ATLAS PUB Note
ATL-PHYS-PUB-2024-018
18th October 2024

Computational Performance of the ATLAS ITk GNN Track Reconstruction Pipeline

The ATLAS Collaboration

The ATLAS event reconstruction chain is projected to increase dramatically in computational cost with the upgrade to the HL-LHC. A particularly expensive step in this chain is track finding, where energy deposits in the inner tracker (ITk) are grouped into subsets of track candidates, which can then be fitted and provided for downstream tasks. In an effort to reduce execution times and harness accelerator hardware such as GPUs, for both offline and online purposes, machine learning approaches are being developed for track finding. A first functional implementation of a graph neural network-based track pattern reconstruction for ITk has been developed, with competitive physics performance compared with traditional methods. This document describes a variety of improvements to the algorithmic implementations and machine learning models, to significantly decrease execution times from minutes to hundreds of milliseconds.

1 Introduction

In preparation for 140–200 pile-up events per bunch crossing in the HL-LHC era, and the significant resources needed to reconstruct charged particles (tracks) in the new full-silicon Inner Tracker ITk [1, 2], ATLAS is pursuing several methods to reduce resource consumption, including those based on machine learning. Charged particles, when traversing the active silicon, leave energy deposits in the channelized readout. These inputs are clustered to form measurements that can either be used individually (as for the pixel detector) or grouped by adjacent clusters (in the strip detector) to form space points, in the following called "hits". These hits are used to reconstruct the charged particle trajectories. The goal of the Graph Neural Network (GNN) based pattern reconstruction is to identify the subsets of hits in the data that correspond to individual charged particles. A first functional implementation of a GNN-based track pattern reconstruction for ITk was presented in Refs. [3, 4]. Updates for improved reconstruction efficiency were described in Ref. [5] and the first studies of GNN-produced track parameter fits were shown in Refs. [5, 6].

The algorithms described in this note aim to reconstruct those primary particles from top quark pair production and soft interactions which have $p_T > 1$ GeV, are produced at transverse radius $R < 26$ cm and $|\eta| < 4$, leave at least 3 hits and that are not electrons, a set of conditions called hereafter *target particles*. Electrons are excluded from the target particles due to their special behavior stemming from bremsstrahlung. For this study, the full Geant4-based ITk simulation is used, with ITk layout version 23-00-03 [7] and pileup of $\mu = \langle 200 \rangle$. A general introduction to track finding with the ITk can be found in Ref. [7]. We describe each stage of the GNN4ITk pipeline in Section 2 along with its computational optimizations. The results of these optimizations are presented in Section 3. Evaluations are performed on either one Nvidia A100 40Gb GPU or one core of an AMD EPYC 7763 CPU, directly running PyTorch in inference mode [8]. We also make use of Numba for compiled CPU operations [9], Pytorch Geometric for graph scatter operations [10], Scikit-learn [11] and Nvidia RAPIDS [12] for graph partitioning, and FRNN for fast nearest neighbor search [13]. Data loading and writing times are omitted, as a final implementation will run in sequence on-device. Unless otherwise mentioned, physics performance is maintained as that reported in Ref. [6].

2 GNN4ITk Pipeline Optimisations

2.1 Overview

In this section, we describe the nominal GNN4ITk track finding pipeline, as well as computational optimisations developed since the physics results reported in Ref. [6]. We focus on those improvements relevant to inference (as opposed to training), which currently runs in the ACORN [14] python codebase, with the exception of the custom CUDA kernels for graph construction (see Sec 2.2). Hits are provided from the Athena space point formation algorithm, each of which are composed either of a single cluster from the pixel sub-detector, or of a stereo pair of clusters from the strip sub-detector. To treat the track finding problem as a graph segmentation problem, hits are first connected pairwise into a graph structure, such that each node represents a hit (with its features attached) and each edge connects a doublet of hits (which may include pair-wise features). If an edge is treated as a hypothesis that the two hits were created by successive energy deposits by the same particle, then one can apply an edge scoring algorithm to label edges as either "true" (are indeed successive hits on a particle track) or "fake" (either belong to different

particles, or are not successive in a particle track). Equipped with a set of edge-wise probabilities for these labels, the graph can be segmented into collections of nodes that are believed to belong to the same track. The output of segmentation is a set of lists, with each entry a track candidate.

These three stages are sketched in Fig. 1.

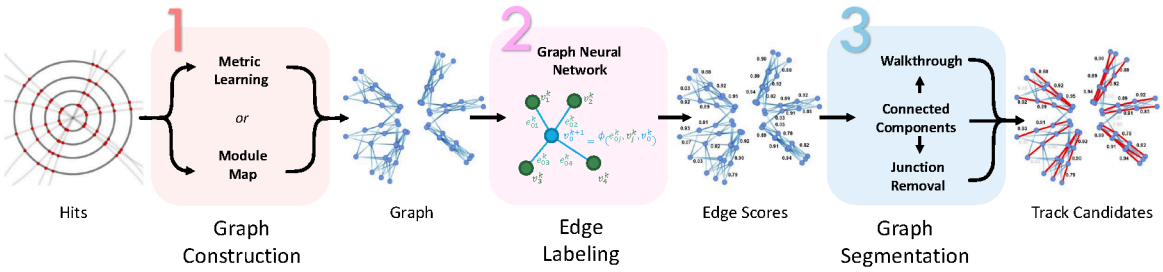

Figure 1: The GNN4ITk track finding inference pipeline, divided in three stages: (1) Graph construction: graphs are built from hits using either Module Map or Metric Learning method; (2) Edge labelling: graph edges are labeled with a score inferred by the GNN model, indicating the likelihood of belonging to the same track; (3) Graph segmentation: track candidates are reconstructed using either only a Connected Component algorithm, or with a combination of Connected Components and Walkthrough or Junction Removal methods.

2.2 Graph Construction

At $\langle\mu\rangle = 200$, a $t\bar{t}$ event contains $O(10^5)$ hits. A fully-connected graph, though easy to build and guaranteeing all true connections, would boast $O(10^{10})$ edges and render the pipeline computationally unfeasible due to GPU memory and time constraints. It is therefore important to control the graph size to a reasonable level while having as many true edges as possible. To characterize a graph, we use two quantities throughout the pipeline as performance metrics: the edge efficiency, denoted by ϵ, defined as the ratio of target true edges contained in the graph to all target truth connections, and the edge purity, denoted by p, defined as the proportion of target true edges to all edges in the graph excluding true edges not from target particles. In this language, graph construction must be highly efficient ($\epsilon \approx 100\%$) to avoid broken track candidates, and sufficiently pure ($p > 1\%$) to satisfy GPU memory constraints. We employ two possible methods to construct graphs: a machine learning model based on *Metric Learning* [15, 16] and a data-driven technique called *Module Map*.

Metric Learning concerns learning the similarity between pairs of data points $\{\mathbf{x}_1, \mathbf{x}_2\}$ in $\mathbb{R}^n$ via a distance function $\tilde{d}(\mathbf{x_1}, \mathbf{x_2})$, and, in conjunction with nearest-neighbor methods, allows clustering data based on a predefined criterion reflected in the choice of $\tilde{d}$. Our approach learns a metric function of the form $\tilde{d}(\mathbf{x}_1, \mathbf{x}_2) = d(f(\mathbf{x}_1), f(\mathbf{x}_2))$, where d is the Euclidean distance and $f : \mathbb{R}^n \to \mathbb{R}^m$ is a multi-layer perceptron (MLP). The network is trained on a contrastive hinge loss defined as

$$l(\mathbf{x}_1, \mathbf{x}_2) = y_{12}\tilde{d}(\mathbf{x}_1, \mathbf{x}_2) + (1 - y_{12}) \max\{0, r - \tilde{d}(\mathbf{x}_1, \mathbf{x}_2)\}. \quad (1)$$

Minimizing the loss is equivalent to finding a transformation f that pulls true hit pairs ($y_{12} = 1$) close together and pushes false hit pairs ($y_{12} = 0$) at least a margin r away from each other. A k-Nearest-Neighbor (kNN) search connects a hit to k nearest hits in a hypersphere of radius r' centered on it, yielding a graph. A second MLP is trained to reduce the graph size, using pairwise geometric features.

The Module Map approach constructs a look-up table containing all triplets of detector modules that could be connected by a track candidate. The module map is built along with a set of geometric cuts on doublets and triplets from 90k simulated $t\bar{t}$ events. Upon inference, all possible triplets present in both the event and the module map are admitted. Geometric cuts are then applied to eliminate unphysical connections and yield the desired graph. Since the results reported in Ref. [6], comprehensive algorithmic optimizations were developed to largely improve the compute time performance of the graph building with the Module Map method, while keeping the same physics performance (the timing result is reported in Table 1).

The algorithm now utilizes a unified doublet module map in conjunction with a triplet module map. The unified doublet module map is constructed by collapsing all triplets in the module map to doublets, and for each doublet geometric cut finding the minima and maxima across all the triplets that it belongs to. The introduction of the unified doublet module map, as a first screening before the triplet selection, effectively reduces approximately 90% of the combinatorial complexity associated with hit connections. The residual connections are filtered through the triplet module map. Furthermore, the entire codebase has been ported to GPU architecture, incorporating custom CUDA kernels specifically developed for managing hit connections via the doublet and triplet module maps, as well as for implementing geometric cuts.

Both methods produce graphs of comparable efficiency and purity, in particular, with $\epsilon > 99\%$ and a number of edges of the order $O(10^6)$ giving a $p \approx 3\%$.

2.3 Edge Classification

A graph neural network (GNN) is trained to discriminate true edges from fake. The architecture comprises an initial encoding step where node and edge features are encoded into a latent space of dimension D using two independent MLPs, one for node features (such as 3D position) and another for edge features (such as position differences). A full list of these features can be found in Ref. [17]. This is followed by message-passing steps, which consist of Interaction Network layers [18]. Each edge's latent representation is updated by an edge network MLP that integrates its current latent state with the latent states of its source and destination nodes. Each node's latent state is then updated by a node network MLP which incorporates its current state with aggregated incoming and outgoing edge latent features. Each iteration of the Interaction Network layer propagates information through the graph, enabling the model to learn complex geometric patterns in particle tracks. The number of these iterations, denoted L, is referred to as the level of message passing. The GNN utilizes a non-recurrent architecture for the Interaction Network layers, meaning each layer has its own set of parameters for node and edge updaters, distinct from those used in other iterations. In the final step, termed the decode step, the edge features in the latent space, resulting from the last GNN iteration, are decoded using an edge decoder MLP into an edge classification score.

The GNN model described in this paper is configured with a latent dimensionality of $D = 128$ and employs a message-passing depth of $L = 8$. The aggregation function utilized is the *sum* function. Each multilayer perceptron (MLP) comprises three linear layers with ReLU activation functions. Batch normalization layers are incorporated after each linear layer. The GNN model was trained using a binary cross-entropy loss function on a dataset of 8,000 $t\bar{t}$ events over approximately 200 epochs and evaluated on a test set of 1,000 $t\bar{t}$ events. The cumulative performance (including that of the first stage) gives an edge-wise efficiency of approximately 98% and a purity of about 95% [5, 6, 17].

To reduce the network's runtime during test inference, two optimization techniques were employed. Firstly, mixed precision was utilized, whereby certain operations use the `torch.float32` data type, while others

employ `torch.float16`, facilitated by PyTorch's Automatic Mixed Precision (AMP) tool [8]. Casting to lower precision enables the usage of the Nvidia GPU's dedicated half-precision Tensor Cores, as well as reduces the memory bandwidth requirement. Secondly, the recent PyTorch *model compilation method* was applied, which accelerates code execution by JIT-compiling it into optimized kernels and optimizing the computation graph using the `torch.compile` method from PyTorch 2.x [19]. This compilation was performed individually for each MLP within the GNN, excluding other components such as aggregation functions.

2.4 Graph Segmentation

Given a set of scored edges, there are several solutions for producing a set of track candidates. The simplest case treats the graph as undirected, applying a score threshold below which edges are removed, and treating all remaining connected subgraphs as track candidates. This is the "Connected Component" (CC) approach, and while computationally inexpensive it is prone to merging tracks (when the score threshold is too low) or splitting tracks (when the score threshold is too high). At the other end of complexity, one can retain the directedness of the graph, and traverse the edges outward from every *source* node (i.e. a node with no incoming edges), forming a track candidate from the longest path constructed for each source node. This "Walkthrough" method was used to produce the physics results reported in Ref. [6], but at that time required several minutes per event. The algorithm has five main steps:

- *Remove Cycles* ensures the directed graph is acyclic.
- *Filter Graph* removes low-scoring edges.
- *Extract Chains* finds connected components that are "chain-like" (every node has at most one incoming edge, one outgoing edge).
- *Topological Sort* orders nodes such that earlier nodes can visit subsequent nodes if a path exists between the two.
- *Build Paths* traverses the graph from each source node, proposing the longest path for each that is compatible with certain score conditions.

To optimize computational performance, two approaches were implemented: the first is a so-called "FastWalkthrough", whereby all previous steps using the slow NetworkX [20] library are replaced with a combination of Pytorch Geometric [10] operations and Numba [9] JIT-compilation of pure python. These replacements improve execution time by two orders of magnitude.

The other approach is to retain the high throughput of the CC algorithm, but with a preprocessing step to avoid merges at graph junction points. This "Junction Removal" (JR) approach has several possible heuristics to resolve junctions. The most performant chooses the highest-scoring incoming edge and outgoing edge for each node, pruning all others from the graph. Combining this pre-processing with CC gives the new algorithm CC+JR, the execution time of which is reported in the following section.

3 Execution Time Results

Reported below are per-event running times of various choices of graph construction and graph segmentation algorithms. In Table 1, execution times from each stage in the pipeline (Fig 1) are reported. Graph construction may either be *Metric Learning* or *Module Map*. Edge scoring is performed with a graph neural network optimised with Pytorch compilation and mixed precision. The different inference time for the Edge Classification between the Metric Learning pipeline and Module Map pipeline is from the different input graph size. The former has about 8×10^5 edges while the latter has about 2×10^6 edges. In an ablation study with the Module Map pipeline, mixed precision is found to reduce the inference time of the GNN edge classifier by 40%, torch compilation achieves a 60% reduction, and combining both techniques leads to a reduction of over 80%. Graph Segmentation (a.k.a. Track building) is performed by the *FastWalkthrough* — first removing low-scoring edges then traversing the remaining graph, selecting the longest path for each source node. Physics performance is that reported previously in Ref. [6].

Stage	**Pipeline**	
	Metric Learning (ms)	**Module Map (ms)**
1. Graph Construction	505	69
2. Edge Classification	108	323
3. Graph Segmentation	118	118
Sum	**731**	**510**

Table 1: Per-event execution times of each stage in the GNN-based ITk track finding pipeline, for both choices of graph construction technique. Stages 1 and 2 are evaluated on a GPU (40GB Nvidia A100). Stage 3 is evaluated on a single CPU core (AMD EPYC 7763).

Table 2 reports the overall performance of graph segmentation algorithms, and Figure 2 shows a breakdown of the timings for the *FastWalkthrough* algorithm. The discrete steps of the FastWalkthrough algorithm are described in Section 2.4, with the bottleneck to performance being the sequential traversal of the graph, even after vectorization and compilation with Numba. In Table 2 a significant improvement can be seen in execution time over an earlier version of the walkthrough algorithm, which was used to produce the physics results in Ref. [6], and referred to here as *CTD23 Walkthrough*. While *CC* is the fastest, it leads to the worst efficiency. *CC+JR* allows for fast inference as well as high track building efficiency.

These results represent a first step towards a GNN-based ITk track finding algorithm that meets the computational requirements of HL-LHC reconstruction. While straightforward optimisations have now been made, there remain a number of possible improvements with further use of custom CUDA kernels, simplification of redundant GNN layers and knowledge distillation, which will continue to target equivalent high-efficiency physics performance.

Stage	Efficiency (Relative Difference, %)	Running Time (ms)
CTD23 Walkthrough	——	42,000
FastWalkthrough	+0.53	120
CC	−1.33	6.0
CC+JR	+0.93	40

Table 2: Per-event execution times and relative difference in integrated physics efficiencies of the various graph segmentation techniques available in Stage 3 (graph segmentation). Differences are calculated relative to the baseline *CTD23 Walkthrough* as $(\epsilon_i - \epsilon_{CTD})/\epsilon_{CTD}$. The score cut on CC set to 0.01, with the minimum and additive thresholds of walkthroughs set to 0.1 and 0.6 respectively. The running times are evaluated on a single CPU core (AMD EPYC 7763). *CTD23 Walkthrough* is the same as that used in Ref. [6].

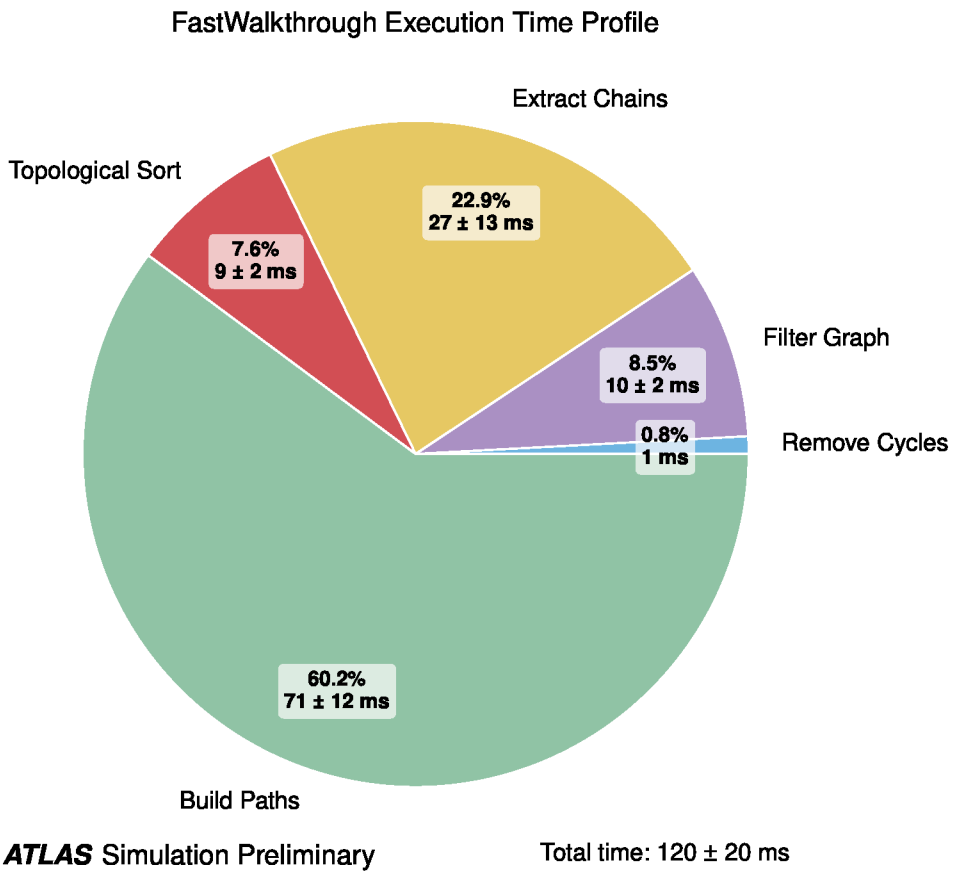

Figure 2: Per-event execution times of each step within the optimized *FastWalkthrough* algorithm. Statistical uncertainties across events are reported, except where they are below precision. Timing is evaluated on a single CPU core (AMD EPYC 7763).

References

[1] *Technical Design Report for the ATLAS Inner Tracker Strip Detector*, 2017, URL: https://cds.cern.ch/record/2257755 (cit. on p. 2).

[2] *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*, 2017, URL: https://cds.cern.ch/record/2285585 (cit. on p. 2).

[3] M. J. Atkinson et al., *Track finding performance plots for a Graph Neural Network pipeline on ATLAS ITk Simulated Data*, 2022, URL: https://cds.cern.ch/record/2807132 (cit. on p. 2).

[4] S. Caillou et al., *ATLAS ITk Track Reconstruction with a GNN-based pipeline*, 2022, URL: https://cds.cern.ch/record/2815578 (cit. on p. 2).

[5] S. Caillou et al., *Physics Performance of the ATLAS GNN4ITk Track Reconstruction Chain*, EPJ Web of Conf. **295** (2024) 03030, URL: https://doi.org/10.1051/epjconf/202429503030 (cit. on pp. 2, 4).

[6] J. D. Burleson et al., *Physics Performance of the ATLAS GNN4ITk Track Reconstruction Chain*, 2023, URL: https://cds.cern.ch/record/2882507 (cit. on pp. 2, 4–7).

[7] ATLAS Collaboration, *Expected tracking and related performance with the updated ATLAS Inner Tracker layout at the High-Luminosity LHC*, ATL-PHYS-PUB-2021-024, 2021, URL: https://cds.cern.ch/record/2776651 (cit. on p. 2).

[8] A. Paszke et al., 'PyTorch: An Imperative Style, High-Performance Deep Learning Library', *Advances in Neural Information Processing Systems*, ed. by H. Wallach et al., vol. 32, Curran Associates, Inc., 2019, URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf (cit. on pp. 2, 5).

[9] S. K. Lam, A. Pitrou and S. Seibert, 'Numba: A llvm-based python jit compiler', *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015 1 (cit. on pp. 2, 5).

[10] M. Fey and J. E. Lenssen, *Fast Graph Representation Learning with PyTorch Geometric*, 2019, arXiv: 1903.02428 [cs.LG], URL: https://arxiv.org/abs/1903.02428 (cit. on pp. 2, 5).

[11] F. Pedregosa et al., *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research **12** (2011) 2825 (cit. on p. 2).

[12] R. D. Team, *RAPIDS: Libraries for End to End GPU Data Science*, 2023, URL: https://rapids.ai (cit. on p. 2).

[13] L. Xue, *FRNN: Fixed Radius Nearest Neighbor*, https://github.com/lxxue/FRNN, 2024 (cit. on p. 2).

[14] M. J. Atkinson et al., *ACORN*, URL: https://gitlab.cern.ch/gnn4itkteam/acorn (cit. on p. 2).

[15] B. Ghojogh, A. Ghodsi, F. Karray and M. Crowley, *Spectral, Probabilistic, and Deep Metric Learning: Tutorial and Survey*, 2022, arXiv: 2201.09267 [stat.ML] (cit. on p. 3).

[16] J. L. Suárez-Díaz, S. García and F. Herrera, *A Tutorial on Distance Metric Learning: Mathematical Foundations, Algorithms, Experimental Analysis, Prospects and Challenges (with Appendices on Mathematical Background and Detailed Algorithms Explanation)*, 2020, arXiv: 1812.05944 [cs.LG] (cit. on p. 3).

[17] S. Caillou et al., 'Novel fully-heterogeneous GNN designs for track reconstruction at the HL-LHC', *26th International Conference on Computing in High Energy & Nuclear Physics*, EPJ Web of Conferences, 2023 09028 (cit. on p. 4).

[18] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende and K. Kavukcuoglu, *Interaction Networks for Learning about Objects, Relations and Physics*, 2016, arXiv: 1612.00222 [cs.AI] (cit. on p. 4).

[19] J. Ansel et al., 'PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation', *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2024, URL: https://pytorch.org/assets/pytorch2-2.pdf (cit. on p. 5).

[20] A. Hagberg, P. Swart and D. Chult, 'Exploring Network Structure, Dynamics, and Function Using NetworkX', 2008 (cit. on p. 5).