Optimising the Configuration of the CMS GPU Reconstruction

Abdulla Ebrahim[1,*], *Andrea* Bocci[2], *Wael* Elmedany[1], and *Hesham* Al-Ammal[1]

[1]University of Bahrain
[2]CERN

Abstract. Particle track reconstruction for high energy physics experiments like CMS is computationally demanding but can benefit from GPU acceleration if properly tuned. This work develops an autotuning framework to automatically optimise the throughput of GPU-accelerated CUDA kernels in CMSSW. The proposed system navigates the complex parameter space by generating configurations, benchmarking performance, and leveraging multi-fidelity optimisation from simplified applications. The autotuned launch parameters improved CMSSW tracking throughput over the default settings by finding optimised, GPU-specific configurations. The successful application of autotuning to CMSSW demonstrates both performance portability across diverse accelerators and the potential of the methodology to optimise other HEP codebases.

1 Introduction

Performance autotuning is an effective technique for optimising applications by automatically finding optimal configurations. Autotuning frameworks explore a search space of parameters like thread block sizes, iteration counts, etc. to maximize a performance metric like throughput or latency [1].

Autotuning has proven particularly useful for GPU computing. GPUs have complex programming models and numerous configurable parameters that affect performance. Manually tuning these parameters is infeasible for large applications. Autotuning provides a principled approach to navigate the optimisation space. It has achieved significant speed-ups for various GPU workloads including deep learning, stencil computations, graphics, and more [2].

One application that can benefit from GPU autotuning is the Compact Muon Solenoid (CMS) experiment's reconstruction software. The CMS detector at the Large Hadron Collider (LHC) produces enormous amounts of data that must be processed to reconstruct particle trajectories. To accelerate reconstruction, CMS exploits GPUs by offloading tracking and other algorithms to GPUs [3].

In this work, we develop an autotuning framework for the CMS High-Level Trigger (HLT) reconstruction software containing multiple GPU kernels. Our goal is to demonstrate significant gains in end-to-end throughput compared to default configurations via rigorous autotuning. The autotuned software can then be deployed to improve experiment efficiency and physics analysis.

*e-mail: asubah@uob.edu.bh

2 Related Work

Efforts to optimise the configuration of computer programs have been the focus of many research studies, as they play a crucial role in speeding up high computational tasks. For example, Ansel et al. [4] developed OpenTuner, an extensible framework for program autotuning, which aims to automate the process of finding the best set of parameters for a given program to optimise its performance.

Multi-fidelity optimisation reduces the cost of evaluating expensive models by first utilizing cheaper low-fidelity models. For example, Lindauer et al. [5] used multi-fidelity to tune hyperparameters in neural network training. We employ similar concepts to accelerate CMSSW autotuning using a simplified GPU track reconstruction framework in place of the low-fidelity model.

In this paper, we explore the application of performance autotuning techniques to optimise the performance of CMSSW GPU kernels for track reconstruction. Our aim is to demonstrate that autotuning can lead to significant improvements in throughput performance, thereby enhancing the scientific productivity of the CMS experiment.

3 Methodology

In this section, we describe the experimental methodology for developing and evaluating our proposed autotuning framework [6]. First, we provide an overview of the autotuning workflow and key components of our approach. Next, we discuss how the parameter search space was defined based on the configurable dimensions of the CMSSW GPU kernels. We then detail the multi-fidelity optimisation strategy leveraged to accelerate the tuning. Finally, we outline the experimental setup including the CMSSW software versions, GPU hardware, and other software configurations used for evaluation. The methodology provides a comprehensive view of our techniques, to enable reproducibility.

3.1 Autotuning Framework

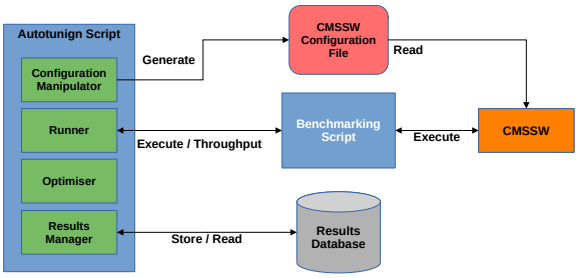

Figure 1. An illustration of the autotuning framework used in this work

Our autotuning framework, as seen in Figure 1, is based on OpenTuner [4], and it comprises four main components: configuration manipulator, runner, optimiser, and results manager. The **configuration manipulator** is responsible for generating CMSSW configuration

Table 1. The types of the tuned parameters and their count.

Type of Parameters	Count
Blocks	6
Threads	23
Strides	3
Total	**32**

files populated with different parameters from the search space. It systematically varies parameters like thread block dimensions and kernel optimisation flags based on the search algorithm's guidance.

The **runner** module executes CMSSW using the generated configurations and measures the resulting performance. To obtain statistically significant results, we utilize a wrapper script around CMSSW that performs multiple benchmark repetitions per configuration [7]. The wrapper discards the first few repetitions to account for caching effects before reporting the average throughput across the remaining repetitions.

The **optimiser** navigates the complex search space to maximize CMSSW throughput using the performance data collected by the runner. It employs an AUCBandit algorithm introduced by the authors of OpenTuner [4]. This algorithm uses multiple search heuristics such as evolutionary algorithms, and monitor their performance. After a number of trials, the algorithm prefers the search heuristics that produces the best performing parameters.

Finally, the **results manager** stores and retrieves all benchmarking data in a database to inform the optimiser. It also tracks the tuning progress to support analysis of the optimisation trajectories taken by the autotuner. The modular design and integration of these components enables efficient autotuning of the CMSSW reconstruction pipelines.

3.2 Search Space

Defining an appropriate search space by identifying the impactful, independent parameters is crucial for effective autotuning. The key parameters we tune include CUDA kernel launch configuration parameters like the number of thread blocks, threads per block, and strides. The number of thread blocks can depend on data size and threads per block, but we focused on six independent block dimensions that can be freely tuned. The threads per block parameter is the most common and is not dependent on other factors. Finally, some algorithms allow the stride parameter to control the data points processed per thread to be tuned freely. Table 1 summarises the parameters space.

3.3 Multi-Fidelity optimisation

To reduce the computational overhead of autotuning, we employ a multi-fidelity optimisation strategy. Instead of tuning the full CMSSW reconstruction directly, we first utilize a minimal GPU tracking implementation called pixeltrack-standalone [8] as a low-fidelity proxy. Pixeltrack-Standalone provides faster feedback during tuning since it only has simplified CMSSW framework logic, but it includes the full GPU kernels. We tune pixeltrack-standalone extensively to find high-performing configurations. These top configurations identified by the low-fidelity framework are then evaluated in CMSSW as the high-fidelity application. This approach allows us to avoid exhaustively tuning CMSSW end-to-end. As shown in Figure 2, the multi-fidelity methodology with pixeltrack as the low-fidelity model provides significant time savings over naively tuning CMSSW directly. The multi-fidelity optimisation enabled efficient search in the vast parameter space.

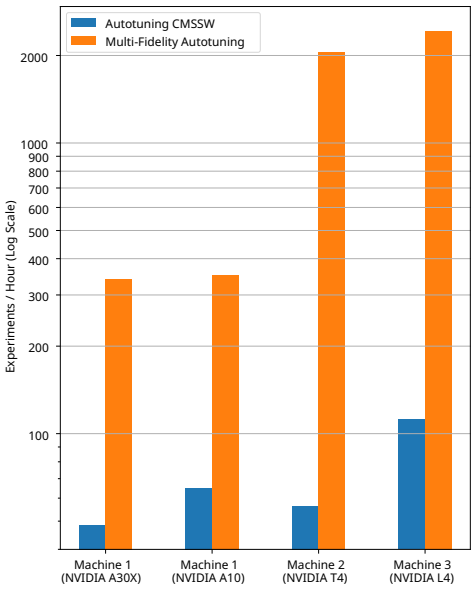

Figure 2. The plot shows the difference in time between directly autotuning CMSSW and Multi-Fidelity Autotuning

Table 2. The configurations used during autotuning and benchmarking the CMSSW.

Machine ID	1	2	3
CMSSW Release	13_0_0	13_0_0	13_0_0
Number of Jobs	1	4	4
Number of CPU Threads	12	32	32
Number of CMSSW Streams	12	24	24
Number of Events	10000	10000	10000

3.4 Benchmarking Methodology and Experimental Setup

For benchmarking, we performed a total of 8 autotuning runs, with 2 runs of 6 hours duration per GPU. This included one run using direct CMSSW autotuning and another run using the multi-fidelity approach per GPU. After each run, the top performing configurations suggested by the autotuner were then benchmarked independently. Each configuration was executed 10 times, with the first 3 results discarded to account for caching effects. The throughput was averaged over the remaining 7 benchmark repetitions to obtain statistically significant performance measurements. This rigorous benchmarking methodology ensured we accurately quantified the improvements obtained from the best autotuned configurations on each GPU. Table 2 summarises our experimental setup, and table 3 shows part of the specifications of the used GPUs.

Table 3. The specifications of the GPUs used in this experiment.

GPU	T4	A30X	A10	L4
Streaming Multiprocessor Count	40	56	72	60
L1 Cache / SM	64KB	192KB	128KB	128KB
L2 Cache	4MB	24MB	6MB	48MB
DRAM	16GB	24GB	24GB	24GB

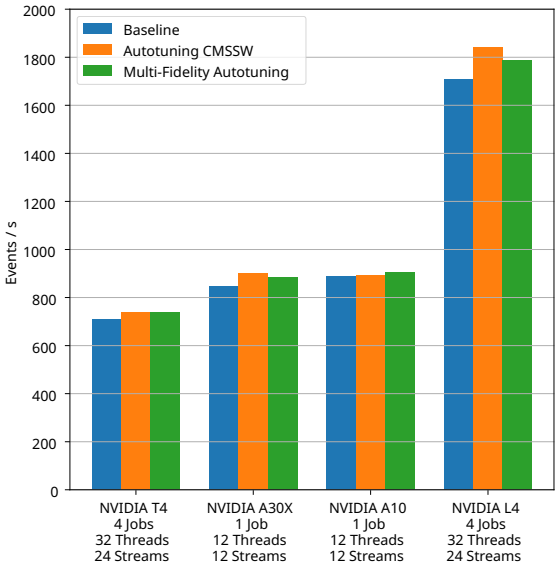

Figure 3. A comparison between the throughput of the baseline parameters, the parameters found after autotuning CMSSW directly, and the parameters found from autotuning the pixeltrack-standalone program.

4 Results and Discussion

Figure 3 shows that the proposed autotuning framework improved the throughput of CMSSW GPU tracking compared to the default parameters. The increase in performance was about 4%, 6%, 2%, and 7% on the T4, A30X, A10, and L4 respectively. In the A10 case, multi-fidelity tuning yielded the best parameters, while direct CMSSW tuning was more effective for the A30X and L4 cases. In the T4 case, both methods produced almost equivalent results. However, multi-fidelity tuning can evaluate many more configurations in the same time frame (refer to Figure 2), making it a versatile tool for initial broad parameter exploration.

A promising approach could be a hybrid method, starting with low-fidelity tuning to explore promising parameter regions, then switching to direct CMSSW tuning for precise measurement and exploitation. This method combines the strengths of both approaches, potentially enhancing tuning efficiency and effectiveness.

Analysing the best parameter configurations found by autotuning (Figure 4) reveals significant variation across the GPUs. Each GPU has different optimal thread block dimensions,

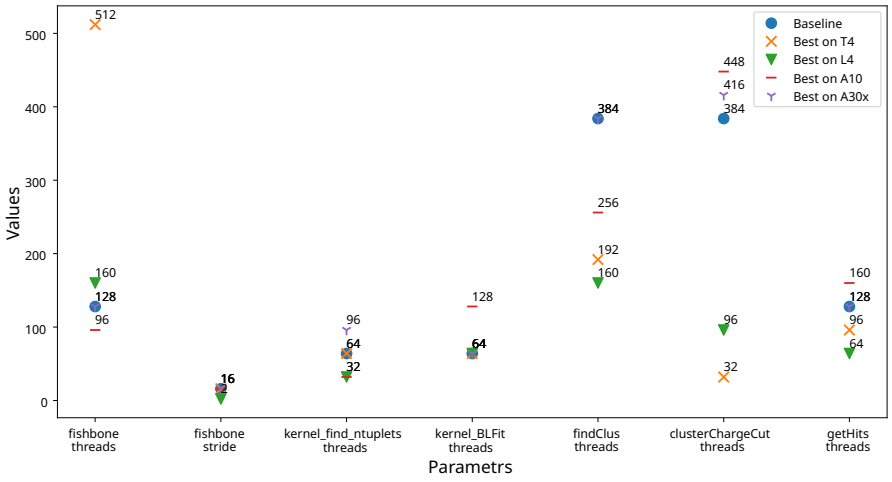

Figure 4. The best parameters found for each GPU.

threads per block, and other kernel launch parameters. This highlights the importance of autotuning, as the best configurations are hardware dependent.

To better understand the performance implications of autotuning, we profiled both the baseline and tuned configurations across all the GPUs. We noticed that kernels requiring a large number of registers tend to increase queuing time for other kernels before execution. This is possibly due to the limited number of registers available per block, suggesting that assigning fewer threads to such kernels could help reduce these execution delays. That is why in Figure 4, all the GPUs benefited from reducing the number of threads in the *findClus* kernel except the A30X which has the largest L1 cache (refer to Table 3).

However, it is important to note that not all kernels are heavily dependent on registers. Some kernels require fewer registers and, as a result, do not show the same queuing effect. This highlights the inherent complexity in finding the optimal configurations, as these are not always obvious due to the hardware-specific differences. Therefore, the value of autotuning becomes evident, as it allows for empirical determination of the best launch parameters for each kernel on every GPU architecture. Autotuning aids in optimising the complex balance between all the variables to enhance the overall performance.

5 Conclusion

In this work, we developed an autotuning framework for optimising CMSSW GPU tracking performance by automatically finding improved kernel launch parameters compared to the defaults. The multi-fidelity optimisation strategy used attempts to navigate the complex configuration space to increase throughput. This initial investigation of autotuning for CMSSW shows its potential to help exploit diverse accelerator hardware. The techniques may be applicable to other HEP codebases. While more extensive benchmarking is required, this work represents early progress in enabling automated optimisation to increase the productivity of GPU-accelerated reconstruction. With further development, autotuning could provide a useful approach to help harness accelerated systems for HEP computing needs.

Acknowledgments

The experiments presented in this paper were carried out using the facilities of the Benefit Advanced AI and Computing Lab at the University of Bahrain (see https://ailab.uob.edu.bh) with support from Benefit Bahrain Company (see https://benefit.bh)

References

[1] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. ACM Computing Surveys **51**, 1–42 (2019).

[2] Ben van Werkhoven, Future Generation Computer Systems, **90**, 347-358 (2019)

[3] Andrea Bocci, David Dagenhart, Vincenzo Innocente, Christopher Jones, Matti Kortelainen, Felice Pantaleo, Marco Rovere EPJ Web Conf. 245 05009 (2020)

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT). 303–316 (2014)

[5] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. The Journal of Machine Learning **23**, 2475–2483 (2022)

[6] CMSSW Autotuning Framework https://github.com/asubah/cmssw-autotuning/

[7] Patatrack Benchmarking Scripts https://github.com/cms-patatrack/patatrack-scripts

[8] Pixeltrack-Standalone https://github.com/asubah/pixeltrack-standalone/tree/autotuning