fundamental for the track reconstruction accuracy. These constraints pushed the choice of a synchronous link protocol

embedding clock and data with deterministic latency; Floors are independent from each other; each is connected to the on-shore laboratory by a bidirectional virtual point-to-point link.

A single floor data stream delivered to shore has a rate of ∼300 Mbps, while shore-to-underwater communication data rate is much lower, consisting only of slow-control data for the apparatus. The small data rate per FCM compared with state-of-the-art technologies led us to designing NaNet[3], an on-shore readout board able to manage multiple FCM data channels.

This is a NaNet customization for the KM3NeT-IT experiment, with added support for a synchronous link protocol with deterministic latency at physical level and for a Time Division Multiplexing protocol at data level (see Fig. 1).
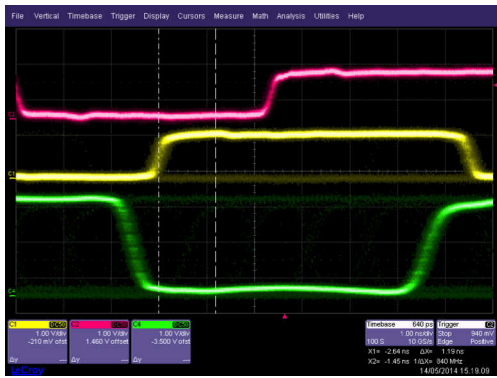
Figure 3: Deterministic latency of NaNet[3] SerDes: phase alignment of the transmitting (upper) and receiving (central) clocks.

The first design stage for NaNet[3] was implemented on the Terasic DE5-net board, which is based on an Altera Stratix-V GX FPGA with four SFP+ channels and a PCIe x8 edge connector. To match time resolution constraint, time delta between wavefronts of three clocks must have ns precision: the first clock is an on-shore reference one (coming from a GPS and redistributed), used for the optical link transmission from NaNet[3] towards the underwater FCM; the second clock is recovered from the incoming data stream by a Clock and Data Recovery (CDR) module at the receiving end of the FCM which uses it for sending its data payload from the apparatus back on-shore; a third clock is again recovered by NaNet[3] decoding the payload at the end of the loop. The deterministic latency constraint must be enforced on the Stratix device as the FCM does on paths both ways to allow correct time stamping of events on the PMT. The established link is synchronous, *i.e.* clock rate is equal for both devices with fixed phase displacement. We developed a test setup to explore latency capabilities of a complete link chain leveraging on the fixed latency native mode of the Altera transceivers, on the NaNet[3] board, and on the hardware fixed latency implementation for a Xilinx device on the FCM board [9]. The external GPS-equivalent clock has been input to the NaNet[3] to clock the transmitting side of the device. A sequence of dummy parallel data are serialised, 8b/10b-encoded and transmitted, together with the embedded serial clock, at 800 Mbps along the fiber towards the receiver side of the FCM system. The FCM system recovers the received clock and transmits the received data and recovered clock back to the NaNet[3] board. The receive side of NaNet[3] deserializes data and produces the received clock.

Testing the fixed latency features of the SerDes hardware implementation is straightforward when taking into account that on every initialisation sequence, *e.g.* for a hardware reset or at SerDes hardware powerup, we should be able to measure the same phase shift between transmitted and received clock, equal to the fixed number of serial clock cycles shift used to correctly align the deserialised data stream. Fig. 3 shows a scope acquisition in infinite persistence mode sampled over 12 h issuing every 10 s a new *reset and align*. The NaNet[3] transmitter parallel clock (the upper line) maintains exactly the same phase difference with the receiver parallel clock (the central line) and with the FCM recovered clock (the lower line).

5   Conclusions and future work

Our NaNet design proved to be an efficient real-time communication channel between the NA62 RICH readout system and GPU-based L0 trigger processor over a single GbE link. With four 10 GbE ports, the currently under development NaNet-10 board will exceed the bandwidth requirements for the NA62 RICH, enabling integration of other detectors in the GPU-based L0 trigger. Along the same lines, the deterministic latency link of the NaNet^3 customization makes it a viable solution for the data transport system of the KM3NeT-IT experiment while the GPUDirect RDMA features imported from NaNet will allow us later on to build a real-time GPU-based platform, to investigate improved trigger and data reconstruction algorithms.

Acknowledgments

References

[1] R. Ammendola et al. APEnet+: a 3D Torus network optimized for GPU-based HPC systems. *Journal of Physics: Conference Series*, 396(4):042059, 2012.

[2] Gianluca Lamanna. The NA62 experiment at CERN. *Journal of Physics: Conference Series*, 335(1):012071, 2011.

[3] A Margiotta. Status of the KM3NeT project. *Journal of Instrumentation*, 9(04):C04020, 2014.

[4] R Ammendola et al. APEnet+ 34 Gbps Data Transmission System and Custom Transmission Logic. *Journal of Instrumentation*, 8(12):C12022, 2013.

[5] A. Aloisio et al. The NEMO experiment data acquisition and timing distribution systems. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2011 IEEE*, pages 147–152, Oct 2011.

[6] B Angelucci et al. The FPGA based trigger and data acquisition system for the CERN NA62 experiment. *Journal of Instrumentation*, 9(01):C01055, 2014.

[7] R Ammendola et al. NaNet: a flexible and configurable low-latency NIC for real-time trigger systems based on GPUs. *Journal of Instrumentation*, 9(02):C02023, 2014.

[8] R. Ammendola et al. NaNet: a low–latency NIC enabling GPU–based, real–time low level trigger systems. *Journal of Physics: Conference Series*, 513(1):012018, 2014.

[9] R. Giordano and A. Aloisio. Fixed latency multi-gigabit serial links with Xilinx FPGA. *IEEE Transaction On Nuclear Science*, 58(1):194–201, 2011.

Chapter 3

GPU in Montecarlo and Offline Analysis

Convenors:

Soon Yung Jun
Piero Vicini
Marco Sozzi

Fast event generation on GPU

J. Kanzaki[1]

[1] KEK, Japan

We use a graphics processing unit (GPU) for fast event generation of general Standard Model (SM) processes. The event generation system on GPU is developed based on the Monte Carlo integration and generation program BASES/SPRING in FORTRAN. For computations of cross sections of physics processes all SM interactions are included in the helicity amplitude computation package on GPU (HEGET) and phase space generation codes are developed for efficient generation of jet associated processes. For the GPU version of BASES/SPRING new algorithm is introduced in order to fully exploit the ability of GPU performance and the improvement of performance in nearly two orders of magnitude is achieved.

The generation system is tested for general SM processes by comparing with the Mad-Graph, which is widely used in HEP experiments, and integrated cross sections and distributions of generated events are found to be consistent. In order to realize smooth migration to the GPU environment, a program that automatically converts FORTRAN amplitude programs generated by the MadGraph into CUDA programs is developed.

Contribution not received.

First experience with portable high-performance geometry code on GPU

John Apostolakis[1], Marilena Bandieramonte[1], Georgios Bitzes[1], Gabriele Cosmo[1], Johannes de Fine Licht[1,]*, Laurent Duhem[3], Andrei Gheata[1], Guilherme Lima[2], Tatiana Nikitina[1], Sandro Wenzel[1,]†

[1]CERN, 1211 Geneve 23, Switzerland
[2]Fermilab, Kirk & Pine Rd, Batavia, IL 60510, USA
[3]Intel, 2200 Mission College Blvd., Santa Clara, CA 95054-1549, USA

The Geant-Vector prototype is an effort to address the demand for increased performance in HEP experiment simulation software. By reorganizing particle transport towards a vector-centric data layout, the projects aims for efficient use of SIMD instructions on modern CPUs, as well as co-processors such as GPUs and Xeon Phi.

The geometry is an important part of particle transport, consuming a significant fraction of processing time during simulation. A next generation geometry library must be adaptable to scalar, vector and accelerator/GPU platforms. To avoid the large potential effort going into developing and maintaining distinct solutions, we seek a single code base that can be utilized on multiple platforms and for various use cases.

We report on our solution: by employing C++ templating techniques we can adapt geometry operations to SIMD processing, thus enabling the development of abstract algorithms that compile to efficient, architecture-specific code. We introduce the concept of modular "backends", separating architecture specific code and geometrical algorithms, thereby allowing distinct implementation and validation. We present the templating, compilation and structural architecture employed to achieve this code generality. Benchmarks of methods for navigation in detector geometry will be shown, demonstrating abstracted algorithms compiled to scalar, SSE, AVX and GPU instructions. By allowing users to choose the most suitable type of processing unit for an application or potentially dispatch work dynamically at runtime, we can greatly increase hardware occupancy without inflating the code base. Finally the advantages and disadvantages of a generalized approach will be discussed.

1 Introduction

Recent evolution of hardware has seen a reawakened focus on vector processing, a concept that had otherwise lain dormant outside the world of video games while CPU clock speeds still rode Moore's law. Most notable is the surge of interest in general purpose GPU programming (GPGPU), but vector instruction sets have existed on convential CPUs since the introduction of *3DNow!* by AMD [3] and *SSE* by Intel [4] in 1998 and 1999, respectively.

---

*Presenting author (johannes.definelicht@cern.ch).
†Corresponding author (sandro.wenzel@cern.ch).

To improve performance on modern hardware, it is essential to make use of their vector processing capabilities to access memory efficiently, whether this means efficient cache access on CPU or efficient use of memory bandwidth on GPU.

1.1 GeantV

The Geant Vector prototype (GeantV) [1] was started as a feasibility study of the potential for adding vectorization to detector simulation programs, in order to address an increasing need for performance in HEP detector simulation software.

At the core of GeantV there is a scheduling engine orchestrating the progress of the simulation. This scheduler works on particle data packed in *structure of arrays* (SOA) form for cache-efficiency and to accommodate SIMD memory access [7]; particles that require similar treatment are grouped into *baskets*, which can be dispatched to the relevant components for vectorized treatment [5]).

The GeantV R&D effort targets multiple platforms for vectorization, including SIMD instructions on CPUs (SSE, AVX) and accelerators such as GPUs and the Xeon Phi (AVX512). Although sometimes limited by the state of compilers, the project pursues *platform independence* to avoid *vendor lock-in*.

1.2 VecGeom

Navigation and particle tracking in detector geometry is responsible for up to 30%–40% of time spent on particle transport in detector simulation (this fraction is highly experiment dependent) [2], thus being an important research topic for vectorization in the GeantV prototype. The main consumers of cycles are the four algorithms shown in Figure 1, which are implemented separately for each geometrical shape.

VecGeom introduces vectorization techniques in the implementation of a new geometry modeller. It is a continuation of the USolids [8] effort, a common library for geometrical primitives, unifying and enhancing the implementations that currently exist in ROOT [9] and Geant4 [10]. The scalar interfaces of VecGeom are therefore made to be compliant with the ones of USolids, allowing free interchangeability between the two. VecGeom and USolids are converging to a single code base.

To accommodate the scheduler of GeantV mentioned above, VecGeom exposes vector signature operating on baskets of particles. These signatures take SOA objects carrying information on the particles in the basket, instead of a single set of parameters. Since these interfaces come in addition to the scalar ones, there is an issue of signature multiplication requiring a lot of boilerplate code, which will be addressed in Section 2.3.

VecGeom does not rely on vectorization alone to achieve performance. In the spirit of USolids, the library employs the best available algorithms to solve the problems listed in Figure 1. This can mean picking and optimizing the best algorithm between existing ones in ROOT, Geant4 and USolids, or it can mean writing them from scratch if deemed relevant. When a new shape is implemented, performance is compared directly to that of the existing libraries. Algorithms are not accepted before their *scalar* version (in addition to the vectorized one) outperforms all existing implementations (more on different kernel versions follows in Section 2.1). After algorithmic optimization, the library uses a number of additional templating techniques for performance which are described in Section 2.1 and 2.4.

2 Templates for portability and performance

Templating is at the heart of VecGeom, being involved in every aspect of optimization. The philosophy is that whenever the parameters of a performance-critical component can be resolved

(a) Determine if point is inside a volume.

(b) Intersection check and distance to intersection with volume.

(c) Minimum bound on distance to volume boundary.

(d) Distance to exit current volume.
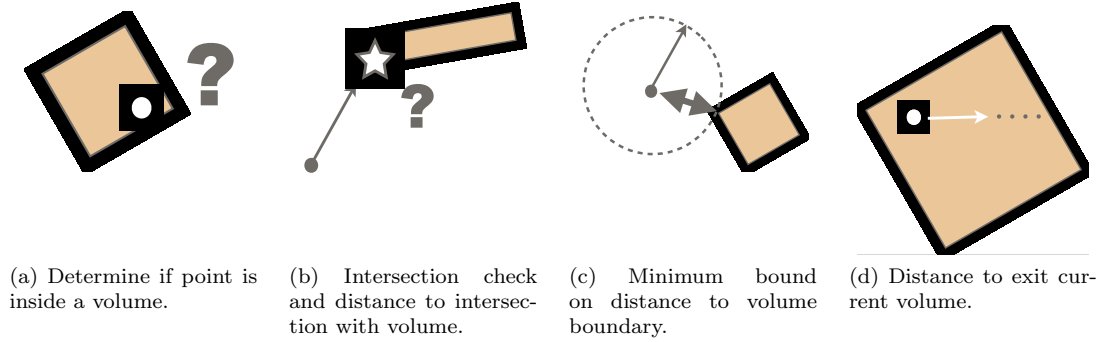
Figure 1: Primary problems solved by a geometry library during particle transport in simulation.

at compile-time, or even just if it has a finite set of configurations (see Section 2.4), templates will be used to generate specialized machine code.

2.1 Thinking in kernels

The approach taken by VecGeom to achieve portability and performance through templates is to separate performance-critical functionality into small plug-and-play *kernels*. Each kernel should have a single and well-defined purpose, delegating subtasks to separate kernels, promoting the sharing of kernels between algorithms. Kernels are built to operate on a single unit of input, meaning no explicit vectorization is typically done in them. Instead, vectorization can be conveyed by the *types* on which operations are performed (as this will often be a template parameter), and is described in Section 3. Logical constructs such as branches have to be abstracted to higher level functions in order to accommodate this. To ensure that kernels can indeed be shared between different contexts, kernels can live either as free C-like functions or as static methods of enclosing classes, but should never relate to an object. When an object-abstraction is useful, classes can instead implement static methods and call them from within their own methods.

2.2 Architecture abstraction

An important motivation for taking the kernel approach is *architecture abstraction*. VecGeom is a performance-oriented library, and as such aims at maximizing hardware utilization by offering to run scalar and SIMD code on CPU, and also run on accelerators. Rather than writing separate implementations for each target architecture and multiplying the required maintenance, VecGeom has opted for a generalized approach. The difference between architectures is isolated to memory management and the *types* that are operated on, with the latter being handled by the kernels. For this, the concept of *backends* was introduced. Backends represent an architecture to which kernels should compile, and are encapsulated in C++ trait classes. These classes contain the relevant types to operate on, as well as some static members, such as a boolean to toggle early returns. Backends are used as a template parameter of kernels that operate on the provided types, and by overloading higher level functions called on these types, the architecture specific instructions necessary to perform operations is decided. An example of a backend class is shown in Listing 1.

```
class VcBackend {
public:
  typedef Vc::double_v double_t;
  typedef Vc::int_v int_t;
  typedef Vc::double_m bool_t;
  static constexpr bool earlyReturns
       = true;
};
```

Listing 1: A C++ trait class representation of a backend, providing types to operate on and allowing other configuration options. Vc is used to wrap vector instruction sets.

```
template <class B>
typename B::double_t DistanceToPlane(
    Vector3D<typename B::double_t> const &point,
    Vector3D<typename B::double_t const &plane) {
  typename B::double_t result;
  result = point.dot(plane);
  // If behind the point is considered inside
  MaskedAssign(result < 0, 0, &result);
  return result;
}
```

Listing 2: Kernel operating on a backend class, such as the one shown in Listing 1.

2.3 Solving code multiplication

The introduction of the vector interfaces led to a *multiplication of signatures* necessary for a shape to implement, as each shape has to provide both the scalar and vector signatures. Because of the way kernels are designed, this quickly led to large amounts of boilerplate code that simply called different instantiations of the the same family of templates. These were not only cumbersome to write for each implemented shape, but also meant a lot of duplicate code to maintain in the future. To improve this, the *curiously recurring template pattern* (named by James Coplien [12]) is employed. This pattern allows a leaf class to inherit from a templated auxiliary class, passing its own type as a template parameter. The auxiliary class will then inherit from the appropriate base class and implement a number of methods calling static methods of the class passed as a template parameter. This can be either the inheriting class itself or a separate class entirely. Using the interface of VecGeom shapes, this meant that only a single templated method per algorithm had to be implemented for each shape. The boilerplate code used to call different configurations was collected in a single global auxiliary class. An example of this technique is provided in Listing 3.

2.4 Shape specialization

VecGeom defines a set of geometrical shapes. Some parameters describing the shapes have a significant impact on the algorithms: an angle configuration can eliminate the need for a trigonometric function, or a tube with no inner radius can just be treated as a cylinder. Creating separate shape classes for specific configurations would cause the number of primitives to explode. Instead, VecGeom takes advantage of this behind the scenes by using the concept of *shape specialization*. By having kernels template on one or more specialization parameters, blocks of code are tweaked or removed entirely from the implementation at compile time. This saves branch mispredictions and register allocation as compared to performing all branches at runtime. The concept is shown in Listing 4.

To avoid to expose end users to specialization, the instantiation of each configuration is done by a factory method. This limits specialization to a finite phase space, but hides the specifics through polymorphism, handing the user an optimized object behind the base primitive's interface. Specialization happens when shapes are *placed* into their frame of reference. Volumes instantiated by other means are not specialized, and will instead branch at runtime.

Figure 2 shows relative benchmarks between the same kernel compiled for three different scenarios: the unspecialized case where branches are taken at runtime, the specialized case

```
template <class B, class S>
class Helper : public B {
public:
  virtual double Distance(Vector3D<double> ...) const {
    // Implement virtual methods by calling the
    // implementation class methods
    S::template Distance<kScalar>(...);
  }
  virtual void Distance(SOA3D<double> ...,
                        double *distance) const {
    // Generate vector types from the input and loop
    // over the kernel
    S::template Distance<kVector>(...);
  }
};
```

Listing 3: Applying the curiously recurring template pattern to eliminate boilerplate code. A shape inheriting from the helper class only needs to provide a single kernel per algorithm, as the helper will use it to implement all the virtual function required by the interface.

```
template <bool hollow>
class TubeImplementation {
public:
  void Distance(...) {
    // Treat outer radius and
        both ends
    ...
    if (hollow) {
      // Treat inner radius
      ...
    }
  }
};
```

Listing 4: Change or remove blocks of code at compile-time. To avoid relying on the optimizer, an auxiliary function can also be introduced to explicitly specialize different cases of a template parameter.

where unused code has been optimized away at compile time, and the vectorized case treating four incoming rays in parallel using AVX instructions.

3 Running on multiple architectures

With the proposed kernel design, supporting multiple architecture means to provide the per-architecture backend trait class, and to write kernels in a generic fashion, letting types lead to the underlying instructions. Mathematical functions typically work by simple overloading on input types. Conditional statements need to be handled more carefully, as conditions evaluated from the input can have different values for each value in the vectors being operated on. This can be done using *masked assignments* and *boolean aggregation*: the overloaded masked assignment function uses an input condition to only assign input elements to a new value if the condition evaluates to true for the instance in question; the overloaded boolean aggregation functions reduce an input condition by aggregators *all*, *any* or *none*. When the input is scalar, these functions trivially assign the variable or return the conditional variable itself, respectively. An example of a kernel using a masked assignment is shown in Listing 2.

VecGeom uses the Vc [11] library to target CPU vector instructions. Vc provides types stored as packed, aligned memory, which are loaded into vector registers. Depending on the supported instruction set, this can compile to SSE, AVX or AVX512 intrinsics, operating on 2, 4 or 8 doubles at a time, respectively. Since vectorization by Vc is explicit, kernels instantiated with these types are guaranteed to use appropriate vector intrinsics. Figure 2 includes a benchmark for a kernel compiled to AVX intrinsics by operating on Vc types.

Supporting CUDA is fairly straightforward, as the VecGeom kernel architecture fits well into a GPU design. Kernel functions are simply annotated with __host__ and __device__ when compiling with nvcc, making them available to both CPU and GPU as needed. Additionally, the mask-based coding style described above means that branching is discouraged by design, showing how the generalized SIMD-oriented approach benefits multiple architectures.
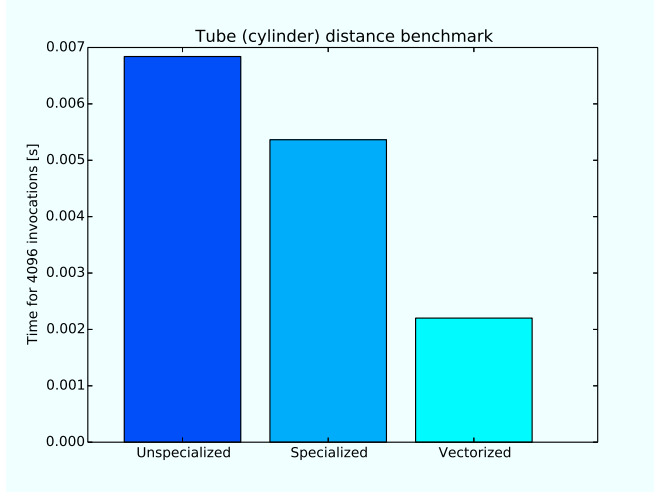
Figure 2: Relative benchmarks of the distance algorithm to a cylinder, which is the special case of a tube with no inner radius. All configurations are compiled from the same kernel. Vectorized kernel compiled to AVX instructions, treating four rays in parallel.
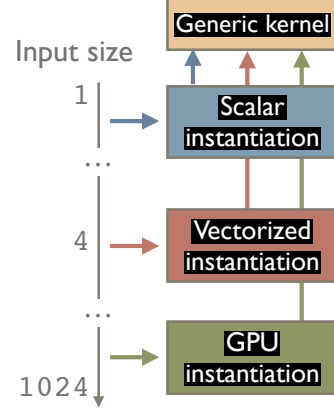
Figure 3: Methods targeting different architectures are instantiated from the same templated kernel.

Once kernels have been instantiated with different backends, it is easy to perform benchmarks such as the ones shown in Figure 4, comparing performance and scaling on different architectures. This can also be used for verification purposes: once an algorithm is validated on one architecture, other architectures can conveniently be validated against this result.

3.1 GPU support

The library support for GPUs was first implemented in CUDA. Although this opposes the desire to avoid vendor lock-in, it was considered the more productive way forward given the current state of CUDA and OpenCL compilers.

The support offered is two-fold: shape primitives and their respective kernels are provided as device functions that can be run from user code, and memory synchronization capabilities are offered to copy geometries built in main memory to GPU memory. Together this offers full flexibility to integrate the GPU in user code, whether it be exclusively or as a target for offloading.

An effort to investigate the use of this approach with OpenCL has been significantly hampered by the lack of support for polymorphism in the current specification.

3.2 Feasibility of architecture independence

Several advantages of architecture independent kernels have been discussed, such as maintainability, hardware utilization and cross-architecture benefits of SIMD data structures. There are of course disadvantages to this approach, the most prominent being loss of important architecture specific optimizations by restricting kernels to use higher-level abstractions. This is recognized in VecGeom by allowing implementation of template specializations of a given backend for kernels or overwriting methods (that are otherwise implemented by a helper as

(a) Distance algorithm, 50 floating point operations per kernel execution.

(b) Inside algorithm, 10 floating point operations per kernel execution.
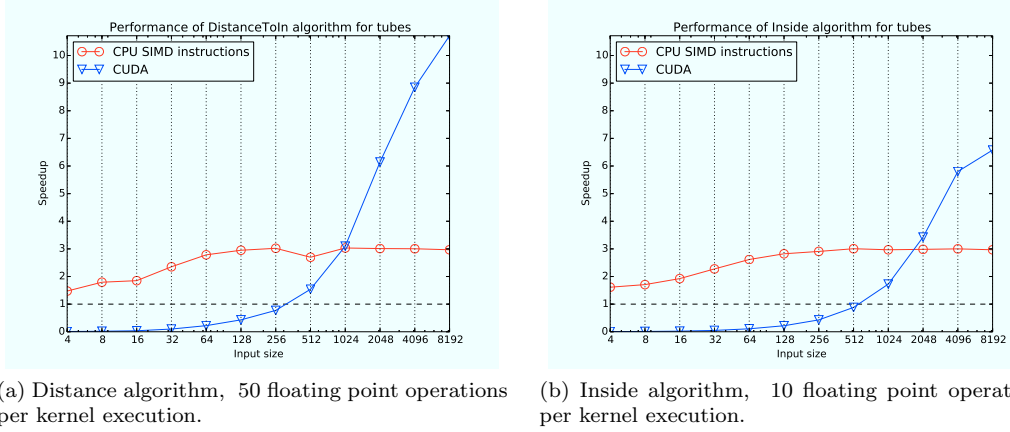
Figure 4: Relative benchmarks of generic kernels run on scalar, AVX and CUDA architectures on consumer-grade hardware (3.4 GHz Ivy Bridge Core i7 and GeForce GTX 680). Speedup with respect to the scalar instantiation of the same kernel shown as the black dashed line. The GPU needs considerably more floating point operations than the CPU to efficiently hide memory latency.

demonstrated in Section 2.3) in leaf classes. In this way, the code base can be extended to include specific implementations without sacrificing the benefits of having general kernels.

Since kernels often template on one or more parameters, and to encourage compiler optimizations in general, all kernels are implemented in header files. In performance intensive code, function calls (virtual function calls in particular) have proven to be a significant performance liability, amplified by the design that encourages high separation of functionality. Avoiding this comes with at least two (correlated) penalties, being compilation time and binary file size. Although this is typically an acceptable trade-off, the impact can be significant, and is monitored as development progresses to ensure a desirable balance.

4   Summary

Through the philosophy of small, reusable and general kernels, and the creation of abstract backends, VecGeom has achieved a small code base with a large degree of portability, supporting both CPU and accelerators. Architecture specific optimizations remain possible by using template specialization and method overwriting.

Benchmarks were presented that demonstrate benefits of vectorization achieved in a general fashion by employing the Vc library, and the gain in the performance obtain by specializing shapes at compile time to remove unused code (all configurations instantiated from the same templated kernel).

GPU support is offered through device instantiations of kernels and memory synchronization API, allowing exclusive or offloading use by an external scheduler.

References

[1] J. Apostolakis et al., *Rethinking particle transport in the many-core era towards GEANT 5*, 2012 J. Phys.: Conf. Ser. **396** 022014, doi:10.1088/1742-6596/396/2/022014.

[2] J. Apostolakis et al., *Vectorising the detector geometry to optimise particle transport*, 2014 J. Phys.: Conf. Ser. **513** 052038, doi:10.1088/1742-6596/513/5/052038.

[3] AMD, *3DNow! Technology Manual*, 21928G/0, Rev. G, March 2000.

[4] Intel Corporation, *Pentium III Processor for the PGA370 Socket at 500 MHz to 1.13 GHz*, 245264-08, Rev. 8, June 2001.

[5] A. Gheata, *Adaptative track scheduling to optimize concurrency and vectorization in GeantV*, Proc. ACAT 2014, forthcoming.

[6] Sandro Wenzel, *Towards a generic high performance geometry library for particle transport simulation*, Proc. ACAT 2014, forthcoming.

[7] Intel Corporation, *Programming Guidelines for Vectorization*, 2014.

[8] Marek Gayer et al., *New software library of geometrical primitives for modeling of solids used in Monte Carlo detector simulations*, 2012 J. Phys.: Conf. Ser. **396** 052035, doi:10.1088/1742-6596/396/5/052035.

[9] R. Brun and F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Meth. A **389** (1997) 81.

[10] S. Agostinelli et al., *Geant4 - a simulation toolkit*, Nuclear Instruments and Methods **A** 506 (2003) 250-303.

[11] M. Kretz and V. Lindenstruth, *Vc: A C++ library for explicit vectorization*, Softw: Pract. Exper., 42: 14091430. doi: 10.1002/spe.1149, December 2011.

[12] J. Coplien, *Curiously Recurring Template Patterns*, C++ Report, February 1995.

Hybrid implementation of the VEGAS Monte-Carlo algorithm

Gilles Grasseau[1], Stanislav Lisniak[1], David Chamont[1]

[1]LLR, Laboratoire Leprince-Ringuet, Ecole polytechnique, 91128 Palaiseau, France

Cutting edge statistical methods involving Monte-Carlo integrations are commonly used in the `LHC` analysis chains. Dealing with high dimensionality integration, the computing time can be a bottleneck to deploy such statistical methods at large scales. In this paper we present the first bricks to build an `HPC` implementation of the well known VEGAS algorithm. Thanks to the open programming standards `OpenCL` and `MPI`, we target to deploy such tool on cluster of nodes handling various hardware like `GPGPU` or 'many-core' computing accelerators.

1 Motivations

Multidimensional integration based on Monte-Carlo (`MC`) techniques [1] are widely used in High Energy Physics (`HEP`) and numerous other computing domains. In `HEP`, they naturally arise from the multidimensional probability densities or from the likelihoods often present in the analysis. Due to computing intensive integrations and large data sets containing millions of events, the situation is difficult for an analysis team if the processing of all samples exceeds 2-3 weeks (elapsed time).

Today, `HPC` programming requires dealing with computing accelerators like `GPGPU` or 'many-core' processors, but also taking into account the development portability and the hardware heterogeneities with the use of open programming standards like `OpenCL`. Among the available `MC` algorithms (MISER, Markov Chain, etc.) the choice has been driven by the popularity and the efficiency of the method. The 'VEGAS' algorithm [2, 3] is frequently used in `LHC` analyses as it is accessible from the ROOT environment [4], while providing reasonably good performance.

The parallel implementation of VEGAS for computing accelerators presents no major obstacle [5], even though some technical difficulties occur when dealing with portability and heterogeneity, mainly due to the lack of libraries and development tools (like performance analysis tools).

Combining `MPI` and `OpenCL`, we present a scalable distributed implementation. Performance will be shown on different platforms (`NVidia K20`, `Intel Xeon Phi`) but also on heterogeneous platform mixing `CPUs`, and different kinds of computing accelerator cards.

The presented work is a canvas to integrate various multidimensional functions for different analysis processes. It is planned to integrate and exploit this implementation in the future analyses by the `CMS` (Compact Muon Solenoid) experiment at `CERN`.

2 Software design

2.1 Introduction to the VEGAS `MC` algorithm

The ROOT-based `MC` integration environment is popular within the High Energy Physics community. This environment actually provides an encapsulation of the GNU Scientific Library (`GSL`) `MC` integration functions [6]. `GSL` offers 3 kinds of `MC` algorithms : classical (or PLAIN), MISER, VEGAS. Since VEGAS converges more rapidly than the two other methods, and is widely used in the `CMS` collaboration data analysis, we will focus on this `MC` integration method.

A High-dimensional integral $I = \int d^n \vec{x} \, f(\vec{x})$ can be approximated by evaluating $f(\vec{x})$ at M points $\vec{x}$, drawn randomly in the domain Ω with the probability density $p(\vec{x})$:

$$\overline{I} = \overline{f_p} = \frac{V}{M} \sum_{\vec{x}} \frac{f(\vec{x})}{p(\vec{x})}, \; V \text{ is the volume of } \Omega,$$

with its estimated variance $\overline{\sigma}^2$:

$$\overline{\sigma}^2 = \frac{\overline{f_p^2} - (\overline{f_p})^2}{M - 1} \simeq \sigma^2, \text{ where } \overline{f_p^2} = \frac{V}{M} \sum_{\vec{x}} (\frac{f(\vec{x})}{p(\vec{x})})^2.$$

The classical `MC` integration method (PLAIN in `GSL` library) uses a uniform density probability $p(\vec{x}) = cst$ which ensures the process' convergence: $\lim_{M \to +\infty} \overline{f_p} \longrightarrow I$. Nevertheless, the convergence is slow and requires a great number of points M to obtain a good approximation of I.

In the VEGAS algorithm, two main ideas are used to reduce the variance and, as a result, accelerate the convergence of the estimated integral $\overline{I}$.

1. *Importance sampling*: the variance tends to zero if the probability density has the form $p(\vec{x}) \propto |f(\vec{x})|$ (quick justification for $f(\vec{x}) > 0$, $\frac{f(\vec{x})}{p(\vec{x})} = cst, \Rightarrow \sigma^2 = 0$, see [1] for more details) . In other words, this means that the function sampling must be concentrated on the largest magnitudes of the function $f(\vec{x})$. Starting with $p(\vec{x})$ uniform, $p(\vec{x})$ gradually approximates $\frac{|f(\vec{x})|}{cst}$, thanks to the contributions of the different function evaluations.

2. *Stratified sampling*: this other strategy samples the highest variance domains in Ω, then subdivides it in sub-domains to decrease the local variance and thus the global variance of $|f(\vec{x})|$. With this strategy, the system converges with a sub-domain partition of Ω which minimizes the variance σ^2. In `GSL` implementation, the probability distribution $p(\vec{x})$ is updated with the local sub-domain variance $\sigma^2(\vec{x})$; these sub-domains are called *boxes* and are used to discretize $p(\vec{x})$.

Depending on the number of points to evaluate (set by user), VEGAS chooses *Importance sampling* or *Stratified Sampling* according to the sampling density of the domain.

2.2 Parallelism and `OpenCL` considerations

The VEGAS algorithm can be sketched with 3 main embedded loops as shown in Fig.: 1. The internal loop evaluates the function $f(\vec{x})$ for M random points $\vec{x}$, according to the probability distribution $p_{k-1}(\vec{x})$. The accumulated values of $f(\vec{x})$ give the estimated integral $\overline{I_k}$. In the same loop, the estimated variance $\overline{\sigma_k}^2$, as well as the new probability distribution $p_k(\vec{x})$, are

updated. Several evaluations of the $\overline{I_k}$ integral are performed in the intermediate loop, in order to compute the Chi-square χ^2 by degree of freedom, thus determining the consistency of the sampling. Finally, the outer-loop is used to control the integral convergence $(\overline{I}, \overline{\sigma}^2, \chi^2)$.

From the parallelization point of view, the most computing intensive loop, i.e. the internal one, must be spread among different computing units, handling the shared variables $\overline{I_k}$, $\overline{\sigma_k}^2$, $p_k(\vec{x})$ with care. It is well-known that opening a *parallel region* (with the `OpenMP` formalism) on the internal loop is much less efficient than opening a *parallel region* on the outer loop. In the same way, `OpenCL` (or `CUDA`) *kernels* must be as large as possible to avoid substantial overhead time to launch kernels and unnecessary work to split the initial kernel in several kernels. It is worth mentioning, in our algorithm, the embedded loops must be split in two *kernels* at the reduction step $(\overline{I_k}, \ldots)$ to synchronize the global memory between the different *work-groups*. As a result, split kernels will generate unexpected overhead time to launch and synchronize : *number_convergence_iterations* $\times$ *number_internal_iterations* $\times$ 2 *kernels*.

Writing and managing a single *kernel* which takes into account all steps of the VE-GAS algorithm, presents no difficulty. It only requires that each computing element evaluates several points contributing to the integral. In addition it substantially increases the computing load per computing unit. However, the only way to synchronize the shared variables is to perform the computations in a single *work-group* (or *block* in `CUDA`). Although this solution works well on `Intel Xeon Phi` (thanks to `OpenCL` 1.2), it does not work properly on `NVidia` hardware for 2 reasons:

Compute: $I = \int_\Omega f(\vec{x}) \, d^n x$, on
a d-dimensional integration domain Ω
Loop *until convergence* $(\overline{\sigma}^2, \chi^2)$
    Loop *internal iteration k*
        $p_k(\vec{x}) \leftarrow 0$;
        Loop *over N points $\vec{x} \in \Omega$*
            $\vec{x} \leftarrow \mathrm{rand}()$;
            $\bar{f} \leftarrow \bar{f} + f(\vec{x})$;
            update $\overline{I_k}$, $\overline{\sigma_k}^2$, $p_k(\vec{x})$;
        End Loop
        update $\overline{I}, \overline{\sigma}^2, \chi^2$;
    End Loop
End Loop

Figure 1: the VEGAS algorithm. The integral $\overline{I}$ of $f(\vec{x})$ is evaluated on the d-dimensional domain Ω. The standard deviation $\overline{\sigma}^2$ and the χ^2 can be used as convergence criteria. $p_k(\vec{x})$ is the probability distribution discretized on a grid.

- The *work-group* size is limited (hardware limit, generally 1024).

- Two `OpenCL` *kernels* cannot be run simultaneously on `NVidia GPGPUs` (the situation is even more dramatic, two `OpenCL` *kernels* cannot be run simultaneously on two different `GPGPUs` in the same process).

With such limitation on `NVidia OpenCL` driver we were constrained to split the kernel and lose efficiency.

2.3 `OpenCL` and `MPI` event dispatchers

The expected speed-up factor of a single accelerator card will not be sufficient to minimize user waiting time when dealing with data-sets containing each 10^6 events to process. The CMS analysis team cannot afford to wait for long processing chains to end (each chain needs several weeks to be processed). With this requirement, it is mandatory to use several accelerator cards

simultaneously, and even more, using several nodes themselves handling several accelerator cards to build a high performance computing application.

To gather the computing power available, a two-level event dispatcher has been implemented: the `OpenCL` *event dispatcher* dealing with the distribution of events among several devices (i.e. several accelerator cards and `CPUs`) and the `MPI` *event dispatcher* to distribute events among the computation nodes.

The `OpenCL` *event dispatcher* takes advantage of the `OpenCL` abstract model, in which *queues* manage *kernels* to be executed on the hosted *devices*. In this way, we can benefit directly of the `CPUs` power without adding a multi-thread programming paradigm to build a hybrid application.

The `OpenCL` dispatcher feeds the device with an event to be processed, as soon as a queue is free. All copies from host node to accelerator cards and all kernels are launched asynchronously, with a dependence graph built thanks to `OpenCL` *events*. The computation of the integral is considered to be finished when the copy of the result - from the *device* to the host - is completed.

The second dispatcher level, called `MPI` *event dispatcher* aggregates computing power

Figure 2: `OpenCL` dispatcher: the events are dispatched to different device queues. The different kernels to be processed are run asynchronously. The synchronization is performed thanks to `OpenCL` *events*

form different nodes by distributing sets of events among several MPI processes (generally one `MPI` process per node). In the same way as the `OpenCL` dispatcher, the *master* `MPI` process sends a set of events to be processed, as soon as a *worker* `MPI` process is idle. Then, the `MPI` worker (or `MPI` master) delegates the treatment of its event set to the `OpenCL` dispatcher.

3 Performance studies

3.1 Benchmark environment

As far as the integration is concerned, we choose for this benchmark the special function *sinus integral* to test the adaptive scheme for an oscillating function, whose result is known:

$$I = \int_0^{2\pi} \prod_{i=0}^n \frac{\sin(x_i)}{x_i} d^n x \simeq 5.7360, \text{ with } n = 5.$$

Product	Version/Options
Compiler	icc-13.0.1 / `-O3`
MPI	`OpenMPI 1.6.5`
`OpenCL`	`Intel 1.2`
`OpenCL`	`NVidia 1.1`

Table 1: Compiler and library versions.

Often, the integration is done iteratively to control the convergence towards acceptable values of the standard deviation σ and chi-square χ^2. In this test case, we want to

perform a fixed number of iterations, evaluating 5×10^5 points per integration. For these parameters, VEGAS draws 2 points per box (discretized volume element of $p_k(\vec{x})$), this means that only 12 boxes are used per dimension for the grid integration domain ($2 \times 12^5 = 497664$).

Three kinds of clusters are available on our platform, called `GridCL`, for benchmarking:

- Two nodes connected with an `InfiniBand` link, each hosting 2 `Intel Xeon E5-2650` processors (8 cores for each processor rated at 2.0 GHz). In addition, each node hosts 2 `NVidia K20M` GPGPU cards.

- Same as above concerning the node characteristics. Each node hosts 2 `Intel Xeon Phi 5110P` accelerator cards.

- The last node based on two `Intel Xeon E5-2650 v2` (Ivy Bridge) processors (8 cores for each processor rated at 2.6 GHz) hosts 6 `NVidia Titan` GPU cards.

The software configuration used to build the VEGAS hybrid application is presented in Tab. 1.

3.2 Results

As shown in Tab. 2, the `OpenCL` implementation of VEGAS presents very good performance on `CPUs` (column 2). This improvement is not only accountable to `OpenCL` parallelization, but also to the `Intel OpenCL` implementation which both *parallelizes and vectorizes* the *kernels*, as `Intel` was claiming [7] (a speed-up of 18.5 is obtained with 32 `MPI` instantiations of the `GSL` implementation - column 1).

The acceleration obtained with the `NVidia K20M` card seems to be good, but compared with `CPUs` performance, the gain is not excellent. As announced in paragraph section 2.2, no gain is obtained with two `K20M` devices (column 4), assuming that the `NVidia OpenCL`

Speed-up	GSL (1)	OCL (2)	OCL (3)	OCL (4)	OCL (5)	MPI (6)
# CPUs	32	32	-	-	32	-
# accelerators	-	-	1	2	2	2/6
K20M node	18.5	50.0	56.0	56.0	94.5	110
Xeon Phi node	18.5	50.0	27.3	54.1	80.3	-
Titan node	18.1	39.1	55.9	55.6	95.3	328

Table 2: Speed-up values obtained with different configurations (the reference time to calculate the speed-up values is the computing time obtained with the `GSL` library): (1) using `GSL` library executed on 32 processors (in fact 16 physical processors, 32 with hyper-threading), (2) `OpenCL` version on the 32 processors, (3) `OpenCL` version on a single accelerator card, (4) `OpenCL` with 2 accelerator cards, (5) `OpenCL` with all devices including `CPUs`, (6) 2 `MPI` processes (6 `MPI` processes on the `Titan` node), each handling one accelerator card (with `OpenCL`).

driver does not handle simultaneously several cards properly. Fortunately, we can bypass the lack of functionality, by launching 2 `MPI` processes each handling one device (see column 6).

Concerning `Intel Xeon Phi`, the speed-up shows that we do not use theses accelerators optimally. We will rely on the `Intel` performance analysis software `VTune` to highlight the bottleneck when we will start to optimize our *kernels*. However, the `Intel OpenCL` driver deals with the 2 cards simultaneously (column 4). Considering all computing devices, good overall performance is obtained (column 5), thanks to the efficiency of `OpenCL CPUs`.

Benchmarking the computing node holding 6 `NVidia Titan` cards, the performance profile is the same as above : moderated speed-up for one card (speed-up = 56). With all 6 cards the

speed up value is identical due to the `NVidia` driver issue, while the speed-up reaches 328 with 6 MPI processes.

The `OpenCL` and `MPI` event dispatchers provide good efficiency (see columns 5 and 6) even if the efficiency decreases when using several nodes. A time sampling has been added into the application to trace the event distribution, the overheads and the idle zones for future optimization work.

4   Conclusion

This preliminary version of our high performance `MC` integration implementation, based on `OpenCL` and `MPI`, already offers good efficiency on `CPUs` and `OpenCL` event dispatchers. The application however still needs several improvements to extract more computing power from accelerator cards, and requires tuning load-balancing parameters to efficiently run on `MPI` event dispatchers. Already substantial speed-up (> 300 - compared with sequential integrations based on `GSL` library) has been reached on the `GridCL` node hosting 6 `NVidia Titan` cards.

Before improving the application with all potential identified optimizations, we are going to focus our activity on a real application currently designed by the `LLR CMS` analysis team. Based on VEGAS integrations, the matrix-element methods (`MEM`) [8, 9], is a well known powerful approach in particle physics to extract maximal information from the events. Knowing that `MEM` require a huge computing power (processing one event can take $60s$), we aim to provide a drastic speed-up to the `MEM` processing chain.

Acknowledgments

References

[1] Stefan Weinzierl. Introduction to Monte Carlo methods. *ArXiv e-prints*, 2000.

[2] G. Peter Lepage. A New Algorithm for Adaptive Multidimensional Integration. *J.Comput.Phys.*, 27:192, 1978.

[3] G. Peter Lepage. VEGAS: An Adaptive Multi-dimensional Integration Program. 1980. Cornell preprint CLNS 80-447.

[4] Rene Brun and Fons Rademakers. ROOT - An Object Oriented Data Analysis Framework. *Nucl. Inst. & Meth. in Phys. Res. A*, 389:81–86, 1997. http://root.cern.ch/.

[5] J. Kanzaki. Monte Carlo integration on GPU. *European Physical Journal C*, 71:1559, February 2011.

[6] M. Galassi et al. *GNU Scientific Library Reference Manual*. Network Theory Ltd, third edition edition, 2009. http://www.gnu.org/software/gsl/.

[7] Intel developper note. Writing Optimal OpenCL Code with Intel OpenCL SDK. page 10, 2011. https://software.intel.com.

[8] Abazov VM et al. A precision measurement of the mass of the top quark. *NATURE*, 429:638–642, 2004.

[9] D. Schouten, A. DeAbreu, and B. Stelzer. Accelerated Matrix Element Method with Parallel Computing. *ArXiv e-prints*, July 2014.

GPUs for Higgs boson data analysis at the LHC using the Matrix Element Method

Doug Schouten[1], Adam DeAbreu[2], Bernd Stelzer[2]

[1]TRIUMF, TRIUMF, 4004 Wesbrook Mall, Vancouver, BC, Canada
[2]Department of Physics, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada

The matrix element method utilizes ab initio calculations of probability densities as powerful discriminants to extract small signals from large backgrounds at hadron collider experiments. The computational complexity of this method for final states with many particles and degrees of freedom sets it at a disadvantage compared to supervised classification methods such as decision trees, k nearest-neighbour, or neural networks. We will present a concrete implementation of the matrix element technique in the context of Higgs boson analysis at the LHC employing graphics processing units. Due to the intrinsic parallelizability of multidimensional phase space integration, dramatic speedups can be readily achieved, which makes the matrix element technique viable for general usage at collider experiments.

1 Introduction

The matrix element method (MEM) in experimental particle physics is a unique analysis technique for characterizing collision events. When used to define a discriminant for event classification, it differs from supervised multivariate methods such as neural networks, decision trees, k-NN, and support vector machines in that it employs unsupervised *ab initio* calculations of the probability density P_i that an observed collision event with a particular final state arises from $2 \to N$ scattering process i. Furthermore, the strong connection of this technique to the underlying particle physics theory provides key benefits compared to more generic methods:

1. the probability density P_i directly depends on the physical parameters of interest;

2. it provides a most powerful test statistic for discriminating between alternative hypotheses, namely P_i/P_j for hypotheses i and j, by the Neyman-Pearson lemma;

3. it avoids tuning on unphysical parameters for analysis optimization[1];

4. it requires no training, thereby mitigating dependence on large samples of simulated events.

The MEM was first studied in [1] and was heavily utilized by experiments at the Tevatron for W helicity [2] and top mass [3, 4] measurements, and in the observation of single top production

---

[1]Rather, optimization is determined by theoretical physics considerations, such as inclusion of higher order terms in the matrix element, or improved modeling of detector resolution.

[5], for example. It has also been used in Higgs searches at the Tevatron [6] and at the Large Hadron Collider (LHC) by both CMS [7] and ATLAS [8] collaborations. The MEM has also been extended in a general framework known as MADWEIGHT [11].

The MEM derives its name from the evaluation of P_i:

$$P_i = \frac{1}{\sigma_i} \sum_{\text{flavor}} \int_{V_n} \mathcal{M}_i^2(\mathbf{Y}) \frac{f_1(x_1, Q^2)\, f_2(x_2, Q^2)}{|\vec{q_1}| \cdot |\vec{q_2}|}\, d\Phi_n(q_1 + q_2; y_1, .., y_n), \qquad (1)$$

where $\mathcal{M}_i$ is the the Lorentz invariant matrix element for the $2 \to n$ process i, $\mathbf{Y}$ is shorthand notation for all the momenta $\vec{y}$ of each of the n initial and final state particles, f_1 and f_2 are the parton distribution functions (PDF's) for the colliding partons, σ is the overall normalization (cross-section), and

$$d\Phi_n(q_1 + q_2; y_1, .., y_n) = (2\pi)^4 \delta^4 \left(q_1 + q_2 - \sum_{i=1}^n y_i\right) \prod_{i=1}^n \frac{d^3 y_i}{(2\pi)^3 2E_i} \qquad (2)$$

is the n-body phase space term. The momenta of the colliding partons are given by q_1 and q_2, and the fractions of the proton beam energy are x_1 and x_2, respectively. The sum in Equation (1) indicates a sum over all relevant flavor combinations for the colliding partons.

The association of the partonic momenta $\mathbf{Y}$ with the measured momenta $\mathbf{X}$ is given by a transfer function (TF), $T(\vec{x}; \vec{y})$ for each final state particle. The TF provides the conditional probability density function for measuring $\vec{x}$ given parton momentum $\vec{y}$. Thus,

$$\hat{p}_i = \int P_i\, T(\mathbf{X}; \mathbf{Y})\, d\mathbf{Y}, \qquad (3)$$

is the MEM probability density for an observed event to arise from process i assuming the parton $\to$ observable evolution provided by the TF's. For well-measured objects like photons, muons and electrons, the TF is typically taken to be a δ-function. For unobserved particles such as neutrinos, the TF is a uniform distribution. The TF for jet energies is often modeled with a double Gaussian function, which accounts for detector response (Gaussian core) and also for parton fragmentation outside of the jet definition (non-Gaussian tail). To reduce the number of integration dimensions, the jet directions are assumed to be well-modeled so that $T(\theta_x, \phi_x; \theta_y, \phi_y) = \delta(\theta^x - \theta^y)\delta(\phi^x - \phi^y)$.

Despite the advantages provided by the MEM enumerated above, an important obstacle to overcome is the computational overhead in evaluating ≥ 1 multi-dimensional integrals for each collision event. For complex final states with many degrees of freedom (eg., many particles with broad measurement resolution, or unobserved particles), the time needed to evaluate $\hat{p}_i$ can exceed many minutes. In realistic use cases, the calculations must be performed multiple times for each event, such as in the context of a parameter estimation analysis where $\hat{p}_i$ is maximized with respect to a parameter of interest, or for samples of simulated events used to study systematic biases with varied detector calibrations or theoretical parameters. For large samples of events, the computing time can be prohibitive, even with access to powerful computer clusters. Therefore, overcoming the computation hurdle is a relevant goal.

This paper presents an implementation of the MEM using graphics processing units (GPU's). The notion of using GPU's for evaluating matrix elements in a multidimensional phase space has been investigated previously [12], although not in the context of the MEM. In order to ascertain the improvements in computing time when utilizing GPU's, the MEM was applied

in the context of a simplified $t\bar{t}H(\to b\bar{b})$ search in LHC Run II. Studying the $t\bar{t}H$ process is important in its own right [13, 14, 15, 16, 17]. Due to the complexity of the final state for this process, it is also an interesting use case in which to study the feasibility of the MEM with the improvements from highly parellelized multi-dimensional integration.

The note is organized as follows: in Section 2, the applicability of GPU architectures to the computational problem at hand is briefly outlined. In Section 3 a simplified $t\bar{t}H$ analysis is presented, which will be used to benchmark the improved computational performance afforded by modern GPU's. In Section 3.1 the specific implementation is outlined together with a summary of the results obtained from a number of GPU and CPU architectures.

2   Parallelized Integrand Evaluation

For dimensions ≥ 3, evaluation of multidimensional integrals is typically only feasible using Monte Carlo methods. In these methods, the integrand

$$ I = \int_{V_m} f(\vec{x}) \, d\vec{x} \tag{4} $$

is approximated by a sum over randomly sampled points in the m-dimensional integration volume V_m

$$ S_N \equiv V_m \underbrace{\frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i)}_{\equiv \bar{f}}, \tag{5} $$

which converges to I by the law of large numbers. For all Monte Carlo integration algorithms, there is a trivial parallelization that can be achieved by evaluating the integrand $f(x_i)$ at points $\{x_i\}_{i=1,..,N}$ simultaneously, since the evaluation of the integrand at each point x_i is independent of all other points $\{x_j\}_{j\neq i}$.

This mode of parallel evaluation is known as data parallelism, which is achieved by concurrently applying the same set of instructions to each data element. In practice, evaluating the functions used in the MEM involves conditional branching, so that the integrand calculation at each x_i does not follow an identical control flow. Nevertheless, it is instructive to proceed with the ansatz of strict data parallelism.

Data parallelism maps very well to the single instruction, multiple data (SIMD) architecture of graphics processing units (GPU's). Modern GPU's contain many individual compute units organized in thread units. Within each thread unit, all threads follow the same instruction sequence[2], and have access to a small shared memory cache in addition to the global GPU memory.

The advent of general purpose programming on GPU's (GPGPU) has vastly increased the computing capability available on a single workstation, especially for data parallel calculations such as in the MEM. Two languages have gained traction for GPGPU, namely CUDA [20] (restricted to GPU's manufactured by NVidia) and OpenCL [21]. Both languages are based on C/C++[3].

---

[2]This has implications for code with complicated control flow, since threads will be locked waiting for other threads in the same unit to be syncronized in the instruction sequence. Careful tuning of the MEM function control flow and the thread unit sizes may improve the performance.

[3]In this work, the AMD Static C++ extensions to OpenCL [22] are used.

3 $t\bar{t}H(\to b\bar{b})$ Search

The fact that the Higgs coupling to the top quark is ≈ 1 hints at a special role played by the top quark in electroweak symmetry breaking. Analysis of $t\bar{t}H(\to b\bar{b})$ production at the LHC can provide a powerful direct constraint on the fermionic (specifically, the top) couplings of the Higgs boson, with minimal model-dependence. The dominant backgrounds to this process, assuming at least one leptonic top decay, arise from the irreducible $t\bar{t}b\bar{b}$ background as well as the $t\bar{t}c\bar{c}$ and $t\bar{t}jj$ backgrounds via 'fake' b-tagged jets. The association of observed jets to
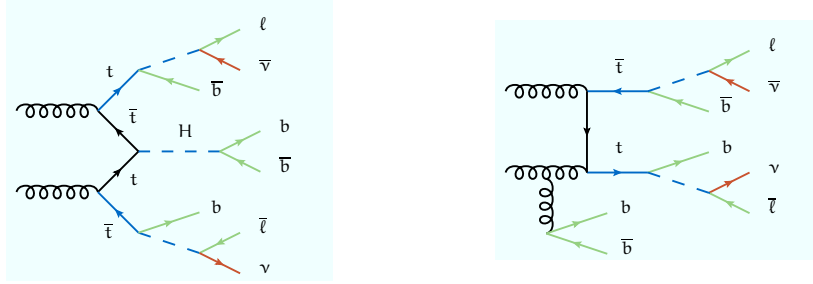

Figure 1: Representative Feynman diagrams for $t\bar{t}H(\to b\bar{b})$ production (left) and the irreducible $t\bar{t}b\bar{b}$ background (right) in the dilepton channel.

the external lines of the leading order (LO) Feynman diagrams in Figure 1 also gives rise to a combinatoric dilution of the signal, since there is an increased probability that a random pair of b partons in a $t\bar{t}b\bar{b}$ event will have $m_{bb'} \approx m_H$. For the fully hadronic $t\bar{t}$ decay, there are $4! \times 4! / 2 = 288$ combinations, assuming fully efficient b-tagging. This benchmark study is performed predominantly for the dileptonic $t\bar{t}$ decay mode, to showcase the application of the method for a very complex final state and also to reduce the large QCD backgrounds.

For the dilepton channel, the observable signature of the final state is $b\bar{b}\ell\bar{\ell} + \not{E}_T$, where $\not{E}_T = \vec{p}^{\,\nu}_T + \vec{p}^{\,\bar{\nu}}_T$. It is not possible to constrain the z components of the neutrino momenta, and using transverse momentum balance removes only two of the remaining four degrees of freedom from the x and y components. Due to the broad resolution of the measured jet energy, there are also four degrees of freedom for the energy of the four b quarks in the final state, so that the MEM evaluation implies an 8-dimensional integration:

$$\hat{p}_i = \frac{1}{\sigma_i} \sum_{\text{jet comb.}} \sum_{\text{flavor}} \int \mathcal{M}_i^2(\mathbf{Y}) \frac{f_1(x_1, Q^2)\, f_2(x_2, Q^2)}{|\vec{q}_1| \cdot |\vec{q}_2|} \, \Phi \,. \tag{6}$$

$$\delta\left(p_x^\nu - \not{E}_T^x - p_x^{\bar{\nu}}\right) \delta\left(p_y^\nu - \not{E}_T^y - p_y^{\bar{\nu}}\right) d^3\vec{p}_\nu \, d^3\vec{p}_{\bar{\nu}} \prod_{j=1}^{N_{\text{jet}}=4} \mathrm{T}\left(E_j^{\text{jet}}; E_j\right) \cdot \left(E_j^2 \sin\theta_j\right) dE_j,$$

where $\Phi = (2\pi)^4 \delta^4(q_1 + q_2 - \sum_{i=1}^n p_y^i) \prod_{i=1}^n \frac{1}{(2\pi)^3 2E_i}$, and the integrals over the lepton momenta are removed by assuming infinitesimal measurement resolution. The outer sum is over all permutations of assigning measured jets to partons in the matrix element. A transformation to spherical coordinates has been performed $d^3\vec{p} \to p^2 \sin(\theta)\, dp\, d\theta\, d\phi$ for the jets, and E is set to $|p|$.

The behaviour of the matrix element function $\mathcal{M}(\mathbf{Y})$ is strongly influenced by whether or not the internal propagators are on shell. It is difficult for numerical integration algorithms to efficiently map out the locations in momentum space of the external lines for which the internal lines are on shell. Therefore, it is advantageous to transform integration over the neutrino momenta to integrals over q^2 (where q is the four momentum) of the top quark and W boson propagators, so that the poles in the integration volume are along simple hyperplanes. This leads to the following coupled equations:

$$
\begin{aligned}
\not{E}_x &= p_x^\nu + p_x^{\bar{\nu}} \\
\not{E}_y &= p_y^\nu + p_y^{\bar{\nu}} \\
q_{W^+}^2 &= (E_{\ell^+} + E_\nu)^2 - (p_x^{\ell^+} + p_x^\nu)^2 - (p_y^{\ell^+} + p_y^\nu)^2 - (p_z^{\ell^+} + p_z^\nu)^2 \\
q_{W^-}^2 &= (E_{\ell^-} + E_{\bar{\nu}})^2 - (p_x^{\ell^-} + p^{\bar{\nu}_x})^2 - (p_y^{\ell^-} + p^{\bar{\nu}_y})^2 - (p_z^{\ell^-} + p^{\bar{\nu}_z})^2 \\
q_t^2 &= (E_b + E_{\ell^+} + E_\nu)^2 - (p_x^b + p_x^{\ell^+} + p_x^\nu)^2 - \\
&\quad (p_y^b + p_y^{\ell^+} + p_y^\nu)^2 - (p_z^b + p_z^{\ell^+} + p_z^\nu)^2 \\
q_{\bar{t}}^2 &= (E_{\bar{b}} + E_{\ell^-} + E_{\bar{\nu}})^2 - (p_x^{\bar{b}} + p_x^{\ell^-} + p_x^{\bar{\nu}})^2 - \\
&\quad (p_y^{\bar{b}} + p_y^{\ell^-} + p_y^{\bar{\nu}})^2 - (p_z^{\bar{b}} + p_z^{\ell^-} + p_z^{\bar{\nu}})^2,
\end{aligned}
\tag{7}
$$

which have been solved analytically in [23]. For the $t\bar{t}H(\to b\bar{b})$ process, there is an additional very narrow resonance from the Higgs propagator. For the same reasoning as above, the following transformation of variables for E_1 and E_2 are employed, which are the energies of the b-quarks from the Higgs decay, respectively:

$$
\begin{aligned}
f &= (E_1 + E_2) \\
m_H^2 &= (E_1 + E_2)^2 - |\vec{p}_1|^2 - |\vec{p}_2|^2 - 2\,|\vec{p}_1|\,|\vec{p}_2|\cos\Delta\theta_{1,2},
\end{aligned}
\tag{8}
$$

where $|\vec{p}| = \sqrt{E^2 - m^2}$. Figure 1 highlights the internal lines which are used in the integration.

3.1 Analysis and Results

The evaluation of the integrand in Equation (6) is broken into components for the matrix element $\mathcal{M}(\mathbf{Y})$, the PDF's, the TF's and the phase space factor. Each of these components is evaluated within a single GPU "kernel" program for each phase space point. Code for evaluating $\mathcal{M}$ is generated using a plugin developed for MadGraph [24]. This plugin allows one to export code for an arbitrary $2 \to N$ process from MadGraph to a format compatible with OpenCL, CUDA, and standard C++. This code is based on HELAS functions [25, 26]. Compilation for the various platforms is controlled with precompiler flags. Model parameters, PDF grids and phase space coordinates are loaded in memory and transferred to the device[4] (GPU) whereafter the kernel is executed. The PDF's are evaluated within the kernel using wrapper code that interfaces with LHAPDF [27] and with the CTEQ [28] standalone PDF library. The PDF data is queried from the external library and stored in (x, Q^2) grids for each parton flavor (d, u, s, c, b), which are passed to the kernel program. The PDF for an arbitrary point is evaluted using bilinear interpolation within the kernel. The precision of the interpolation is within 1% of the

---

[4]In the case of CPU-only computation, the transfer step is unnecessary.

Configuration	Details	Peak Power	Cost (USD, 2014)
CPU	Intel Xeon CPU E5-2620 0 @ 2.00GHz (single core) using `gcc` 4.8.1	95W	400
CPU (MP)	Intel Xeon CPU E5-2620 0 @ 2.00GHz (six cores + hyperthreading) using AMD SDK 2.9 / OpenCL 1.2	95W	400
GPU	AMD Radeon R9 290X GPU (2,816 c.u.) using AMD SDK 2.9 / OpenCL 1.2 on Intel Xeon CPU E5-2620	295W	450
GPU_x	same configuration as GPU, but with minor code modifications to accomodate GPU architecture		

Table 1: Details of the hardware configurations used to benchmark the MEM for GPU and (multicore) CPU's. The peak power is as reported by the manufacturer. The cost is listed in USD for the CPU or GPU only. For the GPU configuration, the code was identical to that used for the CPU configurations. For the GPU_x configuration, the code was modified to accommodate the specific GPU architecture.

values directly queried from the PDF library. An event discriminant D is constructed as

$$D = \log_{10}\left(\frac{\hat{p}_{t\bar{t}H}}{\hat{p}_{t\bar{t}b\bar{b}}}\right) \tag{9}$$

and evaluated for a sample of signal ($t\bar{t}H$) and background ($t\bar{t}b\bar{b}$) events generated in MAD-GRAPH and interfaced with PYTHIA for the parton shower, [29] using the so-called Perugia tune [30]. Jets are reconstructed using the anti-k_T algorithm described in [31] with width parameter $d = 0.4$. Any jets overlapping with leptons within d are vetoed, and b-tagging is performed by matching jets to the highest energy parton within $\Delta R = \sqrt{\Delta\eta^2 + \Delta\phi^2} < d$. A transfer function is defined for b-jets by relating the jet energy to the energy of the matched parton using a double Gaussian distribution.

The analysis is performed at two levels, namely

1. parton level: using the parton momenta from MADGRAPH-generated events directly (by assuming δ-function TF's for all final state particles), and averaging over all permutations for the assignment of the b partons in each event;

2. hadron level: using the outputs from PYTHIA and selecting events with four b-jets, averaging over all the permutations and integrating over the full 8-dimensional phase space as in Equation (6).

The convenient PyOpenCL and PyCUDA [32] packages are used to setup and launch OpenCL and CUDA kernels. Using OpenCL one can also compile the MEM source for a CPU target, and is thereby able to parallelize the MEM across multiple cores (see Table 1). In order to perform the numerical integration, a modified VEGAS implementation in Cython/Python [33] is used. This implementation has a number of improvements compared to previous versions and, importantly, interfaces with the provided integrand function by passing the full grid of phase space points as a single function argument. This allows one to pass the whole integration grid to the OpenCL or CUDA device at once, which facilitates the desired high degree of parallelism. The parton and hadron level MEM is performed with various hardware configurations specified in Table 1. In all configurations except the one labelled GPU_x, the MEM code used is identical. For the GPU_x case, minor modifications were made to replace particular array variables with sets of scalar variables.

3.1.1 Parton Level

Here, the evaluation is performed using the parton momenta, so that all the transfer functions become δ-functions, and the evaluation of D does not involve any numberical integration. In

this benchmark, all possible combinations of b-quarks in the final state are summed. The distribution of D for the signal and background samples is shown in Figure 2. A comparison of the time needed to evaluate $\hat{p}_i$ for all events is shown in Table 2 for various CPU and GPU configurations.

Process	CPU	CPU (MP)	GPU	GPU_x	CPU / GPU_x
signal	255	29	1.8	0.7	364
background	661	91	12	5.4	122

Table 2: Processing time, in seconds, required to evaluate the matrix elements for 10^5 events at parton level, for the various configurations detailed in Table 1. Using GPU's reduces the processing time by a factor greater than $120\times$ compared to a single CPU core for the $t\bar{t}b\bar{b}$ matrix element.

3.1.2   Hadron Level

The analysis at hadron level is closer to what can be optimally achieved in a real world collider experiment. Only the momenta of stable, interacting particles are accessible, and the jet energy resolution must be taken into account. The calculation of $\hat{p}_i$ requires evaluating the eight-dimensional integral in Equation (6). The integration variable transformation for $t\bar{t}H$ and $t\bar{t}b\bar{b}$ matrix element integrals presented in Section 2 are used. At each phase space point in the sum of Equation (5), the $\not{E}_T$ used in Equation (7) is defined as

$$\not{E}_{x,y} = - \left(p_{x,y}^{\ell^+} + p_{x,y}^{\ell^-} + \sum_{j \,\in\, \text{jets}} p_{x,y}^{j} \right).$$ (10)

The processing times per event for the hadron level MEM calculation are shown in Table 3. The relative improvement for the GPU is significantly reduced compared to the parton level analysis. This arises from a number of differences for this scenario. First, the VEGAS stratified sampling and adaptive integration algorithm is run on the CPU in all cases, which damps the GPU improvements in the integrand evaluation. Second, in the evaluation of the integral of Equation (6), significant additional complexity is demanded to solve Equations (7) and (8). Due to the cancellation of large coefficients in these solutions, double floating point precision is required, which reduces the GPU advantage since double precision calculations are performed significantly slower on most GPU's. Furthermore, the number of intermediate variables is significantly larger, which is found to increase the number of processor registers used. Since the number of registers available to each thread unit (or "wavefront" in the parlance of OpenCL) is limited to at most 256 for the GPU used in this study, the overall duty factor of the GPU is significantly reduced, to as low as 10%, since the full number of threads available in each block could not be utilized. It is anticipated that careful tuning of the code to accommodate GPU architecture could greatly improve the relative performance.

3.2   Further Results in the Lepton + Jets Channel

A brief summary is given on the evaluation of the GPU performance on a recent implementation of the $t\bar{t}H(\to b\bar{b})$ search in the lepton + jets channel. The corresponding Feynman diagrams for
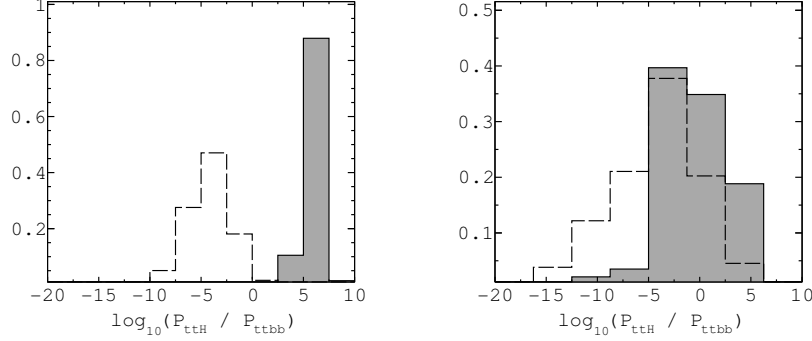
Figure 2: The event discriminant D for $t\bar{t}H$ (filled) and $t\bar{t}b\bar{b}$ (dashed line) events at parton level (left) and at hadron level (right). The distributions are normalized to unit area.

Process	CPU	CPU (MP)	GPU	GPU_x	CPU / GPU_x
signal	312	36.2	7.5	5.9	52.0
background	405	55.1	9.1	7.1	57.3

Table 3: Processing time required to evaluate the matrix elements for a single event at hadron level, for the various configurations detailed in Table 1. Note that this includes a full 8-dimensional integration over phase space for each event. Using GPU's reduces the processing time by at least $50\times$.

signal and background are shown in Figure 3. In this case, only a 6-dim phase space integral has to be evaluated (instead of 8-dim in the case of the dilepton final state). The variable
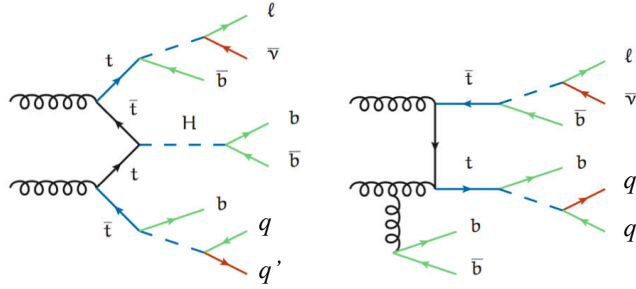
Figure 3: Representative Feynman diagrams for $t\bar{t}H(\to b\bar{b})$ production (left) and the irreducible $t\bar{t}b\bar{b}$ background (right) in the lepton + jets channel

transformation is much simpler. The results of the new study are summarized in Table 4. Using GPU's reduces the processing time by at least $100\times$ for the lepton + jets final state. The overall duty factor of the GPU is increased compared to the dilepton analysis to about 20%.

Process	CPU	CPU (MP)	GPU_x	CPU / GPU_x
signal	≈ 1000	91	7.9	125
background	≈ 3800	347	35	109

Table 4: Processing time required to evaluate the matrix elements for a single event at hadron level, for the various configurations detailed in Table 1. Note that this includes a full 6-dimensional integration over phase space for each event. Using GPU's reduces the processing time by at least $100\times$.

4   Conclusions

The matrix element method can be computationally prohibitive for complex final states. The $t\bar{t}H(\rightarrow b\bar{b})$ benchmark study in this paper has shown that by exploiting the parallel architectures of modern GPU's, computation time can be reduced by a factor ≥ 100 for the matrix element method, at about 10-20% utilization of the GPU. It is anticipated that careful code modifications can add significant further improvements in speed. This can be the subject of future study. However, even with the performance gains in this benchmark study, it is clear that for the MEM, the computing time required with O(10) GPU's is equivalent to a medium-sized computing cluster with O(400) cores (along with its required support and facilities infrastructure). This provides the potential to apply the method generally to searches and measurements with complex final states in experimental particle physics.

The programs described in this work are generic in nature, such that GPU-capable MEM code can be readily derived for an arbitrary $2 \rightarrow N$ process with only few modifications to accommodate transformations of variables or transfer functions. It is envisaged that future work can automate the inclusion of NLO matrix elements and transformations of variables (as in MadWeight) for the matrix element method, thereby providing an optimal methodology for classification and parameter estimation in particle physics.

References

[1] Kunitaka Kondo. Dynamical Likelihood Method for Reconstruction of Events with Missing Momentum. I. Method and Toy Models. *J. Phys. Soc. Jap.*, 57(12):4126–4140, 1988.

[2] V.M. Abazov, et al. Helicity of the W boson in lepton + jets $t\bar{t}$ events. *Phys. Lett. B*, 617:1–10, 2005.

[3] Abulencia, A. et al. Precision measurement of the top-quark mass from dilepton events at cdf ii. *Phys. Rev. D*, 75:031105, Feb 2007.

[4] V.M. Abazov et al. A precision measurement of the mass of the top quark. *Nature*, 429:638–642, 2004.

[5] Aaltonen, T., et al. Observation of single top quark production and measurement of $|V_{tb}|$ with CDF. *Phys. Rev. D*, 82:112005, Dec 2010.

[6] Aaltonen, T., et al. Search for a Standard Model Higgs Boson in $WH \rightarrow \ell\nu b\bar{b}$ in $p\bar{p}$ Collisions at $\sqrt{s} = 1.96$ TeV. *Phys. Rev. Lett.*, 103:101802, Sep 2009.

[7] Serguei Chatrchyan et al. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Phys.Lett.*, B716:30–61, 2012.

[8] Search for the Standard Model Higgs boson in the $H \rightarrow WW^{(*)} \rightarrow \ell\nu\ell\nu$ decay mode using Multivariate Techniques with 4.7 fb-1 of ATLAS data at $\sqrt{s} = 7$ TeV. Technical Report ATLAS-CONF-2012-060, CERN, Geneva, June 2012.

[9] F. Fiedler, A. Grohsjean, P. Haefner, and P. Schieferdecker. The Matrix Element Method and its Application to Measurements of the Top Quark Mass. *NIM A*, 624(1):203–218, 2010.

[10] A. Grohsjean. *Measurement of the Top Quark Mass in the Dilepton Final State Using the Matrix Element Method.* 2010.

[11] P. Artoisenet, V. Lemaitre, F. Maltoni, and O. Mattelaer. Automation of the matrix element reweighting method. *JHEP*, 12(68), 2010.

[12] K. Hagiwara, J. Kanzaki, Q. Li, N. Okamura, and T. Stelzer. Fast computation of MadGraph amplitudes on graphics processing unit (GPU). *European Physical Journal C*, 73:2608, 2013.

[13] F. Englert and R. Brout. Broken symmetry and the mass of gauge vector mesons. *Phys. Rev. Lett.*, 13:321–323, Aug 1964.

[14] Higgs, Peter W. Broken Symmetries and the Masses of Gauge Bosons. *Phys. Rev. Lett.*, 13:508–509, Oct 1964.

[15] G. S. Guralnik, C. R. Hagen, and T. W. B. Kibble. Global conservation laws and massless particles. *Phys. Rev. Lett.*, 13:585–587, Nov 1964.

[16] S. Dittmaier, S. Dittmaier, C. Mariotti, G. Passarino, R. Tanaka, et al. Handbook of LHC Higgs Cross Sections: 2. Differential Distributions. 2012.

[17] C. Degrande, J.M. Gerard, C. Grojean, F. Maltoni, and G. Servant. Probing Top-Higgs Non-Standard Interactions at the LHC. *JHEP*, 1207:036, 2012.

[18] G. Peter Lepage. A new algorithm for adaptive multidimensional integration. *Journal of Computational Physics*, 27(2):192 – 203, 1978.

[19] G. Peter Lepage. VEGAS: An Adaptive Multi-dimensional Integration Program. Technical Report CLNS 80-447, Cornell, March 1980.

[20] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[21] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[22] Ofer Rosenberg, Benedict R. Gaster, Bixia Zheng, Irina Lipov. *OpenCL Static C++ Kernel Language Extension*, 04 edition, 2012.

[23] Lars Sonnenschein. Algebraic approach to solve $t\bar{t}$ dilepton equations. *Phys. Rev. D*, 72:095020, Nov 2005.

[24] Alwall, Johan and Herquet, Michel and Maltoni, Fabio and Mattelaer, Olivier and Stelzer, Tim. MadGraph 5 : Going Beyond. *JHEP*, 1106:128, 2011.

[25] H. Murayama, I. Watanabe, and Kaoru Hagiwara. HELAS: HELicity amplitude subroutines for Feynman diagram evaluations. 1992.

[26] Priscila de Aquino, William Link, Fabio Maltoni, Olivier Mattelaer, and Tim Stelzer. ALOHA: Automatic Libraries Of Helicity Amplitudes for Feynman Diagram Computations. *Comput.Phys.Commun.*, 183:2254–2263, 2012.

[27] M.R. Whalley, D. Bourilkov, and R.C. Group. The Les Houches accord PDFs (LHAPDF) and LHAGLUE. 2005.

[28] Lai, Hung-Liang and Guzzi, Marco and Huston, Joey and Li, Zhao and Nadolsky, Pavel M. and others. New parton distributions for collider physics. *Phys.Rev.*, D82:074024, 2010.

[29] Torbjorn Sjostrand, Stephen Mrenna, and Peter Z. Skands. PYTHIA 6.4 Physics and Manual. *JHEP*, 0605:026, 2006.

[30] Peter Zeiler Skands. Tuning Monte Carlo Generators: The Perugia Tunes. *Phys.Rev.*, D82:074018, 2010.

[31] Matteo Cacciari, Gavin P. Salam, and Gregory Soyez. FastJet User Manual. *Eur.Phys.J.*, C72:1896, 2012.

[32] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[33] G. Peter Lepage. *vegas Documentation*, 2.1.4 edition, 2014.

Fast 3D tracking with GPUs for analysis of antiproton annihilations in emulsion detectors

Akitaka Ariga[1]

[1]Albert Einstein Center for Fundamental Physics, Laboratory for High Energy Physics, University of Bern, Sidlerstrasse 5, 3012 Bern, Switzerland

Fast 3D particle tracking has been a challenge in particle physics for a long time. In particular, the data rate from emulsion detectors is recently about 10-100 TB/day from one microscope. Real-time 3D volume processing and track reconstruction of such a huge data need a large amount of computation, in which the GPU technology plays an essential role. A fast 4π solid angle particle tracking based on GPU technology has been developed for the reconstruction of antiproton annihilations. The results in speed are compared between the multithread CPU processing and the multi-GPU one. By employing multiple GPUs, about two orders of magnitude faster processing has been achieved with an excellent tracking performance in comparison with a single-thread CPU processing.

1 Introduction

The reconstruction of particle trajectory (or *tracking*) is one of the fundamental subjects in experimental particle physics. Thanks to the development of detector technology, particle detectors are tend to employ more and more detector elements and to use higher dimensional data. The importance of 3D volume data processing is getting higher. Since 3D tracking requires a few orders of magnitude larger computation than 2D tracking, it has been essential to solve the computing issue with available computing resources.

Emulsion detector is one of the most classic detectors, however, physicists have attacked to establish the fast 3D volume processing with this detector since many decades.

The basic detector element of emulsion detector is a silver bromide crystal with a typical diameter of 200 nm. The crystals are uniformly distributed in the gelatin medium. An emulsion volume of 1 cc ($\simeq$ an emulsion film of 100 cm^2) consists of about 10^{14} crystals and each crystal works as an independent particle detector. The position resolution of emulsion detector is as good as 50 nm.

In the past, the readout of 10^{14} detectors and the reconstruction of the 3D tracks in emulsion detectors were impossible because of lack of technologies. Due to the difficulty of automation, the analysis had relied on human check on manual microscopes. Efforts to develop automated scanning system has started in late 70's. In order to read out the 3D information stored in emulsion films, scanning microscopes take tomographic images by moving focal plane, and then the tomographic images are processed in real-time manner. This is illustrated in Fig. 1. The scanning speed progressed many orders of magnitudes after several generations [1, 2, 3].
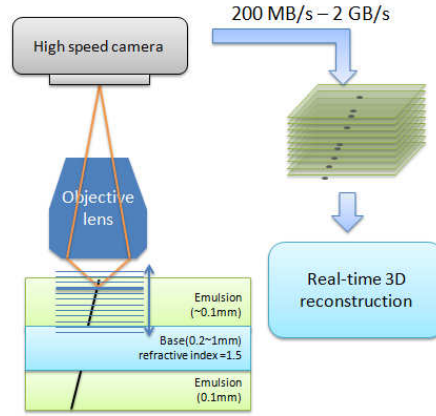
Figure 1: Schematic of scanning microscopes
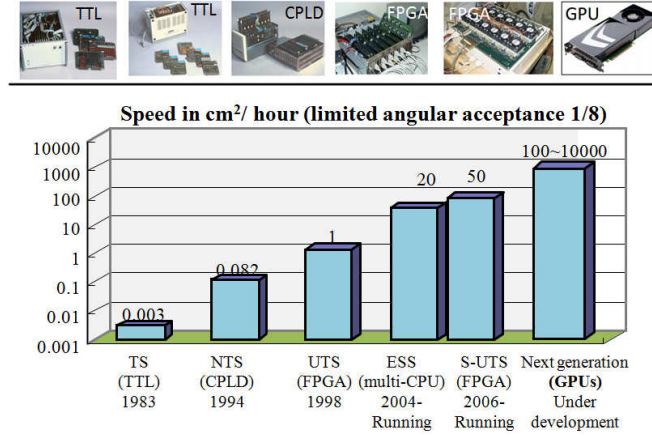
Figure 2: Evolution of scanning speed

The progress of speed in an unit of surface area scanned per hour is shown in Fig. 2, which features the different computing solutions available at those time. The data rate from camera is recently reached to of the order of 1 GBytes/s. To reconstruct 3D tracks in this amount of data, a powerful computing solution is mandatory.

GPU (Graphic Processing Unit) has been developed for the graphics, namely games. Yet, it is recently opened for general purpose computing. A GPU has a number of processors of the order of thousand and it can process tasks in parallel with the processors (parallel processing). An application of GPUs is highly efficent as a computing solution for the case that the algorithm can be parallelized, such as image processing or tracking.

On the other hand, emulsion detectors are intensively used in the field of antimatter physics, especially in the measurement of gravity for antimatter and matter-antimatter annihilation in the AEgIS experiment [4, 5, 6]. In such experimental conditions, the products from the annihilation vertex are emitted isotropically as shown in Fig. 3. However, the conventional
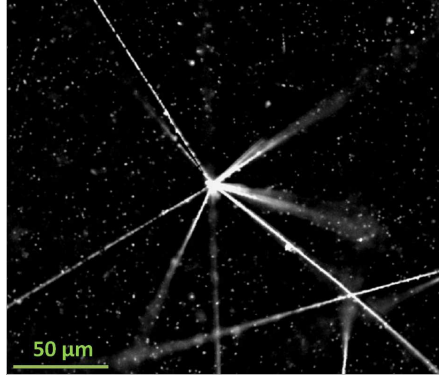
Figure 3: An antiproton annihilation vertex observed in the emulsion detector.

algorithms used in the systems in Fig. 2 reconstruct tracks in limited angular space of $\tan\theta < 0.5$ (θ - angle from the normal of emulsion surface) or, in other word, only 1/8 of full angular space. Thus, a new tracking algorithm which covers full angular space is awaited to achieve a high detection efficiency of antiproton annihilations.

A new algorithm is recently proposed and realized in [7] with a computing solution with GPUs, which is the basis of this article. In this paper, we shortly recall the main feature of the algorithm and report updated results.

2   Tracking algorithm and GPU implementation

The scanning microscope takes a set of tomographic images of 40 layers (*a view*) through an emulsion layer of 50-micron thick as illustrated in Fig. 1. An image size is 1280×1024 pixels, corresponding to 300 μm $\times$ 250 μm. Fig. 4 shows the flow of data processing to obtain 3D position of grains. The volume data (a) is filtered in 3D (b) and then grains in the view are recognized in 3D (c).

(a)   (b)   (c)

Figure 4: Processing of images. (a) Raw data image. (b) 3D-filtered image. (c) Recognized 3D grains marked with a red cross.

Any two combinations of grains in a limited volume define lines (or *seeds*) by connecting the two grains (Fig. 5 (a)). A cylinder along each seeds is defined and the number of grains in

the cylinder is evaluated (Fig. 5 (b)). If the number of grains is bigger than a preset threshold (e.g. 5 grains), the seed is classified as a tracks. The algorithm can cover all angular acceptance with this seed formation.
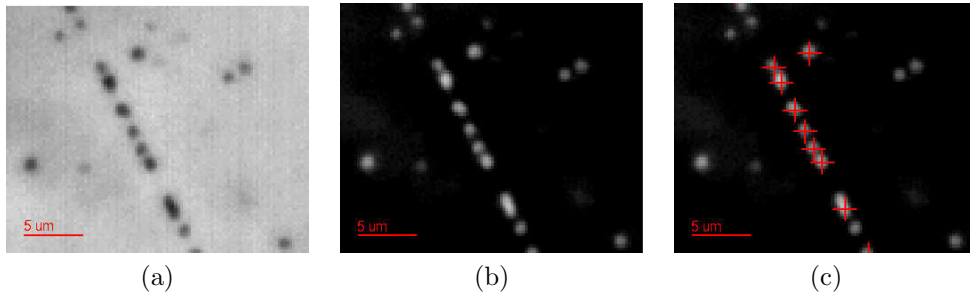

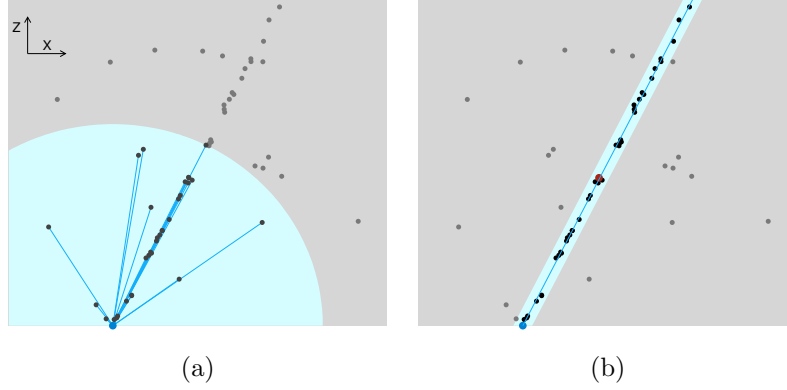(a)   (b)

Figure 5: Schematic of tracking algorithm. (a) For each grain (for example, the blue dot), seeds for a track (blue lines) are defined by looping for grains within a given distance. (b) For each seed (blue line formed by blue and red dots), the number of grains along the seed is counted. If the number of grains is larger than a preset threshold, one can define it as a track.

The 3D volume processing (Fig. 4) and the processing of seeds (Fig.5 (b)) account for a large part of computation time, however, they can be highly parallelized. These parts of algorithm are implemented with the GPU programming.

The further detail of the algorithm can be found in [7].

3   Performance test

The performance of tracking, especially in terms of speed, is compared between the CPUs programming and the GPUs programming. The code is written with C++ and with NVIDIA CUDA libraries (version 5.5) [8] combined with CERN ROOT libraries [9]. A dedicated processing server (Linux Mint 15) equipped with a recent CPU (i7-3930K, 3.2 GHz, 6 cores, 12 logical processors) with 3 state-of-the-art GPUs (NVIDIA GEFORCE TITAN, 837 MHz, 2688 cores, 6.144 GBytes on-board memory [10]) and with a fast memory (DDR3-2400) has been tested for our purpose.

Obtained tracking time is summarized in Table 1 for a single CPU thread case, with and without a GPU. The performance is also checked in two different samples that have different number of grains in the view. "Cosmic sample" was exposed to cosmic rays and it has about 5,000 grains per view. As underlined, the image filtering is the most dominant process for this case. On the other hand "Antiproton sample" was exposed to the antiproton beam at CERN AD and has about 11,000 grains per view. In this case the 3D tracking process holds the major processing time. For both cases, the processing time is remarkably accelerated by implementing GPU-based computing by factors of 40 and 26, respectively.

A multithread programming was then implemented on the basis of the single CPU thread programming. Each CPU thread is linked to one of the three GPU devices. One can use all the

	Cosmic sample (few grains)			Antiproton sample (more grains)		
Process	CPU-based (s/view)	GPU-based (s/view)	Gain	CPU-based (s/view)	GPU-based (s/view)	Gain
Image filtering	<u>3.388</u>	0.051	× 66	3.396	0.051	× 66
3D grain recognition	0.157	0.012	× 13	0.181	0.023	× 13
3D tracking	0.274	0.033	× 8.3	<u>6.999</u>	0.330	× 21
Total processing time	3.819	0.096	× 40	10.576	0.404	× 26

Table 1: Processing time between the CPU-based and the GPU-based programs with a single CPU thread.

GPUs by running several CPU-threads simultaneously. The result is shown in Figure 6. The processing speed (or frequency) is almost proportional to the number of GPUs. The dashed lines shows a fitted tendency with the formula given in the figure. It is well understandable by taking account the overhead time in the process. The frequency can reach up to 15 Hz, which is far enough for the data taking limitation of 5 Hz with the conventional mechanical hardware for [3]. The gain in time for the antiproton sample between the single CPU thread process (0.096 Hz) and the multi-GPU process (8.7 Hz, 3 GPU and 15 CPU threads) is calculated to be 91.

$$freq = \frac{1}{t_{overhead} + \frac{t_{process}}{N_{threads}}}$$
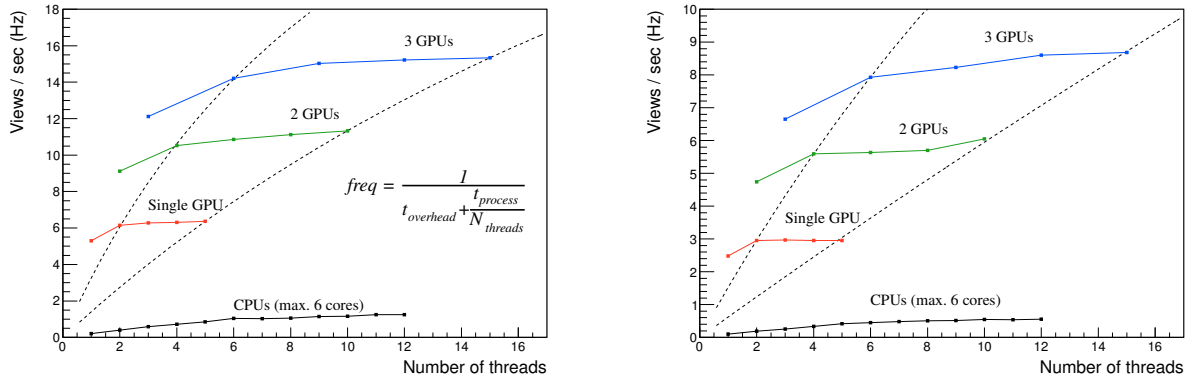
Figure 6: Comparison of scanning speed as a number of views processed per sec, as a function of number of threads. Left: cosmic-ray sample (less grains), right: antiproton sample (more grains).

4 Summary

Fast 3D particle full-angle tracking has been for a long time a challenge when using nuclear emulsion detectors as high accuracy 3D tracking devices. Advances in the GPU technology have opened the way for the actual realization of real-time fast processing.

The proposed algorithm with full angular acceptance, required for the analysis of antiproton interactions, needs a large computing power compared to the conventional algorithms with a

narrower angular acceptance. Nevertheless, the processing speed reached to maximum 15 Hz, which is sufficiently high for the current data taking limitation of 5 Hz.

This new tracking algorithm can be employed by any applications using nuclear emulsion detectors which needs a fast 4π tracking, as the AEgIS experiment at CERN which needs a high detection efficiency for antiproton annihilations.

The development of tracking system will further be extended for the next generation data aquisition system which has about 2 GBytes/s output data rate (currently 0.25 GBytes/s).

Acknowledgments

References

[1] K. Niwa, K. Hoshino and K. Niu, *Auto scanning and measuring system for the emulsion chamber*, in the proceedings of the International Cosmic ray Symposium of High Energy Phenomena, Tokyo, Japan (1974), see pag. 149.
S. Aoki et al., *The fully automated emulsion analysis system*, *Nucl. Instrum. Meth.* **B 51** (1990) 466.
T. Nakano, *Automatic analysis of nuclear emulsion*, Ph.D. Thesis, Nagoya University, Japan (1997).

[2] K. Morishima and T. Nakano, *Development of a new automatic nuclear emulsion scanning system, S-UTS, with continuous 3D tomographic image read-out*, *JINST* **5** *2010 P04011*

[3] N. Armenise et al., *High-speed particle tracking in nuclear emulsion by last-generation automatic microscopes*, *Nucl. Instrum. Meth.* **A 551** (2005) 261;

[4] G. Drobychev et al., *Proposal for the AEGIS experiment at the CERN antiproton decelerator*, available at spsc-2007-017.pdf;
AEGIS PROTO collaboration, *Proposed antimatter gravity measurement with an antihydrogen beam*, *Nucl. Instrum. Meth.* **B 266 (2008) 351**.

[5] C. Amsler, A. Ariga, T. Ariga, S. Braccini, C. Canali, et al., *A new application of emulsions to measure the gravitational force on antihydrogen*, *JINST* **8** P02015 (2013) [arXiv:1211.1370].

[6] AEgIS collaboration, S. Aghion et al., *Prospects for measuring the gravitational free-fall of antihydrogen with emulsion detectors*, *JINST* **8** P08013 (2013).

[7] A Ariga and T Ariga, *Fast 4π track reconstruction in nuclear emulsion detectors based on GPU technology*, *JINST* **9** P04002 (2014).

[8] NVIDIA CUDA web page: http://www.nvidia.com/cuda

[9] CERN ROOT, *A Data Analysis Framework*, web page: http://root.cern.ch/

[10] NVIDIA GEFORCE TITAN web page: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan

GPUs in gravitational wave data analysis

G. Debreczeni[1]

[1] Wigner Research Centre for Physics of the Hungarian Academy of Sciences

Gravitational wave physics is in the doorstep of a new, very exciting era. When the advanced Virgo and advanced LIGO detectors will start their operation, there will be considerable probability of performing the first direct detection of gravitational waves predicted almost 100 years ago by Einsten's theory of General Relativity. However the extraction of the faint signal from the noisy measurement data is a challanging task - and due to the high arithmetic density of the algorithms - requires special methods and their efficient, sophisticated implementation on high-end many-core architectures such as GPUs, APUs, MIC and FPGAs.. The operation-level paralellizability of the algorithms executed so far on single CPU core results - has already resulted - in nearly 2 order of magnitude speedup of the analysis and can directly be trasnlated to detector sensitivity! As such, the developed and applied computational algorithms can be regarded as part of the instrument, thus giving thorough meaning to the notion of "e-detectors". In this talk we will shortly present and discuss the many-core GPU algorithms used in gravitational wave data analysis for extracting the continous waves emitted by isolated, spinning neutron stars and the chirp-like signals of binary NS-NS or NS-BH systems with an outlook for future possibilities.

Contribution not received.

Accelerated Neutrino Oscillation Probability Calculations and Reweighting on GPUs

Richard Graham Calland[1]

[1]University of Liverpool, Department of Physics, Oliver Lodge Bld, Oxford Street, Liverpool, L69 7ZE, UK

Neutrino oscillation experiments are reaching high levels of precision in measurements, which are critical for the search for CP violation in the neutrino sector. Inclusion of matter effects increases the computational burden of oscillation probability calculations. The independency of reweighting individual events in a Monte Carlo sample lends itself to parallel implementation on a Graphics Processing Unit. The library Prob3++ was ported to the GPU using the CUDA C API, allowing for large scale parallelized calculations of neutrino oscillation probabilities through matter of constant density, decreasing the execution time of the oscillation probability calculations by 2 orders of magnitude, when compared to performance on a single CPU core. Additionally, benefit can be realized by porting some systematic uncertainty calculations to GPU, especially non-linear uncertainties evaluated with response functions. The implementation of a fast, parallel cubic spline evaluation on a GPU is discussed. The speed improvement achieved by using the GPU calculations with the T2K oscillation analysis is also noted.

1 Neutrino Oscillation Probability for Long Baseline Experiments

It is established that neutrinos exhibit oscillation between flavour states [1][2]. The standard formalism describes this phenomena using a unitary transition matrix (PMNS) to compute the probability of a neutrino ν_α to change to ν_β. A neutrino travelling a distance L (km) from its source has a probability to change flavour defined as

$$P(\nu_\alpha \rightarrow \nu_\beta) = \delta_{\alpha\beta} - 4\sum_{i>j} Re(U_{\alpha i}^* U_{\beta i} U_{\alpha j} U_{\beta j}^*) \sin^2(\frac{\Delta m_{ij}^2 L}{4E})$$
$$+2\sum_{i>j} Im(U_{\alpha i}^* U_{\beta i} U_{\alpha j} U_{\beta j}^*) \sin(\frac{\Delta m_{ij}^2 L}{2E}) \tag{1}$$

where E is the neutrino energy, $U_{flavour,mass}$ is the mixing matrix, Δm_{ij}^2 is the difference between mass states i,j and $\delta_{\alpha\beta}$ is the Kronecker delta function. However, when the neutrino propagates through matter, an extra potential is added to the equation which manifests from the forward scattering of ν_e on the ambient electrons in matter. This extra potential term requires that the mass matrix of the Hamiltonian be re-diagonalized in order to find the eigenstates
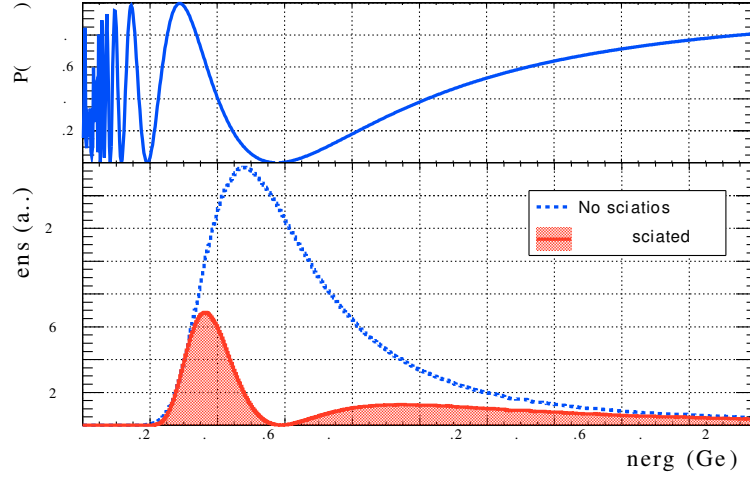
Figure 1: Effect of neutrino oscillation on a mock neutrino energy spectrum. Top plot shows the ν_μ survival probability and the bottom plot shows the mock energy spectrum with and without the survival probability applied.

in matter. The extra computation required to calculate matter effects can be costly, however there is an analytical solution prescribed in [3] which is used in the work presented here.

2 The Tokai-to-Kamioka Experiment

The Tokai-to-Kamioka (T2K) experiment [4] is a second generation long baseline neutrino oscillation experiment, designed to use an intense beam of neutrinos to make precision measurements of the θ_{23} mixing angle, and to look for θ_{13}. The experiment, which is located in Japan, is comprised of 3 sections. The first is located on the east coast of Japan, where the J-PARC accelerator produces a neutrino beam composed of mainly ν_μ by colliding protons onto a graphite target. This beam is measured at the second stage, by a near detector (ND280) situated 280 m from the neutrino source. ND280 can make characterisation measurements of the beam before it has the chance to undergo neutrino oscillation. This stage is important because it can provide valuable constraints on the uncertainties in the neutrino beam flux and cross section models. The final stage, takes place 295 km away at the Super-Kamiokande (SK) 50 kt water Cherenkov detector, where neutrino oscillation parameters are measured by looking for a disappearance of ν_μ-like events and an appearance of ν_e-like events.

Due to the setup of the experiment, there are multiple samples from multiple detectors, which leads to a relatively complicated oscillation analysis involving many systematic uncertainties.

3 Event-by-event Reweighting

Neutrino oscillation analyses in T2K are performed using a large sample of Monte Carlo (MC) simulated events inside the near and far detectors to construct a PDF. The number of MC

events is large due to the multiple interaction modes and reconstruction effects. The events are produced assuming no neutrino oscillation has taken place, so the effect of neutrino oscillation must be calculated as a weight for these MC events. This is visualized in Figure 1. In order to infer the most probable values of the oscillation parameters, the PDF must be reweighted in order to find the response of the detector simulation to varying values of oscillation parameters. This reweighting can be done in two ways. The first, and simplest, is to construct templates for the detector MC. This is essentially using a histogram with a suitable binning, filling the histogram with the MC events and then using the bin centre to calculate a weight for the bin. An alternative method is to keep all MC event information inside memory, and calculating a weight for each MC event. These events can then be binned using the weights to compute a binned likelihood in the same way as using templates. The obvious difference is that the event-by-event method requires orders of magnitudes more calculations, because instead of doing computations per bin, we are instead doing them per event. The advantages of using this method, if one can overcome the computational challenges, are that there is no additional loss of information by compiling events into bins; the shape information of the PDF is retained inside the bin as described in Figure 2. In recent binning schemes used for T2K PDFs, it was found that the difference in predicted number of events from reweighting differs by 1-2% between template and event-by-event methods.

Another advantage of the event-by-event method is in the case where one constructs multiple PDFs from the same set of MC events, as is the case for T2K, where different event topologies are selected from the data based on a few selection criteria. As detector systematic uncertainties are varied, this can cause events to move between sample selections. This systematic effect can easily be modelled using the event-by-event method, but it is difficult to do so with a template method due to the loss of MC information about the event category of each event. This, along with parameters that may shift the kinematics of an event and cause the event to shift between bins in a histogram, is a strong justification to retain all the relevant information about MC events and reweight each event individually.

4 Calculation on GPU

Using the event-by-event method to reweight PDFs (and thus calculate the likelihood) has the consequence of increasing the number of calculations involved by orders of magnitude. In T2K neutrino oscillation analyses, the two largest contributions to CPU time are the evaluation of the oscillation probability including matter effects, and the evaluation of cubic splines that are used to encode the non-linearity of the cross section model on the PDF. These two tasks were offloaded to a GPU using the CUDA toolkit version 5. For the oscillation probability calculation, the library Prob3++ [5] is used to calculate these probabilities on CPU. This library was ported to run on the GPU [6]. The structure of this calculation is as follows. First, the mass and mixing matrices were computed on CPU and copied to GPU memory. Then, an array of neutrino energies taken from the MC was copied to the GPU memory. An array of the same length as the energy array was allocated on GPU memory to store the results. A GPU algorithm (kernel) was then executed with each GPU thread operating on an element of the neutrino energy array. The kernel calculates a weight using the event energy and information from the mass and mixing matrices. The resulting weight is saved to the corresponding element in the weights array, which is copied back to CPU memory when the kernel has finished processing. The time taken to perform this oscillation probability calculation over varying numbers of events in a standalone
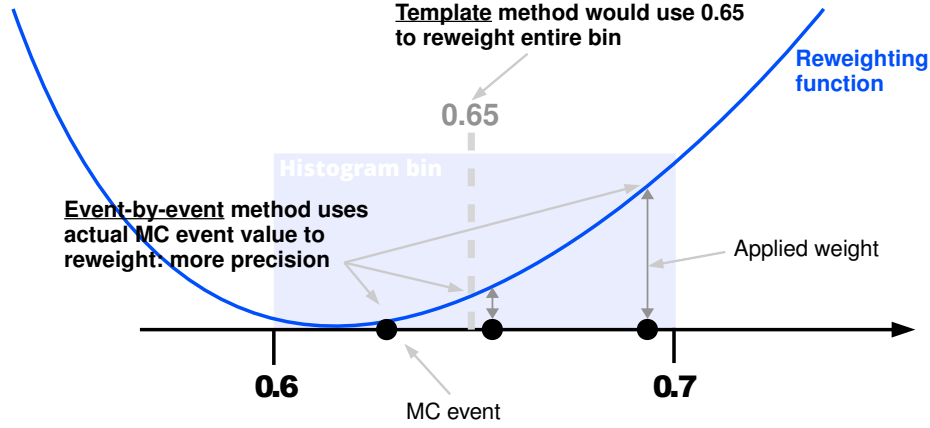
Figure 2: A comparison of template and event-by-event reweighting methods. When calculating weights using a reweighting function, the template method uses the bin centre to calculate a single weight for the whole bin. In contrast, the event-by-event method calculates a weight for each event, using the events energy value. The template method loses information about the shape of the reweighting function, whereas the event-by-event method retains it.

program was studied, with results shown in Figures 3 and 4.

In addition to oscillation probability reweighting, the simulated MC events are also reweighted according to a neutrino cross section model. This model is encoded with cubic splines to efficiently describe the non-linearity of each parameter response without having to rerun the time consuming simulation for every iteration of the fit. When cross section parameters are varied, each response function must be evaluated to produce a new weight, on a per neutrino interaction mode, per event basis. To improve the performance, the array of spline objects used in the CPU code were reformatted into a structure of arrays to better suit the GPU architecture. This structure is on the order of 1 Gb in size, and was copied only once to the GPU as a read-only resource. At each iteration of the analysis, the new parameter values were copied to GPU memory, and were used by the kernel to evaluate a single spline per GPU thread across multiple threads simultaneously. Finally, the resulting weights of the splines were copied back to CPU memory, where they were applied to the events during the construction of the PDF. When compared to the performance of the original array of spline objects structure, the GPU implementation saw a factor of 20 times speed up in a standalone benchmark. This benchmark consisted of evaluating a large number of splines both sequentially (CPU) and simultaneously (GPU) to compare execution times.

A validation study was performed for both the oscillation reweighting and the spline evaluation GPU algorithms. It was found that the relative difference between GPU and CPU versions was on the order of 10^{-12} and 10^{-6} for oscillation reweighting and spline evaluation respectively; both within acceptable tolerance for this application.

The benchmarks found in Figures 3 and 4 were executed on an Intel Xeon E5640 quad-core processor clocked at 2.67 Ghz, using an NVIDIA M2070 GPU which has 448 cores clocked at
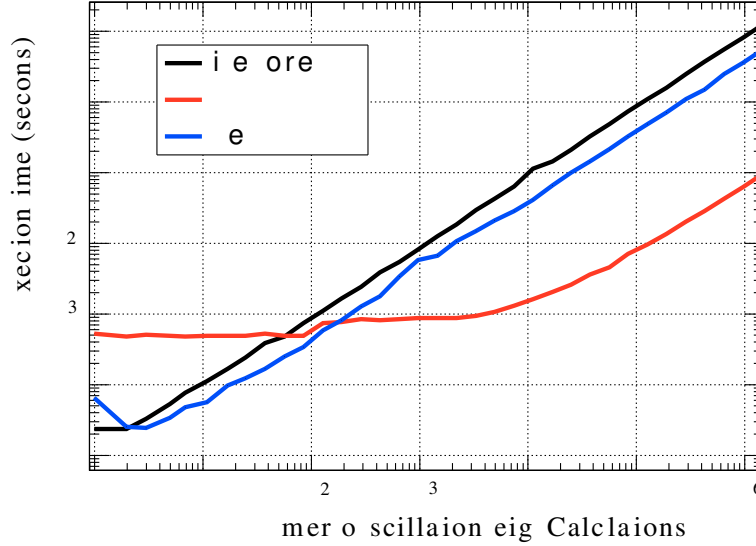
Figure 3: Time taken to evaluate the oscillation probabilities of batches of events, as a function of batch size. Single core CPU time is compared with the CUDA implementation, and also a multi-threaded version using OpenMP.

1.15 GHz. The code was compiled using the CUDA 5 toolkit and gcc version 4.6.3 with the -O2 optimization flag enabled. The OpenMP benchmark was restricted to 4 cores, using the same hardware as the single core benchmark.

5 Conclusion

When offloading both oscillation probability calculations with matter effects, and evaluation of response functions to GPU, the T2K neutrino oscillation analysis saw approximately 20 times speed up in total execution time compared to running everything on a single CPU core. There is much scope to further improve the acceleration of the T2K neutrino oscillation analysis with GPUs. The most obvious way is to perform all reweighting operations on the GPU, instead of offloading two components. However, this may require significant reworking of existing analysis code. Since the generation of the results in this study, the oscillation probability code has been improved to fit the mass and mixing matrices inside constant memory on the GPU. This increased the maximum observed speed increase factor in the standalone benchmark from 130 to 180.

6 Acknowledgments

The author would like to acknowledge the SES (Science Engineering South) Centre for Innovation service (CFI) Emerald HPC cluster, along with the HEP computing staff at the University
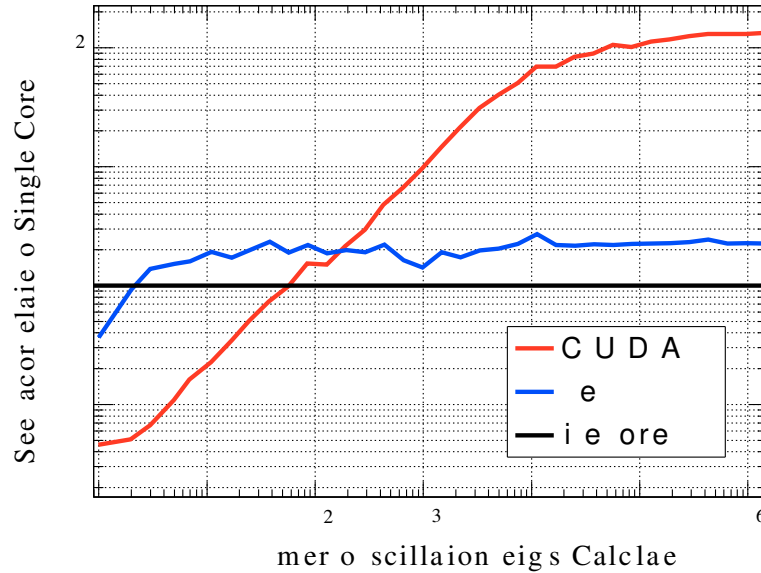
Figure 4: The speedup time relative to single core execution, which shows the speed increase factor for the GPU and OpenMP implementations. The largest speed increase factor seen between CPU and CUDA implementations was 130.

of Liverpool for their support.

References

[1] Y. Fukuda et al. Evidence for oscillation of atmospheric neutrinos. *Phys. Rev. Lett.*, 81:1562–1567, Aug 1998.

[2] Q. R. Ahmad et al. Measurement of the Rate of $\nu_e + d \rightarrow p + p + e^-$ Interactions Produced by ^{8}B Solar Neutrinos at the Sudbury Neutrino Observatory. *Physical Review Letters*, 87(7):071301, August 2001.

[3] V. Barger, K. Whisnant, S. Pakvasa, and R. J. N. Phillips. Matter effects on three-neutrino oscillations. *Phys. Rev. D*, 22:2718–2726, 1980.

[4] T2K Collaboration, K. Abe, N. Abgrall, H. Aihara, Y. Ajima, J. B. Albert, D. Allan, P.-A. Amaudruz, C. Andreopoulos, B. Andrieu, and et al. The T2K experiment. *Nuclear Instruments and Methods in Physics Research A*, 659:106–135, December 2011.

[5] R. Wendell. Prob3++ software for computing three flavor neutrino oscillation probabilities. `http://www.phy.duke.edu/~raw22/public/Prob3++/`, 2012.

[6] RG Calland, AC Kaboth, and D Payne. Accelerated event-by-event neutrino oscillation reweighting with matter effects on a gpu. *Journal of Instrumentation*, 9(04):P04016, 2014.

Sampling secondary particles in high energy physics simulation on the GPU

S.Y. Jun[1]

[1] Fermilab, USA

We present a massively parallel application for sampling secondary particles in high energy physics (HEP) simulation on a Graphics Processing Unit (GPU). HEP experiments primarily uses the Geant4 toolkit (Geant4) to simulate the passage of particles through a general-purpose detector, which requires intensive computing resources due to the complexity of the geometry as well as physics processes applied to particles copiously produced by primary collisions and secondary interactions. The combined composition and rejection methods to sample secondary particles often used in Geant4 may not be suitable for optimal performance of HEP events simulation using recent hardware architectures of accelerated or many-core processors owing to the stochastic nature of the Monte Carlo technique. An alternative approach based on a discrete inverse cumulative probability distribution is explored to minimize the divergence in thread level parallelism as well as to vectorize physics processes for spatial locality and instruction throughput. The inverse cumulative distribution of the differential cross section associated with each electromagnetic physics process is tabulated based on algorithms excerpted from Geant4 and a simple random sampling technique with a linear interpolation is implemented for GPU. Validation and performance evaluation with the alternative technique compared to the conventional composition and rejection method both on GPU and CPU are presented.

Contribution not received.

Implementation of a Thread-Parallel, GPU-Friendly Function Evaluation Library

M.D. Sokoloff[1]

[1] University of Cincinnati, USA

GooFit is a thread-parallel, GPU-friendly function evaluation library, nominally designed for use with the maximum likelihood fitting program MINUIT. In this use case, it provides highly parallel calculations of normalization integrals and log (likelihood) sums. A key feature of the design is its use of the Thrust library to manage all parallel kernel launches. This allows GooFit to execute on any architecture for which Thrust has a backend, currently, including CUDA for nVidia GPUs and OpenMP for single- and multi-core CPUs. Running on an nVidia C2050, GooFit executes as much as 300 times more quickly for a complex high energy physics problem than does the prior (algorithmically equivalent) code running on a single CPU core. This talk will focus on design and implementation issues, in addition to performance.

Contribution not received.

M. D. Sokoloff

Discovering Matter-Antimatter Asymmetries with GPUs

Stefanie Reichert[1] on behalf of the LHCb collaboration

[1]School of Physics and Astronomy, The University of Manchester, Oxford Road, Manchester, M13 9PL, United Kingdom

The search for matter-antimatter asymmetries requires highest precision analyses and thus very large datasets and intensive computing. This contribution discusses two complementary approaches where GPU systems have been successfully exploited in this area. Both approaches make use of the CUDA THRUST library which can be used on supported GPUs. The first approach is a generic search for local asymmetries in phase-space distributions of matter and antimatter particle decays. This powerful analysis method has never been used to date due to its high demand in CPU time. The second approach uses the GOOFIT framework, which is a generic fitting framework that exploits massive parallelisation on GPUs.

1 Introduction

In the neutral charm meson system, the mass eigenstates or physical particles $|D_{1,2}\rangle$ with masses $m_{1,2}$ and widths $\Gamma_{1,2}$ are a linear combination of the flavour eigenstates $|D^0\rangle$ and $|\bar{D}^0\rangle$ which govern the interaction. The linear combination $|D_{1,2}\rangle = p|D^0\rangle \pm q|\bar{D}^0\rangle$ depends on complex coefficients q, p satisfying the normalisation condition $|q|^2 + |p|^2 = 1$. The flavour eigenstates are subject to matter-antimatter transitions, so-called mixing, through box diagrams as illustrated in Fig. 1.
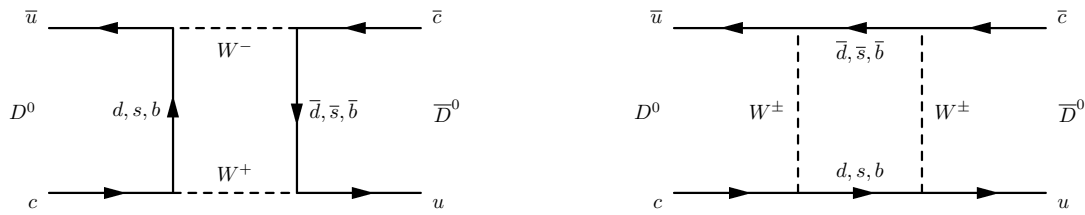
Figure 1: (Left) Box diagram for $D^0 - \bar{D}^0$ mixing via intermediate $W^\pm$ bosons and (right) via intermediate quarks.

The $D^0 - \bar{D}^0$ oscillation is driven by a non-zero difference in masses and widths of the mass eigenstates $\Delta m = m_2 - m_1$ and $\Delta\Gamma = \Gamma_2 - \Gamma_1$, respectively. The mixing parameters x and y are defined as $x = \Delta m/\Gamma$ and $y = \Delta\Gamma/(2\Gamma)$ where Γ denotes the width average $\Gamma = (\Gamma_1 + \Gamma_2)/2$ of

the mass eigenstates. The mass eigenstates are eigenstates to the charge-parity (CP) operator if $CP|D_{1,2}\rangle = \pm|D_{1,2}\rangle$ is fulfilled. Three different types of CP violation can be distinguished. CP violation in decay is implied if the decay amplitude of a D^0 to a final state f and the amplitude of the charge conjugated $\bar{D}^0 \to \bar{f}$ decay differ. In case of $|q/p| \neq 1$, the CP operator transforms the mass eigenstates to states which are not CP eigenstates. This phenomenon is know as CP violation in mixing. Furthermore, CP violation in decay and mixing can interfere. In the charm sector, mixing is well established, while no evidence of CP violation has been measured [1].

The mixing parameters in the charm sector are of the order $\mathcal{O}(10^{-3})$ due to the small mass and width differences Δm and $\Delta \Gamma$ of the mass eigenstates. Therefore, measurements of the charm mixing parameters and searches for CP violation are experimentally challenging and are required to be performed on large datasets. The datasets recorded at the LHCb experiment [2] used for such measurements contain typically several millions of events. Most analysis methods are CPU-expensive but are parallelisable and benefit from the massive parallelisation and speed-up provided by the usage of GPUs. In particular, the presented methods rely on the CUDA THRUST library [3] providing similar functionalities as the C++ Standard Template Library (STL). Functions available in STL such as reduce are also part of the CUDA THRUST library which also enables portability between GPUs and multicore CPUs.

2 Energy test for $D^0 \to \pi^+\pi^-\pi^0$ using the CUDA Thrust library

The energy test [4] is an unbinned model-independent statistical method to search for time-integrated CP violation in $D^0 \to \pi^+\pi^-\pi^0$ decays (charge conjugate decays are implied unless stated otherwise). The method relies on the comparison of two D^0 and $\bar{D}^0$ flavour samples and is sensitive to local CP asymmetries across phase-space. The phase-space is spanned by the invariant mass squared of the daughter particles, e.g. by $m^2(\pi^+\pi^0)$ and $m^2(\pi^-\pi^0)$. The $D^0 \to \pi^+\pi^-\pi^0$ decay is required to originate from a $D^{*+} \to D^0\pi^+$ decay. By determining the charge of the soft pion of the $D^{*+} \to D^0\pi^+$ decay, the data are split in two independent samples containing either D^0 or $\bar{D}^0$ candidates. For both samples, a test statistic T is computed as

$$T \approx \sum_i^n \sum_{j>i}^n \frac{\psi\left(\Delta\vec{x}_{ij}\right)}{n^2 - n} + \sum_i^{\bar{n}} \sum_{j>i}^{\bar{n}} \frac{\psi\left(\Delta\vec{x}_{ij}\right)}{\bar{n}^2 - \bar{n}} - \sum_i^n \sum_j^{\bar{n}} \frac{\psi\left(\Delta\vec{x}_{ij}\right)}{n\bar{n}}, \tag{1}$$

where $n(\bar{n})$ is the size of the $D^0(\bar{D}^0)$ sample, $\psi\left(\Delta\vec{x}_{ij}\right)$ is a metric function and $\Delta\vec{x}_{ij}$ is the distance in the 3-dimensional phase-space between events i and j. A Gaussian, given by

$$\psi\left(\Delta\vec{x}_{ij}\right) = e^{-\Delta\vec{x}_{ij}^2/2\sigma^2}, \tag{2}$$

is chosen as metric function. The width σ is tunable and indicates the radius in which the method is sensitive to local phase-space asymmetries. Thus, σ should be larger than the resolution of $\Delta\vec{x}_{ij}$ and small enough to avoid a dilution of locally varying asymmetries. The three terms in Eq. 1 are the average distance in the D^0, $\bar{D}^0$ and mixed sample. In case of large CP asymmetries, the T-value increases. The T-value computation uses *thrust::transform_reduce*

from the CUDA THRUST library to compute the sums in 1. The function *thrust::transform_reduce* performs a reduction operation, e.g. an addition, after an application of an operator to each element of an input sequence. In this analysis, for each point, the Gaussian metric function having been initialised with the position of event i is evaluated with respect to the position of event j.

The measured T-value does not indicate the presence of CP violation on its own and needs to be compared with the hypothesis of no CP violation. So-called permutation samples are generated where the flavour of each D^0 and $\bar{D}^0$ candidate is reassigned randomly. The permutation samples then reflect the no CP violation hypothesis. The T-value of each permutation sample is calculated resulting in a distribution of T-values reflecting the no CP violation hypothesis. A p-value for the no CP violation hypothesis is calculated as the fraction of permutation T-values greater than the measured T-value. In case that the measured T-value lies outside the range of the permutation T-value distribution, the latter is fitted with a generalised extreme value function [5, 6]

$$f(x; \mu, \sigma, \xi) = N \left[1 + \xi \left(\frac{x - \mu}{\sigma} \right) \right]^{(-1/\xi)-1} \exp \left\{ - \left[1 + \xi \left(\frac{x - \mu}{\sigma} \right) \right]^{-1/\xi} \right\}, \qquad (3)$$

where N is the normalisation, μ the location, σ the scale and ξ the shape parameter. The p-value is then defined as the fraction of the integral above the measured T-value. A statistical uncertainty on the p-value is obtained by propagating the uncertainties on the fit parameters or in case of counting as a binomial standard deviation. On a sample of 700,000 candidates, which is of the same order as the analysed data set of $663,000$ selected events, the energy test with a single permutation requires approximately 10 hours of CPU time whereas the GPU implementation of the energy test runs in 11 minutes. To obtain a reliable p-value, around 1000 permutation samples reflecting the no CP violation hypothesis are required. The results of the energy test on a data sample of $\mathcal{L}_{\text{int}} = 2\,\text{fb}^{-1}$ recorded at a centre-of-mass energy of $8\,\text{TeV}$ are summarised in Table 1 for various metric parameter values. Monte-Carlo studies indicate that $\sigma = 0.3\,\text{GeV}^2$ yields the best sensitivity. The permutation T-value distribution and the measured T-value are illustrated in Fig. 2 along with the visualisation of the asymmetry significance which is obtained by assigning an asymmetry significance to each event similar to the T-value extraction. The analysed data set are found to be consistent with a no CP violation hypothesis at a probability of $(2.6 \pm 0.5)\%$ [7].

$\sigma\ [\text{GeV}^2]$	p-value
0.2	$(4.6 \pm 0.6) \times 10^{-2}$
0.3	$(2.6 \pm 0.5) \times 10^{-2}$
0.4	$(1.7 \pm 0.4) \times 10^{-2}$
0.5	$(2.1 \pm 0.5) \times 10^{-2}$

Table 1: p-values for various metric parameter values. The p-values are obtained with the counting method [7].
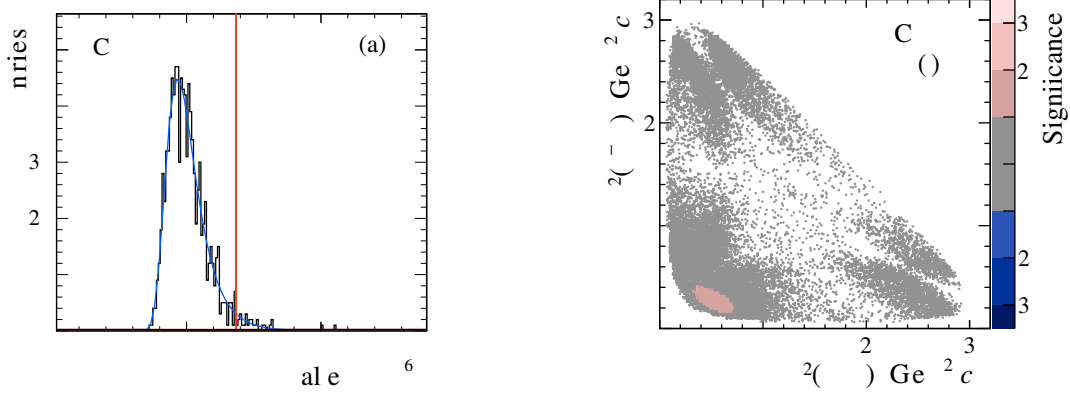
Figure 2: (Left) Permutation T-value distribution and the resulting the fit function. The measured T-value is indicated by the vertical line. (Right) Visualisation of local asymmetry significances. The negative (positive) asymmetry significance is set for the D^0 candidates having negative (positive) contribution to the measured T-value, respectively. The results are given for $\sigma = 0.3\,\mathrm{GeV}^2$ [7].

3   Time-dependent amplitude analysis of $D^0 \rightarrow K_S^0 \pi^+ \pi^-$ with GooFit

A time-dependent amplitude analysis of $D^0 \rightarrow K_S^0 \pi^+ \pi^-$ decays grants direct access to the mixing parameters x and y and allows to search for CP violation in mixing. The three-body decay $D^0 \rightarrow K_S^0 \pi^+ \pi^-$ is treated as a two-body decay $D^0 \rightarrow Rc$ through an intermediate resonance $R \rightarrow ab$ with the amplitude

$$\begin{aligned}
\mathcal{M}_R = \quad & Z(J, L, l, \vec{p}, \vec{q}) B_L^{R \rightarrow ab}(|\vec{q}|) \cdot \\
& \mathcal{T}_R(m_{ab}) B_L^{D \rightarrow Rc}(|\vec{p}|),
\end{aligned} \tag{4}$$

where the angular distribution of the final state particles is denoted by $Z(J = 0, L, l, \vec{p}, \vec{q})$ [8]. For a D^0 meson, $J = 0$, L and l are referring to the orbital angular momentum between R and c and between a and b, respectively. The momenta of c and a in the rest frame of the intermediate resonance R are denoted by $\vec{p}$ and $\vec{q}$. The Blatt-Weißkopf barrier factors [9, 10] are denoted by $B_L(q)$. The dynamical function $\mathcal{T}_R$ is the propagator for the chosen line shape of the resonance, e.g. the Gounaris-Sakurai propagator [11]. A model-dependence of the analysis arises from the choice of intermediate resonances and their line shapes. The two most common models are the so-called isobar model where the line shapes of the resonances in Table 2. are relativistic Breit-Wigners with exception of the $\rho(770)$ which is modelled by a Gounaris-Sakurai shape. An alternative model where the $K_S^0 \pi^\pm$ S-waves are described by the LASS parameterisation [12] and the $\pi^+ \pi^-$ S-wave is formulated with the K-matrix algorithm LASS parameterisation and K-matrix algorithm preserve unitarity by construction in opposition to the unitarity-violating Breit-Wigner line shapes.

The analysis uses the GooFit [14] library to perform the time-dependent Dalitz-plot (Tddp) amplitude analysis fit. The GooFit package is a parallel fitting framework implemented in CUDA and using the Thrust library. GooFit can be run under OpenMP or on GPUs supporting CUDA. In this framework, the most commonly used line shapes for resonance are available, e.g. Breit-Wigner, Gounaris-Sakurai and the LASS parameterisation as well as the Blatt-Weißkopf barrier factors and angular functions. The K-matrix algorithm to describe the $\pi^+\pi^-$ S-wave is under development. In GooFit, the class *TddpPdf* is dedicated to time-dependent amplitude analyses. The user defines the resonances and line shapes entering the probability density function which is then build in *TddpPdf* and is used as fit model to extract the mixing parameters x and y. The *TddpPdf* class requires the invariant mass squared of two daughter pair combinations, the D^0 decay time and its uncertainty as well as

Resonance R	Mass [GeV]	Width [GeV]	Spin
$R \to \pi^+\pi^-$			
$\rho(770)$	0.776	0.146	1
ω	0.783	0.008	1
$f_0(980)$	0.975	0.044	0
$f_0(1370)$	1.434	0.173	0
$f_2(1270)$	1.275	0.185	2
σ_1	0.528	0.512	1
σ_2	1.033	0.099	0
$R \to K_S^0\pi^+$			
$K^*(892)^+$	0.894	0.046	1
$K_0^*(1430)^+$	1.459	0.175	0
$K_2^*(1430)^+$	1.426	0.099	2
$R \to K_S^0\pi^-$			
$K^*(892)^-$	0.894	0.046	1
$K_0^*(1430)^-$	1.459	0.175	0
$K_2^*(1430)^-$	1.426	0.099	2
$K^*(1680)^-$	1.677	0.205	1
non-resonant $K_S^0\pi^+\pi^-$			

Table 2: Intermediate resonances contributing to the isobar model with their mass, width and spin. The numbers and the model are taken from the D0mixDalitz model as implemented in EvtGen [13].

the event number as input. In addition to the amplitude model, background components, efficiencies and resolutions can be included in the fit. The framework allows to veto certain regions in phase-space and the blinding of results. The Thrust library is used to compute the amplitudes of the intermediate resonances and the matrix elements contributing to the logarithm of the likelihood. In addition, the computation of the normalisation integral of the fit model relies heavily on *thrust::transform, thrust::transform_reduce, thrust::make_zip_iterator* and *thrust::make_tuple*. For example, *thrust::transform_reduce* is used to loop over an input sequence created by *thrust::make_zip_iterator* which yields a sequence of tuples from multiple tuples returned by *thrust::make_tuple*. For each element of this sequence, the phase-space integral is computed which is then added to a complex number to yield the total phase-space integral. Due to the parallelisation with the methods from the Thrust library, the fit runs over several hundred thousand or million of events in minutes instead of hours. In Fig. 3, the results of an amplitude analysis fit of a toy data sample generated according to the isobar model with $x = y = 0.005$ is shown.
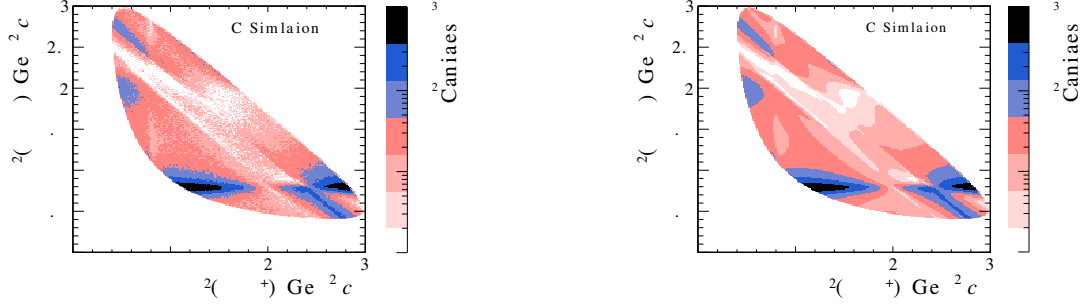
Figure 3: (Left) Toy data and (right) fit model of a toy data sample reflecting the 2012 data taking conditions of LHCb including an efficiency parametrisation. The sample is generated according to the isobar model with $x = y = 0.005$.

4   Conclusion

Searches for CP violation and measurements of the mixing parameters in the charm sector require large statistics which will keep increasing in the future. The search for time-integrated CP violation in $D^0 \to \pi^+\pi^-\pi^0$ decays at LHCb with the energy test is the first analysis at LHCb exploiting the potential provided by GPUs. The energy test could be applied to data for the first time due to the massive parallelisation and reduction of the computing time. A measurement of the mixing parameters x and y with a time-dependent amplitude analysis of $D^0 \to K_S^0\pi^+\pi^-$ decays is currently ongoing with the GOOFIT package.

References

[1] Y. Amhis et al. Averages of B-Hadron, C-Hadron, and tau-lepton properties as of early 2012 and online update at `http://www.slac.stanford.edu/xorg/hfag`. *arXiv: 1207.1158 [hep-ex]*, 2012.

[2] A.A. Alves Junior et al. The LHCb Detector at the LHC. *JINST*, 3:S08005, 2008.

[3] The THRUST libraries. `https://thrust.github.io/`.

[4] M. Williams. Observing CP Violation in Many-Body Decays. *Phys.Rev.*, D84:054015, 2011.

[5] B. Aslan and G. Zech. New test for the multivariate two-sample problem based on the concept of minimum energy. *JSCS*, 75(2):109, 2005.

[6] B. Aslan and G. Zech. Statistical energy as a tool for binning-free, multivariate goodness-of-fit tests, two-sample comparison and unfolding. *Nucl. Instr. Meth. Phys. Res. A*, 537(3):626, 2005.

[7] R. Aaij et al. Search for CP violation in $D^0 \to \pi^-\pi^+\pi^0$ decays with the energy test. *arXiv:1410.4170 [hep-ex]*, 2014.

[8] J. Beringer et al. Review of Particle Physics (RPP). *Phys.Rev.*, D86:010001, 2012.

[9] J. Blatt and V. Weisskopf. *Theoretical Nuclear Physics*. John Wiley & Sons, 1952.

[10] F. Von Hippel and C. Quigg. Centrifugal-barrier effects in resonance partial decay widths, shapes, and production amplitudes. *Phys.Rev.*, D5:624, 1972.

[11] G.J. Gounaris and J.J. Sakurai. Finite width corrections to the vector meson dominance prediction for $\rho \to e^+e^-$. *Phys.Rev.Lett.*, 21:244, 1968.

[12] D. Aston et al. A Study of $K^-\pi^+$ Scattering in the Reaction $K^-p \to K^-\pi^+n$ at 11-GeV/c. *Nucl.Phys.*, B296:493, 1988.

[13] D. J. Lange. The EvtGen particle decay simulation package. *Nucl. Instrum. Meth.*, A462:152, 2001.

[14] The GOOFIT package. `https://github.com/GooFit/GooFit`.

Prospects of GPGPU in the Auger Offline Software Framework

Tobias Winchen[1], Marvin Gottowik[1], Julian Rautenberg[1] for the Pierre Auger Collaboration[2,3]

[1] Bergische Universität Wuppertal, Gaußstr. 20, 42119 Wuppertal, Germany
[2] Pierre Auger Observatory, Av. San Martín Norte 304, 5613 Malargüe, Argentina.
[3] Full author list: `http://www.auger.org/archive/authors_2014_09.html`

The Pierre Auger Observatory is the currently largest experiment dedicated to unveil the nature and origin of the highest energetic cosmic rays. The software framework $\overline{\text{Offline}}$ has been developed by the Pierre Auger Collaboration for joint analysis of data from different detector systems used in the observatory. While reconstruction modules are specific to the Pierre Auger Observatory components of the $\overline{\text{Offline}}$ framework are also used by other experiments. The software framework has recently been extended to analyze also data from the Auger Engineering Radio Array (AERA), the radio extension of the Pierre Auger Observatory. The reconstruction of the data from such radio detectors requires the repeated evaluation of complex antenna gain patterns which significantly increases the required computing resources in the joint analysis. In this contribution we explore the usability of massive parallelization of parts of the $\overline{\text{Offline}}$ code on the GPU. We present the result of a systematic profiling of the joint analysis of the $\overline{\text{Offline}}$ software framework aiming for the identification of code areas suitable for parallelization on GPUs. Possible strategies and obstacles for the usage of GPGPU in an existing experiment framework are discussed.

1 Introduction

Although cosmic rays have been intensively studied for more than 100 years, fundamental questions about the phenomenon remain unclear. In particular, the origin, the acceleration mechanism, and the chemical composition of the highest energetic cosmic rays with energies up to several hundred EeV (1 EeV = 10^{18} eV) remain open questions. As the flux of the highest energy cosmic rays is of the order of one particle per square kilometer per century, direct observation is impracticable. Instead, the Earth's atmosphere is used as a calorimeter in which the cosmic rays are detected indirectly by the particle cascades, or 'air showers', they induce. For detailed reviews on cosmic rays see e.g. references [1, 2].

The currently largest detector for cosmic rays at the highest energies is the Pierre Auger Observatory in Argentina. The Pierre Auger Observatory [3] is designed as a hybrid of two complementary detector systems. The 'surface detector' [4] consists of 1660 water-Cherenkov stations that sample the secondary particles at the ground level. The stations are arranged in a hexagonal grid that covers an area of 3000 km^2. The surface detector array is surrounded by 27 telescopes stationed at four sites to detect the fluorescence light emitted by air molecules that have been excited by the particle cascade. This 'fluorescence detector' [5] provides a

direct calorimetric measurement independent of hadronic interaction models and thus a better energy resolution than the surface detector. Furthermore, as it measures the development of the shower in the atmosphere, it is sensitive to the mass of the primary cosmic ray. However, the fluorescence detector can operate only during clear and moonless nights, whereas the surface detector has no principal constraint on the uptime.

As complementary detector without principal time constraints, the Auger Engineering Radio Array (AERA) [6] detects the radio pulses emitted by the particle cascades, due to geomagnetic and charge-excess effects [7]. Currently, AERA consists of 124 stations in a hexagonal grid covering an area of about 6 km^2. Each station is equipped with an antenna designed for polarized measurements from 30 to 80 MHz.

The Auger $\overline{\text{Off}}\underline{\text{line}}$ Software Framework [8] provides the tools and infrastructure for the reconstruction of events detected with the Pierre Auger Observatory. Components of the framework are also used by other experiments [9]. It is designed to support and incorporate the ideas of physicists for the projected lifetime of the experiment of more than 20 years. This is achieved by a strict separation of configuration, reconstruction modules, event and detector data, and utilities to be used in the algorithms. The reconstruction of radio events detected with AERA is fully integrated in the Auger $\overline{\text{Off}}\underline{\text{line}}$ Software Framework which allows joint analyses of events from all detector systems [10]. The main reconstruction algorithm for the radio reconstruction and its performance profile is described in the next section.

2 Radio Reconstruction in the Auger Offline Framework

The main algorithm for the reconstruction of radio events implemented in several $\overline{\text{Off}}\underline{\text{line}}$ modules is depicted in Fig. 1. After a trigger, the voltage traces of the two channels at each station are read out, and, after noise filtering and signal enhancing, processed as follows. First, as an initial estimate of the event timing in each station, the maxima of the voltage traces is used. Second, from the timing information of the individual stations, the direction of the shower is obtained by triangulation. Third, the antenna pattern for this direction is evaluated, and the E-field trace at each station reconstructed from the voltage traces. Finally, the maximum of the envelope of the E-field trace is used as updated timing information for a new iteration of the procedure. On convergence, the timing information yields the incident direction of the primary particle whereas the E-field distribution on the ground allows a derivation of the energy and particle type of the cosmic ray.

The execution of this algorithm in $\overline{\text{Off}}\underline{\text{line}}$ requires an uncomfortable amount of time. Using the Linux kernel profiler 'perf' [11] we identified two main performance bottlenecks in this algorithm. First, about 15% of the computation time is spent in calculating Fourier transformations with the FFTW library [12]. Second, about 25% of the time is used for the interpolation of the antenna patterns. All other parts of the algorithm use less than 5% of the time. The same bottlenecks are identified using 'google-perftools' [13] or 'Intel VTune amplifier' [14] as alternative profilers.

In the next sections we discuss the elimination of these bottlenecks by implementing the relevant parts on the GPU using the Cuda framework. In both cases we followed a minimum invasive approach that leaves the general interfaces in $\overline{\text{Off}}\underline{\text{line}}$ intact. To select between the CPU and GPU implementation, a preprocessor directive is used.
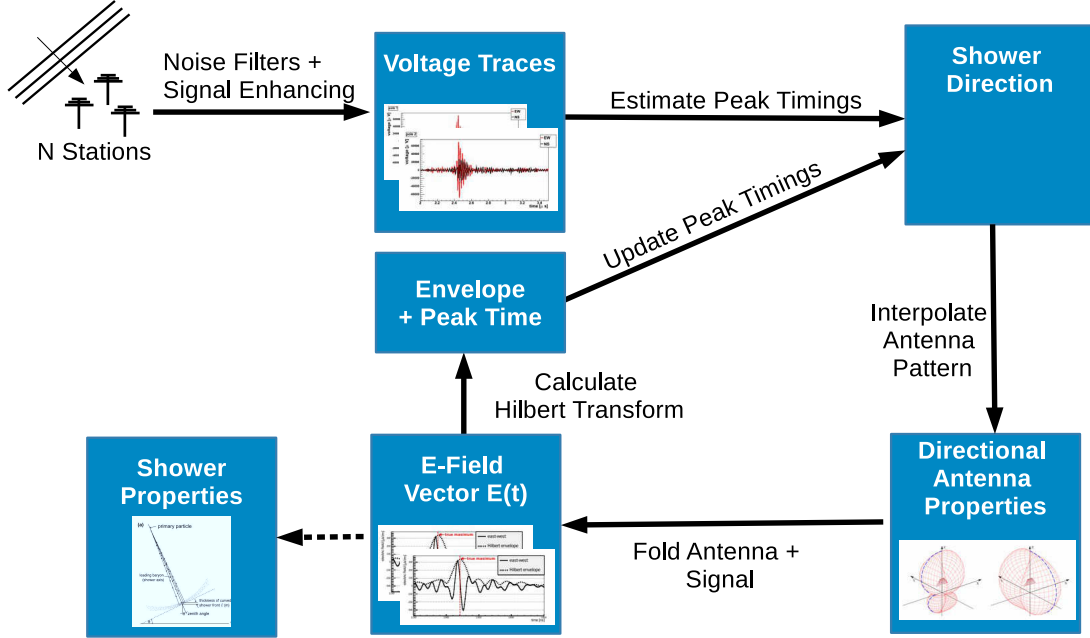
Figure 1: Schematic of the individual steps in the $\overline{\text{Off}\underline{\text{line}}}$ radio reconstruction.

3   Interpolations of Antenna Patterns

To reconstruct the electric field vector from the measured voltage traces the antenna pattern in the direction of the electromagnetic wave must be known. The antenna pattern can be conveniently expressed as a two dimensional complex vector, the 'vector effective length (VEL)'. For each antenna, the VEL is known for discrete frequencies and zenith and azimuth angles from measurements or simulations. Between theses nodes the VEL has to be interpolated for arbitrary directions.

The interpolation of textures is a core task of graphics cards, which have dedicated circuits for the interpolation. The usage of these promises a great speedup, but is available for single precision data of limited size only. In the baseline implementation, the antenna pattern is evaluated in double precision. As a linear interpolation requires six elementary operations, the maximum relative uncertainty from the limited floating-point precision can be estimated as $2.5 \times 10^{-4}\,\%$ in single precision. This is smaller than other uncertainties in the reconstruction and thus negligible here. The largest antenna pattern considered here consists of complex data for the vector effective length at 98 frequencies, 31 zenith angles, and 49 azimuth angles. In single precision this corresponds to approximately 2.4 MB of data which is small compared to
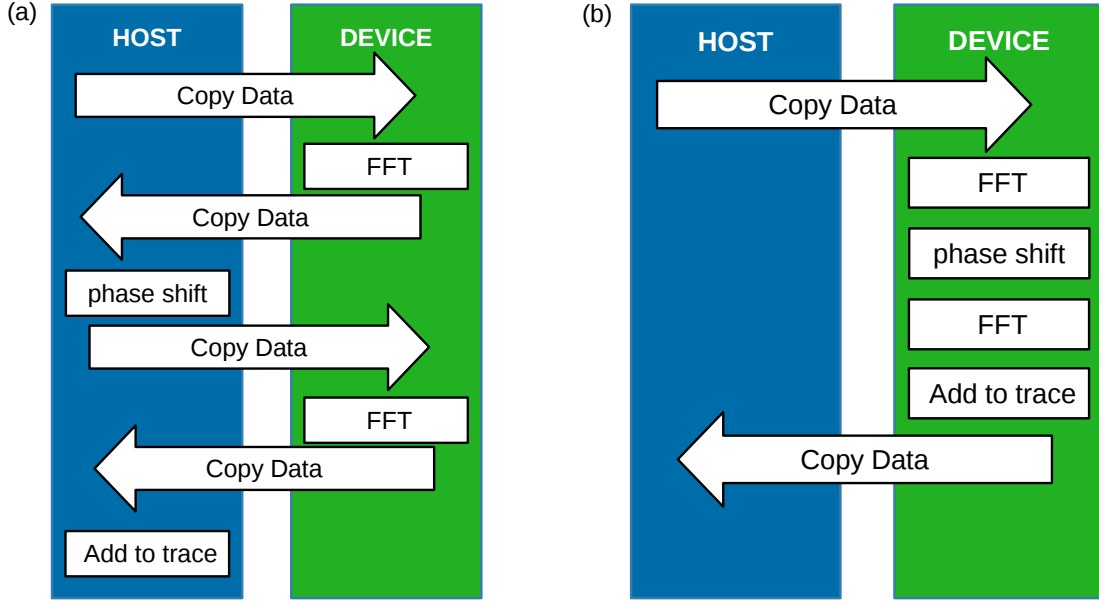
Figure 2: Calculation of the Hilbert envelope **(a)** using the CuFFT wrapper only and **(b)** using a dedicated kernel.

the total available memory size of a few GB on modern GPUs and much below the maximal size of a 3D texture of typically at least 2048×2048×2048 elements. The patterns for all antennas used in AERA can be stored simultaneously on the device.

To speed up the interpolation on the CPU, the pattern has already been buffered in the baseline implementation for look up on repeated access. In the GPU implementation this is unnecessary. Here, the patterns are copied and converted to allow binding to texture memory only once on first access.

4 Fourier Transformations and Hilbert Envelopes

In the baseline implementation in $\overline{\text{Off}\underline{\text{line}}}$, calls to the FFTW library are wrapped in an object-oriented interface. The interface provides several distinct classes for the operation on real or complex data of given dimensionality. Shared functionality is implemented in a base class and propagated by inheritance. In the radio reconstruction, the wrapper operates within a container that stores a trace simultaneously in the time and frequency domains. After modification of either, the other is updated in a lazy evaluation scheme.

To calculate the FFT on the GPU, FFTW calls are replaced by calls to the CUDA FFT library (CuFFT). In contrast to the CPU implementation, all instances of the GPU implementation share static memory on the GPU to avoid time consuming allocations. This is safe here as memory copies are blocking and performed immediately before and after the FFT calculation.

However, in the reconstruction several FFTs are calculated in context of obtaining the Hilbert envelope of the radio traces. The envelope $E(t)$ of a trace $x(t)$ is $E(t) = \sqrt{x(t)^2 + H(x(t)^2)}$ with the Hilbert transform of the signal $H(x(t))$. The Hilbert transformation shifts the phase of
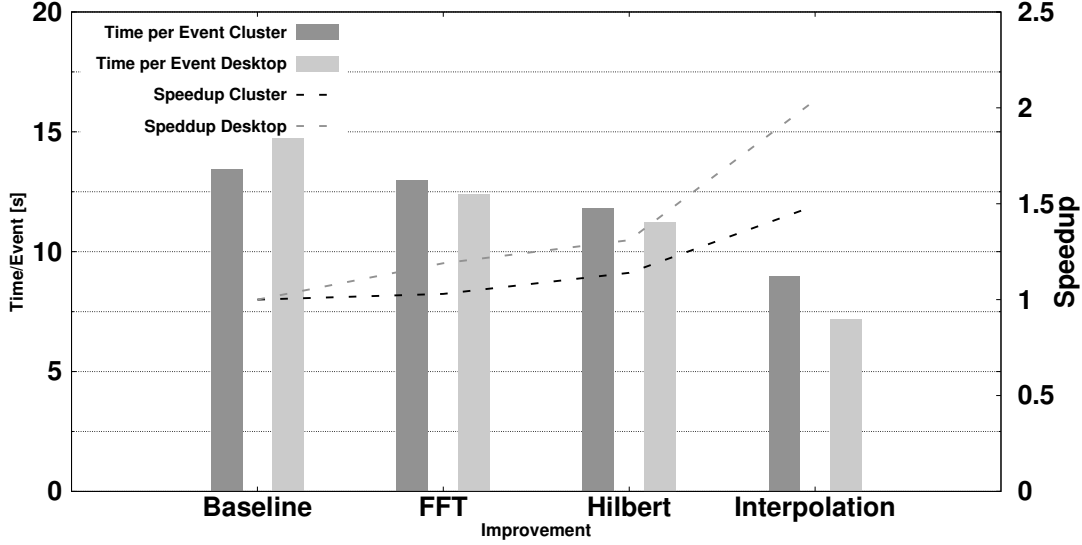
Figure 3: Summary of the speedup achieved in the individual improvements.

negative frequencies, i.e. for band-limited signal frequencies below the mid-frequency, by $-90°$ and positive frequencies by $+90°$. The GPU implementation as described above thus results in a non-optimal memory access pattern for the envelope calculation as shown in Fig. 2 (a). However, with a dedicated computing kernel not only can two memory copies be avoided, but also the phase shift and summation are calculated in parallel (cf. Fig 2 (b)).

5   Discussion

The results obtained from the new GPU implementations are consistent with the results from the baseline implementation. While the FFTs yield identical results on CPU and GPU, the interpolation is not only a different implementation but also only in single precision. This amounts to a relative difference in the directional antenna patterns between the individual implementations of typically below $\pm0.8\%$ and thus small compared to other uncertainties.

The performance improvements obtained by the modifications are summarized in Fig. 3. As test systems we used here a typical recent desktop PC and a combined CPU/GPU cluster. The desktop is equipped with an AMD A8-6600K processor and an NVIDIA GeForce 750 Ti graphics card. The cluster contains 4 Intel Xeon X5650 CPUs and 4 NVIDIA Tesla M2090 GPUs. On the desktop system the GPU implementation of FFT and the Hilbert transformation yield a speedup of 1.3, doing also the interpolations on the GPU increased this speedup to approximately 2. On the cluster system the total achieved speedup is 1.5. The lower speedup on the cluster system is due to the higher relative performance of the cluster CPU and GPU compared to the desktop system.

As only selected isolated parts of the code are moved to the GPU, the time used for computing on the GPU is low compared to the time needed for memory copy, and also only 7% of the copy-time is overlapped by computing time. However, increasing the GPU utilization would

require the traces to be kept permanently on the GPU so that more analysis steps can benefit from porting to the GPU. This, however, would require non-trivial changes in the $\overline{\text{Off}}\underline{\text{line}}$ framework, in particular, modifications of the internal structure and interfaces.

6 Conclusion

The calculation of Fourier transformations and the interpolation of antenna response patterns have been identified as bottlenecks in the AERA event reconstruction using a performance profiler. Eliminating both by re-implementating the calculation in CUDA while keeping the structure of $\overline{\text{Off}}\underline{\text{line}}$ intact yields a speedup of 1.49 to 2.04 depending on the test system. The largest speedup is obtained here on a typical desktop PC equipped with an entry level graphics card. Considering the relative costs of about €500 for a desktop PC and €100 for an entry level GPU, even such selected applications of GPGPU in existing frameworks are a possibility to be considered in planning future computing strategies.

References

[1] K. Kotera and A. V. Olinto. The astrophysics of ultrahigh energy cosmic rays. *Annual Review of Astronomy and Astrophysics*, 49:119–153, 2011.

[2] A. Letessier-Selvon and T. Stanev. Ultrahigh Energy Cosmic Rays. *Reviews of Modern Physics*, 83:907–942, 2011.

[3] J. Abraham et al. (The Pierre Auger Collaboration). Properties and performance of the prototype instrument for the Pierre Auger Observatory. *Nuclear Instruments and Methods in Physics Research Section A*, 523:50, 2004.

[4] I. Allekotte et al. (The Pierre Auger Collaboration). The Surface Detector System of the Pierre Auger Observatory. *Nuclear Instruments and Methods in Physics Research Section A*, 586:409–420, 2008.

[5] J. Abraham et al. (The Pierre Auger Collaboration). The Fluorescence Detector of the Pierre Auger Observatory. *Nuclear Instruments and Methods in Physics Research Section A*, 620:227–251, 2010.

[6] J. Neuser for the Pierre Auger Collaboration. Detection of Radio Emission from Air Showers in the MHz Range at the Pierre Auger Observatory. In *Proceedings of the 6th international Conference on Acoustic and Radio EeV Neutrino Detection Activities (ARENA)*, 2014.

[7] A. Aab et al. (The Pierre Auger Collaboration). Probing the radio emission from air showers with polarization measurements. *Phys. Rev. D*, 89:052002, Mar 2014.

[8] S. Argiro et al. The Offline Software Framework of the Pierre Auger Observatory. *Nuclear Instruments and Methods in Physics Research Section A*, 580:1485–1496, 2007.

[9] R. Sipos et al. The offline software framework of the na61/shine experiment. In *Proceedigs of the International Conference on Computing in High Energy and Nuclear Physics 2012 (CHEP2012)*, 2012.

[10] P. Abreu et al. (The Pierre Auger Collaboration). Advanced functionality for radio analysis in the Offline software framework of the Pierre Auger Observatory. *Nuclear Instruments and Methods in Physics Research Section A*, A635:92–102, 2011.

[11] The Linux Kernel Team. Perf wiki. `https://perf.wiki.kernel.org/index.php/Main_Page`, visited on 2014-09-19.

[12] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[13] Google Inc. gperftools - fast, multi-threaded malloc() and nifty performance analysis tools. `https://code.google.com/p/gperftools/`, visited on 2014-08-10.

[14] The Intel Corporation. Intel VTune Amplifier. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`, visited on 2014-08-02.

Chapter 4

GPU in Lattice QCD

Convenors:

Massimo D'Elia

Designing and Optimizing LQCD codes using OpenACC

Claudio Bonati[1], *Enrico Calore*[2], *Simone Coscetti*[1], *Massimo D'Elia*[1,3], *Michele Mesiti*[1,3], *Francesco Negro*[1], *Sebastiano Fabio Schifano*[2,4], *Raffaele Tripiccione*[2,4]

[1]INFN - Sezione di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
[2]INFN - Sezione di Ferrara, via Saragat 1, 44122 Ferrara, Italy
[3]Università di Pisa, Largo Bruno Pontecorvo, 3, 56127 Pisa, Italy
[4]Università di Ferrara, via Saragat 1, 44122 Ferrara, Italy

An increasing number of massively parallel machines adopt heterogeneous node architectures combining traditional multicore CPUs with energy-efficient and fast accelerators. Programming heterogeneous systems can be cumbersome and designing efficient codes often becomes a hard task. The lack of standard programming frameworks for accelerator based machines makes it even more complex; in fact, in most cases satisfactory performance implies rewriting the code, usually written in C or C++, using proprietary programming languages such as CUDA. OpenACC offers a different approach based on directives. Porting applications to run on hybrid architectures "only" requires to annotate existing codes with specific "pragma" instructions, that identify functions to be executed on accelerators, and instruct the compiler on how to structure and generate code for specific target device. In this talk we present our experience in designing and optimizing a LQCD code targeted for multi-GPU cluster machines, giving details of its implementation and presenting preliminary results.

1 Introduction

Lattice Quantum Chromodynamics (LQCD) simulations enable us to investigate aspects of the Quantum Chromodynamics (QCD) physics that would be impossible to systematically investigate in perturbation theory.

The computation time for LQCD simulations is a strong limiting factor, bounding for example the usable lattice size. Fortunately enough, the most time consuming kernels of the LQCD algorithms are embarassingly parallel, however the challenge of designing and running efficient codes is not easy to met.

In the past years commodity processors were not able to provide the required computational power for LQCD simulations, and several generations of parallel machines have been specifically designed and optimized for this purpose [1, 2]. Today, multi-core architecture processors are able to deliver several levels of parallelism providing high computing power that allow to tackle larger and larger lattices, and computations are commonly performed using large computer clusters of commodity multi- and many-core CPUs. Moreover, the use of accelerators such as GPUs has been successfully explored to boost performamces of LQCD codes [3].

More generally, massively-parallel machines based on heterogeneous nodes combining traditional powerful multicore CPUs with energy-efficient and fast accelerators are ideal targets for LQCD simulations and are indeed commonly used. Programming these heterogeneous systems can be cumbersome, mainly because of the lack of standard programming frameworks for accelerator based machines. In most of the cases, reasonable efficiency requires that the code is re-written targeting a specific accelerator, using proprietary programming languages such as CUDA for nVIDIA GPUs.

OpenACC offers a different approach based on directives, allowing to port applications onto hybrid architectures by annotating existing codes with specific "pragma" directives. A perspective OpenACC implementation of an LQCD simulation code would grant its portability across different heterogeneous machines without the need of producing multiple versions using different languages. However the price to pay for code portability may be in terms of code efficiency.

In this work we explore the possible usage of OpenACC for LQCD codes targeting heterogeneous architectures, estimating the performance loss that could arise with respect to an architecture-specific optimized code. To pursue this goal, we wrote the functions accounting for most of the execution time in an LQCD simulation using plain C and OpenACC directives. As well known, the most compute intensive computational kernel is the repeated application of the Dirac operator (the so called D slash operator $D\!\!\!/$). We wrote and optimized the corresponding routines and then compared the obtained performance of our C/OpenACC implementation with an already existing state-of-the-art CUDA program [4].

In Sec. 2 and Sec. 3 we briefly review the OpenACC programming standard and the LQCD methods respectively, while in Sec. 4 we provide the details of our implementation. In Sec. 5 we present our performance results.

2   OpenACC

OpenACC is a programming framework for parallel computing aimed to facilitate code development on heterogeneous computing systems, and in particular to simplify porting of existing codes. Its support for different architectures relies on compilers; although at this stage the few available ones target mainly GPU devices, thanks to the OpenACC generality the same code can be compiled for different architectures when the corresponding compilers and run-time supports become available.

OpenACC, like OpenCL, provides a widely applicable abstraction of actual hardware, making it possible to run the same code across different architectures. Contrary to OpenCL, where specific functions (called *kernels*) have to be explicitly programmed to run in a parallel fashion (e.g. as GPU threads), OpenACC is based on pragma directives that help the compiler identify those parts of the source code that can be implemented as *parallel functions*. Following *pragma* instructions the compiler generates one or more *kernel* functions – in the OpenCL sense – that run in parallel as a set of threads.

OpenACC is similar to the OpenMP (Open Multi-Processing) language in several ways [5]; both environments are directive based, but OpenACC targets accelerators in general, while at this stage OpenMP targets mainly multi-core CPUs.

Regular C/C++ or Fortran code, already developed and tested on traditional CPU architectures, can be annotated with OpenACC pragma directives (e.g. *parallel* or *kernels* clauses) to instruct the compiler to transform loop iterations into distinct threads, belonging to one or

more functions to run on an accelerator.

Various directives are available, allowing fine tuning of the application. As an example, the number of threads launched by each device function and their grouping can be fine tuned by the *vector*, *worker* and *gang* directives, in a similar fashion as setting the number of *work-items* and *work-groups* in OpenCL. Data transfers between host and device memories are automatically generated, when needed, entering and exiting the annotated code regions. Even in this case data directives are available to allow the programmer to obtain a finer tuning, e.g. increasing performance by an appropriate ordering of the transfers. For more details on OpenACC see [6].

3   Lattice QCD

Lattice QCD (LQCD) is a nonperturbative regularization of Quantum Chromodynamics (QCD) which enables us to tackle some aspects of the QCD physics that would be impossible to systematically investigate by using standard perturbation theory, like for instance the confinement problem or chiral symmetry breaking.

The basic idea of LQCD is to discretize the continuum QCD on a four dimensional lattice, in such a way that continuum physics is recovered as the lattice spacing goes to zero. Fermions are problematic in this respect: a famous no-go theorem by Nielsen and Ninomiya can be vulgarized by saying that in the discretization procedure some of the properties of the fermions will be lost; they will be recovered only after the continuum limit is performed. Several ways to circumvent this difficulty exist and this is the reason for the current lattice fermions zoology: Wilson, staggered, domain-wall and overlap fermions (see e.g. [7]). In the following we will specifically refer to the staggered formulation, which is commonly adopted for the investigation of QCD thermodynamics.

The discretized problem can be studied by means of Monte-Carlo techniques and, in order to generate configurations with the appropriate statistical weight, the standard procedure is the Hybrid Monte Carlo algorithm (HMC, see e.g. [7]). HMC consists of the numerical integration of the equations of motion, followed by an accept/reject step. The computationally more intensive step of this algorithm is the solution of linear systems of the form $\slashed{D}\psi = \eta$, where η is a vector of Gaussian random numbers and $\slashed{D}$ is the discretized version of the Dirac matrix, whose dimension is given by the total lattice size, which is typically $\mathcal{O}(10^5 - 10^6)$. By using an even-odd sorting of the lattice sites, the matrix $\slashed{D}$ can be written in block form

$$\slashed{D} = \begin{pmatrix} m & D_{oe} \\ D_{eo} & m \end{pmatrix}, \quad D_{oe}^\dagger = -D_{eo} ,$$

where m is the fermion mass. The even-odd preconditioned form of the linear system is $(m^2 - D_{eo}D_{oe})\psi_e = \eta_e$, where now ψ_e and η_e are defined on even sites only.

This is still a very large sparse linear problem, which can be conveniently solved by using the Conjugate Gradient method or, more generally, Krylov solvers. The common strategy of this class of solvers is the following: one starts from an initial guess solution, then iteratively improves its accuracy by using auxiliary vectors, obtained by applying D_{oe} and D_{eo} to the solution of the previous step.

From this brief description of the target algorithm it should be clear that, in order to have good performances, a key point is the availability of very optimized routines for the D_{oe} and D_{eo} operators. As a consequence these routines appear as natural candidates for a comparison between different code implementations.
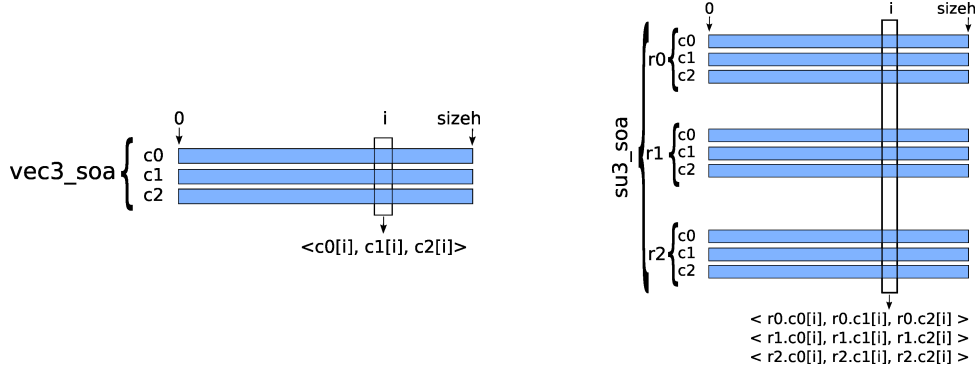
Figure 1: Memory allocation diagrams for the data structures of vectors (left) and $SU(3)$ matrices (right). Each vector or matrix component is a double precision complex value.

Before going on to present some details about the code and the performances obtained, we notice that both D_{oe} and D_{eo} still have a natural block structure. The basic elements of these operators are $SU(3)$ matrices and this structure can be used, e.g., to reduce the amount of data that have to be transferred from the memory (the algorithm is strongly bandwidth limited). The starting point for the development of the OpenACC code was the CUDA code described in [4], which adopts all these specific optimizations.

4 Code Implementation

We coded the D_{eo} and the D_{oe} functions using plain C; OpenACC directives instruct the compiler to generate one GPU kernel function for each of them. The function bodies of the OpenACC version strongly resemble the corresponding CUDA kernel bodies, trying to ensure a fair comparison between codes which perform the same operations.

For both the CUDA and OpenACC versions, each GPU thread is associated to a single lattice site; in the D_{eo} function all GPU threads are associated to even lattice sites, while in the D_{oe} function all GPU threads are associated to odd lattice sites. Consequently each kernel function operates on half of the lattice points.

Data structures are allocated in memory following the *SoA* (Structure of Arrays) layout to obtain better memory coalescing for both vectors and matrices as shown in Fig. 1. The basic data element of both the structures is the standard C99 double complex.

Listing 1 contains a code snippet showing the beginning of the CUDA function implementing the D_{eo} operation. Note in particular the mechanism to reconstruct the x, y, z, t coordinates identifying the lattice point associated to the current thread, given the CUDA 3-dimensional thread coordinates (nt, nx, ny, nz are the lattice extents and $nxh = nx/2$).

Listing 1: CUDA

```
__global__ void Deo(const __restrict su3_soa_d * const u,
                    const __restrict vec3_soa_d * const out,
              const __restrict vec3_soa_d * const in) {

  int x, y, z, t, xm, ym, zm, tm, xp, yp, zp, tp, idxh, eta;
```

```
  vec3 aux_tmp;
  vec3 aux;

  idxh = ((blockIdx.z * blockDim.z + threadIdx.z) * nxh * ny)
       + ((blockIdx.y * blockDim.y + threadIdx.y) * nxh)
       +  (blockIdx.x * blockDim.x + threadIdx.x);

  t =   (blockIdx.z * blockDim.z + threadIdx.z) / nz;
  z =   (blockIdx.z * blockDim.z + threadIdx.z) % nz;
  y =   (blockIdx.y * blockDim.y + threadIdx.y);
  x = 2*(blockIdx.x * blockDim.x + threadIdx.x) + ((y+z+t) & 0x1);

  ...
```

Listing 2 shows the OpenACC version of the same part of the D_{eo} function. The four for loops, each iterating over one of the four lattice dimensions, and the pragma directive preceding each of them are clearly visible. In this case the explicit evaluation of the x, y, z, t coordinates is not needed, since we use here standard loop indices; however we can still control the *thread blocks* size using the vector and gang clauses. In particular, DIM_BLK_X, DIM_BLK_Y and DIM_BLK_Z are the desired dimensions of the *thread blocks*. Since we execute the code on an nVIDIA GPU, as for the CUDA case, also in this case, each GPU thread is actually addressed by 3 coordinates.

Listing 2: OpenACC

```
void Deo(const __restrict su3_soa * const u,
              __restrict vec3_soa * const out,
         const __restrict vec3_soa * const in) {

  int hx, y, z, t;

  #pragma acc kernels present(u) present(out) present(in)
  #pragma acc loop independent gang(nt)
  for(t=0; t<nt; t++) {
    #pragma acc loop independent gang(nz/DIM_BLK_Z) vector(DIM_BLK_Z)
    for(z=0; z<nz; z++) {
      #pragma acc loop independent gang(ny/DIM_BLK_Y) vector(DIM_BLK_Y)
      for(y=0; y<ny; y++) {
        #pragma acc loop independent vector(DIM_BLK_X)
        for(hx=0; hx < nxh; hx++) {

            ...
```

We experimented with other potentially useful optimizations, e.g. combining the D_{eo} and D_{oe} routine in a single function D_{eoe} and mapping it onto a single GPU *kernel*, but the performance was roughly one order of magnitude lower, mainly because of overheads associated to register spilling.

5 Results and Conclusions

We prepared a benchmark code able to repeatedly call the D_{eo} and the D_{oe} functions, one after the other, using the OpenACC implementation or the CUDA one. The two implementations

were compiled respectively with the PGI compiler, version 14.6, and the nVIDIA nvcc CUDA compiler, version 6.0.

The benchmark code was run on a 32^4 lattice, using an nVIDIA K20m GPU; results are shown in Tab. 1, where we list the sum of the execution times of the D_{eo} and D_{oe} operations in nanoseconds per lattice site, for different choices of *thread block* sizes. All computations were performed using double precision floating point values.

	D_{eo} + D_{oe} Funtions	
Block-size	CUDA	OpenACC
8,8,8	7.58	9.29
16,1,1	8.43	16.16
16,2,1	7.68	9.92
16,4,1	7.76	9.96
16,8,1	7.75	10.11
16,16,1	7.64	10.46

Table 1: Execution time in (nsec per lattice site) of the D_{eo} + D_{oe} functions, for the CUDA and OpenACC implementations running on an nVIDIA K20m GPU and using floating point double precision throughout.

Execution times have a very mild dependence on the block size and for the OpenACC implementation are in general slightly higher; if one considers the best *thread block* sizes both for CUDA and OpenACC, the latter is $\simeq 23\%$ slower.

A slight performance loss with respect to CUDA is expected, given the higher level of the OpenACC language. In this respect, our results are very satisfactory, given the lower programming efforts needed to use OpenACC and the increased code maintainability given by the possibility to run the same code on CPUs or GPUs, by simply disabling or enabling pragma directives. Moreover, OpenACC code performance is expected to improve in the future also due to the rapid development of OpenACC compilers, which at the moment are yet in their early days.

The development of a complete LQCD simulation code fully based on OpenACC is now in progress.

References

[1] H. Baier, et al. *Computer Science - Research and Development*, 25(3-4):149–154, 2010. DOI:10.1007/s00450-010-0122-4.

[2] F. Belletti, et al. *Computing in Science and Engineering*, 8(1):50–61, 2006. DOI:10.1109/MCSE.2006.4.

[3] G.I. Egri, et al. *Computer Physics Communications*, 177(8):631–639, 2007. DOI:10.1016/j.cpc.2007.06.005.

[4] C. Bonati, et al. *Computer Physics Communications*, 183(4):853–863, 2012. DOI:10.1016/j.cpc.2011.12.011.

[5] S. Wienke, et al. In *Lecture Notes in Computer Science*, volume 8632 LNCS, pages 812–823, 2014. DOI:10.1007/978-3-319-09873-9-68.

[6] http://www.openacc-standard.org/.

[7] T. DeGrand and C. DeTar. *Lattice methods for quantum chromodynamics*. World Scientific, 2006.

Conjugate gradient solvers on Intel® Xeon Phi™ and NVIDIA® GPUs

O. Kaczmarek[1], C. Schmidt[1], P. Steinbrecher[1], M. Wagner[2]

[1]Fakultät für Physik, Universität Bielefeld, D-33615 Bielefeld, Germany
[2]Physics Department, Indiana University, Bloomington, IN 47405, USA

Lattice Quantum Chromodynamics simulations typically spend most of the runtime in inversions of the Fermion Matrix. This part is therefore frequently optimized for various HPC architectures. Here we compare the performance of the Intel® Xeon Phi™ to current Kepler-based NVIDIA® Tesla™ GPUs running a conjugate gradient solver. By exposing more parallelism to the accelerator through inverting multiple vectors at the same time, we obtain a performance greater than 300 GFlop/s on both architectures. This more than doubles the performance of the inversions. We also give a short overview of the Knights Corner architecture, discuss some details of the implementation and the effort required to obtain the achieved performance.

1 Introduction

In finite temperature Quantum Chromodynamics (QCD) fluctuations of conserved charges, baryon number (B), electric charge (Q) and strangeness (S), are particular interesting observables. They can be measured in experiments at the Relativistic Heavy Ion Collider (RHIC) and the Large Hadron Collider (LHC) and have also been calculated in Lattice QCD (LQCD) with increasing precision [1]. They are derived from generalized susceptibilities

$$\chi_{mnk}^{BQS}(T) = \frac{1}{VT^3} \frac{\partial^{m+n+k} \ln \mathcal{Z}}{\partial \left(\mu_B/T\right)^m \partial \left(\mu_Q/T\right)^n \partial \left(\mu_S/T\right)^k} \Bigg|_{\vec{\mu}=0} , \tag{1}$$

where $\mathcal{Z}$ denotes the partition function of the medium at temperature T and volume V.

In LQCD the required derivatives of $\mathcal{Z}$ w.r.t. the chemical potentials μ can be obtained by stochastically estimating traces over combinations of the inverse and derivatives of the Fermion Matrix M with a sufficiently large number of random vectors η, e.g.

$$\mathrm{Tr} \left(\frac{\partial^{n_1} M}{\partial \mu^{n_1}} M^{-1} \frac{\partial^{n_2} M}{\partial \mu^{n_2}} \dots M^{-1} \right) = \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^\dagger \frac{\partial^{n_1} M}{\partial \mu^{n_1}} M^{-1} \frac{\partial^{n_2} M}{\partial \mu^{n_2}} \dots M^{-1} \eta_k . \tag{2}$$

To control the errors we use 500-1500 random vectors on each gauge configuration. Depending on the desired highest derivative degree this involves several inversion of the Fermion Matrix for each random vector.

#rhs	1	2	3	4	5	6	8
Flop/byte (full)	0.73	1.16	1.45	1.65	1.80	1.91	2.08
Flop/byte (r14)	0.80	1.25	1.53	1.73	1.87	1.98	2.14

Table 1: The arithmetic intensity of the HISQ Dslash for different number of right-hand sides (rhs) using full or reduced 14 float storage (r14) for the Naik links.

For reasons of the numerical costs, staggered fermions are the most common type of fermions for thermodynamic calculations on the lattice. We use the highly improved staggered fermion (HISQ) action [2]. In terms of the smeared links U and Naik links N the Dslash operator reads

$$w_x = D_{x,x'}v_{x'} = \sum_{\mu=0}^{4} \left[\left(U_{x,\mu}v_{x+\mu} - U_{x-\mu,\mu}^{\dagger}v_{x-\mu} \right) + \left(N_{x,\mu}v_{x+3\mu} - N_{x-3\mu,\mu}^{\dagger}v_{x-3\mu} \right) \right] . \quad (3)$$

Here N and U are complex 3×3 matrices and v, w are complex 3-dimensional vectors. Within the inversion the application of the Dslash operator typically consumes more than 80% of the runtime. This part already has a low arithmetic intensity (see Tab. 1) and the average arithmetic intensity (Flop/byte) is further decreased by the linear algebra operation in the conjugate gradient. Thus, the achievable performance is clearly bound by the available memory bandwidth. Given its massively parallel nature and the bandwidth hunger it is well suitable for accelerators. Lattice QCD simulations make extensive use of GPUs for several years now [3]. The MIC architecture is also gaining more attraction and codes are being optimized [4]. A common optimization to reduce memory accesses in LQCD is to exploit available symmetries and reconstruct the gauge links from 8 or 12 floats instead of loading all 18 floats. For improved actions these symmetries are often broken and thus we can only reconstruct the Naik links from 9 or 13/14 floats.

For our and many other applications a large number of inversions are performed on a single gauge configuration. In this case, one can exploit the constant gauge field by grouping the random vectors in small bundles, thus applying the Dslash for multiple right-hand sides (rhs) at once:

$$\left(w_x^{(1)}, w_x^{(2)}, \ldots, w_x^{(n)} \right) = D_{x,x'} \left(v_{x'}^{(1)}, v_{x'}^{(2)}, \ldots, v_x^{(n)} \right) . \quad (4)$$

This increases the arithmetic intensity of the HISQ Dslash as the load of the gauge field occurs only once for the n rhs. Increasing the number of rhs from 1 to 4 already results in an improvement by a factor of more than 2. In the limiting case of assuming the gauge fields do not have to be loaded at all, the highest arithmetic intensity that can be reached is ~ 2.75. At $n = 8$ we have reached already $\sim 75\%$ of the limiting peak intensity, while for 1 rhs we only obtain $25 - 30\%$. For an increasing number of rhs the memory transfers caused by loading the gauge fields are no longer dominating and thus also the impact of reconstructing the Naik links is less pronounced. Note that all numbers given here, as well as the performance data in the following, are for single-precision computations. However, the arguments work also for double-precision and the arithmetic intensity, in this case, is just half of the one in the single-precision case. For the full inverter the linear algebra operations in the conjugate gradient do not allow for the reuse of any constant fields. They therefore limit the achievable speedup.

We summarized some technical data of the accelerators we use for our comparison in Table 2. In the following we will only discuss our implementation for the Intel® Xeon Phi™ . Information about our GPU implementation can be found in [5].

	5110P	7120P	K20	K40
Cores / SMX	60	61	13	15
(Threads/Core) / (Cores/SMX)	4	4	192	192
Clock Speed [MHz]	1053	1238/1333	706	745/810/875
L1 Cache / Core [KB]	32	32	16-48	16-48
L2 Cache [MB]	30	30.5	1.5	1.5
Memory Size [GB]	8	16	5	12
peak fp32/64 [TFlop/s]	2.02/1.01	2.42/1.21	3.52/1.17	4.29/1.43
Memory Bandwidth [GB/s]	320	352	208	288

Table 2: The important technical data of the accelerators we have used in our benchmarks.

2 MIC

The Intel® Xeon Phi™ is an in-order x86 based many-core processor [6]. The coprocessor runs a Linux μOS and can have up to 61 cores combined via a bidirectional ring (see Fig. 1). Therefore, the memory transfers are limited by concurrency reaching only 149 GB/s on a 7120P running a stream triad benchmark [7]. Each core has a private L1 data and instruction cache and a globally visible L2 cache. In the case of a local L2 cache miss, a core can cross-snoop another's core L2 cache and if the data is present avoid a direct memory access. Each core has thirty-
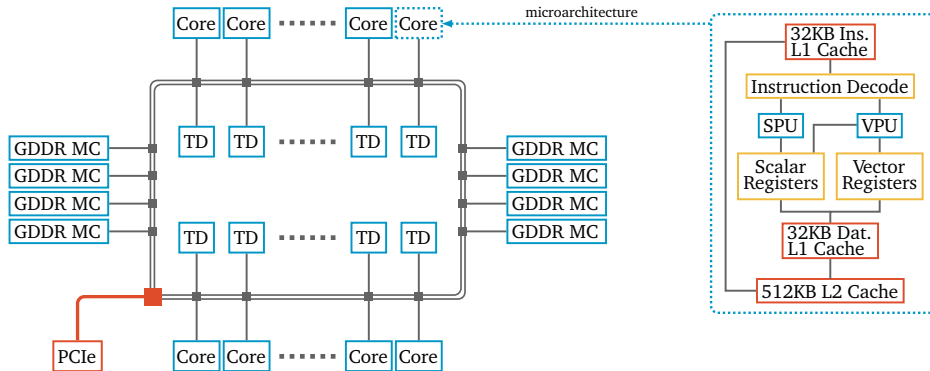

Figure 1: Visualization of the bidirectional ring on the die and the microarchitecture of one core showing the Scalar Processing Unit (SPU), Vector Processing Unit (VPU) and the cache hierarchy. The latter is kept fully coherent through global distributed tag directories (TD).

two 512 bit zmm vector registers and 4 hardware context threads. To fully utilize the Many Integrated Core (MIC) it is, especially for memory-bound applications, necessary to run with four threads per core. This offers more flexibility to the processor to swap the context of a thread, which is currently stalled by a cache miss. The MIC has its own SIMD instruction set extension IMIC with support for fused multiply-add and masked instructions. The latter allows to conditionally execute vector instructions on single elements of a vector register. The coprocessor can stream data directly into memory without reading the original content of an entire cache line, thus bypassing the cache and increasing the performance of algorithms where

the memory footprint is too large for the cache.

Implementation: We have parallelized our program with OpenMP and vectorized it using low-level compiler functions called intrinsics. These are expanded inline and do not require explicit register management or instruction scheduling through the programmer as in pure assembly code. There are 512 bit intrinsics data types for single- and double-precision accuracy as well as for integer values. More than 32 variables of a 512 bit data type can be used simultaneously. With only 32 zmm registers available in hardware, the compiler is, in this case, forced to use "spills", i.e. temporarily storing the contents of a register into L1 cache, and reloading the register when the data is required later, thereby increasing memory bandwidth pressure and cache pollution. When using intrinsics the software has to be designed in a register aware manner; only the explicit management of the registers is taken over by the compiler. We found that the compiler is only able to optimize code over small regions. Thus, the order of intrinsics can have an influence on the achieved performance, thereby making optimizations more difficult. Nonetheless, the use of intrinsics for the Dslash kernel is lightweight requiring only a subset of 9 instructions. Due to the different links needed for the nearest and third-nearest neighbor term we implemented both in separate kernels, thereby reducing cache pollution and simplifying cache reuse for the vectors. For the global sums inside the linear algebra kernels we use the OpenMP reduction clause. In order to avoid explicit barriers, each thread repeats the calculation of the coefficients necessary for the CG in a thread local variable.

Site fusion: One problem of using 512 bit registers involving SU(3) matrix-vector products is that one matrix/vector does not fit into an integer number of zmm registers without padding. Because of this, it is more efficient to process several matrix-vector products at the same time
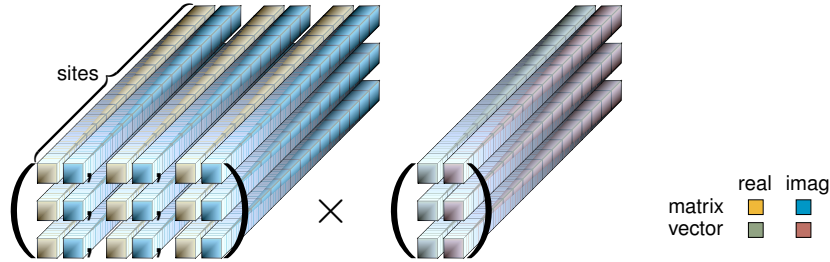

Figure 2: Visualization of a fused matrix-vector product using the 16-fold vectorization scheme. Each lane corresponds to one zmm register, which holds the same element of all 16 sites.

using a site fusion method. A naive single-precision implementation could be to create a "Struct of Arrays" (SoA) object for 16 matrices as well as for 16 vectors. Such a SoA vector object requires 6 zmm registers when it is loaded from memory. One specific register then refers to the real or imaginary part of the same color component gathered from all 16 vectors, thus each vector register can be treated in a "scalar way" (see Fig. 2). These SoA objects are stored in an array using a site ordering technique. Our Dslash kernel runs best with streaming through xy-planes and is specifically adapted for inverting multiple right-hand sides. Therefore, we use a 8-fold site fusion method, combining 8 sites of the same parity in x-direction, which makes the matrix-vector products less trivial and requires explicit in-register align/blend operations. By doing so, we reduce the register pressure by 50% compared to the naive 16-fold site fusion method, leaving more space for the intermediate result of the right-hand sides for each direction
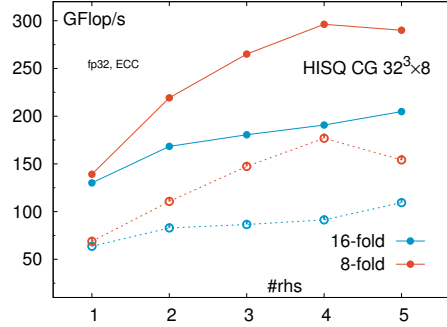
Figure 3: Performance comparison of the 8-fold and 16-fold vectorization scheme measured on a 5110P with enabled ECC. The dashed lines correspond to kernels without software prefetching.

μ. This is why the 8-fold site fusion is 55% faster compared to the 16-fold scheme at 4 rhs (see Fig. 3). For one right-hand side this optimization is insignificant since the 16-fold matrix-vector product requires only 30 of the 32 in hardware available `zmm` registers.

Prefetching: For indirect memory access, i.e. if the array index is a non-trivial calculation or loaded from memory, it is not possible for the compiler to insert software prefetches. The MIC has a L2 hardware prefetcher which is able to recognize simple access pattern. We found that it does a good job for a linear memory access. Thus, there is no need for software prefetching by hand inside the linear algebra operations of the CG. However, the access pattern of the Dslash kernel is too complicated for the hardware prefetcher. Therefore, it is required to insert software prefetches using intrinsics. The inverter runs $2\times$ faster with inserted software prefetches. We unroll the loop over all directions and prefetch always one μ-term ahead. The first right-hand side vector and link of the first μ-term are prefetched at the end of the previous site loop iteration. Considering that there is no reuse of gauge links, it is more efficient to prefetch these into the non-temporal cache. For the vectors we use temporal prefetch hints. It is important to note that software prefeteches are dropped if they cause a page table walk and in order to counterbalance the increased TLB pressure from the multiple right-hand sides, we store, for each lattice site, all rhs contiguously in memory. This approach is 15% faster for large lattices compared to an implementation which stores each right-hand side in a separate array.

3   Comparison

For the Xeon Phi™ we used the Intel® Compiler 14.0 and MPSS 3.3 with disabled instruction cache snooping, huge pages and a balanced processor affinity. For the GPU part we used the NVIDIA® CUDA 6.0 toolkit. For the K40 we enabled GPU boost at the highest possible clock rate 875 MHz. In all benchmarks we left ECC enabled.

In the left panel of Fig. 4 we show the performance as a function of the number of rhs. The maximum number of rhs is limited by memory requirements. We observe roughly the behavior as expected from the increased arithmetic intensity. Comparing the results using four right-hand sides to one right-hand side we find a speedup of roughly 2.05 for the full CG, very close to the increase of 2.16 in arithmetic intensity for the Dslash. Despite the linear algebra operations
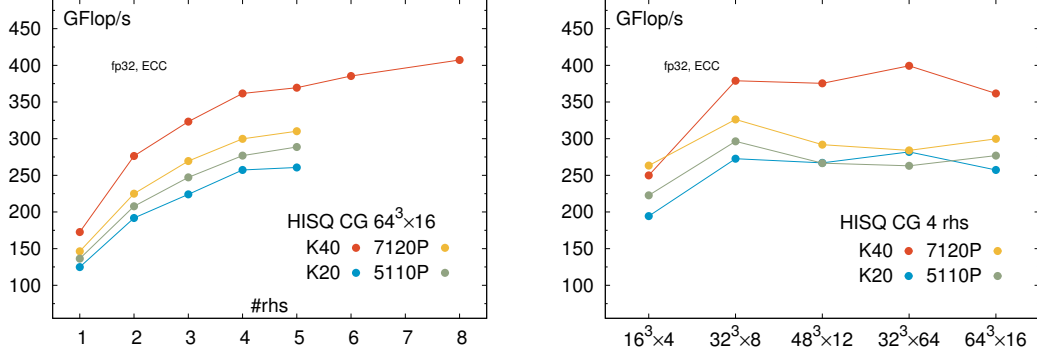
Figure 4: Performance of the HISQ inverter on different accelerators for a $64^3 \times 16$ lattice as a function of the number of rhs (left) and for 4 rhs as a function of the lattice size (right).

that limit the obtainable speedup this is still about 95% of the expected effect. Independent of the number of rhs the ordering with decreasing performance is K40, 7120P, 5110P, K20 with the relative performance roughly given by 1.4, 1.2, 1.1, 1.0 (normalized to K20).

In the right panel we show the performance with 4 rhs as a function of the lattice size. Ignoring the smallest size, the K40 is always superior while the 7120P is faster than the K20. The $32^3 \times 64$ lattice seems to be more expedient for the SIMT model of the GPU.

Acknowledgments: We would like to thank Intel® for providing a Xeon Phi™ system and especially Jeongnim Kim for supporting this work. We acknowledge support from NVIDIA® through the CUDA Research Center program.

References

[1] A. Bazavov et al., Phys. Lett. B **737**, 210 (2014); Phys. Rev. Lett. **113**, 072001 (2014); Phys. Rev. Lett. **111**, 082301 (2013); Phys. Rev. Lett. **109**, 192302 (2012).

[2] E. Follana *et al.* [HPQCD and UKQCD Collaborations], Phys. Rev. D **75**, 054502 (2007); A. Bazavov *et al.* [MILC Collaboration], Phys. Rev. D **82**, 074501 (2010).

[3] See e.g. F. T. Winter *et al.*, arXiv:1408.5925 [hep-lat]; R. Babich *et al.*, arXiv:1109.2935 [hep-lat]; M. A. Clark *et al.*, Comput. Phys. Commun. **181**, 1517 (2010); G. I. Egri *et al.*, Comput. Phys. Commun. **177**, 631 (2007).

[4] See e.g. B. Joó *et al.* ISC 2013, Lecture Notes in Computer Science, Vol, 7905, 40 (2013); R. Li and S. Gottlieb, arXiv:1411.2087 [hep-lat]; Y. C. Jang *et al.*, arXiv:1411.2223 [hep-lat].

[5] O. Kaczmarek et al., arXiv:1409.1510 [hep-lat].

[6] *Intel® Xeon Phi Coprocessor System Software Developers Guide.*

[7] J. D. McCalpin, `http://www.cs.virginia.edu/stream/`

QCDGPU: an Open-Source OpenCL Tool for Monte Carlo Lattice Simulations on Heterogeneous GPU Cluster

Vadim Demchik[1], Natalia Kolomoyets[1]

[1]Dnipropetrovsk National University, Gagarin ave. 72, 49010 Dnipropetrovsk, Ukraine

QCDGPU is an open-source tool for Monte Carlo lattice simulations of the SU(N) gluodynamics and O(N) models. In particular, the package allows to study vacuum thermodynamics in external chromomagnetic fields, spontaneous vacuum magnetization at high temperature in the SU(N) gluodynamics and other new phenomena. The QCDGPU code is implemented in the OpenCL environment and tested on different OpenCL-compatible devices. It supports single- and multi-GPU modes as well as GPU clusters. Also, the QCDGPU has a client-server part for distributed simulations over VPN. The core of Monte Carlo procedure is based on the PRNGCL library, which contains implementations of the most popular pseudorandom number generators. The package supports single-, mixed- and full double-precision including pseudorandom number generation. The current version of the QCDGPU is available online.

1 Introduction

Intensive development of computational methods, along with a rapid progress of computer technology has opened up the possibility of studying many problems of quantum field theory that can not be resolved within the analytical approach. One of such computational methods is lattice Monte Carlo (MC) method, which significantly expanded our understanding of many high-energy phenomena. Lattice MC simulations are based on the procedure of infinite-dimensional path integral calculation with the finite sums on a discrete space-time lattice. Due to locality of basic algorithms, the method is well suited for massively parallel computational environment. Therefore the advent of graphics processing units (GPU) as affordable computational platform reasonably causes a great interest.

Historically, NVIDIA CUDA has become the first widespread platform for general purpose computing on GPUs. However, the requirement of cross-platform compatibility yields to the development of the new open standard computational language OpenCL. Currently the share of OpenCL usage reaches 30% of all scientific researches involving GPUs [1].

Nowadays, there are several software packages to provide MC lattice simulations on GPUs (for example, [2], [3], [4], [5] and [6]). Nevertheless, existing software tools cannot cover many modern high-energy physics problems like investigations of O(N) scalar models or study of SU(N) vacuum in (chromo)magnetic fields. Therefore, we have developed a new software package, QCDGPU to provide the possibility of such kind of explorations. The general aim of

the QCDGPU software [7] is the production of lattice gauge field configurations for O(N) and SU(N) (with or without (chromo)magnetic fields) models with further statistical processing of measurable quantities on GPUs.

2 Structure of QCDGPU

A full platform-independence principle is taken as a basis while constructing the QCDGPU package. Correspondingly, OpenCL was chosen as a GPGPU platform for the package to provide independence on GPU hardware vendors. Host code is implemented in C++ language with minimal external library dependence, which ensures it runs equally well on Windows and Linux operating systems.
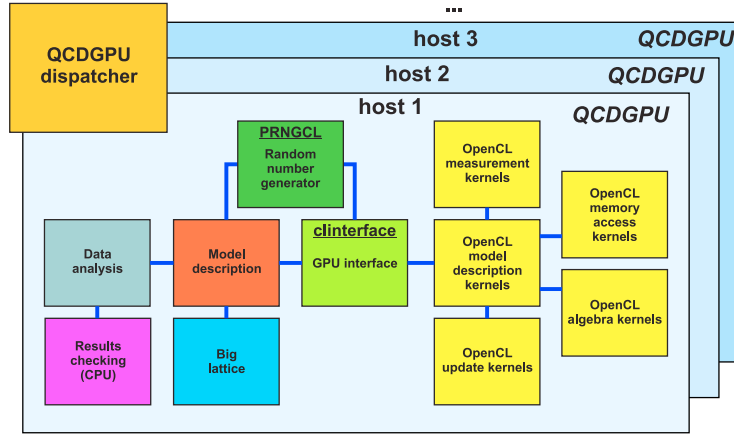

Figure 1: Structure of the QCDGPU package

From the programming point of view the QCDGPU package is a set of independent modules that provide different functions. The structure of QCDGPU is shown as a schematic diagram in Fig. 1.

The core of the package is the CLInterface module which unifies the interaction of the host code with different computing devices. It performs initialization and finalization of computing devices, prepares and adjusts memory objects and OpenCL-kernels for execution, controls and schedules the program flow, etc. All references to memory objects and kernels are organized through integer identifiers. This approach removes direct dealing with OpenCL types and structures, which essentially simplifies programming.

Another important function of the CLInterface module is a caching of previously compiled OpenCL programs to reduce startup time of their execution by creating .bin-files with compiled code for a particular device with distinct computational parameters. The reason to implement an external caching is caused by the fact that not all OpenCL SDK vendors provide correct caching and tracking of dependent source code files. Compilation-specific and additional parameters (like computing platform and device, program options, etc.) are written to the corresponding .inf-files. The uniqueness and code modification is controlled by MD5-hash calculation for each OpenCL source code. Any code or compiler parameters changing leads to a

new rebuilding of the program with corresponding .bin- and .inf-files generation. These files are created for a new startup parameters set. If MD5 and other parameters fit the existing .inf-file, the CLInterface module uses the previously built program from .bin-file. If only MD5-hash is changed (that means kernel source code update), old .bin and .inf files are overwritten. Such a technique provides significant runtime speedup for real MC simulations, because usually a limited set of parameters is used to perform such computer experiments.

Moreover, instant profiling of kernels and host-device data transfer are also performed with the module CLInterface. Due to a slightly slowing up of the package by profiling, it is turned off by default. Device errors handling is performed with CLInterface too. All errors are collected in .log-file. In case of critical errors the program stops.

Model description block is represented by the SUNCL and ONCL modules, which describe SU(N) and O(N) models, correspondingly. The quantum fields in the SU(N) case are represented by $N \times N$ complex matrices living on lattice links. Not all the components of these matrices are independent, so it reduces desired storing data. In the O(N) model case the scalar fields are N-vectors placed in lattice sites. While organizing such matrices and vectors, it is taken into account a common GPU memory architecture that is optimized for the storage of 4-vectors. Such storage format provides additional significant speedup for a program on all computing devices. The whole lattice is a multidimensional set of data that is presented in the QCDGPU package in the following way. The first dimension (fastest index) enumerates lattice site. The next dimension contains the spatial directions of lattice links (in the case of O(N) models this is absent). The last dimension (slowest index) is connected with the gauge group.

The lattice update is realized by decomposing sites in odd and even ones (checkerboard scheme), and, if necessary, into separate parts to handle big lattices. For SU(N) link update a (pseudo)heat-bath algorithm [8, 9] is used, while for O(N) update an improved Metropolis algorithm [10] is implemented.

Pseudorandom numbers for MC simulations are produced with the PRNGCL library [11]. The most widely used generators in HEP lattice simulations are realized in it (RANMAR, RANECU, XOR7, XOR128, MRG32K3a and RANLUX with various luxury levels). Since almost all realized generators (apart from MRG32k3a) have not a general scheme for multi-thread execution, parallelization is made by the random seeding method – each computing thread uses its own seed table, which is uniquely initialized during startup. Initialization of PRNG is made by one integer parameter RANDSERIES. If it is set to zero, system time is chosen for its value. PRNGCL library provides very important feature for MC simulations – repeatability of results on different hardware and operating systems. PRNGCL possesses the possibility to produce pseudorandom numbers with single or double precision to study tiny effects on a lattice.

The possibility to work with big lattices is implemented in QCDGPU through dividing the whole lattice into several parts (Big lattice module). Each part can be simulated on one or several devices. In the case of a multi-GPU system or heterogeneous GPU cluster, dividing the lattice into unequal parts provides the possibility to hide the differences in performance of the different devices. The slowest device gets the smallest lattice part, while the most powerful device operates with biggest lattice portion. The division is implemented along the x spatial axis, because the package is designed mainly for investigation of finite temperature effects, where $L_t < L_s$, $L_{t/s}$ being the lattice size in the temporal and spatial directions, respectively.

In order to hide data transfer among computing devices, additional checkerboard decomposition is used: the odd and even parts of the lattice are updated alternately. In this case, the information exchange of lattice boundary layers is performed in asynchronous mode – while even sites are being updated, transfer of the information at the boundary layer sites takes place and vice versa. That is why it is preferable to divide the lattice into an even number of parts.

In the Data analysis module the statistical analysis of the data collected during one program run is performed. The averages and variances over each run of measured quantities are calculated here. The Data analysis module also provides data distribution analysis for particular measurements.

The Results checking module serves to control the correctness of obtained results. It compares the measurements on the last gauge configuration obtained on GPU with results of traditional sequential approach on CPU. For deep debugging and dynamical comparison of results the Results checking module can independently produce lattice gauge configurations on the same pseudorandom numbers as the device. This module is turned off by default to speed up the program and can be activated.

One of the most widely spread error while developing lattice-based programs is incorrect linking among different lattice sites. Obviously, such an error may bring unnecessary lattice connections and lead to nonphysical results. To overcome such problems the QCDGPU package provides additional lattice initialization with the so-called GID update. Unlike "cold" (ordered) and "hot" (disordered) starts, in this case the lattice is filled with predefined numbers (GID start), which are uniquely determined with each computing thread. Update is also performed with some predefined numbers instead of PRNs. To exclude rounding errors in the heat-bath procedure, the expression for the coefficients of the identity matrix of SU(2) subgroups is changed and the obtained configuration is always accepted. This method provides a good possibility to compare MC simulation results with some alternative realization of the key procedures (for example, mathematical packages).

3   Run description

In actual calculations it is convenient to run QCDGPU in pair with an external program GPU-dispatcher, which prepares the init.dat file with run parameters (like lattice geometry, gauge group, precision and so on). init.dat is an ordinary text file containing those parameters in the form `PARAMETER = value`. The complete list of run parameters may be found in the poster [12].

The program QCDGPU is run in the standard way, `QCDGPU_root_directory> ./QCDGPU init.dat`. During execution it creates several files program$i$.bin and program$i$.inf, where i is index number of the binary file. The .bin files contain compiled OpenCL kernels. Additional information needed for those programs is written in the .inf files.

The possibility of regular saving of the program state, including the state of pseudorandom number generators, is realized in QCDGPU for resuming the interrupted simulation. It allows to interrupt the simulation at any time, and provides basic fault tolerance (power or hardware failure). Saving frequency can be set manually. Saved .qcg-file with the computational state is portable – the computation can be continued on another device.

The result of the program execution is written to the model$n$-yy-Month-dd-hh-mm-ss.txt file, which contains run parameters, average over run values of measured quantities, their variances

over run, the table of averaged over configuration quantities. If Results checking module is on then CPU comparison is included in the output file.

After program execution the file finish.txt is created. It prevents QCDGPU from premature run. To start the program one more time, this file must be removed. GPU-dispatcher creates a new init.dat file, removes finish.txt, and QCDGPU runs with new startup parameters. Such asynchronous task scheduling via operating system tools causes extremely low CPU load in stand-by mode.

Using QCDGPU in pair with GPU-dispatcher allows to perform MC simulations on several hosts at the same time. Each copy of the main computational program is launched on the corresponding host, while parameters passing and launch control are carried out by GPU-dispatcher. It sequentially looks through folders and sets a new task for the first free host.

4   Performance results

To estimate the performance of the QCDGPU package, several devices were used: NVIDIA GeForce GTX970, NVIDIA GeForce GTX980 (NVIDIA CUDA SDK 7.0), AMD Radeon R9 270X, AMD Radeon HD7970 (AMD APP SDK 3.0 Beta), Intel Xeon Phi 31S1P and Intel Xeon CPU E5-2609 (Intel OpenCL SDK 1.2 4.6.0.92). In Table 1 the timings for one sweep in seconds are presented for SU(3) gauge theory for three lattice sizes for single, mixed and double precisions. The mixed precision means that all calculations are done with double precision except for PRNs production. Each sweep consists of the lattice update by multihit heat-bath method (10 hits are set) and measurement of the Wilson action.

Lattice / Device	Single			Mixed			Double		
	16^4	24^4	32^4	16^4	24^4	32^4	16^4	24^4	32^4
NVIDIA GTX980	0.01	0.04	0.13	0.02	0.08	0.25	0.03	0.20	0.69
NVIDIA GTX970	0.01	0.06	0.18	0.02	0.10	0.34	0.04	0.27	0.94
AMD R9 270X	0.02	0.10	0.32	0.05	0.22	0.68	0.14	0.82	2.65
AMD HD7970	0.02	0.08	0.26	0.03	0.14	0.46	0.12	0.66	2.28
E5-2609 CPU	0.19	0.93	2.89	0.23	1.14	3.56	0.59	3.60	11.77
Intel Xeon Phi	0.47	2.29	7.18	0.57	2.65	8.20	1.81	11.3	36.9

Table 1: Timings (in seconds) of one sweep for SU(3) gluodynamics

It can be seen that the computing time depends linearly on the lattice volume for almost all devices. Despite of significant difference in the peak performance for modern GPUs, QCDGPU shows much better ratio of double/single precision timings. The run on Intel Xeon Phi is performed just to test compatibility of the QCDGPU package, the program is not optimized for Intel MIC architecture.

5   Conclusions

QCDGPU package is a new software tool for MC lattice simulations on OpenCL-compatible computing devices. It provides a possibility to study vacuum thermodynamics in extreme

conditions and measuring of nonconventional quantities like Polyakov loop and action spatial distribution, Cartesian components of SU(N) electromagnetic field tensor. QCDGPU is also applicable for simulating scalar O(N) model in the quantum field theory on GPUs.

Underlying full platform-independence principle makes it possible the usage of the QCDGPU package on different OpenCL-compatible computing hardware as well as for most popular operating systems without any source code modifying. Additionally, the package almost does not load the central processor. Together with platform-independence it provides a great opportunity to use the package in local networks as a background software for lattice simulations.

The package demonstrates the best performance results in multi-GPU simulations for trivial parallelization of a task. This approach consists of whole lattice simulation with certain set of adjustable parameters by each involved computing device. The main bottleneck of GPU computing is still relatively low throughput of host-to-device connection. Moreover, interconnection between computing nodes in GPU cluster possesses even lower throughput then host-to-device connection. Obviously, it is better to carry out simulations directly in device memory to eliminate data transfer to/from device. However, for large lattices this is impossible due to the lack of device memory and the only way is to separate lattices into several parts on one host system.

The package instantly provides data production with single, mixed and double precision. The usage of double precision on a regular basis may lead to unnecessary waste of computing time. To overcome this issue internal pseudorandom number generators are built to provide double precision numbers from numbers with single precision by applying a special procedure [13].

Undoubtedly, QCDGPU package is continuously developing to provide additional features that can improve performance and extend the class of tasks solvable with the package. One of such features is autotuning of start-up parameters with internal micro-benchmarks of involved computing devices.

The source codes of the current version of the QCDGPU package and some examples of result files are publicly available [7].

References

[1] High performance computing on graphics processing units, `http://hgpu.org/`.

[2] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi, Comput. Phys. Commun. **181**, 1517 (2010) [arXiv:0911.3191 [hep-lat]].

[3] N. Cardoso and P. Bicudo, Comput. Phys. Commun. **184**, 509 (2013) [arXiv:1112.4533 [hep-lat]].

[4] M. Schrock and H. Vogt, PoS LATTICE **2012**, 187 (2012) [arXiv:1209.4008 [hep-lat]].

[5] M. Bach, V. Lindenstruth, O. Philipsen and C. Pinke, Comput. Phys. Commun. **184**, 2042 (2013) [arXiv:1209.5942 [hep-lat]].

[6] M. Di Pierro, PoS LATTICE **2013**, 043 (2013).

[7] QCDGPU, `https://github.com/vadimdi/QCDGPU`

[8] A. D. Kennedy and B. J. Pendleton, Phys. Lett. B **156** 393 (1985).

[9] N. N. Cabibbo and E. Marinari, Phys. Lett. B **119** 387 (1982).

[10] M. Bordag, V. Demchik, A. Gulov and V. Skalozub, Int. J. Mod. Phys. A **27** 1250116 (2012).

[11] V. Demchik, ch. 12 in "Numerical Computations with GPUs," ed. V. Kindratenko, Springer (2014).

[12] N. Kolomoyets, V. Demchik, GPUHEP (2014), `https://agenda.infn.it/getFile.py/access?contribId=8&sessionId=15&resId=0&materialId=poster&confId=7534`.

[13] V. Demchik and A. Gulov, arXiv:1401.8230 [cs.MS], 1–4 (2014).

CL²QCD - Lattice QCD based on OpenCL

Owe Philipsen[1], Christopher Pinke[1], Alessandro Sciarra[1], Matthias Bach[2]

[1]ITP, Goethe-Universität, Max-von-Laue-Str. 1, 60438 Frankfurt am Main
[2]FIAS, Goethe-Universität, Ruth-Moufang-Str. 1, 60438 Frankfurt am Main

We present the Lattice QCD application CL²QCD, which is based on OpenCL and can be utilized to run on Graphic Processing Units as well as on common CPUs. We focus on implementation details as well as performance results of selected features. CL²QCD has been successfully applied in LQCD studies at finite temperature and density and is available at http://code.compeng.uni-frankfurt.de/projects/clhmc.

1 Lattice QCD at Finite Temperature

Lattice QCD (LQCD) successfully describes many aspects of the strong interactions and is the only method available to study QCD from first principles. The idea is to discretize space-time on a $N_\sigma^3 \times N_\tau$ hypercube with lattice spacing a and treat this system with numerical methods. State-of-the-art lattice simulations require high-performance computing and constitute one of the most compute intensive problems in science. The discretization procedure is not unique and several different lattice theories of QCD have been developed. It is important, in general, to cross check each result using different formulations.

The QCD phase diagram is of great interest both theoretically and experimentally, e.g. at the dedicated programs at RHIC at Brookhaven, LHC at CERN or at the future FAIR facility in Darmstadt[1]. On the lattice, studies at finite temperature T are possible via the identification $T = (a(\beta)N_\tau)^{-1}$. Thus, scans in T require simulations at multiple values of the lattice coupling β. In addition, to employ a scaling analysis, simulations on various spatial volumes N_σ^3 are needed (to avoid finite size effects one typically uses $N_\sigma/N_\tau \approx 3$). Hence, studies at finite T naturally constitute a parallel simulation setup. Currently, these investigations are restricted to zero chemical potential μ, as the sign-problem prevents direct simulations at $\mu > 0$. To circumvent this issue one can use reweighting, a Taylor series approach or one can employ a purely imaginary chemical potential μ_I.

On the lattice, observables are evaluated by means of importance sampling methods by generating ensembles of gauge configurations $\{U_m\}$ using as probability measure the Boltzmann-weight $p[U, \phi] = \exp\{-S_{\text{eff}}[U, \phi]\}$. Expectation values are then

$$\langle K \rangle \approx \frac{1}{N} \sum_m K[U_m] \, .$$

These ensembles are commonly generated using the Hybrid-Monte-Carlo (HMC) algorithm [1], which does not depend on any particular lattice formulation of QCD.

---

[1]See http://www.bnl.gov/rhic/, http://home.web.cern.ch/, and http://www.fair-center.de .

	LOEWE -CSC		SANAM
GPU nodes	600	40	304
GPUs/node	1 × AMD 5870	2 × AMD S10000	2 × AMD S10000
CPUs/node	2 × Opteron 6172	2 × Intel Xeon E5-2630 v2	2 × Xeon E5-2650

Table 1: AMD based clusters where CL^{2}QCD was used for production runs.

The fermions enter in the effective action S_{eff} via the fermion determinant $\det D$, which is evaluated using pseudo-fermions ϕ, requiring the inverse of the fermion matrix, D^{-1}. The fermion matrix D is specific to the chosen discretization. The most expensive ingredient to current LQCD simulations is the inversion of the fermion-matrix

$$D\phi = \psi \quad \Rightarrow \quad \phi = D^{-1}\psi \,,$$

which is carried out with Krylov subspace methods, e.g. conjugate gradient (CG). During the inversion, the matrix-vector product $D\phi$ has to be carried out multiple times. The performance of this operation, like almost all LQCD operations, is limited by the memory bandwidth. For example, in the Wilson formulation, the derivative part of D, the so-called $\not{D}$, requires to read and write 2880 Bytes per lattice site in each call, while it performs *only* 1632 FLOPs per site, giving a rather low numerical density ρ (FLOPs per Byte) of ~ 0.57. In the standard staggered formulation, the situation is even more bandwidth-dominated. To apply the discretization of the Dirac operator on a fermionic field ($D_{KS}\,\phi$) 570 FLOPs per each lattice site are performed and 1584 Bytes are read or written, with a consequent smaller ρ of ~ 0.35. This emphasizes that LQCD requires hardware with a high memory-bandwidth to run effectively, and that a meaningful measure for the efficiency is the achieved bandwidth. In addition, LQCD functions are local, i.e. they depend on a number of nearest neighbours only. Hence, they are very well suited for parallelization.

2   OpenCL and Graphic Cards

Graphics Processing Units (GPUs) surpass CPUs in peak performance as well as in memory bandwidth and can be used for general purposes. Hence, many clusters are today accelerated by GPUs, for example LOEWE -CSC in Frankfurt [2] or SANAM [3] (see Table 1). GPUs constitute an inherently parallel architecture. As LQCD applications are always memory-bandwidth limited (see above) they can benefit from GPUs tremendously. Accordingly, in recent years the usage of GPUs in LQCD simulations has increased. These efforts mainly rely on CUDA as computing language, applicable to NVIDIA hardware *only*[2]. A hardware independent approach to GPU applications is given by the *Open Computing Language* (OpenCL)[3], which is an open standard to perform calculations on heterogeneous computing platforms. This means that GPUs and CPUs can be used together within the same framework, taking advantage of their synergy and resulting in a high portability of the software. First attempts to do this in LQCD have been reported in [4].

An OpenCL application consists of a *host* program coordinating the execution of the actual functions, called *kernels*, on *computing devices*, like for instance GPUs or a CPUs. Although the

---

[2]See `https://developer.nvidia.com/cuda-zone` and `https://github.com/lattice/quda` for the **QUDA** library.
[3]See `https://www.khronos.org/opencl` .

hardware has different characteristics, GPU programming shares many similarities with parallel programming of CPUs. A computing device consists of multiple *compute units*. When a kernel is executed on a computing device, actually a huge number of kernel instances is launched. They are mapped onto *work-groups* consisting of *work-items*. The work-items are guaranteed to be executed concurrently only on the processing elements of the compute unit (and share processor resources on the device).

Compared to the main memory of traditional computing systems, on-board memory capacities of GPUs are low, though increasing more and more[4]. This constitutes a clear boundary for simulation setups. Also, communication between host system and GPU is slow, limiting workarounds in case the available GPU memory is exceeded. Nevertheless, as finite T studies are usually carried out on moderate lattice sizes (in particular $N_\sigma \gg N_\tau$), this is less problematic for the use cases CL^{2}QCD was developed for.

3   CL^{2}QCD Features

CL^{2}QCD is a Lattice QCD application based on OpenCL, applicable to CPUs and GPUs. Focusing on Wilson fermions, it constitutes the first such application for this discretization type [5]. In particular, the so-called Twisted Mass Wilson fermions [6, 7], which ensure $\mathcal{O}(a)$ improvement at maximal twist, are implemented. Recently, the (standard) formulation of staggered fermions has been added. Improved gauge actions and standard inversion and integration algorithms are available, as well as ILDG-compatible IO[5] and the RANLUX Pseudo-Random Number Generator (PRNG) [8]. More precisely, CL^{2}QCD provides the following executables.

- **HMC:** Generation of gauge field configurations for $N_f = 2$ Twisted Mass Wilson type or pure Wilson type fermions using the HMC algorithm [1].

- **RHMC:** Generation of gauge field configurations for N_f staggered type fermions using the Rational HMC algorithm [9].

- **SU3HEATBATH:** Generation of gauge field configurations for $SU(3)$ Pure Gauge Theory using the heatbath algorithm [10, 11, 12].

- **INVERTER:** Measurements of fermionic observables on given gauge field configurations.

- **GAUGEOBSERVABLES:** Measurements of gauge observables on given gauge field configurations.

The host program of CL^{2}QCD is set up in `C++`, which allows for independent program parts using `C++` functionalities and also naturally provides extension capabilities. Cross-platform compilation is provided using the CMake framework.[6] All parts of the simulation code are carried out using OpenCL kernels in double precision. The OpenCL language is based on `C99`. In particular, concrete implementations of basic LQCD functionality like matrix-matrix multiplication, but also more complex operations like the $\displaystyle{\not}{D}$ or the (R)HMC force calculation, are found in the kernel files. The kernels are in a certain way detached from the host part as the latter can continue independently of the status of the kernel execution. This nicely shows

---

[4]For instance, the GPUs given in Table 1 have on-board memory capacities of 1, 12 and 3 GB, respectively.
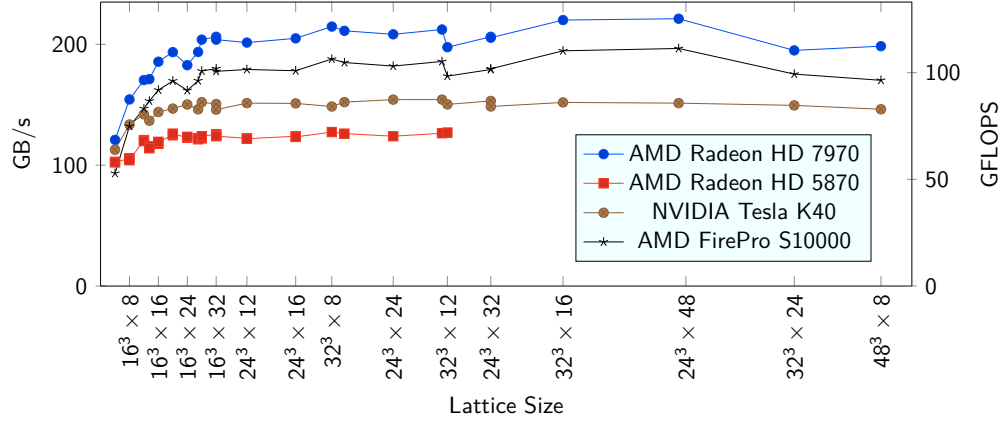
[5]Via LIME, see `http://usqcd.jlab.org/usqcd-docs/c-lime` .

[6]See `http://www.cmake.org` .

OWE PHILIPSEN, CHRISTOPHER PINKE, ALESSANDRO SCIARRA, MATTHIAS BACH

Figure 1: Performace of Wilson $D\!\!\!\!/$ kernel for various lattice sizes on different devices in double precision.

the separation into the administrative part (host) and the performance-critical calculations (kernels).

OpenCL kernels are compiled at runtime which is mandatory as the specific architecture is not known a priori. On the one hand, this introduces an overhead, but on the other hand allows to pass runtime parameters (like the lattice size) as compile time parameters to the kernels, saving arguments and enabling compiler optimization for specific parameter sets. In addition, the compiled kernel code is saved for later reuse, e.g. when resuming an HMC chain with the same parameters on the same architecture. This reduces the initialization time. Kernel code is common to GPUs and CPUs, device specifics are incorporated using macros. It is ensured that that memory objects are treated in a *Structure of arrays (SOA)* fashion on GPUs, which there is crucial for optimal memory access as opposed to *Array of structures (AOS)*.

In general, it is desireable to be able to test every single part of code on its own and to have as little code duplication as possible. This is at the heart of the *Test Driven Development* [13] and *Clean Code* [14] concepts, which we follow during the development of CL²QCD. Unit tests are implemented utilizing the BOOST[7] and CMAKE unit test frameworks. Regression tests for the OpenCL parts are mandatory due to the runtime compilation. In particular, as LQCD functions are local in the sense that they depend only on a few nearest neighbours, one can calculate analytic results to test against and often the dependence on the lattice size is easily predictable. Another crucial aspects to guarantee maintainability and portability of code is to avoid dependence of the tests on specific environments. For example, this happens when random numbers are used (e.g. for trial field configuration). If this is the case, a test result then depends not only on the used PRNG but also on the hardware in a multi-core architecture.

4   Performance of $D\!\!\!\!/$

Our Wilson $D\!\!\!\!/$ implementation, which is crucial for overall performance, shows very good performance on various lattice sizes (Figure 1) and outperforms performances reported in the
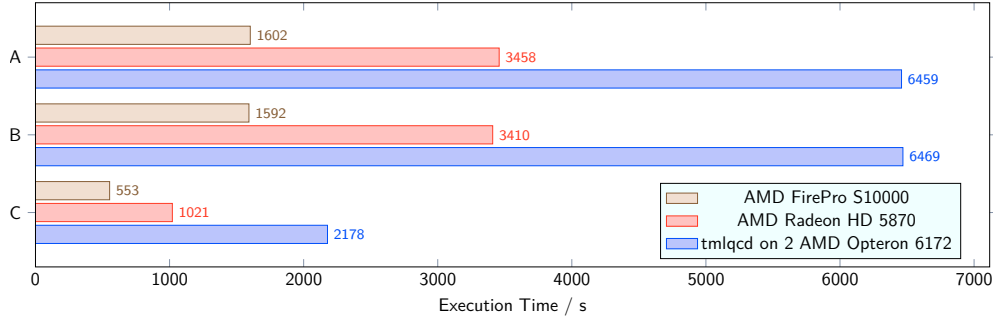
---

[7]See http://www.boost.org .

Figure 2: HMC performance for different Setups A, B and C (setup A having the smallest fermion mass) for $N_\tau = 8, N_\sigma = 24$. The HMC is compared on different GPUs and compared to a reference code [15] running on one LOEWE -CSC node.

literature (see [5]). We are able to utilize $\sim 80\%$ of the peak memory bandwidth on the AMD Radeon HD 5870, Radeon HD 7970 and FirePro S10000. Note that the code runs also on NVIDIA devices as shown in the figure, however, with lower performance since AMD was the primary development platform and no optimization was carried out here.

The staggered D_{KS} implementation, which plays the same role as $\slashed{D}$ regarding the overall speed of the code, shows also good performance on various lattice sizes. We are able to utilize $\sim 70\%$ of the peak memory bandwidth on the AMD Radeon HD 5870 and AMD Radeon HD 7970. Due to its recent development, the implementation of the staggered code can be further optimized. So far no other benchmark for a possible comparison is present in the literature. Again, the code runs also on NVIDIA devices, though also here the performance is lower for the same reasons explained above regarding the Wilson $\slashed{D}$.

5 Algorithmic Performance

The full HMC application also performs very well compared to a reference CPU-based code `tmLQCD` [15] (see Figure 2). The `tmLQCD` performance was taken on one LOEWE -CSC node. Compared to `tmLQCD`, the older AMD Radeon HD 5870 is twice as fast. The newer AMD FirePro S10000 again doubles this performance. This essentially means that we gain a factor of 4 in speed, comparing a single GPU to a whole LOEWE -CSC node. In addition, it is interesting to look at the price-per-flop, which is much lower for the GPUs used then for the used CPUs.

As on-board memory is the biggest limiting factor on GPUs, using multiple GPUs is of great interest [16]. In CL²QCD it is possible to split the lattice in time direction [17].

6 Conclusions and Perspectives

We presented the OpenCL-based LQCD application CL²QCD. It has been successfully applied in finite temperature studies on LOEWE -CSC and SANAM supercomputers (e.g. [18]), providing a well-suited basis for future applications. CL²QCD is available at

http://code.compeng.uni-frankfurt.de/projects/clhmc .

Additional features will be added to CL^{2}QCD according to the needs of the physical studies. In the near future, these will cover the extension of Wilson fermions to $N_f = 2 + 1$ flavours and the implementation of the clover discretization. Adding to that, optimizations of performances of staggered fermions and the inclusion of improved staggered actions are planned.

7 Acknowledgments

References

[1] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Phys. Lett.*, B195:216–222, 1987.

[2] Matthias Bach, Matthias Kretz, Volker Lindenstruth, and David Rohr. Optimized HPL for AMD GPU and multi-core CPU usage. *Comput. Sci.*, 26(3-4):153–164, June 2011.

[3] David Rohr, Sebastian Kalcher, Matthias Bach, A. Alaqeeli, H. Alzaid, Dominic Eschweiler, Volker Lindenstruth, A. Sakhar, A. Alharthi, A. Almubarak, I. Alqwaiz, and R. Bin Suliman. An Energy-Efficient Multi-GPU Supercomputer. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014, Paris, France. IEEE*, 2014.

[4] O. Philipsen, C. Pinke, C. Schäfer, L. Zeidlewicz, and M. Bach. LatticeQCD using OpenCL. *PoS*, LATTICE2011:044, 2011.

[5] Matthias Bach, Volker Lindenstruth, Owe Philipsen, and Christopher Pinke. Lattice QCD based on OpenCL. *Comput.Phys.Commun.*, 184:2042–2052, 2013.

[6] A. Shindler. Twisted mass lattice QCD. *Phys. Rept.*, 461:37–110, 2008.

[7] R. Frezzotti and G.C. Rossi. Chirally improving Wilson fermions. 1. O(a) improvement. *JHEP*, 0408:007, 2004.

[8] Martin Lüscher. A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications*, 79(1):100 – 110, 1994.

[9] M. A. Clark and A. D. Kennedy. Accelerating staggered-fermion dynamics with the rational hybrid monte carlo algorithm. *Phys. Rev. D*, 75:011502, Jan 2007.

[10] M. Creutz. Monte Carlo Study of Quantized SU(2) Gauge Theory. *Phys. Rev.*, D21:2308–2315, 1980.

[11] Nicola Cabibbo and Enzo Marinari. A new method for updating SU(N) matrices in computer simulations of gauge theories. *Physics Letters B*, 119(4-6):387 – 390, 1982.

[12] A.D. Kennedy and B.J. Pendleton. Improved heatbath method for monte carlo calculations in lattice gauge theories. *Physics Letters B*, 156(5-6):393 – 399, 1985.

[13] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[15] K. Jansen and C. Urbach. tmLQCD: a program suite to simulate Wilson Twisted mass Lattice QCD. *Comput. Phys. Commun.*, 180:2717–2738, 2009.

[16] R. Babich et al. Scaling Lattice QCD beyond 100 GPUs. 2011.

[17] M. Bach, V. Lindenstruth, C. Pinke, and O. Philipsen. Twisted-Mass Lattice QCD using OpenCL. *PoS*, LATTICE2013:032, 2014.

[18] Owe Philipsen and Christopher Pinke. Nature of the roberge-weiss transition in $N_f = 2$ qcd with wilson fermions. *Phys. Rev. D*, 89:094504, May 2014.

cuLGT: Lattice Gauge Fixing on GPUs

Hannes Vogt[1], Mario Schröck[2]

[1]Institut für Theoretische Physik, Universität Tübingen, Germany
[2]INFN Roma Tre, Italy

We adopt CUDA-capable Graphic Processing Units (GPUs) for Landau, Coulomb and maximally Abelian gauge fixing in 3+1 dimensional SU(3) and SU(2) lattice gauge field theories. A combination of simulated annealing and overrelaxation is used to aim for the global maximum of the gauge functional. We use a fine grained degree of parallelism to achieve the maximum performance: instead of the common 1 thread per site strategy we use 4 or 8 threads per lattice site. Here, we report on an improved version of our publicly available code (www.cuLGT.com) which again increases performance and is much easier to include in existing code. On the GeForce GTX 580 we achieve up to 470 GFlops (utilizing 80% of the theoretical peak bandwidth) for the Landau overrelaxation code.

1   Introduction

In lattice QCD, gauge fixing is necessary to study the fundamental (gauge-variant) two point functions of QCD and to compare with continuum results. Lattice gauge fixing is an optimization problem with very many degrees of freedom and many local maxima. Each local maximum corresponds to a so-called Gribov copy. These Gribov copies may have an effect on gauge-variant quantities. One way to get a unique gauge copy is to search for the global maximum of the functional. This task is very time-consuming and an efficient implementation on modern hardware is desirable. Since the optimization algorithms are nicely parallelizable, graphic processing units (GPUs) are perfectly suited for these algorithms. Here, we report on latest improvements to cuLGT, a CUDA-based software for lattice gauge fixing that evolved from the first GPU gauge fixing code presented in [1]. In its first version, cuLGT offered standalone applications for Landau, Coulomb and maximally Abelian gauge fixing in 3+1 dimensional SU(3) gauge theories using a combination of overrelaxation and simulated annealing [2]. One of the aims of cuLGT2 was to offer the possibility to integrate the gauge fixing routines in existing lattice QCD frameworks, like the MILC code [3]. In the following, we will restrict the discussion to Landau gauge and the overrelaxation algorithm. For a more complete treatment we refer to the original work [2].

An alternative GPU gauge fixing software using a Fourier accelerated steepest descent algorithm is available from the authors of [4].

2 Lattice Gauge Fixing

On the lattice, a link variable $U_\mu(x) \in \mathrm{SU}(N_c)$ transforms with respect to gauge transformations $g(x) \in \mathrm{SU}(N_c)$ as

$$U_\mu \to U_\mu^g = g(x)U_\mu(x)g^\dagger(x + \hat{\mu}).$$

The continuum Landau gauge condition,

$$\partial_\mu A_\mu(x) = 0,$$

translates on the lattice to the discrete derivative

$$\Delta(x) = \sum_\mu \left(A_\mu(x) - A_\mu(x - \hat{\mu}) \right) = 0, \tag{1}$$

where the connection between the continuum gauge fields $A_\mu(x)$ and the lattice links is established by

$$A_\mu(x) = \frac{1}{2iag} \left[U_\mu(x) - U_\mu^\dagger(x) \right]_{\mathrm{traceless}}.$$

In each local maximum of the Landau gauge fixing functional

$$F^U[g] = \frac{1}{N_d N_c V} \sum_x \sum_\mu \mathrm{Re\,tr} \left[g(x)U_\mu(x)g^\dagger(x + \hat{\mu}) \right] \tag{2}$$

the lattice Landau gauge condition (1) is satisfied. In the normalization of (2), $N_d = 4$ denotes the number of space-time dimensions and $V = N_s^3 N_t$ is the lattice volume. Instead of considering the complete functional (2), we rewrite it in a sum of local terms by factoring out the gauge transformation at lattice point x,

$$F^U[g] = \frac{1}{2N_d N_c V} \sum_x \mathrm{Re\,tr} \left[g(x)K(x) \right].$$

Then, we optimize

$$\mathrm{Re\,tr} \left[g(x)K(x) \right] = \mathrm{Re\,tr} \left[g(x) \sum_\mu \left[U_\mu(x) + U_\mu^\dagger(x - \hat{\mu}) \right] \right]. \tag{3}$$

with respect to $g(x)$. All other (inactive) gauge transformations are set to the identity. The local functional (3) only depends on links that start or end at lattice site x.

The local maximum at each site can be found directly as

$$g(x) = K^\dagger(x)/\sqrt{\det K^\dagger(x)}$$

for the gauge group SU(2). For $N_c > 2$ one iterates over the SU(2) subgroups. To overcome the problem of critical slowing down, the authors of [5] proposed to apply an overrelaxation update by replacing $g(x)$ by $g^\omega(x)$ with $\omega \in [1, 2)$. Since only transformations at neighboring sites interfere, we can use a *checkerboard* layout and sweep first over the *black* and then over the *white* lattice sites. The algorithm is then iterated until the discrete gauge condition (1) is satisfied up to a given numerical precision.

3 GPU optimizations

Most GPU applications in lattice QCD are bound by the bandwidth of global device memory. Therefore, the highest focus should be on ideal memory usage. For an overview of optimization techniques for lattice QCD we refer to [6]. In the following, we will shortly summarize the optimizations that led to the performance of cuLGT1. In Sec. 4 we will report on the improved code cuLGT2.

3.1 Pre-cuLGT (first version)

For maximal throughput the memory access should be coalesced, i.e. consecutive threads should read from consecutive memory addresses. Therefore, we first reorder the lattice into its black and white sublattices according to the checkerboard layout. Within each sublattice we order data such that the site index runs faster than Lorentz and matrix indices.

In order to save bandwidth we do not load full $N_c \times N_c$ matrices from global memory, but only parts of it. Then, we use the properties of the gauge group to reconstruct the full matrix in registers/local memory. For the gauge group SU(3) we use a 12 parameter representation, i.e. the first 2 rows of the matrix.

With these optimizations we already get a remarkable speedup of a factor of ~ 30 over a single core CPU implementation[1] for the SU(3) overrelaxation code. The performance of 120 GFlops is of course far away from the theoretical peak performance of this GPU, however the correct measure of the utilization of the GPU is the achieved memory throughput. Therefore, on the r.h.s. of Fig. 1 we show the throughput in percent of the theoretical peak bandwidth. For this version of the code we only use 20% of the theoretical bandwidth[2].

3.2 cuLGT1

Register pressure turned out to be the main performance bottleneck in the first version of the code. There we used one thread to calculate the update of a site. A simple calculation shows that we would already need 144 registers for storing the 8 links that are involved in a site update (and additional registers are needed for temporary storage in the computation). Since the register limit of GPUs of the Fermi generation is 63 registers (32 bit) per kernel, a lot of data is spilled to the slow global memory. To relax the register pressure we introduced a 8-thread-per-site strategy in [2], where we keep one link (18 parameters) in each thread. The communication among these 8 threads (summation of links to get $K(x)$ and distribution of the result $g(x)$) is done via shared memory. With this optimization we increase performance by a factor of three, using more than 60% of the bandwidth. This version of the code is currently available on our website.

4 cuLGT2

With the development of cuLGT2 we wanted to solve several structural problems of cuLGT1: (a) the parameterization of links was hard-coded, switching to other parameterization would

---

[1]We used our own reference CPU code which runs slightly faster than the publicly available MILC code [3]. However, for a *fair* comparison of CPU vs. GPU performance we would need a highly optimized multi-threaded CPU code.

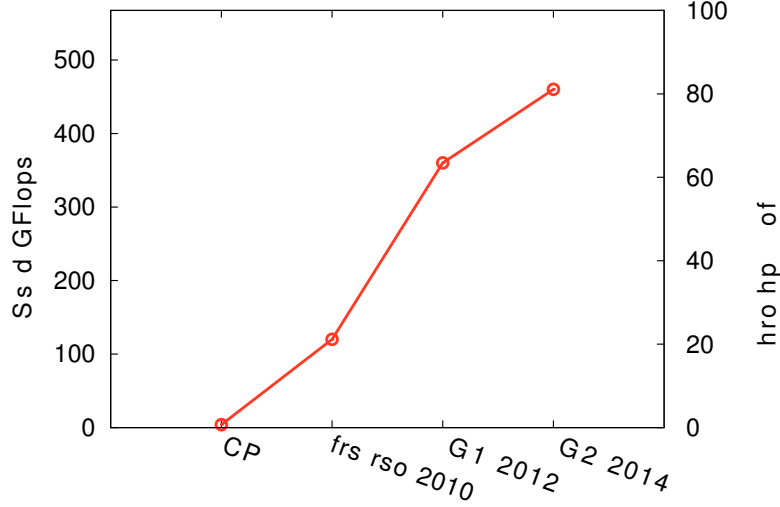[2]With the 12 parameter representation.

Figure 1: Evolution of the performance of cuLGT from the first version to cuLGT2 for the SU(3) Landau gauge overrelaxation code (32^4 lattice in single precision) on a GeForce GTX 580. The CPU performance is from a single core of an Intel Core i7-2600.

have needed a lot of code changes; (b) related to the former point, SU(2) and SU(3) implementations needed a lot of code duplication; (c) switching from the 8-threads-per-site strategy to 4 threads per site was not easily possible; (d) all these points prevented to implement a tool to automatically choose the optimal kernel settings for different GPUs, a technique that is already successfully used in the QUDA library [7].

To systematically solve these issues we decided to completely rewrite major parts of the code. In the following, we leave out most details of the implementation but focus only on the link management. This part might be useful for many other lattice GPU applications, since it allows developers to use high level constructs for writing GPU optimized code. A multi-GPU version of cuLGT2 is not yet available. We refer to cuLGT1 for lattice sizes that need multi-GPU.

4.1 Link management

Already in cuLGT1 we used two separate classes to represent a link in global memory `SU3<Link>` and local memory `SU3<Matrix>`. The former takes care of the memory pattern (details about available memory patterns in [2]) and allows to access a link by specifying the site and the Lorentz index. The data is stored in a linear `float` or `double` array of length $L = V N_d (2N_c^2)$, where $2N_c^2$ is the number of parameters of the $N_c \times N_c$ complex matrix. For using the 12 parameter representation one just reads/stores the first two rows of the matrix. Other representations where the parameters are not just a subset of the full matrix, like the minimal 8 parameter representation, are not provided. Changing the datatype of the global array was

also not possible[3].

With cuLGT2, we decided to introduce an additional abstraction that easily allows changing how links are represented in memory. An implementation of a `ParameterizationType` defines the datatype and the number of elements. For transformations from one representation to another we define a mediator that overloads the assignment operator for the specific types. Links in global memory (`GlobalLink`) are now defined with two template parameters to specify the memory *pattern* and the *parameterization*. The `LocalLink` takes only the *parameterization* type as template parameter. Two examples for `ParameterizationType`s for SU(3) are

```
class SU3Full                        class SU3Vector4
{                                    {
   typedef float TYPE;                  typedef float4 TYPE;
   const static int SIZE = 18;          const static int SIZE = 3;
}                                    }
```

on the l.h.s. a 18 parameter representation with floats, usually used in `LocalLink`; on the r.h.s. a 12 parameter representation as three float4, usually used in `GlobalLink`. A (simplified) example code to perform a gauge transformation is

```
 1 typedef GlobalLink<GPUPattern,SU3Vector4> GLOBALLINK;
 2 typedef LocalLink<SU3Full> LOCALLINK;
 3 void transformLink(Site s, int dir, LOCALLINK gLeft, LOCALLINK gRight)
 4 {
 5    GLOBALLINK global(s, dir);
 6    LOCALLINK local;
 7    local = global;
 8    local = gLeft*local*gRight.hermitian();
 9    global = local;
10 }
```

In line 1 and 2 the parameterizations for the `GlobalLink` and `LocalLink` are defined. Changing the gauge group to SU(2) would only require to set appropriate SU(2) parameterizations here. In line 7 a `GlobalLink` is assigned to a `LocalLink`. The full matrix is implicitly reconstructed. In line 8 the link is transformed. `LocalLink` overloads the multiplication operator and defines a function to compute the hermitian conjugate. The actual implementation of these operations is in the class of the `ParameterizationType`. In line 9 the modified link is written back to global memory, discarding the third line.

4.2   Performance

Although the primary design goal of cuLGT2 was not on performance improvements, we got a speedup compared to cuLGT1. The main improvement in Fig. 1 comes from the use of the 4-threads-per-site strategy instead of 8 threads per site. In Fig. 2 we compare the performance of the code on different GPUs. Only on the Tesla K20s 8-threads-per-site performs better. We do not yet know why this happens and why the result is that bad compared to the GTX Titan. Additional tuning is needed for the K20s.

---

[3]Using for example `float4`/`double2` instead of `float`/`double` would improve bandwidth utilization for memory accesses that cannot be coalesced, like access to links that are neighbors of the current site $U_\mu(x + \hat{\mu})$.
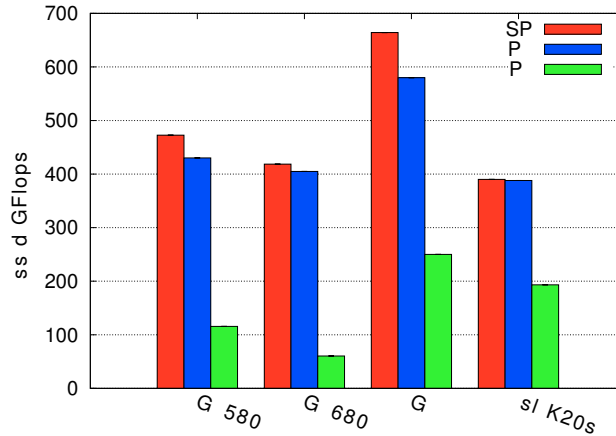
Figure 2: Performance of the cuLGT2 SU(3) Landau gauge overrelaxation code (32^4 lattice) on different GPUs in SP (left), mixed precision (middle) and DP (right)

5   Summary

With the development of cuLGT2 we successfully solved several design issues of cuLGT1. The gauge fixing software is now well modularized which allows us to run the gauge fixing routines from the MILC code. Additionally, the software automatically chooses the optimal kernel setup for different architectures at runtime by trying (a) 4 or 8 threads per site update, (b) different register limits by setting `__launchbounds()` (c) switching texture loads on or off. With the 4-threads-per-site strategy and the improved link management the performance of the code was remarkably improved.

Acknowledgments

References

[1] Mario Schröck. The chirally improved quark propagator and restoration of chiral symmetry. *Phys.Lett.*, B711:217–224, 2012.

[2] Mario Schröck and Hannes Vogt. Coulomb, Landau and Maximally Abelian Gauge Fixing in Lattice QCD with Multi-GPUs. *Comput.Phys.Commun.*, 184:1907–1919, 2013.

[3] MILC Code. `http://www.physics.utah.edu/~detar/milc/`.

[4] Nuno Cardoso, Paulo J. Silva, Pedro Bicudo, and Orlando Oliveira. Landau Gauge Fixing on GPUs. *Comput.Phys.Commun.*, 184:124–129, 2013.

[5] Jeffrey E. Mandula and Michael Ogilvie. Efficient gauge fixing via overrelaxation. *Phys.Lett.*, B248:156–158, 1990.

[6] M.A. Clark, R. Babich, K. Barros, R.C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Comput.Phys.Commun.*, 181:1517–1528, 2010.

[7] QUDA library. `http://lattice.github.io/quda/`.

Chapter 5

GPU in Other Applications

Convenors:

Felice Pantaleo
Claudio Bonati

Novel GPU features: Performance and Productivity.

P. Messmer[1]

[1] NVIDIA

The huge amount of computing power needed for signal processing and off-line simulation makes High-Energy Physics an ideal target for GPUs. Since the first versions of CUDA, considerable progress has been made in demonstrating the benefit of GPUs for these processing pipelines and GPUs are now being deployed for production systems. However, early experiments also showed some of the challenges encountered in HEP specific tasks, including memory footprint, complex control flow, phases of limited concurrency and portability. Many of these concerns have been addressed with recent changes to the GPU hardware and software infrastructure: Unified memory, dynamic parallelism, and priority streams are just some of the features at the developers disposal to fully take advantage of the available hardware. In addition, recently added boards like the TK1 processor for embedded high performance, low power applications enable now CUDA accelerated applications from the sensor to the offline simulations. In this talk I will present some of the latest additions to the GPU hardware, provide an overview of the recent changes to the software infrastructure and will walk through features added in the latest CUDA version.

Contribution not received.

Commodity embedded technology and mobile GPUs for future HPC systems

F. Mantovani[1]

[1] Barcelona Supercomputing Center, Spain

Around 2005-2008, (mostly) economic reasons led to the adoption of commodity GPU in high-performance computing. This transformation has been so effective that in 2013 the TOP500 list of supercomputers is still dominated by heterogeneous architectures based on CPU+coprocessor. In 2013, the largest commodity market in computing is not the one of PCs or GPUs, but mobile computing, comprising smartphones and tablets, most of which are built with ARM-based System On Chips (SoCs). This leads to the suggestion that, once mobile SoCs deliver sufficient performance, mobile SoCs can help reduce the cost of HPC. Moreover mobile SoCs embed GPUs that are in many cases OpenCL/CUDA capable, therefore most of the computing experience gained over these years can be highly beneficial.

Since the end of 2011 the Mont-Blanc project is tackling the challenges related to the use of mobile SoCs in an HPC environment, developing a prototype, evaluating heterogeneous CPU+GPU architectures and porting libraries and scientific applications to ARM based architectures. In view of the experiences within the Mont-Blanc project at the Barcelona Supercomputing Center, this contribution will show preliminary results about performance evaluation of heterogeneous CPU+GPU computation on mobile SoCs and will describe possibilities and challenges involved in developing high-performance computing platforms from low cost and energy efficient mobile processors and commodity components.

Contribution not received.

F. MANTOVANI

Using Graphics Processing Units to solve the classical N-Body Problem in Physics and Astrophysics

Mario Spera[1,2]

[1]INAF-Osservatorio Astronomico di Padova, Vicolo dell'Osservatorio 5, I-35122, Padova, Italy;
[2]Sapienza, Universitá di Roma, P.le A. Moro 5, I-00165 Rome, Italy;

`mario.spera@oapd.inaf.it`   `mario.spera@live.it`

Graphics Processing Units (GPUs) can speed up the numerical solution of various problems in astrophysics including the dynamical evolution of stellar systems; the performance gain can be more than a factor 100 compared to using a Central Processing Unit only. In this work I describe some strategies to speed up the classical N-body problem using GPUs. I show some features of the N-body code `HiGPUs` as template code. In this context, I also give some hints on the parallel implementation of a regularization method and I introduce the code `HiGPUs-R`. Although the main application of this work concerns astrophysics, some of the presented techniques are of general validity and can be applied to other branches of physics such as electrodynamics and QCD.

1 Introduction

The N-body problem is the study of the motion of N point-like particles interacting through their mutual forces that can be expressed according to a specific physical law. In particular, if the reciprocal interaction is well approximated by the Newton's gravity law, we refer to the classical, gravitational N-body problem. The differential equations that describe the kinematics of the N-body system are

$$\begin{cases} \ddot{\mathbf{r}}_i & = -G \sum\limits_{\substack{j=1 \\ j \neq i}}^{N} \frac{m_j}{r_{ij}^3} \mathbf{r_{ij}} \\ \mathbf{r}_i\left(t_0\right) & = \mathbf{r}_{i,0} \qquad\qquad i = 1,2,...,N \\ \dot{\mathbf{r}}_i\left(t_0\right) & = \dot{\mathbf{r}}_{i,0} \end{cases} \tag{1}$$

where t is the time, $\mathbf{r}_i$, $\dot{\mathbf{r}}_i$ and $\ddot{\mathbf{r}}_i$ are the position, the velocity and the acceleration of the i-th particle, respectively, G is the universal gravitational constant, m_j indicates the mass of the particle j, $\mathbf{r}_{i,0}$ and $\dot{\mathbf{r}}_{i,0}$ represent the initial position and velocity and r_{ij} is the mutual distance between particle i and particle j. Although we know that the solution of the system of equations (1) exists and is unique, we do not have its explicit expression. Therefore, the best way to solve the N-body problem is numerical. The numerical solution of the system of equations (1) is considered a challenge despite the considerable advances in both software

development and computing technologies; for instance, it is still not possible to study the dynamical evolution of stellar systems composed of more than $\sim 10^6$ objects without the need of theoretical approximations. The numerical issues come mainly from two aspects:

- *ultraviolet divergence (UVd)*: close encounters between particles $(r_{ij} \to 0)$ produce a divergent mutual force $(F_{ij} \to \infty)$. The immediate consequence is that the numerical time step must be very small in order to follow the rapid changes of positions and velocities with sufficient accuracy, slowing down the integration.

- *infrared divergence (IRd)*: to evaluate the acceleration of the i-th particle we need to take into account all the other $N-1$ contributions because the gravitational force never vanishes $(F_{ij} \to 0 \Leftrightarrow r_{ij} \to \infty)$. This implies that the N-body problem has a computational complexity of $O\left(N^2\right)$.

To control the effects of the UVd and smooth the behavior of the force during close encounters, a parameter ϵ (*softening parameter*) is introduced in the gravitational potential. This leads to an approximate expression for the reciprocal attraction which is

$$\mathbf{F_i} = -G\frac{m_i m_j}{\left(r_{ij}{}^2 + \epsilon^2\right)^{\frac{3}{2}}}\mathbf{r_{ij}}. \tag{2}$$

In this way, the UVd is artificially removed paying a loss of resolution at spatial scales comparable to ϵ and below. An alternative approach concerns the usage of a regularization method, that is a coordinate transformation that modifies the standard N-body Hamiltonian removing the singularity for $r_{ij} = 0$. We briefly discuss this strategy in Sec. 3.

On the other hand, the issues that come from the IRd can be overcome using:

1. *approximation schemes*: the direct sum of inter-particle forces is replaced by another mathematical expression with lower computational complexity. To this category belongs, for instance, the *tree scheme*, originally introduced by Barnes and Hut, which is one of the most known approximation strategies [1];

2. *hardware acceleration*: it is also possible to use more efficient hardware to speed up the force calculation maintaining the $O\left(N^2\right)$ computational complexity of the problem.

For what concerns hardware advances, Graphics Processing Units (GPUs) can act as computing accelerators of the evaluation of the mutual forces. This approach is extremely efficient because a GPU is a highly parallel device which can have up to $\sim 3,000$ cores and run up to $\sim 80,000$ virtual units (GPU threads) that can execute independent instructions at the same time. Since the evaluations of mutual distances can be executed independently, a GPU perfectly matches the structure of the N-body problem. Nowadays, the overwhelming majority of N-body simulations are carried out exploiting the GPU acceleration.

2 The direct N-body code `HiGPUs`

In this section I give an overview of the most common strategies adopted to numerically solve the N-body problem using a GPU. I describe the `HiGPUs` code as an example of GPU optimized N-body code [2]. `HiGPUs`[1] is a direct summation N-body code that implements a Hermite 6th

---

[1]`http://astrowww.phys.uniroma1.it/dolcetta/HPCcodes/HiGPUs.html`

order time integrator. It is written combining utilities of C and C++ and it uses CUDA (or OpenCL), MPI and OpenMP to exploit GPU workstations as well as GPU clusters. The main features of `HiGPUs` and of other GPU optimized N-body codes are the following:

1. THE HERMITE ALGORITHM: the Hermite time integrators (4th, 6th and 8th order) represent the state of the art of direct N-body simulations([4], [3]). In particular, the 4th order scheme is, by far, the most widely used algorithm in this context being particularly efficient in terms of ratio between computing time and accuracy. The Hermite integrators are based on Taylor series of positions and velocities and their most important feature is that they have high accuracy even though they need to evaluate the distances between particles just once per time step. This is a huge advantage in using Hermite integrators if we consider that, for example, a standard Runge-Kutta method needs to evaluate accelerations 4 times per integration step and it is "only" 4th order accurate.

2. BLOCK TIME STEPS: stars in an N-body system can have very different accelerations; this corresponds to have a large variety of evolutionary time-scales. In this context, it is convenient to assign to all the objects their own time step which becomes a function of the physical parameters that describe the kinematic state of the corresponding particle. In order to avoid time synchronization issues among the N bodies and to simplify the parallelization process, the time step is forced to be a power of two. Thus, particles are sub-divided in several groups (blocks) that share the same time step [4]. In this way we need to update positions and velocities only of $m \leq N$ particles per time step; in particular, bodies with smaller time steps will be updated more often than particles with bigger time steps for which the kinematic state will be estimated using Taylor expansions only. This implies that the computational complexity per time step is reduced from $O\left(N^2\right)$ to $O\left(mN\right)$.

3. THE `Bfactor` VARIABLE: another important aspect concerns the GPU load. For example, a GeForce GTX TITAN Black GPU can run a maximum of 30,720 threads in parallel, therefore we need to fittingly distribute the work load to fully exploit this kind of GPU. When using the
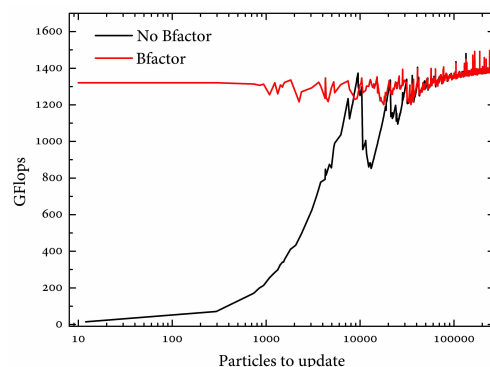

Figure 1: The effects of the `Bfactor` optimization on the performance of a GeForce GTX TITAN Black (measured in billions of floating point operations per second, GFlops) as a function of the number of particles that need to be updated (according to the block time steps distribution). The results were obtained using the N-body code `HiGPUs` with $N \simeq 260,000$.

block time steps strategy it is quite common to have $m < 30,720$ therefore we introduce the variable `Bfactor` that can increase the number of running threads and further split and distribute the work load among the GPU cores. For instance, if we have $m < 30,720$ and `Bfactor`$= B$ we run mB threads and each thread calculates the accelerations due to N/B bodies. The optimal `Bfactor` value must be determined step by step. We show in Fig. 1 the differences in terms of GPU performance running a typical N-body simulation with and without the `Bfactor` variable. It is evident that, for a GeForce GTX TITAN Black, when the particles that must be updated are $m \lesssim 30,720$ we obtain significantly higher performance when the `Bfactor` optimization is turned on. A similar optimization strategy can be found in [5].
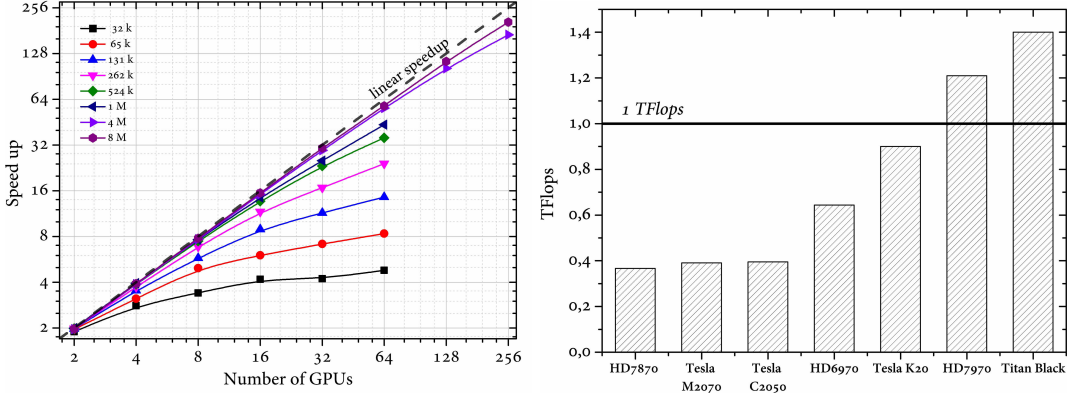
Figure 2: Left panel: speed up of the code `HiGPUs` as a function of the number of used GPUs for different N-body systems with $3.2 \times 10^4 \lesssim N \lesssim 8 \times 10^6$. The dashed line corresponds to the maximum computing efficiency (linear speedup). Right panel: performance (in TFlops) of `HiGPUs` running on single, different GPUs. For the scalability tests we used the IBM Data-Plex DX360M3 Linux Infiniband Cluster provided by the Italian supercomputing consortium CINECA.

4. PRECISION: it is well known that the maximum theoretical performance of all the GPUs in double precision (DP) is lower than their capability to execute single precision (SP) operations. For N-body problems it is important to use DP to calculate reciprocal distances and to cumulate accelerations in order to reduce round-off errors as much as possible. All the other instructions (square roots included) can be executed in SP to speed up the integration. Some authors use an alternative approach based on an emulated double precision arithmetic (also known as double-single precision or simply DS). In this strategy a DP variable is replaced with two, properly handled, SP values; in this way only SP quantities are used against a slightly larger number of operations that must be executed [6].

5. SHARED MEMORY: the GPU shared memory (SM) is a limited amount of memory (in general $\lesssim 65$kB) and it is shared between all the GPU threads in the same block and can be used for fast data transactions (on average, SM is about 10 times faster than "standard" memory). During the evaluation of the N-body accelerations, the best strategy is to cyclically load SM until all the pair-wise forces are computed.

2.1 Performance results

Fig. 2 shows the scalability of the code `HiGPUs` on a GPU cluster (left panel) and its performance using single, different GPUs (right panel). Form Fig. 2, it is apparent that GPUs are extremely well suited to solve the N-body problem: we reach a computing efficiency of $\sim 92\%$ using 256 GPUs and ~ 8 million bodies and a sustained performance of ~ 1.4 TFlops on just one GPU (GeForce GTX TITAN Black). More details can be found in [2] and [7].
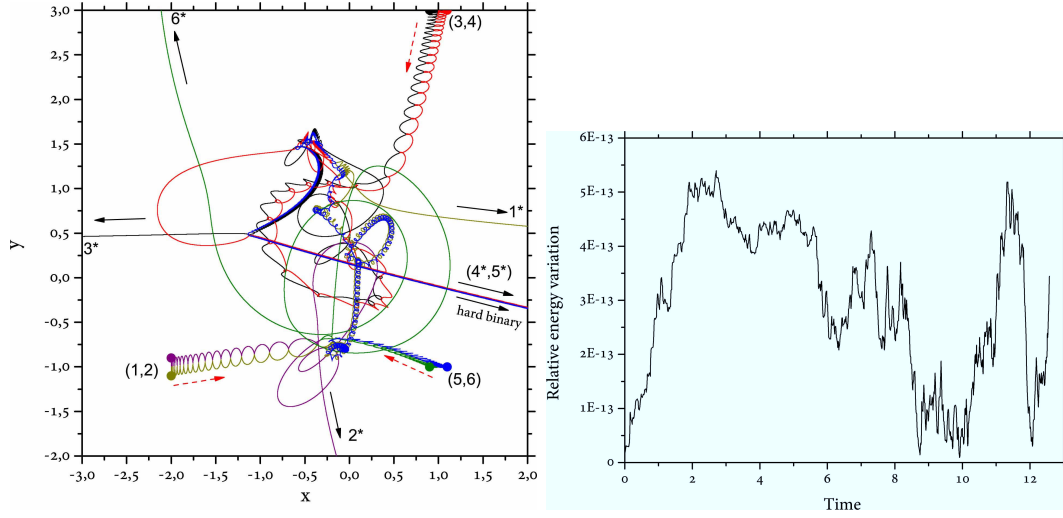
Figure 3: Left panel: trajectories obtained for the modified version of the Pythagorean 3-body problem, where the three particles are replaced with three binaries, using a regularized algorithm. Three binary systems, indicated with (1,2), (3,4) and (5,6), are initially placed at the vertices of a right triangle with null velocities. During the dynamical evolution, the particles 1, 2, 3 and 6 are ejected form the system with high velocities (outside directions are indicated with arrows) while particles 4 and 5 form a very hard binary. Right panel: relative energy variation during the integration of the Pythagorean problem.

3 Regularization

Close encounters between (two or more) particles are critical in N-body simulations because of the UVd of the gravitational force. An attempt to remove the small-scale singularity of the interaction potential is referred as an attempt of *regularization*. The Burdet-Heggie method ([8], [9]), the Kustaanheimo-Stiefel algorithm [10] or the Mikkola's algorithmic formulation (MAR, [11]) are some of the most famous examples of regularization. In general, all these methods are quite expensive in terms of implementation effort and computing time but, if we use them to integrate few bodies only, they become both faster and much more accurate than standard N-body integrators. It is not convenient to implement regularization methods on a GPU because of their mathematical construction and because they can integrate, in general, a maximum of few tens of bodies. Nevertheless, during the dynamical evolution of an N-body system, we can identify the groups of particles that are in tight systems or that are experimenting a close encounter, and regularize them. This process can be done in parallel by the CPU, by means of OpenMP, establishing a 1 to 1 correspondence between groups that must be regularized and CPU threads. At the same time, given that the GPU kernels are asynchronous, the regularization process can be performed while the GPU works in background. This describes the parallel scheme adopted to implement the MAR in the code `HiGPUs-R` which is still under development. A test application to demonstrate the advantages of regularization is shown in Fig. 3. It represents a modified version of the so called Pythagorean 3-body problem

(e.g. [12]) integrated with `HiGPUs-R`. Standard N-body integrators, such as the Hermite 4th order scheme, cannot evolve this system, because either the time step becomes prohibitively small throughout the dynamical evolution, or, if we fix a minimum time step, the solution is completely inaccurate. The only chance is to use a regularized code which is very fast (~ 10 seconds of simulations to obtain the trajectories in the left panel of Fig. 3) and maintains a very good total energy conservation (see the right panel of Fig. 3).

4   Conclusions

In this work I have presented and discussed the main strategies adopted to speed up the numerical solution of the N-body problem using GPUs. I have also shown the main advantages in using regularization methods and described a new parallel scheme to implement the Mikkola's algorithmic regularization in the context of a GPU N-body code. I have used the direct N-Body code `HiGPUs` as reference and I have given an overview of the code `HiGPUs-R` that is a new regularized version of `HiGPUs`. The development of fast and regularized N-body codes such as `HiGPUs-R` is of fundamental importance to investigate a large number of astrophysical problems (ranging from the dynamical evolution of star clusters to the formation of double black hole binaries).

5   Acknowledgments

References

[1] J. Barnes and P. Hut, Nature **324** 446 (1986)

[2] R. Capuzzo-Dolcetta, M. Spera and D. Punzo, JCP **236** 580 (2013)

[3] Nitadori, K., & Makino, J., New Astronomy **13** 498 (2008)

[4] S.J. Aarseth, *Gravitational N-body simulations: tools and algorithms*, Cambridge Univ. Press, UK (2003)

[5] L. Nyland, M. Harris and J. Prins, GPU gems 3 **31** 677 (2007)

[6] E. Gaburov, S. Harfst and S. Portegies Zwart, New. Ast. **14** 630 (2009)

[7] R. Capuzzo-Dolcetta, M. Spera, Comp. Phys. Comm. **184** 2528 (2013)

[8] C.A. Burdet, Z. Angew. Math. Phys. **18** 434 (1967)

[9] D.C. Heggie, Astr. and Space Science Lib. **39** 34 (1973)

[10] P. Kustaanheimo and E. Stiefel, Jour. für Reine und Angew. Math. **218** 204 (1965)

[11] S. Mikkola and K. Tanikawa, Celest. Mech. Dyn. Astr. **74** 287 (1999)

[12] S. Mikkola and S.J. Aarseth, Celest. Mech. Dyn. Astr. **57** 439 (1993)

Fast Cone-beam CT reconstruction using GPU

Giovanni Di Domenico[1]

[1]Dipartimento di Fisica e Scienza della Terra, via Saragat 1, 44122 Ferrara, Italy
[2]INFN - Sezione di Ferrara, via Saragat 1, 44122 Ferrara, Italy

Fast 3D cone-beam CT reconstruction is required by many application fields like medical CT imaging or industrial non-destructive testing. We have used GPU hardware and CUDA platform to speed-up the Feldkamp Davis Kress (FDK) algorithm, which permits the reconstruction of cone-beam CT data. In this work, we present our implementation of the most time-consuming step of FDK algorithm: back-projection. We also explain the required transformations to parallelize the algorithm for the CUDA architecture. Our FDK algorithm implementation in addition allows to do a rapid reconstruction, which means that the reconstruction is completed just after the end of data acquisition.

1 Introduction

There are a lot of application areas that benefit of the GPU computing, among them we find the image reconstruction in medical physics applications [1]. In the last years, the computational complexity of image reconstruction has increased with the progress in imaging systems and reconstruction algorithms. Moreover, fast image reconstruction is required in an imaging diagnostic department to allow the technologist to review the images while the patient is waiting or to obtain the output in real-time imaging applications. Many researchers have worked to accelerate Cone-Beam CT (CBCT) reconstruction using various accelerators, such as GPU [3, 4, 5, 6], Cell Broadband Engine (CBE) [7], and field programmable gate array (FPGA) [8]. The Common Unified Device Architecture (CUDA) [2] is a software development platform, invented by nVIDIA, that allows us to write and execute general-purpose application on graphics processing unit (GPU). Scherl et al. [4] have developed one of the first GPU-accelereted CT reconstruction using CUDA and they have compared the performance results of CUDA implementation with the CBE ones concluding that the CUDA-enable GPU is well suited for high-speed CBCT recontruction. In this paper, we propose a CUDA based method capable of accelerating CBCT reconstruction based on FDK algorithm, showing some optimization techniques for the backprojection step.

2 Theory

The ideal geometry for cone-beam CT, see Fig.1, is made by an x-ray source that moves on a circle through the x-y plane around z, the axis of rotation. The (u,v,w) denotes the rotated reference frame, where β is the angle of rotation, R_s is the radius of the source circle, while R_d is the source to detector distance. The line from the x-ray source through the origin O hits the

detector at the origin of its 2D-coordinate system (u,v) and is orthogonal to the detector plane. The CT measurement can be converted to a form that is closely modeled by a line integral
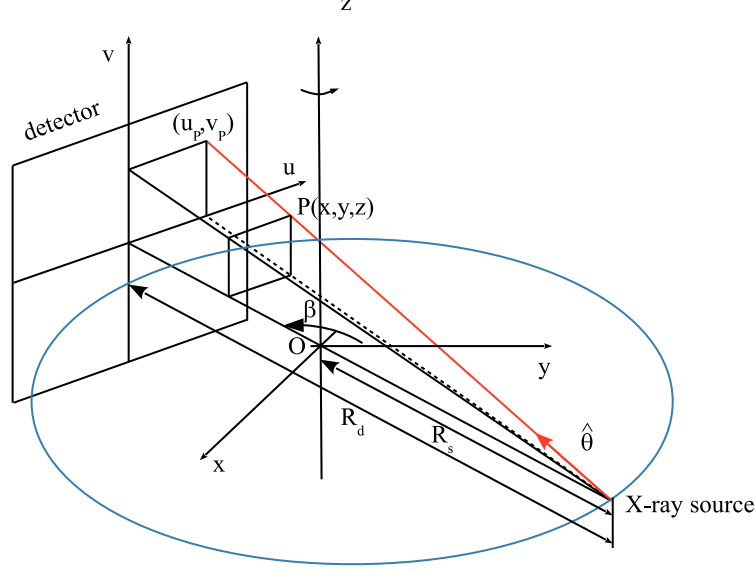

Figure 1: Cone-beam CT geometry.

through the continuous object function

$$g(u, v, \beta) = \int_0^\infty f(\vec{r}_o(\beta) + \alpha\hat{\theta})d\alpha \tag{1}$$

where $f(\vec{r})$ represents the object function, the x-ray attenuation coefficient, and the data function $g(u, v, \beta)$ is the line integral through the object in the direction of unit vector $\hat{\theta}$ from the source location $\vec{r}_o(\beta)$. For the model considered here, the detector is taken to be a flat panel array with bin location (u_p, v_p)

$$\vec{d}_o(u_p, v_p, \beta) = (R_d - R_s)(-sin\beta, cos\beta, 0) + u_p(cos\beta, sin\beta, 0) + v_p(0, 0, 1) \tag{2}$$

The goal of image reconstruction is to find $f(\vec{r})$ from the knowledge of $g(u, v, \beta)$.

2.1 FDK algorithm

The FDK algorithm [9] reconstructs the function $f(\vec{r})$ accurately only in the plane of trajectory. Outside of this plane, the algorithm approximate $f(\vec{r})$. The FDK algorithm applies a filtered backprojection technique to solve the reconstruction task in a computationally efficient way. Due to its efficiency it has been implemented successfully on almost ever commercially available medical CT image system and still maintain its state of the art status in modern computed tomography. The FDK algorithm is organized in three steps:

1. cosine weighting:

$$g_1(u,v,\beta) = g(u,v,\beta)\frac{R_d}{\sqrt{R_d^2 + u^2 + v^2}} \tag{3}$$

2. ramp filtering:

$$g_2(u,v,\beta) = g_1(u,v,\beta) \otimes h_{ramp}(u) \tag{4}$$

3. backprojection:

$$\hat{f}(x,y,z) = \frac{1}{2}\int_0^{2\pi} (\frac{R_s}{R_s - xsin\beta + ycos\beta})^2 g_2(u,v,\beta)d\beta \tag{5}$$

Typically, the algorithm input data consist of K-projections, the size of each projection is $N_u \times N_v$, the output data is N^3-voxel volume V.

3 Materials and Methods

The CUDA programming model assumes that both the host (CPU) and the device (GPU) maintain their own separate memory spaces, referred to as host memory and device memory, respectively. Since a CUDA kernel works only on data present in device memory and due to the limited GPU memory capacity, it is not easy to store the entire volume and the projections in device memory. For example: a 512^3 voxel volume requires al least 512 MB of memory space, so we have decided to store the entire volume (or a 512^3 portion if the volume size is greater than 512^3) in device memory and to load the projections into device memory when needed and remove it after its backprojection. Our first implementation of FDK algorithm
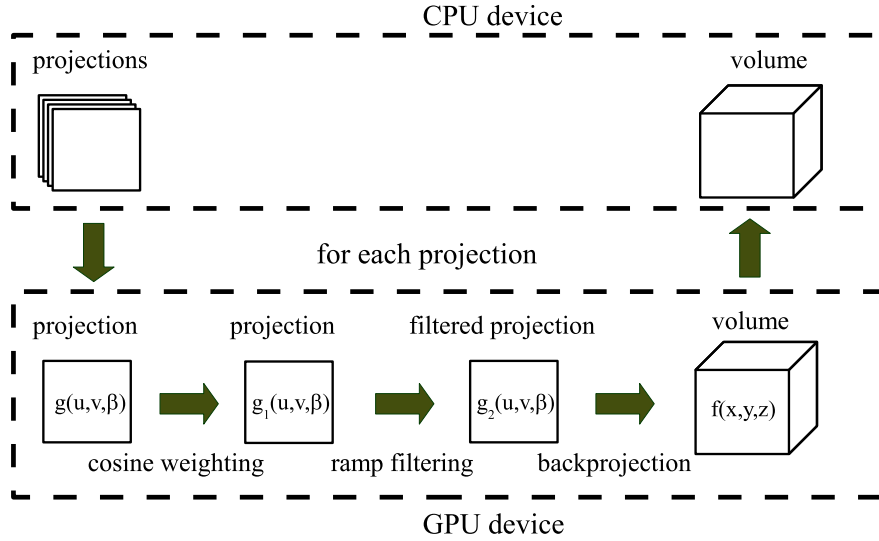

Figure 2: Overview of implemented naive method.

on CUDA-enabled GPU is a naive method, as shown in fig. 2, we transfer the first projection

P_1 to the device global memory and perform the cosine weighting, the ramp filtering and the backprojection steps to the volume V in the device global memory. This operation is repeated on the remaining projections to obtain the final accumulated volume. The naive code assigns in the backprojection step every voxel to a different CUDA threads. Two data sets have been chosen for testing the naive implementation in CUDA:

- **dataset 1** consists of 200 projections acquired on 360° circular scan trajectory. The size of each projection is 1024x512.

- **dataset 2** consists of 642 projections acquired on 360° circular scan trajectory. The size of each projection is 256x192.

Reconstruction tests have been performed on a desktop PC: it is equipped with a Intel Core i7-3770 CPU, 16 GB main memory and a nVIDIA GeForce GTX-Titan with 4GB device memory. Our implementation runs on Linux with CUDA v5.5. The filtering step has been implemented by using the CUFFT library of CUDA package. Table 1 shows the execution time needed for the CUDA implementation of FDK algorithm compared with the OpenMP [10] implementation.

	step1 [s]	step2 [s]	step3 [s]	Total [s]
CUDA				
dataset1	0.57	5.69	8.05	14.31
dataset2	0.45	1.81	3.1	5.36
OpenMP				
dataset1	0.67	5.09	206.95	212.71
dataset2	0.21	1.26	80.96	82.43

Table 1: Execution time of CUDA implementation compared to OpenMP. step 1: cosine weighting, step 2: ramp filtering, step 3: backprojection.

The results obtained show a remarkable time speed-up in the backprojection step implemented in CUDA by factor ≈ 25 respect to the OpenMP implementation, while the other ones are comparable. Starting from these results we have worked on the optimization of the backprojection step, a memory intensive operation, by using the techniques suggested in the CUDA programming guide [2]. The two techniques used are:

1. **Memory coalescing**: we organize the threads inside a block to access contiguous memory location and each thread computes the backprojection of a given number of voxels along z. This approach allows to save the number of add-multiply operations by incrementing the (u,v) coordinates.

2. **Global memory access reduction**: we use kinds of memory that have a cache mechanism (texture and constant memory). Moreover, texture memory has a GPU's hardware bilinear interpolation mechanism that speeds up the calculation of update voxel value. Also, we increase the number of projections processed by a single thread to reduce the number of access to volume global memory.

We have accomplished the task of backprojection optimization on CUDA-enabled GPU by using RabbitCT [11] an open platform for worldwide comparison in backprojection performance. We have implemented two new versions for the backprojection kernel, the version v1 where we

introduce the use of constant and texture memory on GPU, and the version v2 where in addition we load more projections in texture memory before the backprojection step. A processed dataset of a rabbit, suitable for cone beam 3D reconstruction, is available for testing the own kernel implementation. It consist of N=496 projection images I_n, the size of a projection image is 1248x960 pixels. The performance results of the new kernels respect to the naive kernel, named v0, are shown in table 2. Both the new versions of backprojection kernel show an improvement

Version	Size	Time [s]	Occupancy	Error [HU]	GUPs
v0	512	10.32	84	0.03	6.01
v1	512	4.77	89	0.16	12.90
v2	512	3.0	95	0.16	20.67

Table 2: Comparison of execution time, percentage of GPU occupancy , reconstruction error in Hounsfield Unit (HU), billion of voxel updates per second (GUPs), for the various version of backprojection kernel on the RabbiCT dataset.

in the execution time, the speed-up of version v1 is of a factor 2 and the speed-up of version v2 is of a factor 3 with a moderate increase of reconstruction error. At the end of optimization step the new backprojection kernels have been integrated in CBCT reconstruction code.

4   Results and Discussion

The reconstructed images of dataset 1 obtained by using CUDA and OpenMP implementation of the FDK method are shown in figure 3. The maximum relative difference between the
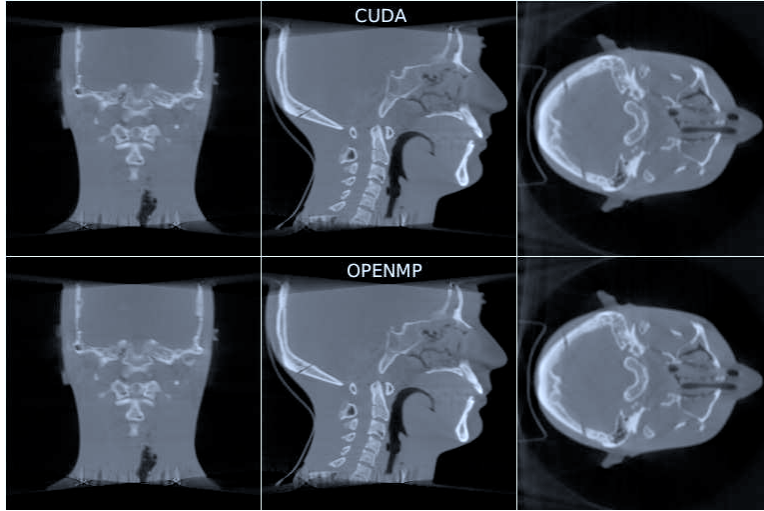

Figure 3: Comparison between GPU and CPU reconstructed CBCT images of dataset 1.

corresponding pixels in the reconstructed volumes, defined in eq. 6, is less than 9.0×10^{-3}.

$$\max_{(x,y,z) \in V} \frac{|I_{CUDA}(x,y,z) - I_{OMP}(x,y,z)|}{I_{OMP}(x,y,z} \tag{6}$$

The total execution times of various CUDA implementation of FDK reconstruction software are in table 3 compared to the OpenMP implementation.

	v0 [s]	v1 [s]	v2 [s]	OpenMP [s]
dataset1	14.31	8.49	7.68	212.7

Table 3: Total execution time of various versions of CUDA implementation of the FDK reconstruction algorithm compared with the OpenMP implementation.

Using GPU + CUDA in CBCT reconstruction we are able to accelerate the reconstruction process of a factor greater than 25. On dataset 1 the reconstruction time is less than 8 s for a volume of 512^3, and this result suggests CUDA implementation permits a real time reconstruction of CBCT data.

Acknowledgment

References

[1] G. Pratx and L. Xing, Med. Phys. **38** 2685 (2011).

[2] nVIDIA Corporation: CUDA C Programming guide Version v5.5 (July 2013).

[3] F. Xu, K. Mueller, Phys. Med. Biol. **52** 3405 (2007).

[4] H. Scherl, B. Keck, M. Kowarschik, J. Hornegger, IEEE NSS Conf. Proc. **6** 4464 (2007).

[5] G. Yan, J. Tian, S. Zhu, Y. Dai, C. Qin, J. X-ray Sci. **16** 225 (2008).

[6] Y. Okitsu, F. Ino, K. Hagihara, Parallel Comput. **36** 129 (2010).

[7] M. Kachelrieβ, M. Knaup, O. Bockenbach, Med. Phys. **34** 1474 (2007).

[8] N. Gac, S. Mancini, M. Desvignes, Proc. 21st ACM Symp. Applied Computing 222 (2006).

[9] L.A. Feldkamp, L.C. Davis, J.W. Kress, J. Opt. Soc. Am. **A1** 612 (1984).

[10] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP*. The MIT Press (2008).

[11] C. Rohkohl, B Keck, H.G. Hofmann,J. Hornegger, Med. Phys. **36** 3940 (2009).

GPU-parallelized model fitting for realtime non-Gaussian diffusion NMR parametric imaging

Marco Palombo[1,2,3]*, Dianwen Zhang*[4]*, Xiang Zhu*[5]*, Julien Valette*[2,3]*, Alessandro Gozzi*[6]*, Angelo Bifone*[6]*, Andrea Messina*[7]*, Gianluca Lamanna*[8,9]*, Silvia Capuani*[1,7]

[1]IPCF-UOS Roma, Physics Department, "Sapienza" University of Rome, Rome, Italy
[2]CEA/DSV/I2BM/MIRCen, Fontenay-aux-Roses, France
[3]CEA-CNRS URA 2210, Fontenay-aux-Roses, France, France
[4]ITG, Beckman Institute, UIUC, Urbana, Illinois, United States
[5]College of Economics and Management, CAU, Beijing, China
[6]IIT, Center for Neuroscience and Cognitive Systems @ UniTn, Rovereto, Italy
[7]Physics Department, "Sapienza" University of Rome, Rome, Italy
[8]INFN, Pisa Section, Pisa, Italy,
[9]INFN, Frascati Section, Frascati (Rome), Italy

The application of graphics processing units (GPUs) for diffusion-weighted Nuclear-Magnetic-Resonance (DW-NMR) images reconstruction by using non-Gaussian diffusion models is presented. The image processing based on non-Gaussian models (Kurtosis and Stretched Exponential) currently are time consuming for any application in realtime diagnostics. Non-Gaussian diffusion imaging processing was implemented on the massively parallel architecture of GPUs, by employing a scalable parallel Levenberg-Marquardt algorithm (*GPU-LMFit*) optimized for the Nvidia CUDA platform. Our results demonstrate that it is possible to reduce the time for massive image processing from some hours to some seconds, finally enabling automated parametric non-Gaussian DW-NMR analyses in realtime.

1 Introduction

In this contribution we focused on the application of graphics processing units (GPUs) accelerated computing in reconstruction of diffusion weighted Nuclear-Magnetic-Resonance (DW-NMR) images by using non-Gaussian diffusion models, such as the diffusional kurtosis imaging (DKI) [1] and the stretched exponential model imaging (STREMI) [2], which allow to increase the sensitivity and specificity of the DW-NMR maps in healthy [3, 4, 5] and pathological subjects [6, 7]. However, the post-processing of DW-NMR images based on these models currently requires too long times for any realtime diagnostics. Typically, for the elaboration of diffusion maps, 10^6-10^7 voxels have to be managed. For each voxel a typical algorithm used to obtain non-Gaussian maps calculates at least three parameters by non-linear functions optimization. This is computationally demanding and takes some hours on recent multi-core processors(i.e. CPU Intel Xeon E5 and E7) to obtain a whole brain map. The aim of this work is to implement non-Gaussian diffusion imaging processing on the massively parallel architecture of GPUs and

optimize different aspects to enable on-line imaging.

To achieve this goal, diffusion images were acquired in a fixed mouse brain and two different algorithm to reconstruct DKI and STREMI maps were implemented on GPU. Successively, diffusion images were processed and non-Gaussian diffusion maps were obtained by using both the conventional currently used algorithm working on CPU and the new algorithms, based on a highly parallelized Levenberg-Marquardt (LM) method on GPU.

More specifically, we are concerned with a model-based approach for extracting tissue structural information from DW-NMR imaging data. Non-linear relations describing the DW-NMR signal attenuation have to be fitted to experimental dataset voxel-by-voxel by using the LM algorithm.

Certain features of the proposed implementation make it a good candidate for a GPU-based design: a) Independence between voxels across the three-dimensional brain volume allows voxel-based parallelization, b) Within each voxel, certain computation steps of data analysis are intrinsically iterative and independent, allowing further parallelization (i.e. fitting parameters estimation), c) Relatively simple mathematical operations are needed and these can be handled effectively by the GPU instruction set and d) Memory requirements are moderate during each step of the algorithm.

2   Theory

Non-Gaussian DW-NMR models.   Describing water molecules displacement with a Gaussian Motion Propagator, NMR signal decay recorded using a diffusion-sensitized sequence may be expressed as a b-value function according to the following equation [8]:

$$S(b) = S(0)\exp(-bD) \tag{1}$$

where D is the apparent diffusion coefficient and $b = (\Gamma\delta g)^2\Delta_{eff}$ with Γ the nuclear spin gyromagnetic ratio, g the diffusion sensitizing gradient strength, δ the diffusion sensitizing gradient duration and Δ_{eff} the effective diffusion time, depending on the particular diffusion sensitized sequence used. In 3D space, observing diffusion displacement in a generic direction and working in an anisotropic environment, D is no more a scalar, but a tensor (DT) and the coupling of non diagonal terms in DT has to be taken into account. Nevertheless, due to proprieties of Fourier Transform and Gaussian Propagator, the mono-exponential form of the signal decay can be conserved by introducing the so called b-factor [9]:

$$\ln\left(\frac{S(b)}{S(0)}\right) = -\sum_{i,j=1}^{3} b_{i,j}D_{i,j} \tag{2}$$

where $b_{i,j} = (\Gamma\delta)^2\Delta_{eff}\int_0^{t_{seq}} dt(\int_0^t dt' g(t')g(t')^T)_{i,j}$ is the correspondent term of b-factor matrix to the relative term of DT. Due to its propriety, DT is diagonalizable to obtain scalar invariant quantities as MD and FA and the reference frame in which DT is diagonal.

The formalism exposed is based on the assumption that molecular diffusion occurs in a homogeneous environment, implying a linear relationship between mean square displacement and diffusion time. However, in a complex system with pores and traps on many length scales, this simple relation is lost and Eqs.1 and 2 are no longer valid [10, 11]. We refer to these cases as "non-Gaussian" diffusion process. In the last decade several different approaches have been

introduced in NMR field to describe DW-NMR signal decay in case of non-Gaussian diffusion. Here we treat two of the most employed ones: (i) the DKI [1] and (ii) the STREMI [2, 10].

(i) DKI is based on the assumption that DW-NMR signal can be described by the following relation [1]:

$$\frac{S(b)}{S(0)} = \left\{ \left[\exp\left(-bD_{app} + b^2 A_{app}\right) \right]^2 + \eta^2 \right\}^{1/2} \tag{3}$$

where η is the background noise, and $A_{app} = \frac{1}{6} D_{app}^2 K_{app}$, with D_{app} and K_{app} the apparent diffusion coefficient and the apparent diffusional kurtosis, estimated in the direction parallel to the orientation of diffusion sensitizing gradients, respectively.

(ii) The STREMI assumes the following relation to describe the signal decay [2, 10]:

$$\ln(\frac{S(b)}{S(0)}) = -\sum_{i=1}^{3} (b_i^*)^{\gamma_i} A_i \tag{4}$$

where A_i is the generalized diffusion coefficient estimated along the direction identified by the ith eigenvector of DT, named $\vec{\epsilon}_i$; γ_i is the stretching exponent estimated along the ith direction of DT reference frame (being between 0 and 1); and b_i^* is the projection of bvalue along $\vec{\epsilon}_i$ (with components ϵ_{1i}, ϵ_{2i}, ϵ_{3i}, in the laboratory reference frame), i.e. $b_i^* = \vec{b} \cdot \vec{\epsilon}_i$.

3 Materials and Methods

DW-NMR data acquisition. An *ex vivo* healthy mouse brain, fixed in paraformaldehyde and stored in PBS [12], was scanned at 7.0T (BRUKER Biospec). An imaging version of PGSTE sequence was performed with $TE/TR = 25.77/4000ms$, $\Delta/\delta = 40/2ms$, number of averages $NA = 14$; 16 axial slices with thickness $STH = 0.75mm$, field of view $FOV = 6cm$, matrix 128x128 with in plane resolution of $470\mu m^2$ were acquired with 10 b-values ranging from 100 to $8000s/mm^2$ along 30 no-coplanar directions plus 5 $b = 0s/mm^2$

DW-NMR data analysis. Both DKI parametric maps (K_{app}-maps) and STREMI parametric maps (γ-maps) were estimated in each direction parallel to the orientation of diffusion sensitizing gradient, but K_{app}-maps were obtained by fitting on a voxel-by-voxel basis Eq.(3) to the DW image signal intensities (for $b \leq 3000s/mm^2$), while γ-maps, were obtained by similar procedure, using Eq.(4).

Finally, the non-Gaussian diffusion parametric maps MK and $M\gamma$ were computed by averaging across the 30 directions in the corresponding K_{app}- and γ-maps, respectively.

GPU implementation. A modern GPU device can have a number of "multiprocessors" (MP), each of which executes in parallel with the others. Using the Nvidia compute unified device architecture (CUDA), multiple thread blocks (and thus multiple fittings) can execute concurrently with many parallel threads on one multi-processor. Here we implemented an efficient and robust fitting algorithm, based on a highly parallelized LM method on GPU. The LM algorithm is based on an iterative numerical optimization procedure that minimizes the sum of squared model residuals. The function we used to perform LM fittings on GPU device is the single precision *GPU-LMFit* function, introduced and described in details elsewhere [13]. *GPU-LMFit* uses a scalable parallel LM algorithm optimized for using the Nvidia CUDA platform.
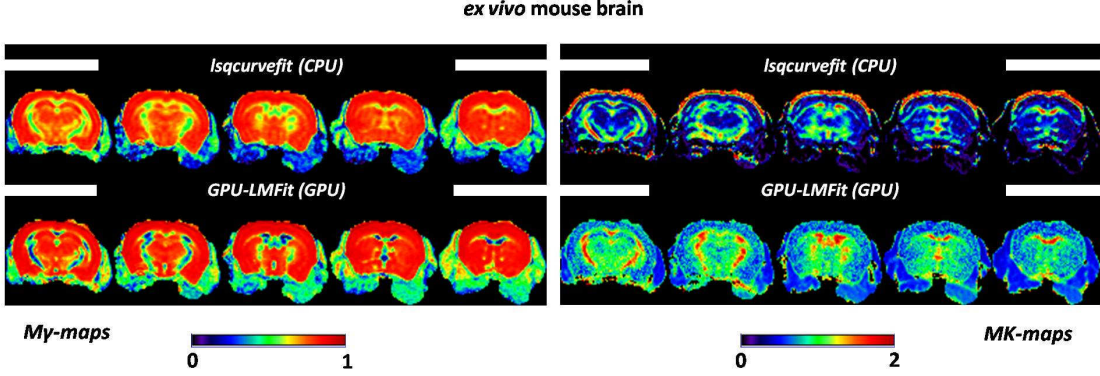
ex vivo mouse brain

Figure 1: Comparison between CPU and GPU non-Gaussian diffusion maps reconstruction. Reported maps were obtained by using the *lsqcurvefit* on Intel Xeon E5- E5430 CPU and *GPU-LMFit* on Nvidia Quadro K2000 GPU (see Table 1 for details).

The code kernel calls *GPU-LMFit* to perform the LM algorithm on each CUDA block, which is mapped to a single voxel. Because the processing of different voxels is totally independent, the CUDA blocks do not need to synchronize, and the kernel launches as many blocks as voxels contained in a particular slice to speed up performance.

The code was optimized to be fully integrated within Matlab (The Mathworks, Natick, MA, USA) scripts.

A multi-core central processing unit (CPU) Intel Xeon E5430 processor at 2.66GHz with 8 thread, and a Nvidia GPU Quadro K2000, with 2Gb of dedicated memory, supporting 1024 threads per block and a maximum number of 64 registers per thread, were used for the analysis and the cross-comparison of CPU and GPU performance. In particular, *lsqcurvefit* function with Parallel Computing Toolbox was used to test multi-core CPU performance.

Each analyzed DW-NMR image dataset is of $\sim$ 150 Mb, and requires that the total number of fittings to be performed to create a single image of the non-Gaussian parametric map is in the range of $(0.5 - 5)\text{x}10^6$. In the kernel function, we choose to distribute the computation in 4096 CUDA blocks, each of which has 8 threads concurrently executing to compute a fitting. Therefore, the kernel function was called multiple times in the program to complete all fittings for an image, and each call to this function requires 7-15 Mb global memory on GPU.

4 Results and Discussion

Non-Gaussian diffusion parametric maps $M\gamma$ and MK, obtained by using *lsqcurvefit* on multiple CPU threads and *GPU-LMFit on GPU*, are displayed in Figure 1. The specific performances of the CPU and GPU employed are reported in Table 1, while the cross-comparison between *lsqcurvefit* and *GPU-LMFit* results is reported in Figure 2.

The Figures 1 and 2 show that the GPU approach for STREMI is in agreement with conventional CPU one. Conversely, for DKI, *GPU-LMFit* slightly overestimates MK values with respect to *lsqcurvefit*. However, it is important to note that MK-maps obtained by *GPU-LMFit*
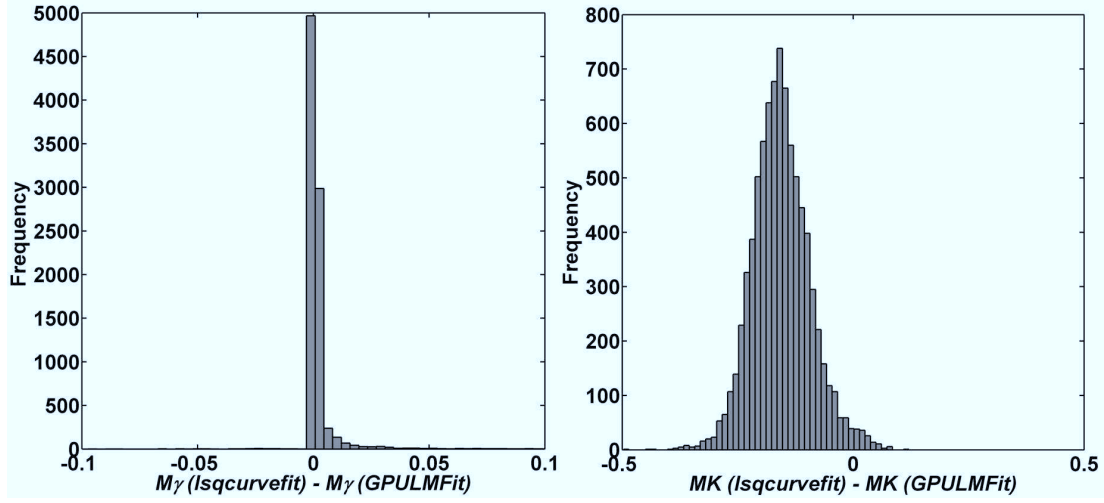
Figure 2: Cross-comparison between *lsqcurvefit* and *GPU-LMFit* results. The frequency histograms of the difference between $M\gamma$ (left) and MK (right) values derived with *GPU-LMFit* and *lsqcurvefit* are computed from all the voxels of the parametric maps comprised within the *ex vivo* mouse brain investigated.

show a contrast-to-noise ratio different from the *lsqcurvefit* ones (see Figure 1): they better discriminate between white and gray matter regions within mouse brain, even if they are more grainy and noisy than the *lsqcurvefit* ones. The grainy and noisy characteristic of these maps can be due to the single precision implementation of the *GPU-LMFit* function. Indeed, each K_{app}-map is obtained by fitting, voxel-by-voxel, Eq.3 to get the two constants D_{app} and A_{app}. Then, K_{app} is estimated by keeping the ratio: $K_{app} = \frac{6A_{app}}{D_{app}^2}$. It is therefore clear that using single or double precision operators in K_{app} estimation can make the difference. In particular, the *lsqcurvefit* is a double precision Matlab function, while the *GPU-LMFit* version used here is single precision. Therefore, further developments are actually in progress to release a double precision version of *GPU-LMFit*, in order to increase the quality of non-Gaussian diffusion maps obtained on GPU.

Finally, from Table 1 it is possible to appreciate the relative speed-up obtainable by using *GPU-LMFit*, which for a medium level GPU like the Quadro K2000 is ~ 240x. In terms of computational time, this speed-up factor means that the GPU implementation reported here allows to reduce the time for massive image processing from some hours to some seconds (see

Performance Parameters	CPU	GPU
Average Speed (fit/sec.)	50	12000
Computational Time (sec.)	7200	30
Speed-up factor	1	240

Table 1: Comparison of CPU and GPU performance in non-Gaussian diffusion maps reconstruction.

MARCO PALOMBO, DIANWEN ZHANG, XIANG ZHU, JULIEN VALETTE, . . .

Table 1). Moreover, numerical simulations were performed on high level Nvidia GPU, the Nvidia Titan, to test the additional speed-up factor obtainable by employing one of the most powerful GPU now available (results not reported here). Simulation results suggest that an additional speed-up factor of 6.6x with respect to the Nvidia Quadro K2000 GPU is achievable by using the Nvidia Titan GPU. This demonstrates that automated parametric non-Gaussian DW-NMR analysis in realtime is now really possible by using the GPU approach proposed in this work.

5 Conclusion

In this contribution we focused on the application of GPUs in the reconstruction of DW-NMR images based on non-Gaussian diffusion models. This application can benefit from the implementation on the massively parallel architecture of GPUs, optimizing different aspects and enabling online imaging. A pixel-wise approach by using a fast, accurate and robust parallel LM minimization optimizer, called *GPU-LMFit*, was implemented in CUDA and fully integrated in Matlab. Our results show that the GPU application proposed here can further improve the efficiency of the conventional LM model fittings, reducing the time for DW-NMR image-processing from hours to seconds, finally enabling automated parametric non-Gaussian DW-NMR analysis in realtime. Moreover, another important feature of the architecture of the proposed approach is that it can allow multiple GPUs applications [13], where the measured experimental data in the host computer memory is separately passed to the global memories of multiple GPUs, and then the host program launches the kernel functions on each GPU device. Therefore, another natural development of this work, behind the upgrade to the double precision version, is the multiple GPUs application in order to further improve the efficiency of the LM model fittings with *GPU-LMFit*.

Acknowledgements

References

[1] J.H. Jensen *et al.*, Magn. Reson. Med. **53 (6)** 1432 (2005).
[2] S. De Santis, *et al.*, Magn. Reson. Med. **65 (4)** 1043 (2010).
[3] M. Palombo, *et al.*, J. Magn. Reson. **216** 28 (2012).
[4] J. GadElkarim, *et al.*, IEEE J. Emerg. Sel. Top. Circ. Syst. **3** 432 (2013).
[5] M. Palombo, *et al.*, Magn. Reson. Med.(2014), DOI: 10.1002/mrm.25308.
[6] S. Capuani, *et al.*, Magn. Reson. Imag. **31** 359 (2013).
[7] F. Grinberg, *et al.*, PloS one **9(2)** e89225 (2014).
[8] E.O. Stejskal, J.E. Tanner, J. Chem. Phys. **42(1)** 288 (1965).
[9] P.J. Basser, *et al.*, Biophys. J. **66(1)** 259 (1994).
[10] M. Palombo, *et al.*, J. Chem. Phys. **135 (3)** 034504 (2011).
[11] M. Palombo, *et al.*, Nature Sci. Rep. **3** 2631 (2013).
[12] L. Dodero , *et al.*, PloS one **8(10)** e76655 (2013).
[13] X. Zhu , D. Zhang, PloS one **8(10)** e76665 (2013).

Estimation of GPU acceleration in NMR medical imaging reconstruction for realtime applications

Marco Palombo[1,2,3], Matteo Bauce[4], Andrea Messina[4], Gianluca Lamanna[5,6], Silvia Capuani[1,4]

[1]IPCF-UOS Roma, Physics Department, "Sapienza" University of Rome, Rome, Italy
[2]CEA/DSV/I2BM/MIRCen, Fontenay-aux-Roses, France
[3]CEA-CNRS URA 2210, Fontenay-aux-Roses, France, France
[4]Physics Department, "Sapienza" University of Rome, Rome, Italy
[5]INFN, Pisa Section, Pisa, Italy,
[6]INFN, Frascati Section, Frascati (Rome), Italy

Introduction. Recently new Nuclear-Magnetic-Resonance (NMR) methods have been developed to measure physical parameters that are directly correlated with properties of biological tissues, such as the Diffusional Kurtosis Imaging (DKI) [1] and the Stretched Exponential Model Imaging (STREMI) [2]. Such methods are based on non-Gaussian diffusion measurements. These techniques are particularly interesting because they allow to increase the sensitivity and specificity of the NMR imaging for healthy and pathological cerebral conditions [3, 4, 5, 6].
The DKI is essentially based on the assumption that DW-NMR signal can be described by the following relation [1]:

$$\frac{S(b)}{S(0)} = \left\{ \left[\exp\left(-bD_{app} + b^2 A_{app}\right) \right]^2 + \eta^2 \right\}^{1/2} \tag{1}$$

where η is the background noise, $A_{app} = \frac{1}{6} D_{app}^2 K_{app}$, with D_{app} and K_{app} respectively the apparent diffusion coefficient and the apparent diffusional kurtosis, estimated in the direction parallel to the orientation of diffusion sensitizing gradients, and $b = (\Gamma \delta g)^2 \Delta_{eff}$ with Γ the nuclear spin gyromagnetic ratio, g the diffusion sensitizing gradient strength, δ the diffusion sensitizing gradient duration and Δ_{eff} the effective diffusion time, depending on the particular diffusion sensitized sequence used.
On the contrary, the STREMI assumes the following relation to describe the signal decay [2, 7]:

$$\frac{S(b)}{S(0)} = \exp\left[-(b)^{\gamma} A\right] \tag{2}$$

where A is the generalized diffusion coefficient and γ is the stretching exponent, being between 0 and 1.
Currently, the post-processing of NMR images based on these new techniques requires a too long time (2 hours on a multi-core Intel Xeon E5430 CPU) for any use in realtime diagnostics. In this contribution we focused on the application of graphics processing units (GPUs) accelerated computing to improve the speed in reconstruction of diffusion weighted nuclear magnetic

resonance (DW-NMR) images by using non-Gaussian diffusion models. The aim of this work is to use model-related numerical simulations of DW-NMR signal in realistic conditions, to estimate the possible speed-up factor achievable by using GPU computing for non-Gaussian diffusion mapping.

Methods. *Synthetic DW-NMR data generation.* Brain regions characterized by highly coherent axonal bundles, with different geometrical organization were considered to simulate DW-NMR signal from water molecules diffusing within a realistic medium. In human brain such a region can be identified by the Corpus Callosum (CC) that, according to the estimated axonal diameter distributions within it, can be mainly subdivided into two regions: the first, characterized by denser and smaller axons, includes the genu and splenium (GS-CC); the other, characterized by less dense and bigger axons, includes the body (B-CC).

To simulate DW-signal, a Monte-Carlo (MC) simulation was implemented in C++. A total of 10^4 point like spins were randomly placed in a 2D plane of 0.5x0.5 mm^2, resembling a voxel of the DW-NMR image dataset. A random walk at a rate $\Delta t \sim 2.5 \times 10^{-5}$ s per step, with bulk diffusivity (D_0) set to 1.4×10^{-3} mm^2/s and particle step size $\Delta x = (4D_0\Delta t)^{1/2}$ was performed between randomly packed axons. Axonal diameter distribution and density were numerically reproduced within each voxel of the synthetic image. Specifically, the chosen mean axon diameter $\pm SD$ and axon density percentage were: 2 ± 0.5 μm, ~ 0.50, for G-CC and S-CC; 6 ± 1.5 μm, ~ 0.35 for B-CC. Assuming axons as infinitely long coaxial cylinders, the DW-NMR signal decay for each voxel of the synthetic image due to a Pulsed Field Gradient Stimulated Echo sequence (PGSTE) was simulated through spin phase accumulation. The DW-NMR signal for the whole image was obtained by averaging the signal from each voxel. The parameters of the sequence were chosen to be similar to those of experiments we performed and report elsewhere [5]. We simulated different synthetic images, at different resolutions: 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024 and 2048x2048, to test the performance of CPU and GPU in analyzing images at different resolutions using a voxel-by-voxel fitting approach.

Synthetic DW-NMR data analysis. DKI and STREMI metrics were estimated by fitting on a voxel-by-voxel basis Eqs.(1) and (2) to the synthetic DW image signal intensities, respectively. This procedure was performed for all the image resolution investigated.

Here we used an efficient and robust fitting algorithm, named *GPU-LMFit*, based on a highly parallelized Levenberg-Marquardt (LM) method on GPU, introduced and described in details elsewhere [8]. *GPU-LMFit* uses a scalable parallel LM algorithm optimized for using the Nvidia CUDA platform. The code kernel calls *GPU-LMFit* to perform the LM algorithm on each CUDA block, which is mapped to a single voxel. Because the processing of different voxels is totally independent, the CUDA blocks do not need to synchronize, and the kernel launches as many blocks as voxels contained in a particular slice to speed up performance. The code was optimized to be fully integrated within Matlab (The Mathworks, Natick, MA, USA) scripts. A multi-core central processing unit (CPU) Intel Xeon E5430 processor at 2.66GHz with 8 thread, an Nvidia GPU GeForce GT650m and an Nvidia GPU Titan were used for the analysis and the cross-comparison of CPU and GPU performance. In particular, *lsqcurvefit* function with Parallel Computing Toolbox was used to test multi-core CPU performance.

Results and Discussion. An example of two simulated voxels with realistic geometry and local magnetic field inhomogeneities; the resulting DW-NMR signal and the fitted curves is reported in Figure 1-a), b) and c). The results of the performance test, obtained for each

synthetic image resolution, is instead reported in Figure 1-d).

Numerical results reported in Figure 1-d) suggest that for typical clinical images, whose resolution ranges from 64x64 to 256x256, an expected speed-up factor of ~ 100x (for the Nvidia GeForce GT650m) and ~ 1000x (for the Nvidia Titan) with respect to CPU performance is achievable by using massive parallel GPU computing to perform non-linear fitting of non-Gaussian diffusion models to DW-NMR dataset. Despite the results presented here are based on a simplistic simulation, where several effects including noise are neglected, they are in good agreement with experimental results. In real experiments, noise effects is not negligible, specially at high b-values, and can decrease the performance of both the CPU and GPU algorithms used. Indeed, high noise fluctuations in experimental data may introduce many spurious local minima in the likelihood function to be minimized in fitting routine. This implies that conventional fitting pipeline, based on LM algorithm, often fail in finding the global minimum, becoming strongly dependent on the fitting parameters initialization. Further investigations are therefore in progress in order to optimize the LM based fitting algorithms to make them less sensitive to noise fluctuations.

Conclusion. In this contribution we focused on the application of GPUs in the reconstruction of DW-NMR images based on non-Gaussian diffusion models. By using model-related numerical simulations, the performances of LM based fitting algorithm on CPU and GPU were tested for synthetic images at different resolutions and on two different GPUs: a low and a high-level Nvidia GPU. Our numerical results suggest that the implementation of LM algorithm on GPU makes it excellent for extensive GPU-based applications such as massive MRI processing, further improving the efficiency of the conventional LM model fittings on CPU. Specifically, an expected speed-up factor of ~ 100x (for the Nvidia GeForce GT650m) and ~ 1000x (for the Nvidia Titan) with respect to CPU (Intel Xeon E5430) performance is achievable by using massive parallel GPU computing. These results strongly suggest the GPU computing as a powerful tool for enabling automated parametric non-Gaussian DW-NMR analysis in realtime.

Acknowledgements

References

[1] J.H. Jensen *et al.*, Magn. Reson. Med. **53 (6)** 1432 (2005).

[2] S. De Santis, *et al.*, Magn. Reson. Med. **65 (4)** 1043 (2010).

[3] M. Palombo, *et al.*, J. Magn. Reson. **216** 28 (2012).

[4] J. GadElkarim, *et al.*, IEEE J. Emerg. Sel. Top. Circ. Syst. **3** 432 (2013).

[5] M. Palombo, *et al.*, Magn. Reson. Med.(2014), DOI: 10.1002/mrm.25308.

[6] S. Capuani, *et al.*, Magn. Reson. Imag. **31** 359 (2013).

[7] M. Palombo, *et al.*, J. Chem. Phys. **135 (3)** 034504 (2011).

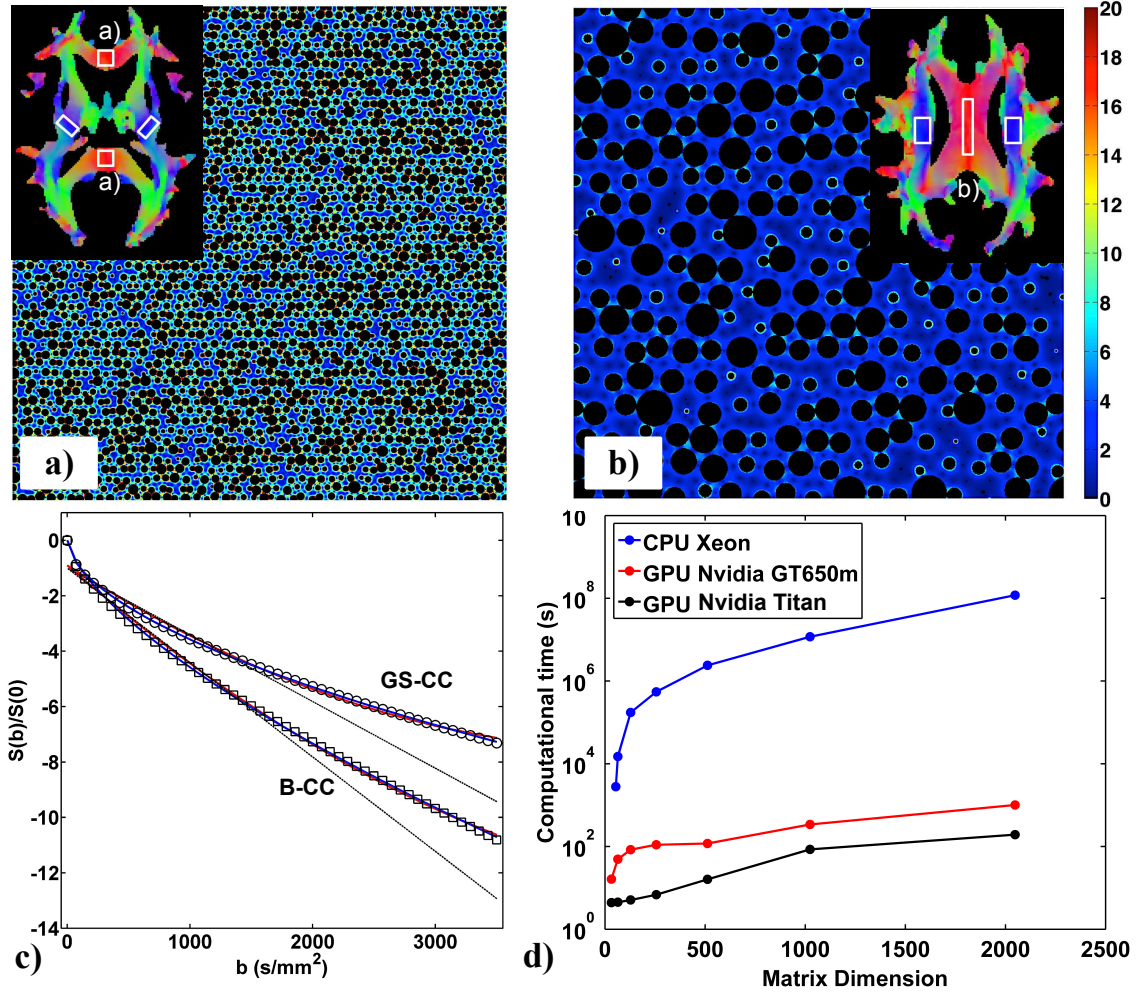[8] X. Zhu , D. Zhang, PloS one **8(10)** e76665 (2013).

Figure 1: a-b) Local axonal geometry and magnetic field inhomogeneities strength spatial distribution computed with $\Delta\chi^{H_2O-TISSUE} = -0.010$ ppm (in IS) and static magnetic field of 3.0 T, for a representative voxel selected within GS-CC and B-CC, respectively, depicted in the WM maps reported in insets. c) S(b)/S(0), as a function of b, for the two representative voxels in GS-CC (circles) and B-CC (squares). Straight lines represent Eq.(1) (red) and (2) (blue), fitted to the simulated data. As comparison, dotted black lines represent the curves of equation $-bD_{app}$, with D_{app} values estimated by the fitting procedures to the data until $b \leq 2000$ s/mm^2. d) Performance test of the low-level, high-level Nvidia GPUs and high performance CPU described in the main text, for different values of image resolutions (matrix dimension).

Chapter 6

Poster Session

CUDA Implementation of CG Inverters for the Faddeev-Popov Matrix

Attilio Cucchieri[1], Tereza Mendes[1]

[1]Instituto de Física de São Carlos, Universidade de São Paulo, Caixa Postal 369, 13560-970 São Carlos, SP, Brazil

The **strong force** is one of the four fundamental interactions of nature (along with gravity, electromagnetism and the weak nuclear force). It is the force that holds together protons and neutrons in the atomic nucleus. The strong interaction is described by Quantum Chromodynamics (QCD) [1], a quantum field theory with local gauge invariance, given by the $SU(3)$ group symmetry. A unique feature of the strong force is that the particles that feel it directly (quarks and gluons) are completely hidden from us, i.e. they are never observed as free particles. This property is known as **color confinement** and makes QCD much harder to handle than the theories describing the weak and electromagnetic forces. Indeed, it is not possible to study QCD analytically in the limit of small energies, or large spatial separations, which corresponds to several processes of interest, including the mechanism of color confinement. To gain insight into these issues, physicists must rely on numerical simulations performed on supercomputers. These studies are carried out using the **lattice** formulation of QCD [2, 3, 4, 5, 6]. Similar studies are done also for other non-Abelian $SU(N)$ gauge theories.

A **propagator** of a field is a two-point function, i.e. a correlation function between values of the field at two different points in space-time [7]. In quantum mechanics, the propagator determines the evolution of the wave function of a system and, for a particle, it gives the probability amplitude of propagating from a point in space-time to another [8]. More generally, **Green's functions** (i.e. n-point functions) carry all the information about the physical and mathematical structure of a quantum field theory. Thus, the study of the long-range —or infrared (IR)— behavior of propagators and vertices is an important step in our understanding of QCD. In particular, the confinement mechanism for color charges could manifest itself in the IR behavior of (some of) these Green's functions.

For gauge theories, such as QCD, the local gauge invariance implies that Green's functions are usually gauge-dependent quantities and can be evaluated only after a specific gauge condition is imposed. Among the possible choices, the so-called Landau (or Lorenz) gauge condition is particularly interesting, since it preserves the relativistic covariance of the theory. The status of lattice studies of IR propagators in Landau gauge has been reviewed in [9].

On the lattice, the **minimal Landau gauge** condition is usually implemented by (numerically) finding local minima of a functional [10]. As a consequence, in this gauge, the path integral over gauge-field configurations is restricted to the set of transverse configurations for which the so-called Landau-gauge **Faddeev-Popov matrix** (FP) is semi-positive-definite [11].

Thus, this matrix should encode all the relevant (non-perturbative) aspects of the theory, related to the color-confinement mechanism.

For a given (thermalized and gauge-fixed) lattice gauge-field configuration $U_\mu(x) \in SU(N)$, with $\mu = 0, 1, 2$ and 3, the FP matrix $\mathcal{M}_U$ in minimal Landau gauge is defined by its action on a function $\gamma^c(x)$ as (see, for example, [12] and [13])

$$
(\mathcal{M}_U \gamma)^b (x) = \sum_\mu \Gamma_\mu^{bc}(x) \left[\gamma^c(x) - \gamma^c(x + e_\mu) \right] + \Gamma_\mu^{bc}(x - e_\mu) \left[\gamma^c(x) - \gamma^c(x - e_\mu) \right]
$$

$$
+ f^{bdc} \left[A_\mu^d(x) \gamma^c(x + e_\mu) - A_\mu^d(x - e_\mu) \gamma^c(x - e_\mu) \right] .
$$

Here, $b, c, d = 1, \ldots, N^2 - 1$ are color indices, f^{bcd} are the (anti-symmetric) structure constants of the $SU(N)$ gauge group, $A_\mu^d(x)$ are the gauge fields defined by the relation

$$
A_\mu^d(x) \lambda^d = \left. \frac{U_\mu(x) - U_\mu^\dagger(x)}{2iag_0} \right|_{\text{traceless}} ,
$$

where λ^d are the generators of the $SU(N)$ group, g_0 is the bare coupling constant, a is the lattice spacing and

$$
\Gamma_\mu^{bc}(x) = \frac{1}{8} \, \text{Tr} \left(\left\{ \lambda^b, \lambda^c \right\} \left[U_\mu(x) + U_\mu^\dagger(x) \right] \right) .
$$

We note that, in the $SU(2)$ case, one finds $\Gamma_\mu^{bc}(x) = \delta^{bc} \, \text{Tr} \, U_\mu(x)/2$ and $f^{bcd} = \epsilon^{bcd}$, where ϵ^{bcd} is the completely anti-symmetric tensor. Also, note that the FP matrix $\mathcal{M}_U$ becomes the lattice Laplacian if $\Gamma_\mu^{bc}(x) = \delta^{bc}$ and $A_\mu^d(x) = 0$.

The **inverse** of the FP matrix enters the evaluation of several fundamental Green's functions of the theory, such as the ghost propagator, the ghost-gluon vertex, the Bose-ghost propagator, etc. These functions can be computed on the lattice through **Monte Carlo simulations** [14]. However, the numerical inversion of the FP matrix is rather time consuming, since it is a very large (sparse) matrix with an extremely small (positive) eigenvalue, thus requiring the use of a **parallel preconditioned conjugate-gradient** (CG) algorithm [15]. Moreover, this inversion has to be done in double precision and, for each lattice configuration, one has to consider hundreds of different sources, corresponding to different kinematic combinations. One should also stress that, in a lattice simulation, one cannot study momenta smaller than $2\pi/L$, where L is the lattice side. Thus, numerical studies of Green's functions in the IR limit (small momenta) require very large lattice volumes and a careful extrapolation of the data to the infinite-volume limit. In fact, inversion of the FP matrix is the performance bottleneck for these numerical studies.

In this study we considered four preconditioned conjugate-gradient algorithms. In particular, for the preconditioner matrix $\mathcal{P}$ we used:

  i) the diagonal elements (with respect to color and space-time indices) of the FP matrix,

  ii) the diagonal elements (with respect to space-time indices only) of the FP matrix,

  iii) the usual lattice Laplacian [16, 17],

iv) the FP matrix with $A_\mu^d(x) = 0$.

In the former two cases described above, the inversion of $\mathcal{P}$ can be done exactly and it does not require inter-GPU communication. On the other hand, for the latter two choices, we employed a (non-preconditioned) CG algorithm. We tested the above choices for the preconditioning step using double and single precision. For the last two cases (choices **iii** and **iv**) we also considered two different stopping criteria for the CG algorithm used to invert the preconditioner matrix $\mathcal{P}$. The code has been written using `CUDA` and `MPI` and tested on multiple GPUs (Tesla S1070 and Tesla K20) interconnected by InfiniBand.
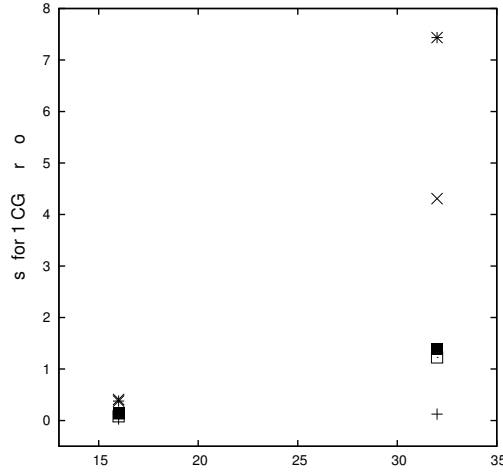

Figure 1: Weak scaling using lattice volumes $V = L^4 = 16^4$ and 32^4 with 1 and 16 Tesla S1070 GPUs, respectively. The data correspond to the CG algorithm without preconditioning (+), with preconditioning **iii** ($\times$), **iv** ($*$), **iii** in single precision ($\square$) and **iv** in single precision ($\blacksquare$).

Our results can be seen in the three plots in Figs. 1 and 2. In particular, we show the processing time for one CG iteration as a function of the lattice side for a fixed *lattice volume/number of GPUs* ratio (weak scaling, Fig. 1) and as a function of the number of GPUs at a fixed lattice size L (strong scaling, Fig. 2). One clearly sees that the overhead due to the inversion of the preconditioner matrix $\mathcal{P}$ is quite large for the cases **iii** and **iv** (see Fig. 1). However, this overhead can be drastically reduced by applying the preconditioner step in single precision (see again Fig. 1). Moreover, this reduction in the processing time for one CG iteration does not affect the convergence properties of the method, i.e. the number of CG iterations necessary to satisfy a given convergence criterion is essentially unchanged when moving from double to single precision. From Fig. 2 we also see that the overhead due to inter-GPU communication is not particularly critical for the algorithms considered, even when the inversion of the preconditioner matrix $\mathcal{P}$ requires inter-GPU communication and for a rather small lattice volume, at least when going from 1 to 4 GPUs.

Of the four preconditioners considered here, the last two are quite effective in reducing the number of CG iterations, typically by a factor of 3–4. We plan to study other preconditioners for the FP matrix, including the so-called even-odd preconditioning, and to fine-tune the
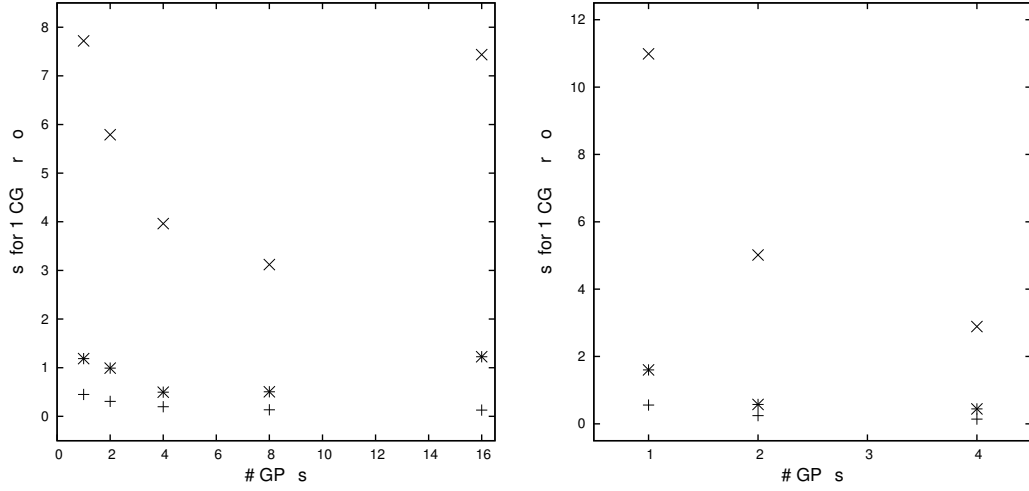
Figure 2: Strong scaling using lattice volume $V = 32^4$ with, respectively, 1,2,4,8,16 Tesla GPUs (left plot) and 1,2,4 Kepler GPUs (right plot). The data correspond to the CG algorithm with preconditioning **i** (+), with preconditioning **iv** (×) and with preconditioning **iii** in single precision (∗).

implementation of the various algorithms.

References

[1] *An Elementary Primer for Gauge Theory*, K. Moriyasu, (World Scientific, Cingapura, 1983).

[2] *Quarks, gluons and lattices*, M. Creutz, (Cambridge University Press, Cambridge UK, 1983).

[3] *Introduction to Quantum Fields on a Lattice*, J. Smit, (Cambridge University Press, Cambridge UK, 2002).

[4] *Lattice gauge theories. An introduction*, H.J. Rothe, (World Scientific, Cingapura, 2005).

[5] *Lattice Methods for Quantum Chromodynamics*, T. DeGrand e C. DeTar, (World Scientific, Cingapura, 2006).

[6] *Quantum Chromodynamics on the Lattice*, C. Gattringer e C.B. Lang, (Springer, Berlim D, 2010).

[7] *Quantum and Statistical Field Theory*, M. Le Bellac, (Oxford University Press, Oxford UK, 1995).

[8] *Modern Quantum Mechanics*, J.J. Sakurai, edited by San Fu Tuan, (revised edition, Addison-Wesley Publishing Company, Reading MA, USA, 1994).

[9] A. Cucchieri and T. Mendes, PoS **QCD-TNT09** 026 (2009).

[10] L. Giusti, M.L. Paciello, C. Parrinello, S. Petrarca and B. Taglienti, Int. J. Mod. Phys. **A16** 3487 (2001).

[11] N. Vandersickel and D. Zwanziger, Phys. Rept. **520** 175 (2012).

[12] D. Zwanziger, Nucl. Phys. **B412** 657 (1994).

[13] A. Cucchieri, T. Mendes and A. Mihara, Phys. Rev. **D72** 094505 (2005).

[14] *A guide to Monte Carlo simulations in Statistical Physics*, D.P. Landau e K. Binder, (Cambridge University Press, Cambridge UK, 2000).

[15] *Iterative methods for sparse linear systems*, Y. Saad, (2nd edition, SIAM, Philadelphia, PA, 2003).

[16] S. Furui and H. Nakajima, Phys. Rev. **D69** 074505 (2004).

[17] A. Sternbeck, E.-M. Ilgenfritz, M. Muller-Preussker and A. Schiller, Phys. Rev. **D72** 014507 (2005).

Implementation of a Data Transfer Method for GPU-based Online Tracking for the PANDA Experiment

L. Bianchi[1]

[1] Forschungszentrum Jlich, Germany

The PANDA experiment (antiProton ANnihilation at DArmstadt) is a new hadron physics experiment currently being built at FAIR, Darmstadt (Germany). PANDA will study fixed target collisions of phase space-cooled antiprotons of 1.5 to 15 GeV/c momentum with protons and nuclei at a rate of 20 million events per second. To distinguish between background and signal events, PANDA will utilize a novel data acquisition mechanism. Instead of relying on fast hardware-level triggers initiating data recording, PANDA uses a sophisticated software-based event filtering scheme involving the reconstruction of the whole incoming data stream in realtime. A massive amount of computing power is needed in order to sufficiently reduce the incoming data rate of 200 GB/s to 3 PB/year for permanent storage and further offline analysis. An important part of the experiment's online event filter is online tracking, giving the base for higher-level discrimination algorithms. To cope with PANDA's high data rate, we explore the feasibility of using GPUs for online tracking. This talk presents the status of the three algorithms currently investigated for PANDA's GPU-based online tracking; a Hough transform, a track finder based on Riemann paraboloids, and a novel algorithm called the Triplet Finder. Their performances and different optimizations are shown. Currently having a processing time of 20 s per event, the Triplet Finder in particular is a promising algorithm making online tracking on GPUs feasible for PANDA.

Contribution not received.

Photon Propagation with GPUs in IceCube

Dmitry Chirkin[1] for the IceCube collaboration[*]

[1]UW-Madison, Madison, WI, U.S.A.

A typical muon event in IceCube creates in excess of 10^7 Cherenkov photons in the sensitive wavelength range, presenting a considerable computational challenge when more than a billion of such muons need to be simulated to represent just a few days of the detector data. However, we note that describing propagation of a large number of photons in a transparent medium is a computational problem of a highly parallel nature. All of the simulated photons go through the same stages: they are emitted, they may scatter a few times, and they get absorbed. These steps, when performed in parallel on a large number of photons, can be done very efficiently on a GPU. The IceCube collaboration uses parallelized code that runs on both GPUs and CPUs to simulate photon propagation in a variety of settings, with significant gains in precision and, in many cases, speed of the simulation compared to the table lookup-based code. The same code is also used for the detector medium calibration and as a part of an event reconstruction tool.

1 Introduction

IceCube (shown on the right, see [1]) is a cubic-kilometer-scale high-energy neutrino observatory built at the geographic South Pole. IceCube uses the 2.8 km thick glacial ice sheet as a medium for producing Cherenkov light emitted by charged particles created when neutrinos interact in the ice or nearby rock. Neutrino interactions can create high-energy muons, electrons or tau leptons, which must be distinguished from a background of downgoing atmospheric muons based on the pattern of emitted Cherenkov light. The photons that reach the detector have wavelengths in the visible range (300 to 550 nm), are emitted at the Cherenkov angle of about 41°, and are subject to both scattering (with scattering length varying between values of 5 to 90 m within the detector volume) and absorption (20-280 m) [2]. Since scattering and absorption lengths

---

[*]http://icecube.wisc.edu
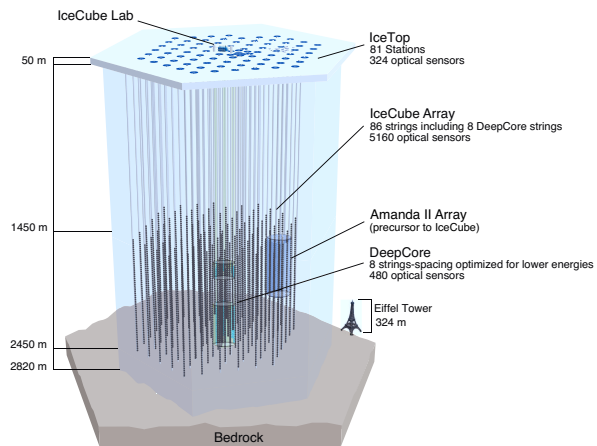
are comparable to each other, and to the typical distances at which the events are observed (20-150 m), existing analytical approximations (single scattering or, on the other end, diffusive) cannot be used, and the photons must be propagated individually in the simulation. Results of such propagation can optionally be tabulated, but the parameter space that must be explored in such an approach is large (in 8 dimensions or more), necessitating certain simplifications. Additionally, the tables are slow to generate, quite large, and may still suffer from certain issues (e.g., related to binning or interpolation strategy).

2 Photon Propagation Software

The photon propagation code (PPC) [3] was initially written to study the feasibility of direct photon propagation for simulation of events in IceCube. The simple nature of photon propagation physics allowed us to focus on the code optimization, to make sure the simulation ran as fast as possible. The simulation was written in C++, then re-written entirely in Assembly for the 32-bit i686 architecture with SSE[1] vector optimizations. The Assembly version of the program used the SSE instructions for updating photon direction when scattered and for locating the optical sensors near the photon segment, while the calculation of the scattering angle was performed by keeping all intermediate variables inside the registers of the float point unit (FPU) stack. The calculation speed improved by 25-37% compared to the C++ version.

Shortly thereafter we were able to demonstrate that significant acceleration of the photon propagation by factors of 150 or more (compared to running on a single CPU core) is possible by using graphics processing units (GPUs). We confirmed this with a version of software that employs the Nvidia GPUs (graphics processing units) via the CUDA[2] programming interface, and a second version that uses OpenCL[3], supporting both Nvidia and AMD GPUs, and also multi-CPU environments. The IceCube collaboration also uses a software module called CLSIM[4] that was independently written for OpenCL, and verified to produce results statistically indistinguishable from those produced with PPC.

The reason for the substantial acceleration of our simulation on GPUs is the highly parallel nature of the simulation of the photon propagation. All of the simulated photons go through the same steps before getting absorbed or hitting a sensor (see Figure 1): photon propagation between the scattering points, calculation of the scattering angle and new direction, and evaluation of whether the current photon segment intersects with any of the optical sensors of the detector array. The GPUs are designed to perform the same computational operation in parallel across multiple threads. Each thread works on its own photon for as long as the photon exists. When the photon is absorbed or hits the detector the thread receives the new photon from a pool of photons for as long as that pool is not empty. Although a single thread runs slower than a typical modern computer CPU core, running thousands of them in parallel results in the much faster processing of photons from the same pool on the GPU.

---

[1]Streaming SIMD Extensions, an extension to the CPU instruction set that operates on vector data.

[2]Compute Unified Device Architecture, a programming model by Nvidia that provides access to the computing features of their GPU cards.

[3]Open Computing Language, a framework for accessing features of devices working on multiple data elements in parallel. This includes devices like CPUs, GPUs, and accelerator cards from multiple manufacturers.

[4]OpenCL-based photon-tracking simulation modeling scattering and absorption of light in ice or water.
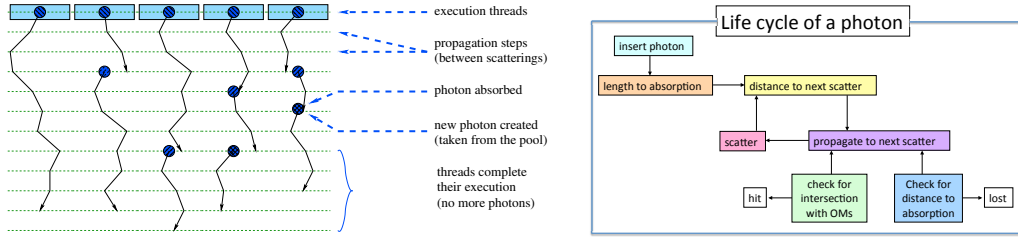
Figure 1: Parallel nature of the photon propagation simulation: tracking of photons entails the computationally identical steps: propagation to the next scatter, calculation of the new direction after scatter, and evaluation of intersection points of the photon track segment with the detector array. These same steps are computed simultaneously for thousands of photons.

3 Simulation with photon propagation code

The direct photon simulation software is typically used in the two scenarios shown in Figure 2. In the first scenario the in-situ light sources of the detector are simulated for calibrating the detector and the properties of the surrounding ice. It is possible to very quickly re-simulate the detector response to a variety of ice scattering and absorption coefficients which are finely tabulated in depth bins. This allows for these coefficients to be fit directly, by finding the combination that is a best simultaneous fit to all of the in-situ light source calibration data [2]. For the 10 meter depth bins, 200 coefficients are fitted (with scattering and absorption defined in 100 layers spanning 1 km of depth of the detector), with nearly a million possible ice parameter configurations tested in less than a week on a single GPU-enabled computer. This method is intractable with a table-based simulation, as each new parameter set would require generation of the new set of parametrization tables, each generation taking on the order of a week of computing time on a $\sim$ 100-CPU cluster.

In the second scenario the Cherenkov photons created by the passing muons and cascades are simulated as part of the larger simulation of the detector response to atmospheric and other fluxes of muons and neutrinos. The simulation is able to account for some effects that are difficult to implement with the table-based simulation, because their simulation would lead to additional degrees of freedom, thus increasing the size of the parametrization effort and tables many-fold. One of these is the tilt of the ice layers, *i.e.*, dependence of the ice parameters not only on the depth, but also on the xy surface coordinates [2]. Another effect, recently discovered, is ice anisotropy [4], which manifests itself as a roughly 13% variation in scattering length depending on the orientation.
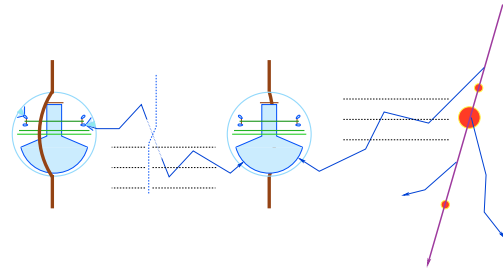

Figure 2: Typical simulation scenarios: photons emitted by the detector are tracked as part of the calibration procedure (left). Cherenkov photons emitted by a passing muon and cascades along its track are tracked to simulate the typical IceCube events (right).

4 IceCube GPUs

The IceCube collaboration operates or otherwise has access to multiple GPU clusters. Recent work on fitting the ice was done at a cluster of 32 Nvidia 690 (2 GPUs each) and 32 AMD 7970 cards (for 96 GPUs altogether). The fitting works by simulating several slightly different ice configurations, waiting for the results, and finding the best configuration. Then creating several slightly modified configurations (based on the best configuration calculated in the previous step), and so on.

We have studied the relative performance of different GPU cards currently in use within our collaboration and summarize our findings in Table 1.

To optimize the use of the GPU-enabled computers further we employ the DAG[5] tools. This involves separating simulation segments into tasks, and assigning these tasks to DAG nodes. DAG assigns separate tasks to different computer nodes; execution of photon propagation simulation is performed on dedicated GPU nodes.

Card	Performance	TDP
AMD 7970	1.77	250
AMD 7950	1.57	200
Nvidia 690	2	300
Nvidia 680	1	195
Nvidia 660	0.62	140
Nvidia 650 ti	0.43	110
Nvidia m2070	0.66	225

Table 1: Relative performance of the tested GPU cards. TDP is the manufacturer's specified maximum card heat generation in Watts.

5 Concluding remarks

We report on a photon propagation tool that replaces the older table-based approach in certain situations, while achieving more precision, better description of the physics of the process and shorter run time. This tool is capable of running on both CPU cores and GPU hardware, achieving significant speed up (factors in excess of ~ 100) on the latter.

References

[1] M. G. Aartsen and others (IceCube collaboration). Evidence for high-energy extraterrestrial neutrinos at the icecube detector. *Science*, 342:1242856, 2013.

[2] M. G. Aartsen et al. (IceCube collaboration). Measurement of south pole ice transparency with the icecube led calibration system. *Nucl. Inst. Meth. A*, 711:73–89, 2013.

[3] Dmitry Chirkin for the IceCube Collaboration. Photon tracking with gpus in icecube. *Nucl. Inst. Meth. A*, 725:141–143, 2013.

[4] Dmitry Chirkin for the IceCube Collaboration. Evidence of optical anisotropy of the south pole ice. *proceedings to 33th ICRC, Rio de Janeiro*, 2013, arXiv:1309.7010.

---

[5]Directed Acyclical Graph, used to represent dependencies in a software chain

Studying of SU(N) LGT in External Chromomagnetic Field with QCDGPU

Natalia Kolomoyets[1]*, Vadim Demchik*[1]

[1]Dnepropetrovsk National University, Gagarin ave. 72, 49010 Dnepropetrovsk, Ukraine

It is well-known that in extreme conditions the behavior of a system is different than in standard ones, see for example [1]. The presence of external (chromo)magnetic field is a particular case of these nonstandard conditions. One of the most powerful methods to investigate quantum field theory (also in extreme conditions) is the lattice Monte Carlo (MC) approach, which allows to investigate problems which cannot be solved analytically. Lattice methods became popular with the increase in performance of computers and the use of GPUs significantly increases the computing performance.

Major research collaborations develop software for MC simulations (see, for example, [2] for a review) that is adopted for their tasks and hardware. However, due to the severe competition between hardware manufacturers, cross-platform principles have to be considered while creating such type of programs.

The open-source package QCDGPU is a package that enables to perform MC lattice simulations of SU(N) gauge theories and O(N) models on AMD and nVidia GPUs. The main feature of the package is the possibility of investigating physical phenomena in presence of an external chromomagnetic field. All functions needed for MC simulations are performed on GPU. The CPU part of the program computes averages of measured quantities over the MC run, prepares the computational device, provides input/output functions, etc. The GPU kernels are implemented in OpenCL to enable execution on devices by various vendors. The host part of the code is written in C++. The package is optimized for execution on GPUs, but can also run on CPUs.

QCDGPU performs gauge configuration production as well as the measurements of the lattice observables. In particular, the package allows to measure the most common lattice observables: mean values of plaquette and action, the Wilson loop, the Polyakov loop, its square and 4-th power. Also some nonconventional measurements are implemented: measurement of components of the $F_{\mu\nu}$ tensor, spatial distribution of the Polyakov loop and action. The last quantities are useful for the investigation of phenomena in presence of external chromomagnetic fields.

The QCDGPU package allows to investigate SU(N) gluodynamics in n-dimensional space. By default the 4D hypercubic lattice is considered but the number n can be changed through the run parameters.

The standard Wilson action for lattice SU(N) theories is considered,

$$S_W = \beta \sum_{x,\nu>\mu} \left(1 - \frac{1}{N} \operatorname{Re} \operatorname{Tr} U_{\mu\nu}(x) \right),$$

summation is performed on all the lattice sites and on all the pairs of space-time directions μ and ν, $\beta = 2N/g^2$ is inverse coupling, $U_{\mu\nu}(x)$ is the plaquette in the point $x = (x, y, z, t)$,

$$U_{\mu\nu}(x) = U_\mu(x)U_\nu(x + \hat{\mu})U_\mu^\dagger(x + \hat{\nu})U_\nu^\dagger(x), \tag{1}$$

$\hat{\mu}$ is the unit vector in direction μ. The field variables $U_\mu(x)$ are SU(N) matrices. For SU(2) and SU(3) groups the following symmetry properties are used to represent link variables:

$$\text{SU(2)}: \quad U = \begin{pmatrix} u1 & u2 \\ -u2^* & u1^* \end{pmatrix}$$

$$\text{SU(3)}: \quad U = \begin{pmatrix} u1 & u2 & u3 \\ v1 & v2 & v3 \\ w1 & w2 & w3 \end{pmatrix}, \quad \vec{w} = \vec{u}^* \times \vec{v}^*.$$

So SU(2) matrices are defined through 4 real numbers and SU(3) matrices are defined through 12 reals instead of 18. Due to the GPU architecture, those matrices are represented as the appropriate number of 4-component vectors. For SU(2) this is one `double4` (`float4`) structure,

`U.uv1 = (Re u1, Im u1, Re u2, Im u2),`

while in SU(3) case three such structures are needed for one matrix:

`U.uv1 = (Re u1, Re u2, Re u3, Re v3);`
`U.uv2 = (Im u1, Im u2, Im u3, Im v3);`
`U.uv3 = (Re v1, Re v2, Im v1, Im v2).`

Other matrix elements are calculated during execution, when they are needed.

For link update the standard multi-hit heat-bath algorithm [3, 4] is implemented. To parallelize lattice update the checkerboard scheme is used.

One of the main purposes of QCDGPU is to enable the investigation of phenomena in external chromomagnetic field. The external Abelian chromomagnetic field is realized through twisted boundary conditions (t.b.c.) [5]. This approach is similar to one used in [6]. The field is introduced as additional flux through plaquettes. This allows to work with the external field as with a continuous quantity. The t.b.c. have the following form:

$$U_\mu'(N_x - 1, y, z, t) = U_\mu(N_x - 1, y, z, t)\Omega, \qquad \Omega = e^{i\varphi\delta_{\mu 2}}, \qquad \forall\, y, z, t,$$

$\varphi = a^2 N_x H$ is the field flux in z direction through the $N_x \times 1$ stripe of plaquettes, see Fig. 1.
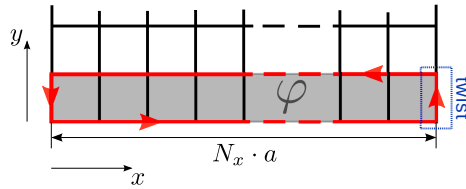

Figure 1: Chromomagnetic flux φ through a stripe of plaquettes

This method is valid for values of the external field flux small enough to neglect the commutators with the external field.

The possibility of varying the external field flux in the QCDGPU package allows to study the dependence of the deconfinement phase transition temperature, the gluon magnetic mass, etc., on the applied external field. One more problem that can be investigated with QCDGPU is the vacuum magnetization in SU(N) gluodynamics.

A general approach to investigate this phenomenon on the lattice is to find the (global) minimum of the action as function of the applied external field. A non-trivial minimum of the action means that a non-zero chromomagnetic field exists in the ground state.

The possibility of vacuum magnetization was investigated recently for SU(3) lattice gluodynamics [5]. The simulations were performed on 4×16^3 lattice at $\beta = 6$. There are two neutral chromomagnetic field components in this group. They correspond to the 3-rd and 8-th Gell-Mann matrices. So the action was considered as function of two field variables. Three sections of this 2D surface were investigated: $\varphi_8 = 0$, $\varphi_3 = 0$ and $\varphi_8 = 6.17\varphi_3$, to compare results with [7]. For the section $\varphi_8 = 0$ the non-trivial minimum of the action was obtained at 95% confidence level (Fig. 2).
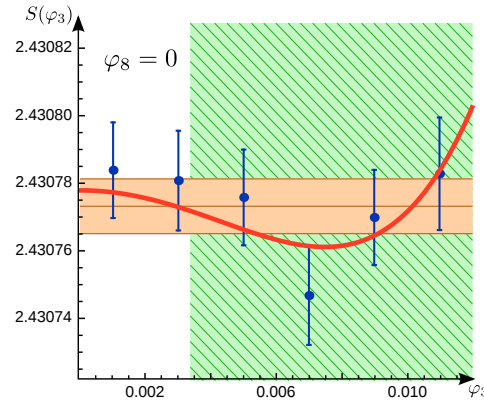

Figure 2: Dependence of action S on applied external field flux φ_3 at $\varphi_8 = 0$. The bars correspond to the 95% CI in the bins, the horizontal stripe is zero level (mean value and 95% CI), the thick curve corresponds to the χ^2-fit result. 95% CI for the position of the minimum is shown by the hatched background

From χ^2 analysis the following estimates and 95% confidence interval (CI) were obtained for generated field:

$$\varphi_3 = \left(7.48^{+11.3}_{-4.11}\right) \times 10^{-3} \qquad \text{or} \qquad H_3 = \left(488^{+734}_{-268}\right) \text{ MeV}^2.$$

The estimated temperature is $T \simeq 370$ MeV.

Also the possibility of direct measurement of the Cartesian components of the SU(N) electromagnetic field tensor $F_{\mu\nu}$ is implemented in the QCDGPU package. The expansion of the plaquette (1) for small lattice spacing,

$$U_{\mu\nu}(n) = 1 + ia^2 F^b_{\mu\nu}(n)\lambda^b - \frac{1}{2}a^4 F^b_{\mu\nu}(n)F^{b'}_{\mu\nu}(n)\lambda^b\lambda^{b'} + \mathcal{O}(a^5), \tag{2}$$

where λ^b are the gauge group generators, is used to extract the field strength. This is done by multiplying (2) by the generators λ^b, and taking the trace [8]:

$$a^2 F_{\mu\nu}^b = -i \operatorname{Tr}\left[U_{\mu\nu}\lambda^b\right] + \mathcal{O}(a^4).$$ (3)

This function of the QCDGPU package provides an alternative approach to investigate the vacuum magnetization phenomenon [8]. The method is based on a study of the distribution function of the total chromomagnetic field strength H. This total field consists of the condensed $\vec{H}_c$ and "quantum" parts. The Cartesian components of $\vec{H}$ are obtained directly from MC run by Eq. (3). They are supposed to be Gaussian ones with mean values $\vec{H}_c$ and variance σ^2. The variance can be obtained as usual sample variance over the MC run. Thus, the absolute values of the field $\vec{H}$ have the following distribution (in dimensionless units):

$$p(\eta) = \frac{16\eta}{\zeta\pi^{3/2}}\, e^{-\zeta^2/4} e^{-4\eta^2/\pi} \sinh \frac{2\zeta\eta}{\sqrt{\pi}},$$

$\eta = H/H_0$, $\zeta = H_c\sqrt{2}/\sigma$, $H_0 = 4\sigma/\sqrt{2\pi}$ is the mean value of H at $H_c = 0$. Consequently, the mean value of η is

$$\bar{\eta} = \frac{16}{\zeta\pi^{3/2}}\, e^{-\zeta^2/4} \int_0^\infty \eta^2 e^{-4\eta^2/\pi} \sinh \frac{2\zeta\eta}{\sqrt{\pi}}\, d\eta = f(\zeta).$$

This can be associated with its estimator from lattice simulations and $f(\zeta)$ must be inverted numerically to calculate the condensate field.

The open-source QCDGPU package is developed as a tool for MC lattice simulations of SU(N) gluodynamics in external chromomagnetic field. The package performs gauge configurations production as well as measurements. All procedures for MC simulations are realized in OpenCL and are optimized for execution on GPUs. CPU performs only averaging of measured quantities over run and auxiliary work for the interaction with the compute device. Apart from common lattice quantities, QCDGPU provides measurement of several non-standard ones. This allows to investigate such problems as the dependence of different quantities on the applied chromomagnetic field, spontaneous vacuum magnetization, etc. The introduction of fermionic fields is among the plans for future developments.

References

[1] M. N. Chernodub, Phys. Lett. B **549** 146 (2002), arXiv:hep-ph/0208105.

[2] V. Demchik and N. Kolomoyets, Comput. Sci. Appl. v. 1, no. 1, 13 (2014), arXiv:1310.7087 [hep-lat].

[3] A. D. Kennedy and B. J. Pendleton, Phys. Lett. B **156** 393 (1985).

[4] N. N. Cabibbo and E. Marinari, Phys. Lett. B **119** 387 (1982).

[5] V. I. Demchik, A. V. Gulov, N. V. Kolomoyets, to be published

[6] M. Vettorazzo and P. de Forcrand, Nucl. Phys. B **686** 85 (2004), arXiv:hep-lat/0311006.

[7] V. V. Skalozub and A. V. Strelchenko, Eur. Phys. J. C **33**, 105 (2004), arXiv:hep-ph/0208071.

[8] V. Demchik, A. Gulov and N. Kolomoyets, arXiv:hep-lat/1212.6185v1 (2012).

[9] P. Cea and L. Cosmai, arXiv:hep-lat/0101017.

Fast algorithm for real-time rings reconstruction.

R. Ammendola[1], M. Bauce[2,3,4], A. Biagioni[2], S.Capuani[3,5], S. Chiozzi[6,7], A. Cotta Ramusino[6,7], G.Di Domenico[6,7], R. Fantechi[2,8], M. Fiorini[6,7], S.Giagu[2,3], A. Gianoli[6,7], E. Graverini[8,9], G. Lamanna[8,10,*], A. Lonardo[2], A. Messina[2,3], I. Neri[6,7], M. Palombo[5,11,12], F. Pantaleo[9,10], P. S. Paolucci[2], R. Piandani[8,9], L. Pontisso[8], M.Rescigno[2], F. Simula[2], M. Sozzi[8,9], P. Vicini[2]

[1]INFN Sezione di Roma "Tor Vergata", Via della Ricerca Scientifica 1, 00133 Roma, Italy
[2]INFN Sezione di Roma"La Sapienza", P.le A. Moro 2, 00185 Roma, Italy
[3]University of Rome "La Sapienza", P.le A.Moro 2, 00185 Roma, Italy
[4] CERN, Geneve, Switzerland
[5]IPCF-UOS Roma, Physics Department, "Sapienza" University of Rome, Rome, Italy
[6]INFN Sezione di Ferrara, Via Saragat 1, 44122 Ferrara, Italy
[7]University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy
[8]INFN Sezione di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
[9]University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
[10] INFN Sezione di Frascati, Italy
[11]CEA/DSV/I2BM/MIRCen, Fontenay-aux-Roses, France
[12]CEA-CNRS URA 2210, Fontenay-aux-Roses, France, France
* Corresponding author. E-mail: gianluca.lamanna@cern.ch

The GAP project is dedicated to study the application of GPU in several contexts in which real-time response is important to take decisions. The definition of real-time depends on the application under study, ranging from answer time of μs up to several hours in case of very computing intensive task. During this conference we presented our work in low level triggers [1] [2] and high level triggers [3] in high energy physics experiments, and specific application for nuclear magnetic resonance (NMR) [4] [5] and cone-beam CT [6]. Apart from the study of dedicated solution to decrease the latency due to data transport and preparation, the computing algorithms play an essential role in any GPU application. In this contribution, we show an original algorithm developed for triggers application, to accelerate the ring reconstruction in RICH detector when it is not possible to have seeds for reconstruction from external trackers.

1   Introduction

Low level trigger RICH reconstruction is very challenging. But the possibility to use in trigger decisions the information coming from Cherenkov rings, would be very useful in building stringent conditions for data selection. Standard algorithms are difficult to use in an on-line environment: some of them requires an external seed, some of them is not adequate in terms of resolution and noise immunity, all of them are too slow. In particular, in high intensity experiments high data rate requires fast data processing setting the maximum complexity allowed for the on-line reconstruction algorithms. The recent development in high throughput processors

provides new opportunities in using information coming from rings in RICH detectors directly in the first trigger level. The GPUs (Graphics Processing Units), in particular, are designed with a parallel architecture in order to exploit the huge computing power offered by a big number of computing cores working together on the same problem. This kind of structure (the so called SIMD (Single Instruction Multiple Data)) allows the design of new parallel algorithms suitable for pattern recognition and ring fitting, with very high computing throughput and relatively small latency. In this paper we will discuss a new algorithm heavily based on the possibility to run concurrently several computing threads on the same processor. The GPU implementation of this algorithm and the results in terms of performances will be discussed.

2 Requests for an on-line ring reconstruction

Most of standard single ring algorithms [7] can not be trivially used in real-time application. The geometrical fits, like for instance Gauss-Newton, Chernov-Lesort and others, are very accurate, but most of them require an initial seed to start, while the algebraic fits, like for instance Taubin, Crawford and others, are less accurate, but usually faster. In both cases it is not easy to extend the ring reconstruction to a multi-ring problem. Specialized algorithms, like Hough transform, fitQun, APfit, based on likelihood and similar, are usually slow with respect to the requirements imposed by high intensity experiments. A good algorithm candidate to be used in low level trigger applications should have the following characteristics:

- seedless,

- multi-Ring reconstruction,

- fast enough to cope with event rate of tens of MHz,

- accurate (events reconstruction with offline resolution).

In addition, depending on the specific experiment requirements, noise immunity and stability of the fitting time with respect to the complexity should be required. Presently no suitable algorithms are available on the market.

3 A physics case: the NA62 RICH

As physics case for the study described in this paper, we consider the NA62 RICH detector. The NA62 experiment at CERN [8] has the goal of measuring the branching ratio of the ultra-rare decay of the charged kaon into a pion and a neutrino anti-neutrino pair. The main interest in this decay is linked to the high precision theoretical prediction of its branching ratio, at the level of few percent, since it is almost free of non-parametric theoretical uncertainties. Because of this, a precise measurement with a sample of 100 events would be a stringent test of the Standard Model, being also highly sensitive to new physics. The trigger is a key system to obtain such a result. In its standard implementation, the FPGAs on the readout boards of each sub-detector participating to the L0 trigger, compute simple trigger primitives. The maximum latency allowed for the synchronous L0 trigger is related to the maximum data storage available on the data acquisition boards; its value was chosen to be rather large in NA62 (up to 1 ms). Such a time budget allows in principle the use of more complex but slower trigger

implementations at this level. This would have the benefit of increasing the trigger selectivity for the $K^+ \to \pi^+ \nu \bar{\nu}$ process, and would allow the design of new trigger condition to collect additional physics processes.

The NA62 RICH [9] is a 17 meters long, 4 meters in diameter, 1 atm Neon filled detector, used to distinguish pions and muons in the 15-35 GeV/c range. The Cherenkov light is focused on two spots equipped with about 1000 small (16 mm in diameter) phototubes each. The ring maximum radius is about 20 cm, while the average number of hits is 20 per ring. Typical NA62 events are one charged particle in final state ($K^+ \to \pi^+ \pi^0$ and $K^+ \to \mu^+ \nu$), but interesting three tracks events are also possible ($K^+ \to \pi^+ \pi^+ \pi^-$, Lepton Flavor Violation modes, etc.).

4 The Almagest

Most of the algorithms mentioned above are efficient for single rings. A strategy to implement a multi-rings reconstruction should use a two step process: first collect the hits (pattern recognition) from the same circle and then perform the fit. Obviously the pattern recognition is the most time consuming and difficult part. We present a new idea based on elementary geometry to address this problem.

In his treatise on astronomy, the Almagest, Ptolemy derived several geometrical results. In particular, in Euclidean geometry, one of Ptolemy's theorems connects the lengths of the four sides and the two diagonals of a cyclic quadrilateral, that is quadrilateral whose vertexes all lie on a single circle. *In a convex quadrilateral, if the sum of the products of its two pairs of opposite sides is equal to the product of its diagonals, then the quadrilateral can be inscribed in a circle.* In the particular case shown in figure 1, the theorem states that:

$$\overline{AC} \cdot \overline{BD} = \overline{AB} \cdot \overline{CD} + \overline{BC} \cdot \overline{DA}$$

This is an useful condition for our problem, since it does not require any other information but the distances between hits inside the RICH. Using this theorem it is possible to decide if a point should be considered for a single ring fit or not. Assuming that the maximum number of hits coming from the NA62 RICH is 64, checking all the combinations of all the hits in groups of four (more than 600000), to decide if the points belong to the same ring, is unfeasible in on-line reconstruction.

To reduce the number of tests, one possibility is to choose few triplets, i.e. a set of three hits assumed to belong to a single ring, and iterate through all the other hits while checking whether Ptolemy's relation is satisfied. In this way, in the worst case scenario of a 64 hit NA62 event, the number of iterations for each triplet turns out to be 61. The hits collected in this phase will be used to fit the ring with a classical single-fit algorithm (like the Taubin method). The problem converges only if the points of the starting triplet come from the same ring. Obviously the choice of the test triplets is decisive for the efficiency of the algorithm. Considering events with two or three rings like, for example, the one shown in figure 2, to maximize the probability that all the points of a triplet belong to the same ring we decided to use 8 triplets and in particular the ones obtained with three points from the top, bottom, right, left and from the U and V diagonal directions.

A key point of this approach is the possibility to run more than one "triplet" at the same time. Modern processors offer the possibility to vectorialize the algorithms in order to exploit parallel computing. In particular the so called streaming processors have the SIMD (Single Instruction Multiple Data) architecture that allow to run the same program on different sets
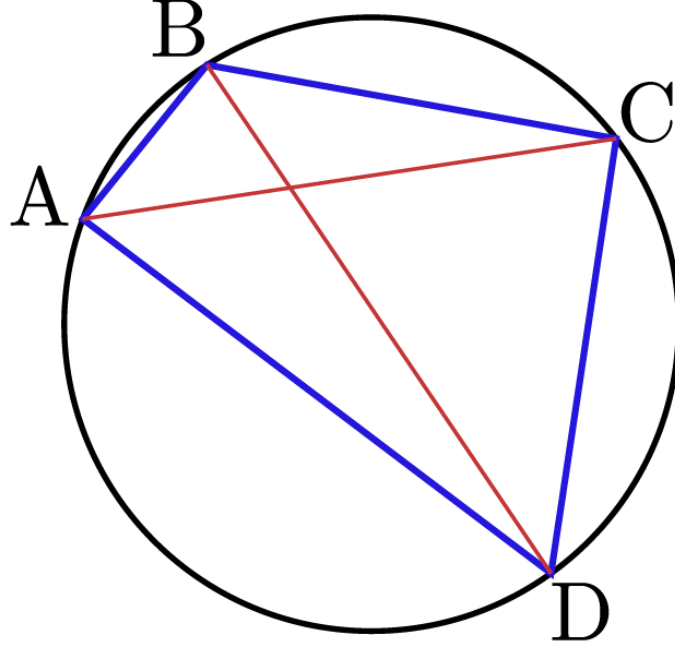
Figure 1: Ptolemy's Theorem.

of data concurrently. Among SIMD processors, the GPUs (Graphics Processing Units) offer a huge computing power, as will be discussed in the next section.

In figure 3 a preliminary result on the reconstruction efficiency for the multi-ring selection is shown. The rings are obtained using GEANT4 Montecarlo, including electromagnetic showers, delta rays and hadronic interaction. For this reason, additional spurious hits are possible. The efficiency strongly depends on hits distribution: asymmetric hits distribution for a ring, for instance, could introduce a bias in the triplets selection described above, as well as points too close each other could affect the single ring (Taubin) fit. Further cuts on hits distance and position can improve the efficiency significantly. The average time to reconstruct a multi-ring event is about 1 μs, using one single GPU.

5   The GPU

Graphic processors represent a viable way to fill the gap between a multi-level system, with a hardware-based lowest level trigger, and a system which would not require preliminary real-time hardware processing on partial event information. Indeed GPUs do provide large raw computing power even on a single device, thus allowing to take complex decisions within a latency that can match significant event rates. This computing power is given by the different processor architecture with respect to the standard CPUs. In GPU more transistors are devoted to computing with respect to the CPU, the number of cores in a GPU can be easily of the order
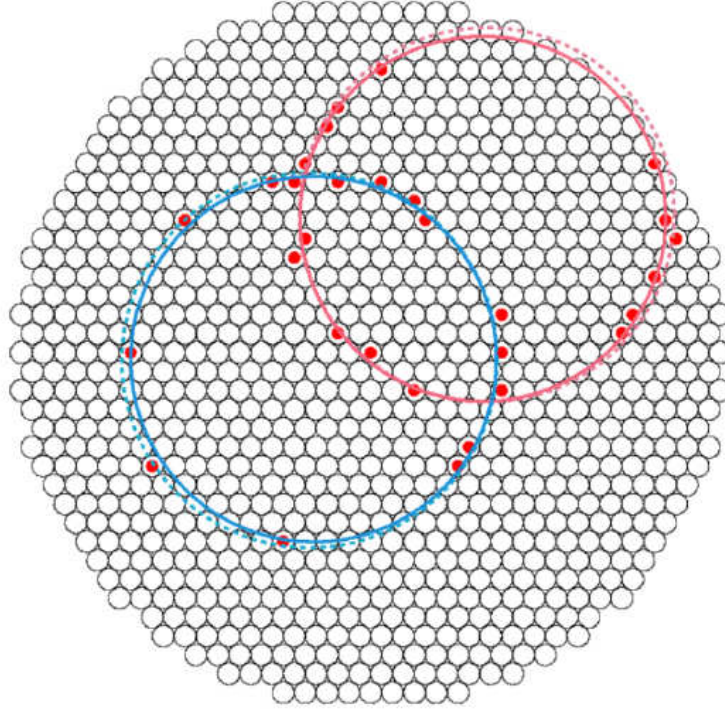
Figure 2: Two rings reconstructed by Almagest's algorithm (solid line). The dotted ring represent the real ring position. The small circles represent the PMTs positions within the NA62 RICH.

of few thousands, with respect to the few units in multi-cores CPU. In addition the computing dedicated structure of the GPU is obtained by reducing the cache dimension, increasing the memory bus and simplifying the control stage.

In recent times GPUs are used for General Purpose computing (GPGPU), and not only for graphics applications. In the High Performance Computing sector there are two major GPU vendors: AMD and NVIDIA. For the moment we are concentrating on NVIDIA, because they can be programmed at a lower level allowing better control of the hardware through the software. GPU parallelism, on Graphic cards produced by NVIDIA, is exposed for general-purpose computing thanks to an architecture called Compute Unified Device Architecture (CUDA). The CUDA architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). A SM is designed to execute hundreds of threads concurrently. Each thread is execute on a single CUDA core. The CUDA cores share a very fast on-chip memory, to allow data exchange and temporary caching. The computing power of recent GPUs can easily exceed 4 Teraflops.
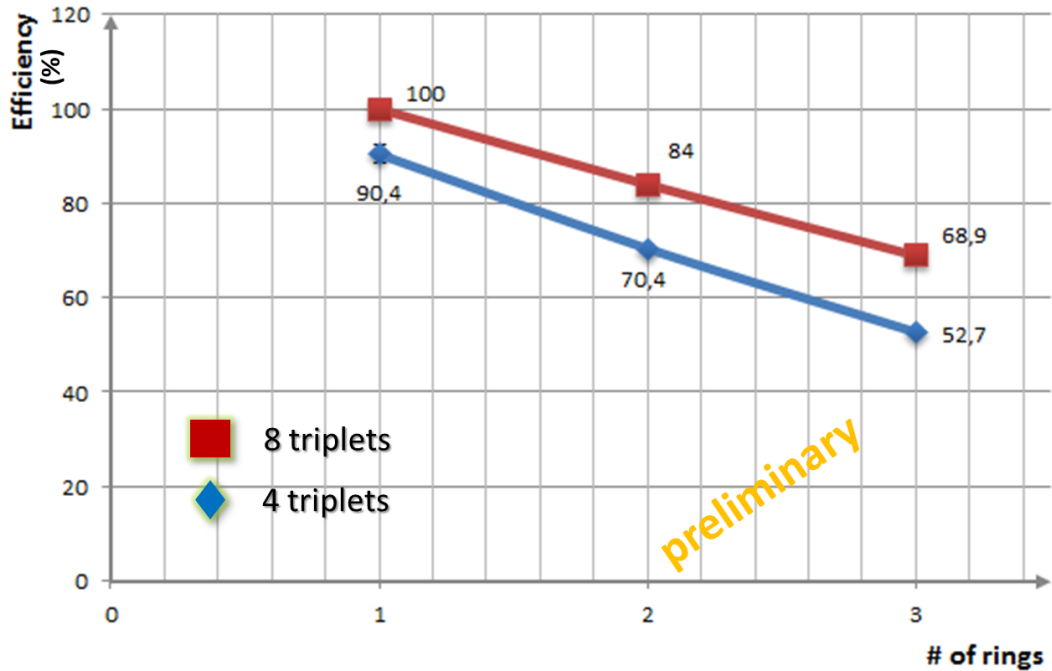
Figure 3: Preliminary efficiency of the algorithm as a function of the number of rings, for 4 triplets (horizontal and vertical) and 8 triplets (horizontal, vertical and diagonals) version. As expected, the efficiency increases with the number of triplets considered.

6 GPU in real-time

The use of GPUs in lowest trigger levels requires a careful assessment of their real-time performance. A low total processing latency and its stability in time (on the scales of hard real-time[1]) are indeed requirements which are not of paramount importance in the applications for which GPUs have been originally developed. This issue is related to the fact that in common usage of GPUs as graphics co-processors in computers, data are transferred to the GPU - and results are transferred back - through the PCIExpress computer bus. Moreover, in order to better exploit the parallel architecture of GPUs, their computing cores should be saturated, thus requiring the computation on a significant number of events after a buffering stage.

GAP (GPU Application Project) is focused on the use of GPU in real-time, both in High Energy Physics and medical imaging. We use two techniques to reduce the data transportation latency: PFRING and NANET. The first is a new fast capture driver, designed to avoid any redundant operation on the data received in a standard NIC (Network Interface Card)[10], while the second is a board named NANET [11], based on an FPGA, in which the possibility to copy directly the data from the FPGA to the GPU is implemented. Preliminary results, on single ring search kernel [12], shows similar performance (figure 4 and figure 5). In the case of

---

[1]A system is defined as "hard" real-time if a task that temporally exceed its deadline causes irrecovarable damage to the system.

PFRING we used a TESLA K20 board, while for the NANET test we used, a little bit older, TESLA M2070. The K20 is almost a factor of two faster than the M2070 in executing the kernel, but all the other components of the latency are almost the same (both boards are used as X16 PCIExpress gen.2). The main component of the total latency is the gathering time
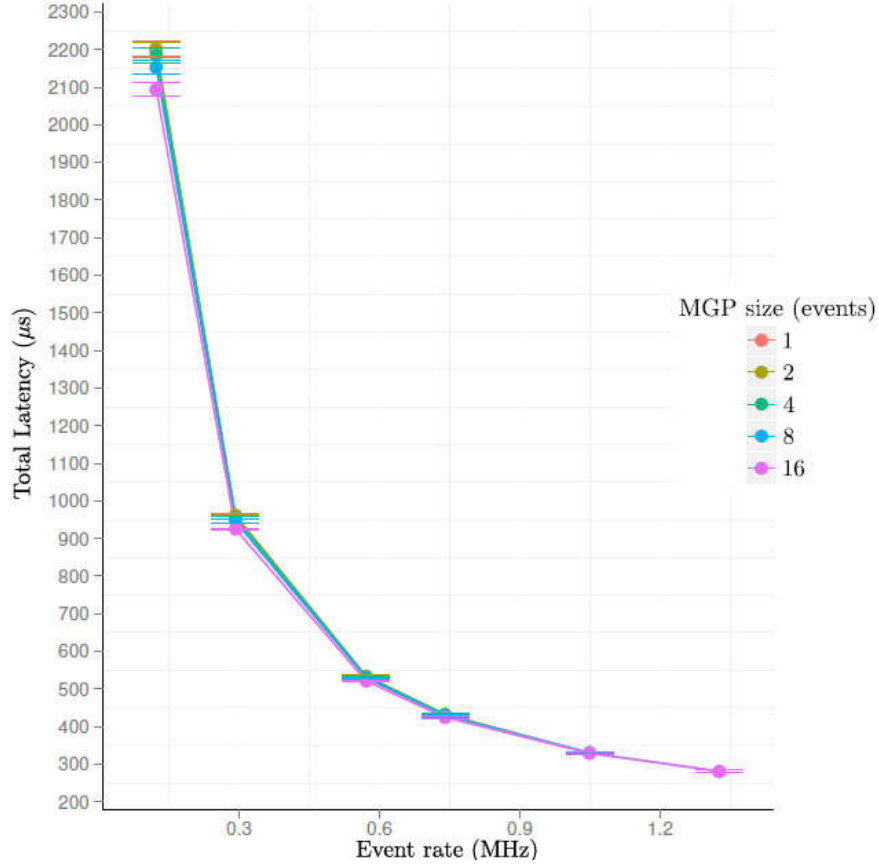

Figure 4: Total time (including data transfer and computing for a single ring kernel) as a function of the rate, for a buffer of 256 events, for PFRING driver solution. The time is independent on the MGP (the number of events per data packet from the detector).

needed to collect a sufficient number of events to better exploit the GPU computing power and to hide the overhead latency due to data transmission. Times of the order of 150/200 μs per event is the limit of the present technology. The latency time fluctuation is well below the μs. This is very important in using GPU for synchronous low level trigger. The behaviour of the GPUs, with respect to the CPUs, is almost deterministic (the non-deterministic component is due to the GPU-CPU interaction in data transfer and kernel loading) and very stable.
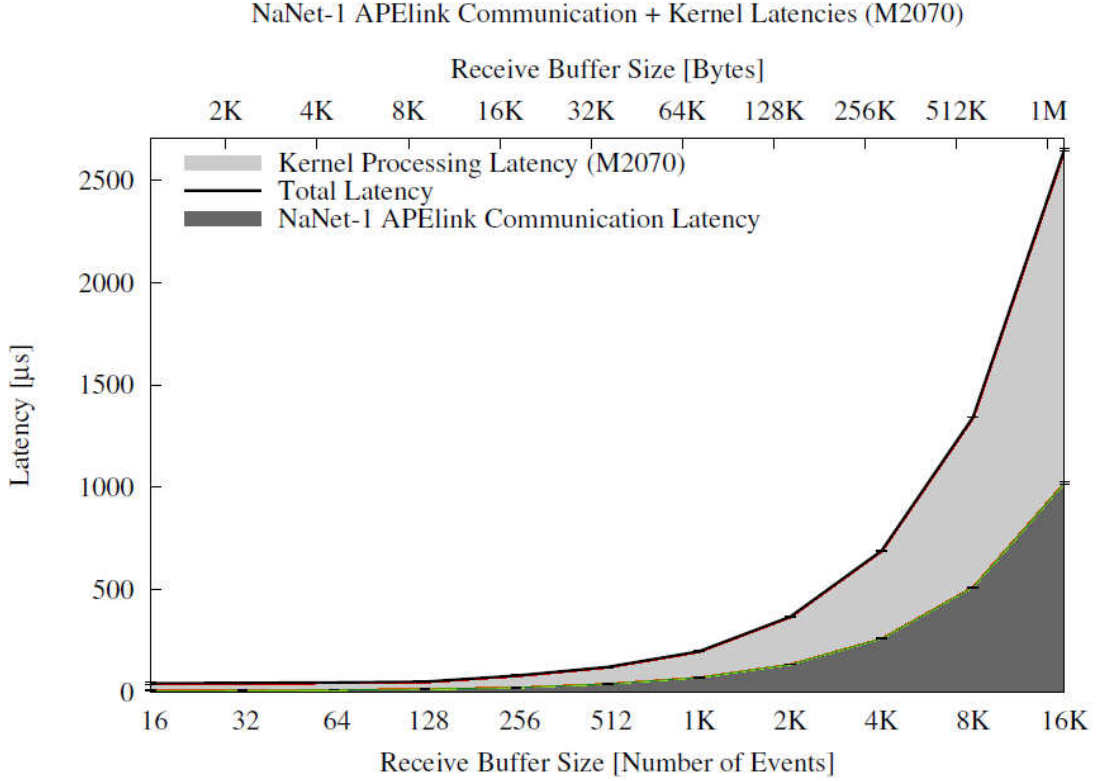
Figure 5: Total time (including data transfer and computing for a single ring kernel) as a function of the buffer dimension, for NANET board solution.

7   Conclusions

We have presented a new algorithm for ring reconstruction. This algorithm, based on Ptolemy's theorem on quadrilaterals, is suitable to be used for circular pattern recognition. A parallel implementation of this algorithm is proposed to run on a GPU, the standard video processor. The computing power for this kind of device is huge for parallel application. Preliminary tests, using the NA62 RICH as physics case, are very encouraging. The maximum latency is in the order of 150/200 μs, while the computing time is 50 ns for single ring, and 1 μs for multi-rings events. Rare events search experiment, like NA62, would benefit on the possibility to exploit on-line ring reconstruction to define selective trigger for data acquisition. A "demonstrator" is in preparation for testing, in parassitic way, during the next NA62 run.

Acknowledgment

References

[1] M. Fiorini et al. *GPUs for the realtime low-level trigger of the NA62 experiment at CERN.* GPU in HEP 2014 conference.

[2] A. Lonardo et al. *A FPGA-based Network Interface Card with GPUDirect enabling real-time GPU computing in HEP experiments.* GPU in HEP 2014 conference.

[3] M. Bauce et al. *The GAP project: GPU applications for High Level Trigger and Medical Imaging* GPU in HEP 2014 conference.

[4] M. Palombo et al. *Estimation of GPU acceleration in NMR medical imaging reconstruction for realtime applications.* GPU in HEP 2014 conference.

[5] M. Palombo et al. *GPU-parallelized model fitting for realtime non- Gaussian diffusion NMR parametric imaging.* GPU in HEP 2014 conference.

[6] G. Di Domenico et al. *Fast Cone-beam CT reconstruction using GPU* GPU in HEP 2014 conference.

[7] N. Chernov *Circular and linear regression: Fitting circles and lines by least squares* Monographs on Statistics and Applied Probability (Chapman & Hall/CRC), Volume 117 (2010).

[8] NA62 Collaboration, *NA62: Technical Design Document, https://cds.cern.ch/record/1404985,* 2010.

[9] B. Angelucci*et al.*, Nucl. Instrum. Meth. A **621** (2010) 205. (*see also M.Piccini at the same conference RICH2013*)

[10] *http://www.ntop.org*

[11] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti and F. Simula *et al.*, J. Phys. Conf. Ser. **396** (2012) 042059.

[12] G. Collazuol, G. Lamanna, J. Pinzino and M. S. Sozzi, Nucl. Instrum. Meth. A **662** (2012) 49.

Participants

- Alves Junior, Antonio Augusto (INFN Roma1, Roma, Italy)
  antonio.augusto.alves.junior@roma1.infn.it

- Arezzini, Silvia (INFN Pisa, Pisa, Italy)
  silvia.arezzini@pi.infn.it

- Ariga, Akitaka (University of Bern, Bern, Switzerland)
  akitaka.ariga@lhep.unibe.ch

- Bauce, Matteo, (INFN Roma1, Roma, Italy)
  matteo.bauce@roma1.infn.it

- Belgiovine, Mauro (INFN Bologna, Bologna, Italy)
  mauro.belgiovine@bo.infn.it

- Berényi,Dániel (Wigner Research Centre for Physics, Budapest, Hungary)
  berenyi.daniel@wigner.mta.hu

- Bianchi, Ludovico (Forschungszentrum Jülich, Jülich, Germany)
  l.bianchi@fz-juelich.de

- Bonati, Claudio (INFN Pisa, Pisa, Italy)
  claudio.bonati@pi.infn.it

- Bouhadef Bachir (INFN Pisa, Pisa, Italy)
  bachir.bouhadef@pi.infn.it

- Bozza, Cristiano (University of Salerno, Fisciano, Italy)
  kryss@sa.infn.it

- Calderini, Giovanni (LPNHE Paris, Paris, France & Universiy of Pisa, Pisa, Italy)
  giovanni.calderini@pi.infn.it

- Calland, Richard (University of Liverpool, Liverpool, United Kingdom)
  rcalland@hep.ph.liv.ac.uk

- Calore, Enrico (INFN Ferrara, Ferrara, Italy)
  enrico.calore@fe.infn.it

- Campora Perez, Daniel Hugo (CERN, Geneva, Switzerland)
  dcampora@cern.ch

- Cesini, Daniele (CNAF, Bologna, Italy)
  daniele.cesini@cnaf.infn.it

- Chamont, David (Lab. Leprince-Ringuet - IN2P3 - CNRS - École polytechnique, Palaiseau, France)
  chamont@llr.in2p3.fr

- Chirkin, Dmitry (University of Wisconsin-Madison, Madison, U.S.A.)
  dima@icecube.wisc.edu

- Collazuol, Gianmaria (INFN Padova & University of Padova, Padova, Italy)
  gianmaria.collazuol@pd.infn.it

- Corvo, Marco (Univesity of Ferara, Ferrara, Italy)
  marco.corvo@fe.infn.it

- Coscetti, Simone (INFN Pisa, Pisa, Italy)
  simone.coscetti@pi.infn.it

- Cucchieri, Attilio University of São Paulo, São Carlos, Brazil)
  attilio@ifsc.usp.br

- Cwiek, Arkadiusz (National Centre for Nuclear Research, Warsaw, Poland)
  arkadiusz.cwiek@fuw.edu.pl

- D'Elia, Massimo (INFN Pisa & University of Pisa, Pisa, Italy)
  delia@df.unipi.it

- Dankel, Maik (CERN, Steinfurt, Germany)
  maik.dankel@cern.ch

- de Fine Licht, Johannes (CERN, Copenhagen, Denmark)
  johannes.definelicht@cern.ch

- Debreczeni, Gergely (Wigner Research Centre for Physics, Budapest, Hungary)
  debreczeni.gergely@wigner.mta.hu

- Di Domenico, Giovanni (University of Ferrara, Ferrara, Italy)
  didomenico@fe.infn.it

- Fiorini, Massimiliano (INFN Ferrara, Ferrara, Italy)
  fiorini@fe.infn.it

- Gallorini, Stefano (INFN Padova, Padova, Italy)
  stefano.gallorini@pd.infn.it

- Gianelle, Alessio (INFN Padova, Padova, Italy)
  alessio.gianelle@pd.infn.it

- Grasseau, Gilles (Lab. Leprince-Ringuet - IN2P3 - CNRS - École polytechnique, Palaiseau, France)
  gilles.grasseau@llr.in2p3.fr

- Herten, Andreas (Forschungszentrum Jülich, Jülich, Germany)
  a.herten@fz-juelich.de

- Howard, Jacob (University of Oxford, Oxford, United Kingdom)
  j.howard1@physics.ox.ac.uk

- Jun, Soon Yung (Fermilab, Batavia, U.S.A.)
  syjun@fnal.gov

- Kaletta, Dietmar (University of Tuebingen, Tuebingen, Germany)
  dietmar.kaletta@uni-tuebingen.de

- Kanzaki, Junichi (KEK, Tsukuba, Japan)
  junichi.kanzaki@cern.ch

- Kisel, Ivan (Goethe University & Frankfurt Institute for Advanced Studies, Frankfurt am Main, Germany)
  i.kisel@compeng.uni-frankfurt.de

- Kolomoyets, Natalia (Dnepropetrovsk National University, Dnepropetrovsk, Ukraine)
  rknv7@mail.ru

- Lamanna, Gianluca (INFN Pisa, Pisa, Italy)
  gianluca.lamanna@pi.infn.it

- Lonardo, Alessandro (INFN Roma1, Roma, Italy)
  alessandro.lonardo@roma1.infn.it

- Mantovani, Filippo (Barcelona Supercomputing Center, Barcelona, Spain)
  filippo.mantovani@bsc.es

- Mazza,Simone Michele (INFN Milano, Milano, Italy)
  simone.mazza@mi.infn.it

- Mesiti,Michele (INFN Pisa & University of Pisa, Pisa, Italy)
  michele.mesiti@pi.infn.it

- Messina, Andrea (INFN Roma1, Roma, Italy)
  andrea.messina@roma1.infn.it

- Messmer,Peter (NVIDIA, Zurich, Switzerland)
  pmessmer@nvidia.com

- Mila,Giorgia (INFN Torino, Torino, Italy)
  giorgia.mila@to.infn.it

- Minuti, Massimo (INFN Pisa, Pisa, Italy)
  massimo.minuti@pi.infn.it

- Morganti, Lucia (CNAF, Bologna, Italy)
  lucia.morganti@cnaf.infn.it

- Nagy-Egri,Máté Ferenc (Wigner Research Centre for Physics, Budapesz, Hungary)
  nagy.mate@wigner.mta.hu

- Neri,Ilaria (INFN Ferrara & University of Ferrara, Ferrara, Italy)
  neri@fe.infn.it

- Palombo, Marco (MIRcen, I2BM, DSV, CEA, Fontenay-aux-Roses, France; CNR Roma, Roma, Italy; Sapienza University of Rome, Rome, Italy)
  mrc.palombo@gmail.com

- Pantaleo, Felice (CERN, Geneva, Switzerland)
  felice.pantaleo@cern.ch

- Piandani, Roberto (INFN Pisa, Pisa, Italy)
  roberto.piandani@pi.infn.it

- Pinke, Christopher (Goethe University, Frankfurt am Main, Germany)
  pinke@th.physik.uni-frankfurt.de

- Pinzino, Jacopo (INFN Pisa & University of Pisa, Pisa, Italy)
  jacopo.pinzino@pi.infn.it

- Pontisso, Luca (INFN Rome, Rome, Italy)
  pontissoluca@gmail.com

- Rei, Luca (INFN Genoa, Genoa, Italy)
  luca.rei@ge.infn.it

- Reichert, Stefanie (University of Manchester, Manchester, United Kingdom)
  stefanie.reichert@hep.manchester.ac.uk

- Reid, Ivan (Brunel University, Uxbridge, United Kingdom)
  ivan.reid@brunel.ac.uk

- Rescigno, Marco, (INFN Roma1, Roma, Italy)
  marco.rescigno@roma1.infn.it

- Rinaldi, Lorenzo (University of Bologna, Bologna, Italy)
  lorenzo.rinaldi@bo.infn.it

- Rodrguez, Juan José (CIEMAT, Madrid, Spain)
  jjr@ciemat.es

- Ruggeri,Andrea (Computer Team Srl, Pisa, Italy)
  andrea@computerteam.it

- Schroeck, Mario (INFN Roma3, Roma, Italy)
  mario.schroeck@roma3.infn.it

- Sokoloff, Michael (University of Cincinnati, Cincinnati, U.S.A.)
  mike.sokoloff@uc.edu

- Soldi, Dario (University of Torino, Bra, Italy)
  dario.soldi@to.infn.it

- Sozzi, Marco (INFN Pisa & Univerity of Pisa, Pisa, Italy)
  marco.sozzi@pi.infn.it

- Spera, Mario (INAF - Astronomical Observatory of Padova, Padova, Italy)
  mario.spera@oapd.inaf.it

- Steinbrecher, Patrick (University of Bielefeld, Bad Oeynhausen, Germany)
  p.steinbrecher@physik.uni-bielefeld.de

- Stelzer, Bernd (Simon Fraser University, Vancouver, Canada)
  bernd.stelzer@cern.ch

- Tupputi, Salvatore Alessandro (CNAF, Bologna, Italy)
  salvatore.a.tupputi@cnaf.infn.it

- Vicini,Piero(INFN Roma1, Roma, Italy)
  piero.vicini@roma1.infn.it

- Vogt, Hannes (Universitaet Tuebingen, Tuebingen, Germany)
  hannes.vogt@uni-tuebingen.de

- vom Bruch, Dorothea (University of Heidelberg, Heidelberg, Germany)
  dorotheavombruch@gmail.com

- Winchen, Tobias (University of Wuppertal, Wuppertal, Germany)
  winchen@uni-wuppertal.de

The total execution times of various CUDA implementation of FDK reconstruction software are in table 3 compared to the OpenMP implementation.

	v0 [s]	v1 [s]	v2 [s]	OpenMP [s]
dataset1	14.31	8.49	7.68	212.7

Table 3: Total execution time of various versions of CUDA implementation of the FDK reconstruction algorithm compared with the OpenMP implementation.

Using GPU + CUDA in CBCT reconstruction we are able to accelerate the reconstruction process of a factor greater than 25. On dataset 1 the reconstruction time is less than 8 s for a volume of 512^3, and this result suggests CUDA implementation permits a real time reconstruction of CBCT data.

Acknowledgment

References

[1] G. Pratx and L. Xing, Med. Phys. **38** 2685 (2011).

[2] nVIDIA Corporation: CUDA C Programming guide Version v5.5 (July 2013).

[3] F. Xu, K. Mueller, Phys. Med. Biol. **52** 3405 (2007).

[4] H. Scherl, B. Keck, M. Kowarschik, J. Hornegger, IEEE NSS Conf. Proc. **6** 4464 (2007).

[5] G. Yan, J. Tian, S. Zhu, Y. Dai, C. Qin, J. X-ray Sci. **16** 225 (2008).

[6] Y. Okitsu, F. Ino, K. Hagihara, Parallel Comput. **36** 129 (2010).

[7] M. Kachelrieβ, M. Knaup, O. Bockenbach, Med. Phys. **34** 1474 (2007).

[8] N. Gac, S. Mancini, M. Desvignes, Proc. 21st ACM Symp. Applied Computing 222 (2006).

[9] L.A. Feldkamp, L.C. Davis, J.W. Kress, J. Opt. Soc. Am. **A1** 612 (1984).

[10] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP*. The MIT Press (2008).

[11] C. Rohkohl, B Keck, H.G. Hofmann,J. Hornegger, Med. Phys. **36** 3940 (2009).