EPJ Web of Conferences **214**, 05041 (2019) https://doi.org/10.1051/epjconf/201921405041
*CHEP 2018*

# Performance optimization of the air shower simulation program for the Cherenkov Telescope Array

*Luisa* Arrabito[1,*], *Konrad* Bernlöhr[2], *Johan* Bregeon[1], *Gernot* Maier[3] for the CTA Consortium,
*Philippe* Langlois[4], *David* Parello[4], and *Guillaume* Revy[4]

[1]Laboratoire Univers et Particules, Université de Montpellier Place Eugène Bataillon - CC 72, CNRS/IN2P3, F-34095 Montpellier, France
[2]Max-Planck-Institut für Kernphysik, P.O. Box 103980, D-69029 Heidelberg, Germany
[3]Deutsches Elektronen-Synchrotron, Platanenallee 6, 15738 Zeuthen, Germany
[4]Université de Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan, Université de Montpellier, Laboratoire d'Informatique Robotique, CNRS, France

**Abstract.** The Cherenkov Telescope Array (CTA), currently under construction, is the next-generation instrument in the field of very high energy gamma-ray astronomy. The first data are expected by the end of 2018, while the scientific operations will start in 2022 for a duration of about 30 years. In order to characterize the instrument response to the Cherenkov light emitted when cosmic ray showers develop in the atmosphere, detailed Monte Carlo simulations will be regularly performed in parallel to CTA operation. The estimated CPU time associated to these simulations is very high, of the order of 200 millions HS06 hours per year. Reducing the CPU time devoted to simulations would allow either to reduce infrastructure cost or to better cover the large phase space. In this paper, we focus on the main computing step (70% of the whole CPU time) implemented in the CORSIKA program, and specifically on the module responsible for the propagation of Cherenkov photons in the atmosphere. We present our preliminary studies about different options of code optimization, with a particular focus on vectorization facilities (SIMD instructions). Our proposals take care, as automatically as possible, of the hardware portability constraints introduced by the grid computing environment that hosts these simulations. Performance evaluation in terms of running-time and accuracy is provided.

## 1 Introduction

The Cherenkov Telescope Array Observatory [1] aims at building and operating the world largest instrument to observe the very high energy gamma-ray sky. The instrument is being built as two arrays of telescopes (one per Earth hemisphere) exploiting the imaging Cherenkov technique to reconstruct the nature, direction and energy of cosmic-ray particles interacting in the atmosphere. High energy gamma-rays from the cosmos interact with the atoms of the atmosphere and convert into electron-positron pairs that then generate a large electromagnetic shower. Electrons and positrons, that go faster than light in the air, cause

*e-mail: arrabito@in2p3.fr

the emission of Cherenkov light (mostly optical blue light), which propagates through the atmosphere to the ground, where it is collected by CTA telescopes. Images of the showers, produced through the Cherenkov light cones, and acquired in the telescope camera are used to reconstruct the properties of the incident particle.

The instrument response functions (IRFs) are then used to go from a list of events to the estimation of the properties of gamma-ray sources. In this scheme, IRFs are produced through Monte Carlo simulations of the whole process: atmospheric air shower, production and propagation of the Cherenkov light, optical telescope response and camera image acquisition. Given the size of the CTA arrays (several $km^2$), the energy range (tens of GeV up to 300 TeV) and the very large parameter phase space, producing accurate IRFs requires large volume of simulations. These simulations are, as of today, mostly run on the EGI (European Grid Initiative) grid resources, that are gracefully made available to the CTA Consortium [2]. Up to 10000 CPU cores can be available at a time, and more than 100 millions HS06 CPU hours have been consumed each year in the past 5 years. In this context, the optimization of the simulation software has multiple purposes: faster production, higher statistics, better coverage of the phase space and eventually cost reductions when CTA is in production phase. Given the large amount of CPU consumed every year for CTA simulations, even a modest improvement of the code performances would allow to save several millions of CPU hours. In this paper we present our first performance optimization of the air shower simulation software used in CTA, called CORSIKA, which is described in Section 2. In Section 3, we present the results of the program profiling and in Section 4 the work on the optimization.

## 2 Corsika, an air shower simulation program

CORSIKA [3], for COsmic Ray SImulations for KAscade, is a program for detailed simulation of extensive air showers initiated by high energy cosmic ray particles. Originally developed at KIT (Karlsruhe Institute for Technology) in the early 90's for the KASCADE experiment, CORSIKA has been rapidly adapted by other experiments for their specific use case. Nowadays, it is the reference air shower simulation software for all cosmic-ray, gamma-ray and neutrino astronomy experiments (*e.g.* Auger, CTA, IceCube, etc.). For a given event, basically, the program tracks the initial particle until its first interaction in the atmosphere, then handles the development of the electromagnetic and hadronic cascades, particle energy losses, interaction processes and the production of the multiple secondary particles that are then propagated until they decay/interact or go below a given energy threshold, or reach the ground.

CORSIKA consists of a single main source code, written in Fortran 77, handling the description of the atmosphere, the stack of particles to propagate and the so-called particle transport. Physics interactions are handled in customized external packages. For the case of Cherenkov astronomy, an additional *IACT/atmo* package [4], written in C, implements the geometry of 3D arrays of Cherenkov telescopes, an improved description of the atmosphere (allowing the use of external atmospheric models) and the photon propagation through the atmosphere. In total CORSIKA consists of more than $10^5$ lines of code. The description and response of the detectors are usually implemented in a separated software package.

### 2.1 The CTA use case

As explained in the introduction, in CTA we are interested in measuring the Cherenkov light produced within electromagnetic air showers. Two steps must then be considered: first the generation of Cherenkov photons, second the propagation of these photons to the ground.

The generation of Cherenkov photons is described by equation 1:

$$\frac{d^2N}{dxd\lambda} = \frac{2\pi\alpha z^2}{\lambda^2}\left(1 - \frac{1}{\beta^2 n^2(\lambda)}\right) \quad (1)$$

where $N$ is the number of photons per unit length of particle path and per unit of wavelength $\lambda$, $z$ the charge of the particle, $\alpha$ the fine structure constant, $n$ the refraction index and $\beta$ the ratio between particle and light speed. While the energy loss of electrons and positrons is both small and smooth in the atmosphere, the production of Cherenkov light changes rapidly with the refraction index. This implies that to get a good description of the Cherenkov light production, small enough steps in the atmosphere are mandatory.

At high energies air showers are composed of hundred of thousands of particles, and even more Cherenkov photons are produced at each step, each of which must be propagated to the ground. The propagation of light through the atmosphere is a very intensive computationally process, indeed for each photon the air refraction index profile must be computed and applied through an interpolation process of external atmospheric tables.

Cherenkov production is implemented in a dedicated subroutine (*cerenk*) of the CORSIKA main program, while the photon propagation is handled in the *raybnd* function of the *IACT/atmo* package. Particle tracks are subdivided into several steps and for each step, the number of emitted Cherenkov photons is calculated from equation 1 within the *cerenk* subroutine. In order to reduce the computing time of photon propagation, all the computations are applied to bunches of typically 5 photons rather than to individual photons. Particle steps are further subdivided into sub-steps so that a single photon bunch is emitted at each sub-step. At each sub-step iteration the *raybnd* function is called to calculate the bending of the photon bunch due to the refraction in the atmosphere and its propagation toward the ground. Finally, the coordinates of the photon bunches intersecting the telescope geometry are recorded and saved in the CORSIKA output (*telout* function in *IACT/atmo*).

## 3 Corsika profiling

Before starting any optimization work on CORSIKA performances, we have performed a code profiling to identify which parts of the program are the most computationally intensive, using the Linux *perf* tool[1]. The profiling was executed on a dedicated server (running CentOS 7.4.1708 on a x86_64 Intel Xeon E5-2650 at 2.20 GHz) using a set of standard input parameters, as those used in the official CTA productions. The program execution duration is controlled by varying the number of generated showers. After having checked the stability of the profiling results for different run durations, we have set to 1000 the number of simulated showers, which for the chosen set of input parameters takes about 5 minutes.

Linux *perf* is an a event-based sampling tool. As sampling event, we have used the cycles event with a sampling frequency of 99 Hz. In order to easily visualize the profiling results, we have processed the sampled data with the *Flame Graph* tool[2]. The obtained profile is shown in Figure 1, where each box corresponds to a particular function or subroutine. The width of the boxes is proportional to the CPU spent in that function. The hierarchy of the function calls is also represented, each box having its parent function just below.

In more detail, Flame Graph reports 82% of the total CPU as consumed by the *cerenk* subroutine, of which about 50% is spent in the *raybnd* function, which we recall is responsible for the photon propagation through the atmosphere. These results confirm that photon propagation is one the most CPU intensive parts of CORSIKA. As explained in Section 2.1, *raybnd*

[1] https://perf.wiki.kernel.org/index.php/Main_Page
[2] http://www.brendangregg.com/flamegraphs.html

is called once per photon bunch. Given that each shower produces on average hundreds of thousands of photon bunches (in the energy range relevant for CTA), in our reference run *raybnd* is called hundreds of millions times. Looking then at the CPU consumption within *raybnd*, we see that a large fraction (65%) is spent in mathematical functions (with the following sharing: 32% *exp*, 19% *asin* and 14% *sincos*), while another fraction (21%) is due to the atmospheric interpolation process, employing a binary search algorithm (*rpol* function). Finally, compatible results have also been obtained using different profiling tools, such as Callgrind[3] and a gdb based profiler[4] [5].

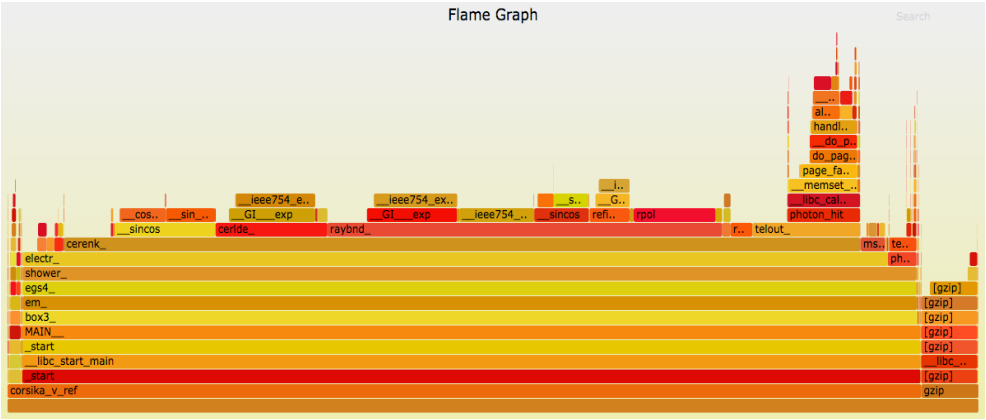


**Figure 1.** Corsika profile produced with Linux *perf* and the *FlameGraph* tool.

## 4 Optimization strategy

Given the size and the complexity of CORSIKA, it seems difficult to achieve a significant optimization by means of optimization options provided by the compiler[5]. Even if compiler has been greatly improved in the last decade, whole program optimization is still a difficult task. Inter procedural analysis (IPA) is often limited by pointer aliasing and compiler heuristics. Nevertheless, in order to get eventual hints for further manual optimization, we have performed extensive tests, measuring the execution times for many different combinations of compilation options. The results of these tests are presented in Section 4.1.

The next step consisted in applying some manual transformations, focusing on the hot parts of the program identified by the profiling. As presented in the previous section, about 40% of the overall CPU time is consumed for the photon propagation through the atmosphere. Since photon bunches propagate independently one from each other, the corresponding code might benefit from the application of vectorization techniques. The basic idea is to group photon bunches by 4 (or 8 according to the available instruction set on the target machine) into vectors and treat them all at once. In this context, we have also considered the use of high-performance (vectorized) mathematical libraries in specific parts of the program. Indeed, we have measured that 50% of the total CPU is consumed by calls to the *libm*. In Section 4.2, we present the first transformations that we have performed in this sense.

[3] http://valgrind.org/docs/manual/cl-manual.html
[4] https://poormansprofiler.org/
[5] https://gcc.gnu.org

The vectorization approach is very promising, since it would bring a speed-up of about a factor 4 (or 8) for the concerned parts of the program, with relatively little code rewriting. Also, this approach is particularly interesting in the context of CTA, since vector instruction sets are widely available on the grid computing resources currently used for the CTA productions.

Finally, since Monte Carlo showers simulation is a massively parallel process, we did not consider the parallelization approach as beneficial in our case. However, the parallelization at the shower level may be interesting for experiments working at very high energies, for which the simulation of a single shower may take several days on a single core.

### 4.1 Automated optimization

In order to explore the compiler optimization capabilities, we have prepared a test framework to easily compile and execute CORSIKA by varying the optimization options. First of all, we have run a reference version of the program, compiled with the CentOS 7 gcc default version 4.8.1. Secondly, we have chosen a list of compiler optimization options, subdivided in the following four categories: 1) standard optimization: *O1, O2, O3*; 2) loop optimization: *ftree-loop-if-convert, ftree-loop-distribution, ftree-loop-distribute-patterns, ftree-loop-im, ftree-vectorize, funroll-loops, funroll-all-loops, floop-nest-optimize*; 3) advanced instruction sets enabling: *mavx, mavx2*; 4) link-time optimization: *flto*. Then, we have compiled CORSIKA with all possible combinations of these options, obtaining in total more than 3000 program versions. Finally, each version was executed in exactly the same conditions, i.e. on the same host and with the same input parameters. The execution time has been measured with the Linux *perf* tool and compared to the reference version. Moreover, since CORSIKA is a stochastic simulation program, in order to ensure reproducible results, we have fixed the seed for the random number generators and checked that the output physics quantities (spatial coordinates and arrival time of the photon bunches) were identical to the reference version. The result of this test campaign is that we could achieve at most 1.03 of speed-up for some specific set of options [5]. Finally, we have tried out the *ffast-math* flag, which optimizes the arithmetic expressions, without preserving however strict IEEE compliance. Even in this case, we did not observe any significant improvement (maximum speed-up of 1.06) [5]. Moreover, as expected, the numerical values of the output physics quantities were slightly different from the reference version, so that their validity would have to be carefully checked. We also mention that running these same tests with higher gcc versions (7.3.1 and 8.2.1) and *O3* flag did not show any performance improvement.

The whole test campaign took about 10 days of CPU, so that repeating the experience several times was not easily affordable. Nevertheless, by repeating our measurements several times for a few selected program versions, we have observed fluctuations of the order of a few percents. These results confirm that the optimization of a complex and large software, as CORSIKA, is beyond the compiler capabilities, while any eventual improvement necessarily go through manual transformations.

### 4.2 Manual optimization through vectorization

Before starting the actual vectorization work, we have identified simple transformations requiring little code rewriting and focusing only on the photon propagation module, which is one of the most costly parts of the program (cf. Section 3). In a first phase, we identified and removed a few redundant calls to functions performing the atmospheric tables interpolation. This first transformation improved performances by a factor 1.09 which is already higher than what was obtained through compiler optimizations (cf. Section 4.1). In addition,

the transformed code exposes some vectorization possibilities for calls to the *exp* function, which is one of the most frequently called mathematical functions as shown by the profiling (cf. Section 3). Our second transformation thus consisted in applying a vectorized *exp* to a selected portion of the program, where 2 or 3 subsequent calls to *exp* could be easily grouped into vectors.

For this purpose, we have relied on external vectorized mathematical libraries. Among the various available, we have tested the SIMD vector *libm* [6] and the CERN's VDT [7] library. The main advantage of these two libraries with respect to others (i.e. Intel's SVML[6], AMD's *libm*[7]), is that they are open source, fully written in high-level languages (C and C++) and hence portable across systems. Another library we may consider in future is the *libmvec*[8] recently added to GNU glibc to support auto-vectorization in gcc. It is also open source, even if not portable on all architectures.

The speed-up achievable with the two tested libraries depends on the considered function. In the case of *exp*, the SIMD vector *libm* announces a speed-up of a factor 4, while VDT library reports an interesting 77 speed-up factor (for SIMD vectors of length 4). These differences may be explained by the fact that the SIMD vector *libm* certifies a certain level of accuracy, while there is no guarantee for VDT. Since performance and accuracy are intimately related, we plan to carry out a detailed study to determine the minimum required accuracy in air shower simulations. A similar study [7] has proven that typical High Energy Physics (HEP) applications do not require the high degree of accuracy, which is commonly provided by standard mathematical libraries [8]. The performance result obtained after our second transformation is an overall speed-up of 1.16 for both the tested libraries. Regarding the accuracy, we verified that we have obtained strictly identical numerical results on the output physics quantities with respect to the reference version.

Encouraged by these results, we have then extended the vectorization to other mathematical functions within the *IACT/atmo* module. This, however required a slightly larger code rewriting, such as the unrolling of the loop over the particle sub-steps (cf. Section 2.1) in the *cerenk* subroutine. This third transformation allowed us to apply mathematical vector libraries also to trigonometric functions (*sin*, *cos* and *asin*) mainly used for the refraction angle calculation and the photon trajectory projection. The obtained speed-up thus increased to a value of 1.20. More details about the three transformations just discussed can be found in [5].

Finally, in order to apply vector operations also to arithmetic expressions, we have tested different high-level libraries, which provide an abstraction of low level SIMD instructions. Among them we mention: bSIMD[9], Vc [9], UME (Unified Multicore Environment) [10], xsimd[10]. In our preliminary non-exhaustive evaluation, we have considered the portability, the support of the most common vector types and operations for different instruction sets (SSE4, AVX, AVX2, AVX-512) as well as the support of vectorized mathematical functions. We found that bSIMD proposes a proprietary and a public version, however it is not clear which functionalities will be supported in the public version. Vc is an open source project widely used within the HEP community. It is the backend of the VecCore library[11], but it does not support mathematical functions. UME is a similar project developed at CERN. It has the advantage over Vc to support AVX-512 instruction set, but with apparently lower overall performances [11]. Finally, we did not test yet xsimd, an open source project, which

[6]https://software.intel.com/en-us/node/583201
[7]http://developer.amd.com/tools-and-sdks/archive/libm/
[8]https://sourceware.org/glibc/wiki/
[9]https://developer.numscale.com/bsimd/documentation/v1.17.6.0/
[10]https://github.com/QuantStack/xsimd
[11]https://github.com/root-project/veccore

**Table 1.** Speed-up obtained with automatic and manual transformations of CORSIKA. The reported speed-up is the cumulated value after each transformation. For each version of the program, the accuracy has been checked by comparing the output numerical values with those obtained with a CORSIKA reference version.

| Transformation | Speed-up | Comments |
|---|---|---|
| Automatic transformation with gcc | 1.03 | Using special optimization options |
| Manual refactoring | 1.09 | Eliminating redundant calls to binary search in interpolation process |
| First manual vectorization | 1.14 | Using vectorized *exp* |
| Extended manual vectorization | 1.20 | Grouping photon bunches by 4 and using vectorized mathematical functions |

seems to cover all the desired functionalities. Our future plan is to conduct a more detailed evaluation of all these products for their later application to CORSIKA.

## 5 Perspectives

In the optimization work presented in Section 4.2, we have restricted ourselves to the Cherenkov propagation module, where we have applied vectorized mathematical functions only to a few selected instructions. The next step will consist in extending the vectorization to a number of arithmetic expressions we have already identified both in the Cherenkov propagation and in the Cherenkov production modules. Moreover, the implementation of some conditional expressions is non optimal with respect to compiler branch prediction. However, since the Cherenkov production module is written in Fortran (*cerenk* subroutine), it is not possible to rely on one of the above mentioned high-level libraries for vectorization (cf. Section 4.2), so that we envisage to fully reimplement it as an external C++ module. A detailed study of the memory access patterns will most likely give us additional hints about further optimization opportunities.

For the longer term, we plan to consider a complementary approach for optimization, which consists in reducing the format of certain variables (i.e. from double precision to single precision), still preserving the desired numerical accuracy of the output physics quantities. Indeed, format reduction can improve the overall performances of a program and has an even more direct impact on the vectorized portions. For this purpose, we will test one or more of the available tools for the identification of the candidate variables for reduction [12] [13].

Finally, another area of improvement concerns the algorithm used for the atmospheric tables interpolation applied in several places (cf. Section 2.1). Atmospheric tables contain the values of various physics quantities (air density, thickness, refraction index, etc.) at given altitudes. The goal of the interpolation procedure is to calculate the values of these quantities at intermediate altitudes (along the particle propagation path). Since the atmospheric profiles are almost exponential, a linear interpolation is performed between *log* values. The drawback of this simple approach is that it implies massive calls to *exp* to later retrieve the interpolated values. In order to dramatically reduce the number of calls to *exp*, we plan to evaluate ad-hoc interpolation schemes that replace linear interpolation of the log values.

## 6 Conclusions

In this work, we have demonstrated that vectorization techniques can be successfully employed in the CORSIKA air shower simulation program. Focusing on one of the most CPU

consuming parts of the program, i.e. the Cherenkov propagation module, we have identified, applied and evaluated successive manual transformations summarized in Table 1. The highest speed-up (1.20) has been obtained by applying 3 consecutive transformations, where the last step consisted in grouping photon bunches by 4 in the Cherenkov production module and applying vectorized mathematical functions in the photon propagation module. If we consider that a typical Monte Carlo campaign takes about 100 millions of CPU hours, a performance gain of 1.20 potentially allows a reduction of 17 millions CPU hours. Given the limited scope of the program transformations performed so far, we think that further optimizations should be achievable. In Section 5, we have presented several optimization options, mostly based on vectorization, precision reduction and optimization of the interpolation algorithm. Finally this work will serve as a basis for the application of vectorization techniques in the context of the recently started project of *Next Generation Corsika* [14].

We gratefully acknowledge financial support from the agencies and organizations listed here: http://www.cta-observatory.org/consortium_acknowledgments.

## References

[1] Actis M. et al. (CTA Consortium), *Design concepts for the Cherenkov Telescope Array CTA: an advanced facility for ground-based high-energy gamma-ray astronomy*, Experimental Astronomy **32**, 193-316 (2011)

[2] Arrabito L. et al., *The Cherenkov Telescope Array production system for Monte Carlo simulations and analysis*, Journal of Physics Conference Series, **898**, 052013 (2017)

[3] Heck D., Knapp J., Capdevielle J. N., Schatz G., Thouw, *CORSIKA: a Monte Carlo code to simulate extensive air showers*, Tech. Rep. FZKA-6019, Forschungszentrum Karlsruhe. https://publikationen.bibliothek.kit.edu/270043064 (1998)

[4] Bernlöhr K., *Simulation of imaging atmospheric Cherenkov telescopes with CORSIKA and sim_telarray*, Astroparticle Physics **30**, 149-158 (2008)

[5] https://gite.lirmm.fr/cta-optimization-group/cta-optimization-project/wikis

[6] Lauter C., *A new open-source SIMD vector libm fully implemented with high-level scalar C*, 50th Asilomar Conference on Signals, Systems and Computers, Nov 2016, Pacific Grove, United States (2016)

[7] Piparo D. et al., *Speeding up hep experiment software with a library of fast and auto-vectorisable mathematical functions*, Journal of Physics: Conference Series, **513**, 052027 (2014)

[8] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754-2008 (2008)

[9] M. Kretz, *Extending C++ for Explicit Data-Parallel Programming via SIMD Vector Types*, Goethe University Frankfurt, Dissertation (2015)

[10] Karpinski P., McDonald J., *An Embedded Domain Specific Language for General Purpose Vectorization*, Kunkel J., Yokota R., Taufer M., Shalf J. (eds) High Performance Computing. ISC High Performance 2017. Lecture Notes in Computer Science, **10524** Springer, Cham (2017)

[11] Moneta L. et al., *Vectorization of ROOT Mathematical Libraries*, CHEP Parallel 372 these proceedings (2018)

[12] C. Rubio-Gonzalez, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, *Precimonious: tuning assistant for floating-point precision*, SC '13, 1-12 (2013)

[13] C. Rubio-Gonzalez, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, *Floating-point precision tuning using Blame analysis*, ICSE 2016, 1074-1085 (2016)

[14] https://arxiv.org/abs/1808.08226