UNIVERSITY OF BAHRAIN

College of Information Technology

Department of Computer Science

Performance Auto-tuning Framework for GPU Applications

A Thesis Submitted in Partial Fulfilment of the Requirements for the Doctor of Philosophy Degree in Computing and Information Sciences

Submitted By

Abdulla Ebrahim Ali Subah

20103079

Supervised By

Dr. Wael Elmedany

(Associate Professor)

Co-Superviser

Dr. Hesham Al-Ammal

(Assistant Professor)

University of Bahrain

October 2024

APPROVAL

University of Bahrain
Deanship of Graduate Studies
And scientific Research

جامعة البحرين
عمادة الدراسات العليا والبحث العلمي

Dissertation Deposit Form

Author Name:	Abdulla Ebrahim Ali Subah		
Email:	asubah@uob.edu.bh	Contact No:	36334258
College:	**Information Technology**		
Department:	**Computer Science**		
Thesis Title:	Performance Auto-tuning Framework for GPU Applications		
Year:	2024		

Author's Declaration: I agree to the following conditions:

The above thesis will be available (in the University of Bahrain library and externally) and reproduced as necessary at the discretion of the University of Bahrain. It may also be digitized by the University of Bahrain and made available via the university's repository or via other parties on the Internet.

This condition shall apply to ALL copies including electronic copies.

The above thesis has been provided on the understanding that it is copyright protected material and that no quotation from this thesis may be published without proper acknowledgement.

Signature of the Author:		Date:	05-Oct-24

Approved at the University Council 6th meeting, 2013
In accordance with Decision No. 736/2013 dated 14/3/2013
Modified by University Council Decision No. 2043/2013 dated 20/11/2013.

DECLARATION

I declare that this work is the result of my own investigation an
that it has not already been accepted in substance for any degree,
nor is it currently submitted for any degree.

Signed: ---------- Date: 9/12/2024
Abdulla Ebrahim Ali Subah (Candidate)

The Thesis defense committee considers the thesis titled:

Performance Auto-tuning Framework for GPU Applications

Submitted by Abdulla Ebrahim Ali Subah to the College of Information Technology is satisfactory and acceptable for the Doctor Philosophy in Computation and Information Sciences.

The Defense Committee:

Dr.Wael Elmedany — Department of Computer Science
(Supervisor) — University of Bahrain

Dr. Hesham Alamal — Department of Computer Science
(co-supervisor) — University of Bahrain

Prof. Nikolas Bessis — Edge Hill University
(Member) — UK

Dr. Riadh Ksantini — Department of Computer Science

ABSTRACT

Optimizing GPU applications for performance and portability across diverse architectures is challenging due to the complexity of GPU programming and hardware diversity. This thesis presents a performance auto-tuning framework for GPU applications to address these challenges.

Efficient search techniques were developed to reduce the search space and computational overhead in auto-tuning GPU kernels. OpenTuner was extended to support GPU kernel autotuning, incorporating advanced search algorithms like basin hopping and Bayesian optimization. A machine learning-based search space reduction method using boosted trees predicted promising kernel parameter configurations. Multi-fidelity optimization balanced exploration and exploitation by evaluating configurations at different fidelity levels.

An autotuning interface was developed and loosely integrated with the CMS Software (CMSSW) used in high-energy physics experiments. This loose coupling allows the autotuner to work with other software packages and enables other autotuners to optimize CMSSW, enhancing flexibility and portability. The framework was evaluated using real-world GPU kernels from CMSSW across different GPU architectures. A benchmarking methodology proposed by other researchers was applied to compare different search techniques, providing practical insights into autotuner benchmarking methodologies. The optimized kernels outperformed default configurations, improving execution speed and resource utilization. The framework effectively reduced the search space and computational overhead, meeting the objectives of enhancing performance and portability.

Limitations include limited hardware diversity, focus on specific machine learning models, and emphasis on single-objective tuning without considering other factors like power efficiency. Future work involves expanding to other hardware architectures, experimenting with advanced machine learning and reinforcement learning techniques, applying the autotuner to different software packages, developing dynamic tuning mechanisms, and enhancing user interfaces.

Contents

APPROVAL **A**

DECLARATION **B**

ABSTRACT **C**

TABLE OF CONTENTS **D**

LIST OF FIGURES **H**

LIST OF TABLES **L**

ACKNOWLEDGMENTS **N**

DEDICATION **O**

PUBLICATIONS **P**

LIST OF ABBREVIATIONS **Q**

1 Introduction **1**

1.1 Background and Motivation 2

1.2 Problem Statement 3

1.3 Research Objectives 5

1.4 Thesis Contributions 6

1.5 Thesis Structure 7

2 Background **9**

2.1 GPU Computing 9

2.1.1 GPU Architecture 10

2.1.2 GPU Programming Platforms 13

2.2 Performance Optimization for GPUs 17

2.2.1 GPUs Common Optimization Techniques 17

2.2.2 Performance Portability 21

2.3 Autotuning 23

2.3.1 Concept and Importance 24

2.3.2 Autotuning for GPUs 26

3 Literature Review **30**

3.1 Autotuning Frameworks 30

3.2 Autotuning Benchmarking Methodologies 34

3.3 Optimization Techniques Used in Autotuning 37

3.4 Summary and Research Gaps 41

4 Methodology **42**

4.1 Overview of Methodology Components 42

4.2 Research Design 43

4.3 CMSSW Framework 44

4.3.1 Overview of CMSSW 45

4.3.2 Relevance to GPU Kernel Autotuning 46

4.3.3 Integration of Autotuning in CMSSW 47

4.4 Data Collection 48

4.4.1 GPU Kernels 48

4.4.2 Parameter Space 49

4.4.3 Performance Metrics 50

4.4.4 Data Collection Methodology 51
4.5 Evaluation Criteria for Autotuners 52
4.6 The Proposed Framework 54
4.6.1 Introduction to the Autotuning Framework 54
4.6.2 Framework Architecture 55
4.6.3 Parameter Space Definition 57
4.6.4 Search Techniques 60
4.7 Experimental Setup 62
4.7.1 Hardware 62
4.7.2 Software 64

5 Results and Discussion 65
5.1 Baseline Performance Analysis 66
5.1.1 Maximum Throughput 66
5.1.2 Distribution of Configurations 68
5.1.3 Autotuning Convergence 73
5.1.4 Configurations Validity 77
5.2 Seed Configurations 80
5.2.1 Maximum Throughput 81
5.2.2 Distribution of Configurations 82
5.2.3 Autotuning Convergence 84
5.2.4 Configurations Validity 85
5.3 Reducing Search Space using Boosted Trees 87
5.3.1 XGBoost for Feature Selection 89
5.3.2 Maximum Throughput 96
5.3.3 Distribution of Configurations 97
5.3.4 Autotuning Convergence 98
5.3.5 Configurations Validity 98
5.4 Multi-fidelity Autotuning 103
5.4.1 Events Count Reduction 104

5.4.2 Pixeltrack Standalone Framework 104
5.4.3 Integration with Full Framework Autotuning 106
5.5 Performance Portability 106
5.5.1 Performance Portability Analysis 106

6 Conclusion and Future Work 109
6.1 Achieving the Research Objectives 109
6.2 Implications of the Research 112
6.3 Limitations 114
6.4 Future Work 115
6.5 Concluding Remarks 116

References 117

Abstract (in Arabic)

List of Figures

2.1 A comparison between the architectures of a GPU and a CPU. 11
2.2 GPU memory hierarchy. 12
2.3 An illustration of how the memory is accessed in the threads using cache lines and memory banks. 18
2.4 An illustration of the difference between interleaved and coalesced memory access patterns. 18
2.5 Diverged code flow chart vs execution flow chart. 19
2.6 Timeline comparison between data copy and kernel execution 20

4.1 A diagram of the autotuning framework used in this study. . 55

5.1 Distribution of configuration throughputs for different autotuning techniques across GPU architectures. The x-axis represent the frequency and y-axis is the throughput. The vertical black dashed line indicate the baseline performance for that GPU. The red dashed line is the distribution mean. . 69
5.2 Convergence patterns of different autotuning techniques across GPU architectures. The x-axis represent the throughput and y-axis is the time. The horizontal dashed line indicate the baseline performance for that GPU. 74
5.3 Percentage of valid and invalid configurations generated by different search techniques for the T4 GPU. 78

5.4 Percentage of valid and invalid configurations generated by different search techniques for the A10 GPU. 79
5.5 Percentage of valid and invalid configurations generated by different search techniques for the L4 GPU. 79
5.6 Percentage of valid and invalid configurations generated by different search techniques for the L40S GPU. 80
5.7 Distribution of throughput values achieved by different search techniques across GPU architectures when seeded with the T4-optimized baseline configuration. The black dashed line indicates the baseline performance, while the red dashed line shows the mean throughput for each distribution. 83
5.8 Convergence behaviour of different search techniques across GPU architectures when initialized with a seed configuration. Each plot shows the evolution of throughput over iterations, with the black dashed line indicating the baseline performance. 86
5.9 Percentage of valid and invalid configurations generated by different search techniques on T4 when initialized with a seed onfiguration. 87
5.10 Percentage of valid and invalid configurations generated by different search techniques on A10 when initialized with a seed configuration. 88
5.11 Percentage of valid and invalid configurations generated by different search techniques on L4 when initialized with a seed configuration. 88
5.12 Percentage of valid and invalid configurations generated by different search techniques on L40S when initialized with a seed configuration. 89

5.13 Feature importance ranking for T4 GPU based on XGBoost weight metric. 92
5.14 Feature importance ranking for A10 GPU based on XGBoost weight metric. 93
5.15 Feature importance ranking for L4 GPU based on XGBoost weight metric. 94
5.16 Feature importance ranking for L40S GPU based on XGBoost weight metric. 95
5.17 Distribution of throughput values achieved by different search techniques using XGBoost-guided search across GPU architectures. The black dashed line indicates the baseline performance, while the red dashed line shows the mean throughput for each distribution. 99
5.18 Convergence plots of different search techniques using XGBoost-guided search across GPU architectures. Each plot shows the evolution of throughput over iterations, with the black dashed line indicating the baseline performance. 100
5.19 Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on T4 GPU. 101
5.20 Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on A10 GPU. 102
5.21 Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on L4 GPU. 102
5.22 Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on L40S GPU. 103

5.23 Configuration evaluation rates across different GPUs using original CMSSW (10,000 events), reduced events (1,000), and pixeltrack standalone framework approaches. 105
5.24 Performance portability matrix showing application efficiency when running different GPUs (rows) using configurations optimized for specific architectures (columns). Diagonal values represent the maximum achievable performance for each GPU using its respective optimized configuration. . . . 107

List of Tables

2.1 Summary of Background Topics 29

3.1 Comparison of Autotuning Frameworks and Their Features . 33
3.2 Comparison of Heterogeneous Computing Benchmark Suites 37
3.3 Summary of Autotuning Search Techniques 40

4.1 CPU Specifications 63
4.2 GPU Specifications 63
4.3 Software Environment 64

5.1 Maximum throughput (events/second) achieved by different auto-tuning techniques across GPU models 67
5.2 Maximum throughput (events/second) achieved by different search techniques with seeded initialization across GPU architectures. The seeded configuration was manually optimized for T4. Bold values indicate the best performance for each GPU. 81
5.3 XGBoost Model Configuration and Dataset Details 96
5.4 Maximum throughput (events/second) achieved by different search techniques with reduced parameter space using XGBoost across GPU architectures. Bold values indicate the best performance for each GPU. 97
5.5 Maximum throughput (events/second) achieved using different fidelity levels 104

6.1 Mapping of Research Objectives to Thesis Contributions . . 111

ACKNOWLEDGMENTS

To my thesis supervisors Dr. Hesham Al-Ammal Assistant Professor and Dr. Wael Elmedany Associate Professor at University of Bahrain, I would like to express my gratitude for their guidance, support, and patience.

I would also like to express my gratitude to my internship supervisor at CERN / CMS Dr. Andrea Bocci, the Applied Physicist at CERN for his guidance, support, and patience.

This work was partially sponsored by Shaikh Ebrahim bin Mohammed Al Khalifa Center for Culture and Research (SEARCH), my deepest gratitude to Shaikh Mai bint Mohammed Al Khalifa the president of the board of trustees for her usual generous support.

Thanks to all the staff of the Department of Computer Science and the Information Technology College in the University of Bahrain.

Thanks to all the Patatrack research group members, and CMS collaboration members for their support.

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`.

Abdulla Ebrahim Subah

DEDICATION

To my family, for their unconditional love and support.

To my teachers, for every piece of knowledge they taught me.

PUBLICATIONS

Ebrahim, Abdulla, Bocci, Andrea, Elmedany, Wael, & Al-Ammal, Hesham. (2024). Optimising the configuration of the CMS GPU reconstruction. EPJ Web of Conf., 295, 11015. doi: 10.1051/epjconf/202429511015

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
AUC	Area Under Curve
BAT	Benchmark suite for AutoTuners
BO	Bayesian optimization
CLTune	OpenCL kernel autotuning tool
CMS	Compact Muon Solenoid
CMSSW	CMS Software
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithms
GDDR	Graphics Double Data Rate
GPU	Graphics Processing Unit
GPTune	Autotuning tool incorporating transfer learning
HBM	High Bandwidth Memory
HLT	High Level Trigger
HPC	High-Performance Computing
I/O	Input/Output

LHC Large Hadron Collider

ML Machine Learning

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PSO Particle Swarm Optimization

ROCm Radeon Open Compute

SHOC Scalable Heterogeneous Computing benchmark suite

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor

SMAC3 Sequential Model-based Algorithm Configuration, version 3

SYCL Single-source Heterogeneous Programming for OpenCL

TDP Thermal Design Power

Chapter 1

Introduction

In today's world, a huge amount of data is being generated, and problems are becoming more complex. This has increased the need for powerful computing solutions. Graphics Processing Units (GPUs), which were originally created to handle graphics tasks, have become important in high-performance computing. They are used in areas like scientific research, machine learning, and data analysis because many calculations can be processed in parallel.

However, getting the best performance from GPUs is not easy. Various settings like memory use and task division need to be adjusted by developers. The diversity of GPUs and frequent updates add to the challenge. Issues like managing threads, optimizing memory access, and balancing workloads across the GPU are often faced by developers.

Performance auto-tuning can help with these problems. Different settings are automatically tried by auto-tuning tools, and the best ones are chosen based on a defined performance criteria. This allows developers to focus more on improving their algorithms instead of spending time on low-level optimizations.

This thesis aims to make the optimization process for GPU applications simpler and better. Development time can be saved, performance can be improved, and applications can work well on different GPUs with an effective

auto-tuning framework.

1.1 Background and Motivation

The Compact Muon Solenoid Software (**CMSSW**) framework is a large, modular software system developed for the CMS experiment at the Large Hadron Collider (LHC) (CMS Collaboration, 2024). It has been ported to use GPUs to take advantage of their parallel processing capabilities and to meet the increasing computational demands of high-energy physics research (CMS Collaboration, 2021).

A unique aspect of CMSSW is that it runs multiple GPU kernels in multiple processes simultaneously. This means that several kernels may be executing at the same time on the same GPU, originating from different parts of the application or even different workflows. This concurrent execution introduces additional complexity to the performance optimization process. Resource contention and interference between kernels can significantly impact overall performance. Optimizing each kernel in isolation may not lead to optimal performance when they are run together.

Moreover, CMSSW operates on different high-performance computing (HPC) sites with various hardware configurations. These sites may have different types of GPUs from different vendors or generations, which adds complexity to performance optimization. The framework's modular nature allows many developers to contribute independently to different parts of the system. Each module may have its own set of GPU kernels, further increasing the diversity of workloads running concurrently.

Manual optimization of GPU code in such a modular and concurrent environment requires significant effort and expertise (van Werkhoven, 2019). Developers need to adjust various parameters, such as thread block sizes and memory usage patterns, to suit specific hardware and to account for the effects of multiple kernels running at the same time. This process is

time-consuming and may not be feasible for all modules or developers.

Having a common auto-tuning interface and auto-tuner within CMSSW would help address these challenges. An auto-tuning framework can automatically explore different optimization parameters and select the best configurations for each GPU kernel, considering the concurrent execution and resource sharing on the available hardware. This would:

- **Improve Performance**: By finding optimal settings for each kernel in the context of concurrent execution, better overall performance can be achieved without manual tuning.
- **Enhance Performance Portability**: Applications can run efficiently on different GPUs at various HPC sites, adapting automatically to the hardware and the multiprocess environment.
- **Reduce Development Effort**: Developers can focus on the functionality of their modules, relying on the auto-tuner to handle performance optimization in a complex, multi-kernel environment.

The motivation for this thesis stems from the need to simplify and enhance the performance optimization process for GPU-accelerated applications within CMSSW, especially given its unique characteristics of running multiple kernels in multiple processes at the same time. By integrating an effective auto-tuning framework into CMSSW, the overall efficiency of the software can be improved, supporting the goals of high-energy physics research.

1.2 Problem Statement

Despite advances in GPU technologies and programming models, optimizing GPU applications for top performance remains difficult (Ryoo et al., 2008). Challenges include:

- **Complex Optimization Spaces**: A large and multidimensional parameter space for GPU optimization exists, including factors like thread block sizes, grid dimensions, memory access patterns, and algorithm-specific parameters (van Werkhoven, 2019).

- **Architectural Diversity**: Varying architectural features are found in GPUs from different vendors or even different generations from the same vendor, requiring specific optimizations for each platform (Reyes & de Sande, 2012).

- **Manual Effort and Expertise**: A lot of manual effort and deep expertise in GPU architectures are demanded by traditional optimization techniques, which may not be feasible for all developers or scalable for all applications (Ashouri, Killian, Cavazos, Palermo, & Silvano, 2018).

- **Lack of Integration with Existing Frameworks**: Many auto-tuning solutions are standalone tools that are not easily integrated into existing application frameworks, limiting practical use (Ansel et al., 2014).

The problem addressed in this thesis is the development of a performance auto-tuning framework specifically designed for GPU applications that can:

1. **Efficiently Explore the Optimization Space**: Strategies to navigate the large parameter space effectively without exhaustive search are implemented.

2. **Adapt to Different GPU Architectures**: Performance portability is provided by automatically tuning applications for different hardware configurations.

3. **Integrate with Existing Software Frameworks**: Auto-tuning capabilities are seamlessly incorporated into existing application frameworks, such as the CMSSW framework used in high-energy physics.

1.3 Research Objectives

The main goal of this research is to design and implement a performance auto-tuning framework that tackles the challenges mentioned. Specific objectives include:

- **Develop Efficient Search Techniques**: Search algorithms are implemented to reduce the search space and computational overhead of auto-tuning.
- **Integrate the Auto-tuning Framework with CMSSW**: The practical use of the framework is demonstrated by integrating it into the CMSSW framework, optimizing real-world GPU kernels used in high-energy physics experiments.
- **Evaluate the Framework**: The effectiveness of the auto-tuning framework in improving performance and ensuring portability across different GPU architectures is assessed.
- **Provide Insights into Auto-tuners Benchmarking Methodologies**: Insights are contributed to the broader knowledge by analysing the results and identifying best practices and potential areas for future research.

1.4 Thesis Contributions

This thesis presents five primary contributions to the field of GPU kernel optimization:

1. **Multi-Process GPU Kernel Autotuning Framework**: To the authors' knowledge, the optimization of multiple GPU kernels running concurrently across multiple processes has not been previously addressed. OpenTuner was extended to support this capability, addressing a gap in GPU optimization research and enabling practical optimization in real-world scenarios like CMSSW where multiple kernels are executed simultaneously.

2. **Enhanced Search Space Optimization Techniques**: Multiple optimization strategies were implemented and integrated into OpenTuner including:

   - Machine learning approach using boosted trees for configuration prediction
   - Multi-fidelity optimization technique for balanced resource usage
   - Basin Hopping and Bayesian Optimization for improved global optima detection

   These additions improve the efficiency and effectiveness of the autotuning process across different scenarios.

3. **CMSSW GPU Optimization Framework**: An autotuning framework was designed and implemented within CMSSW to optimize GPU kernels across different architectures. This integration provides a practical solution for automated GPU kernel optimization in high-energy physics applications.

4. **Reproducible Benchmarking Implementation**: Established benchmarking methodologies from literature were implemented to evaluate autotuning performance. The implementation includes multiple measurement repetitions, controlled testing environments, and detailed documentation of experimental conditions to ensure data accuracy and reproducible results.

5. **Cross-Architecture Performance Analysis**: The autotuning framework's performance was analysed and evaluated across multiple GPU architectures, providing insights into achieving performance portability in heterogeneous computing environments.

These contributions address the challenges of optimizing GPU applications within a complex, modular, and concurrent environment like CMSSW. By extending existing tools and integrating new techniques, the thesis provides practical solutions that can benefit the high-energy physics community and other fields that rely on GPU-accelerated applications.

1.5 Thesis Structure

The thesis is organized as follows:

- **Chapter 2: Background**

  Foundational knowledge on GPU computing is provided, including GPU architectures and programming platforms. Common performance optimization techniques for GPUs are discussed, and the concept and importance of auto-tuning, particularly for GPUs, are introduced.

- **Chapter 3: Literature Review**

  A review of existing auto-tuning frameworks and methodologies is presented. Various optimization techniques used in auto-tuning are

examined, and gaps in current research that this thesis aims to address are identified.

- **Chapter 4: Methodology**

  The research design and methodology adopted in the thesis are outlined. An overview of the CMSSW framework is provided, data collection methods are discussed, evaluation criteria for auto-tuners are defined, and the proposed auto-tuning framework's architecture and search techniques are introduced.

- **Chapter 5: Results and Discussion**

  Results of applying the auto-tuning framework are presented, including baseline performance analysis and the effects of seeded configurations. The impact of reducing the search space using machine learning techniques is explored, multi-fidelity auto-tuning is discussed, performance portability is evaluated, and limitations and future work are considered.

- **Chapter 6: Conclusion and Future Work**

  The thesis concludes with a summary of the contributions made, a discussion of the study's limitations, and potential directions for future research are outlined.

- **References**

  A list of all the scholarly works cited throughout the thesis.

Through this structure, a comprehensive examination of performance auto-tuning for GPU applications is provided, contributing valuable insights and practical solutions to the high-performance computing community.

Chapter 2

Background

This background chapter aims to provide the foundational knowledge necessary to understand GPU computing, the need for performance optimization, and the potential of auto-tuning in addressing these challenges. By exploring these topics, the chapter sets the stage for the development of an auto-tuning framework that can effectively optimize GPU applications, particularly in the context of HPC.

2.1 GPU Computing

GPU computing leverages the highly parallel architecture of graphics processors to accelerate a wide range of computationally intensive tasks. Originally designed to handle the simultaneous calculations required for rendering complex graphics, GPUs have found applications far beyond their original purpose (Luebke et al., 2004). Their ability to perform many calculations in parallel makes them particularly well-suited for problems that can be broken down into independent, concurrent operations (Buck et al.,

2004).

GPUs are used in diverse domains such as machine learning, scientific simulations, cryptography, and big data analytics. In machine learning, GPUs accelerate the training of deep neural networks (Steinkraus, Buck, & Simard, 2005), enabling breakthroughs in areas like computer vision (Fung & Mann, 2005). Scientific simulations, from molecular dynamics (Anderson, Lorenz, & Travesset, 2008) to climate modelling (Demeshko, Maruyama, Tomita, & Matsuoka, 2013), benefit from the increased computational power of GPUs. In the field of high energy physics, GPUs are increasingly used to process the vast amounts of data generated by particle accelerators and detectors (Bocci, Innocente, Kortelainen, Pantaleo, & Rovere, 2020).

However, harnessing the full potential of GPU computing presents unique challenges (Ryoo et al., 2008). Effective utilization of GPU resources requires careful consideration of hardware architecture, memory management, and parallelization strategies. These factors significantly influence application performance and efficiency. As such, optimizing GPU applications often involves complex tuning processes, which can be time-consuming and require specialized expertise.

2.1.1 GPU Architecture

GPUs architectures are significantly different from CPUs architecture. CPUs are designed for general-purpose computing with a focus on low-latency operations and complex control flow, whereas GPUs are optimized for parallel processing and high throughput. This fundamental difference is reflected in their respective architectures (see Figure 2.1), with CPUs consisting of a few complex cores and GPUs containing numerous simpler cores designed for concurrent execution of similar tasks.

The parallel nature of GPU architecture is closely tied to the concepts of

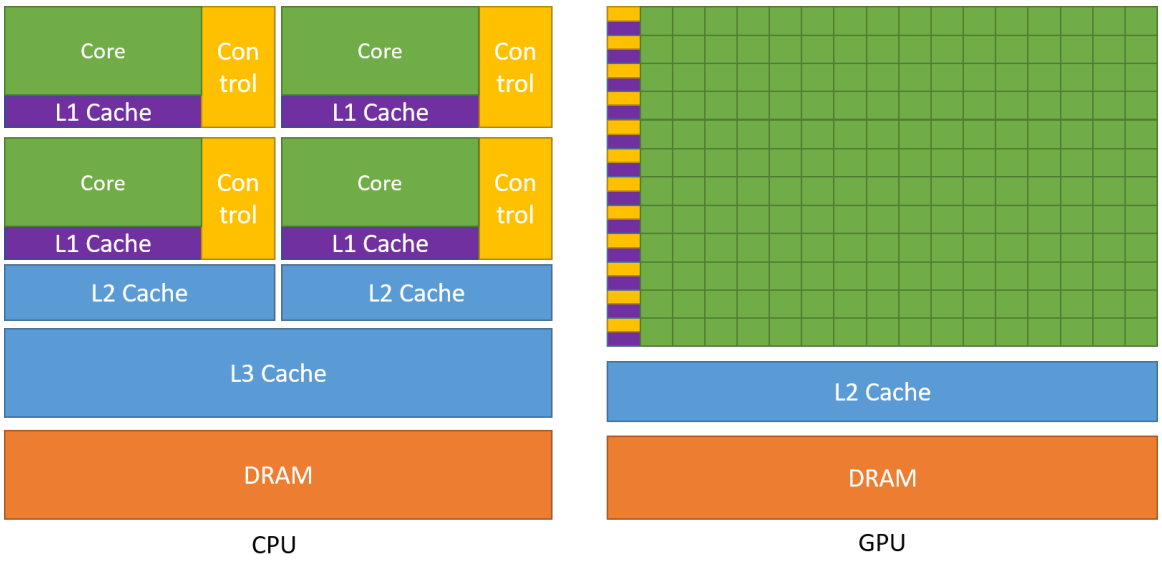

Figure 2.1: A comparison between the architectures of a GPU and a CPU (NVIDIA Corporation, 2024a).

Single Instruction, Multiple Data (SIMD) and Single Instruction, Multiple Thread (SIMT). SIMD, a paradigm originally developed for vector processors, allows a single instruction to operate on multiple data points simultaneously. GPUs extend this concept with SIMT, which enables the execution of a single instruction across multiple independent threads. This approach is particularly well-suited to graphics rendering and other highly parallelizable computations, where the same operation is often performed on large sets of data.

The fundamental building blocks for parallel computation in modern GPUs architectures are Streaming Multiprocessors (SMs). Each SM contains multiple CUDA cores (in NVIDIA GPUs) (NVIDIA Corporation, 2024a) or compute units (in AMD GPUs) (AMD, Inc., 2024), as well as special function units, load/store units, and register files. SMs are designed to execute groups of threads, known as warps or wavefronts, in a SIMT fashion. This design allows for efficient scheduling and execution of parallel workloads, as multiple warps can be active simultaneously within an SM, en-

abling latency hiding through fine-grained multithreading. The number and configuration of SMs vary across different GPU models and generations, directly impacting the overall parallel processing capability of the GPU.

Memory hierarchy in GPUs is structured to support their parallel processing capabilities (refer to Figure 2.2). Unlike CPUs, which rely heavily on large, multi-level caches to reduce memory access latency, GPUs employ a more specialized memory hierarchy. At the highest level, GPUs utilize high-bandwidth memory, such as GDDR or HBM, to feed data to their numerous processing cores. Within the GPU, a shared memory or L1 cache is typically accessible by a group of threads, allowing for efficient data sharing and reducing the need for costly global memory accesses.

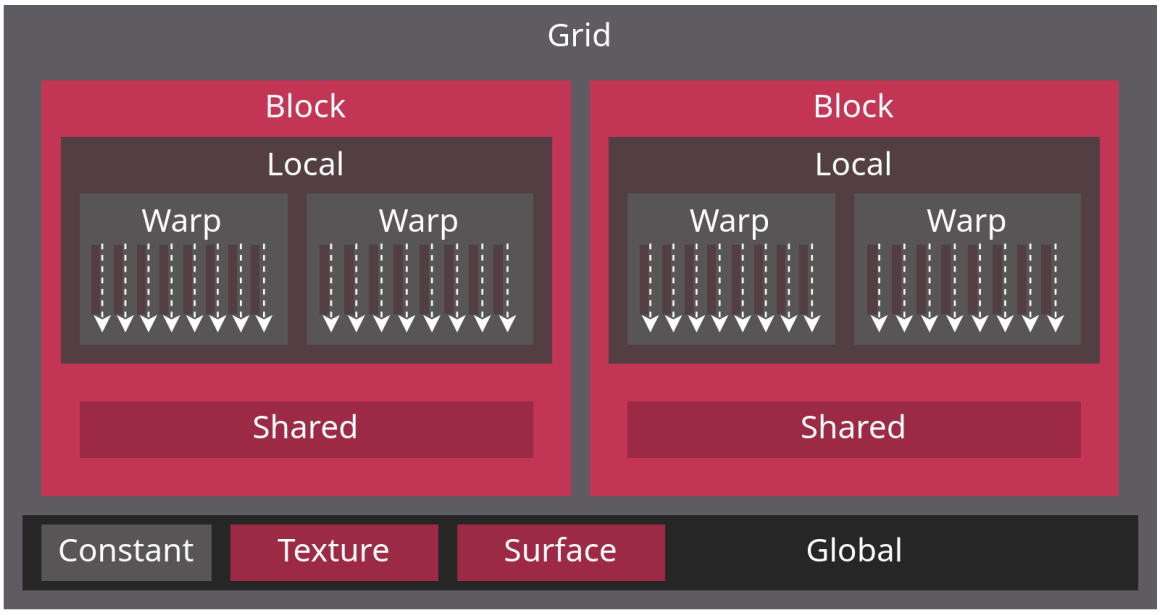

Figure 2.2: GPU memory hierarchy (AMD, Inc., 2024).

Memory hierarchy in GPUs is structured to support their parallel processing capabilities (refer to Figure 2.2). Unlike CPUs, which rely heavily on large, multi-level caches to reduce memory access latency, GPUs employ a more specialized memory hierarchy. At the highest level, GPUs utilize high-bandwidth memory, such as GDDR or HBM, to feed data to their numerous processing cores. Within the GPU, a shared memory or L1 cache is

typically accessible by a group of threads, allowing for efficient data sharing and reducing the need for costly global memory accesses.

The performance characteristics of GPUs are not uniform across all types of computations. GPUs excel in scenarios with high arithmetic intensity, where the ratio of computational operations to memory operations is high. Conversely, algorithms with complex control flow, frequent synchronization requirements, or irregular memory access patterns may not fully leverage the GPU's parallel architecture, potentially leading to suboptimal performance.

An effective utilization of GPU performance often requires careful optimization and tuning (Tillmann, Karcher, Dachsbacher, & Tichy, 2013). This process involves considerations such as thread block size selection, memory access patterns, load balancing, and minimizing data transfers between the CPU and GPU. Tools such as profilers and performance analysis frameworks play a crucial role in identifying bottlenecks and guiding optimization efforts.

The combination between algorithmic innovations and hardware advancements will likely drive further improvements in GPU performance characteristics, potentially unlocking new frontiers in computational capabilities and enabling more complex simulations, larger-scale data analysis, and more sophisticated artificial intelligence models.

2.1.2 GPU Programming Platforms

GPU programming platforms have evolved significantly over the past two decades, offering developers diverse approaches to harness the parallel processing capabilities of graphics processing units for general-purpose computing tasks. These platforms provide abstractions and frameworks that facilitate the development of high-performance applications across various hardware architectures. Among the popular GPU programming platforms,

CUDA, ROCm, OpenCL, and SYCL have emerged as key players in the field of heterogeneous computing.

CUDA (Compute Unified Device Architecture), introduced by NVIDIA in 2007 (NVIDIA Corporation, 2024a), has established itself as a widely adopted GPU programming model. It provides a proprietary platform and application programming interface (API) that enables developers to utilize NVIDIA GPUs for parallel computing tasks. CUDA extends the C programming language (see listing (2.1.1)), allowing programmers to define kernel functions that are executed concurrently across multiple threads on the GPU. The model incorporates a hierarchical memory structure, including global, shared, and local memory, which can be leveraged to optimize data access patterns and improve performance. CUDA's ecosystem encompasses a comprehensive set of libraries, tools, and development environments, facilitating the creation of efficient GPU-accelerated applications in various domains, such as scientific computing, machine learning, and computer vision.

While CUDA has gained significant traction, AMD's ROCm (Radeon Open Compute) has emerged as an open-source alternative for GPU programming. ROCm provides a platform for developing and deploying GPU-accelerated applications on AMD hardware. It offers a set of tools, libraries, and compilers that enable developers to write portable code across different GPU architectures. ROCm supports multiple programming models, including HIP (Heterogeneous-Computing Interface for Portability), which allows for the conversion of CUDA code to run on AMD GPUs with minimal modifications (see listing (2.1.1)). This approach aims to enhance code portability and reduce vendor lock-in, enabling developers to target a broader range of hardware platforms.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Listing (2.1.1): CUDA / HIP code example.

OpenCL (Open Computing Language) represents a standardized, cross-platform framework for parallel programming on heterogeneous systems. Developed by the Khronos Group, OpenCL provides a unified programming model that can target CPUs, GPUs, FPGAs, and other accelerators from various vendors. The OpenCL architecture consists of a host program that coordinates the execution of kernels on one or more compute devices (see listing (2.1.2)). It employs a hierarchical memory model similar to CUDA, with global, local, and private memory spaces. OpenCL's platform-agnostic nature allows developers to write portable code that can run on diverse hardware configurations, making it an attractive option for applications requiring broad compatibility.

```
__kernel void VecAdd(__global float* A,
                     __global float* B,
                     __global float* C)
{
    int i = get_global_id(0);
    C[i] = A[i] + B[i];
}
```

Listing (2.1.2): OpenCL example.

SYCL (pronounced "sickle") is an open standard for heterogeneous computing developed by the Khronos Group (Khronos, 2020). It provides a higher-level programming model that builds upon the concepts of OpenCL while offering a more modern, C++ based approach. SYCL aims to simplify GPU programming by providing a single-source, cross-platform abstraction layer. The standard allows developers to write code that can be executed on CPUs, GPUs, and other accelerators using standard C++ syntax and semantics. SYCL employs a host-device execution model, where a single source file contains both host and device code (see listing (2.1.3)). The model leverages C++ templates and lambda functions to define kernels, reducing the complexity of managing separate host and device code bases.

```
int main() {
    ...
    // Create a queue to work on default device
    queue q;

    // Submit a command group to the queue
    q.submit([&](handler& h) {
        // Accessors for the buffers assuming they are created and
            initialized
        auto accA = bufA.get_access<access::mode::read>(h);
        auto accB = bufB.get_access<access::mode::read>(h);
        auto accC = bufC.get_access<access::mode::write>(h);

        // Define and dispatch the kernel
        h.parallel_for(range<1>(N), [=](id<1> i) {
             accC[i] = accA[i] + accB[i];
        });
    });
    ...
}
```

Listing (2.1.3): SYCL example using the DPC++ implementation.

2.2 Performance Optimization for GPUs

While the performance benefits of GPUs are substantial, it is crucial to acknowledge that effectively leveraging GPU capabilities presents its own set of challenges (Ryoo et al., 2008; Reyes & de Sande, 2012). The development of GPU-accelerated applications often requires specialized programming models and languages, such as CUDA or OpenCL, which introduce additional complexity to the software development process. Moreover, the optimization of algorithms for GPU execution frequently necessitates a fundamental rethinking of computational approaches to maximize parallelism and minimize data transfer overheads.

2.2.1 GPUs Common Optimization Techniques

Coalesced memory access patterns in GPU computing are crucial for optimizing performance. In NVIDIA GPUs, data is transferred between Streaming Multiprocessors (SMs) and memory in cache lines (see figure 2.3), typically 128 bytes (NVIDIA Corporation, 2024a). GPUs execute threads in warps, usually 32 threads, and when these threads access adjacent memory locations, it's called coalesced access (see figure 2.4). This pattern is efficient because it reduces the number of memory fetches required (Schmidt, Gonzalez-Dominguez, Hundt, & Schlarb, 2018), as a single cache line can serve multiple threads within a warp. Additionally, coalesced access minimizes bank conflicts in shared memory, where multiple threads accessing the same memory bank can cause serialization (Li, Wei, Sun, Annavaram, & Kim, 2019). By contrast, interleaved access patterns, where threads access non-adjacent memory locations, are less efficient. Implementing coalesced memory access can significantly improve GPU performance by reducing memory bandwidth usage and minimizing bank conflicts, making it a key optimization technique in GPU programming.

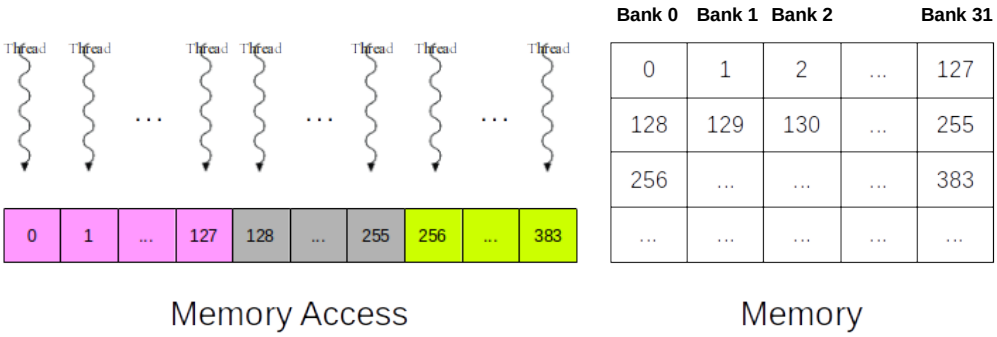

Figure 2.3: An illustration of how the memory is accessed in the threads using cache lines and memory banks. (Mohamed, 2020).

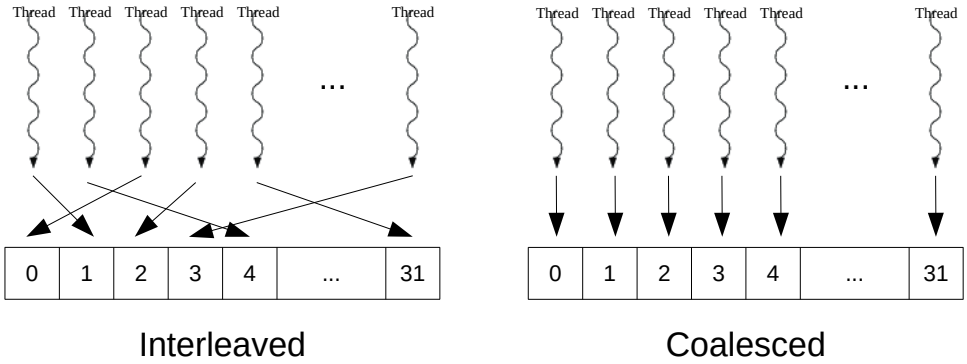

Figure 2.4: An illustration of the difference between interleaved and coalesced memory access patterns. (Mohamed, 2020).

Thread divergence (see figure 2.5), a situation where different threads within a warp take divergent execution paths, can severely impact GPU performance (Lin & Wang, 2020). Techniques to mitigate this issue include branch prediction, warp-level programming, and algorithm redesign to promote more uniform execution across threads. Additionally, the careful selection of thread block sizes and grid configurations can significantly influence the efficiency of GPU kernels.

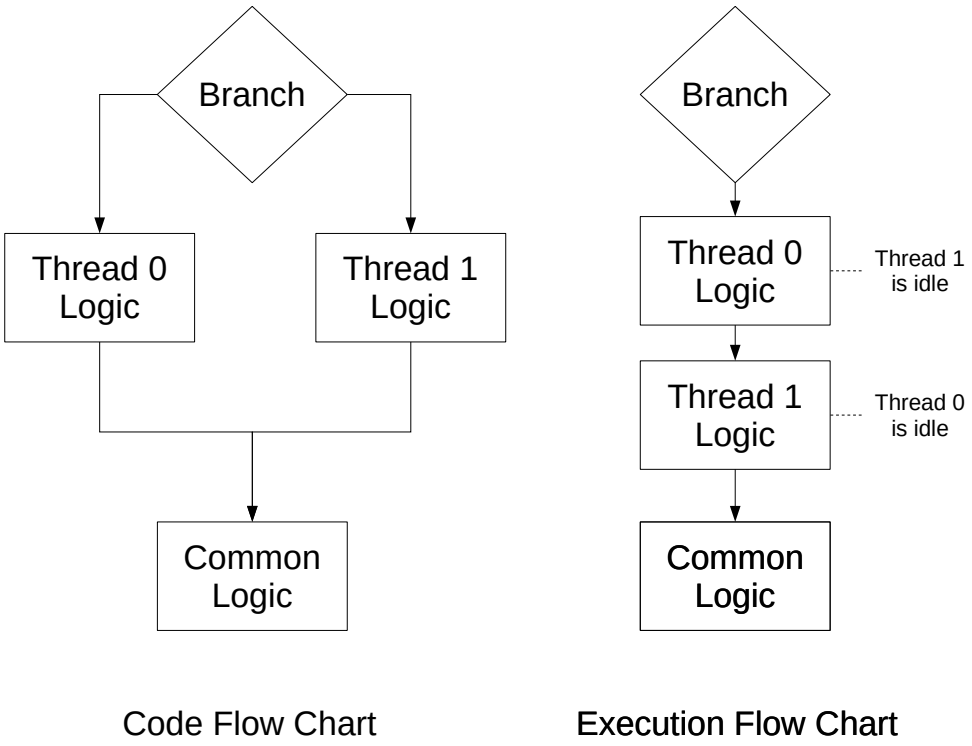

Figure 2.5: Diverged code flow chart vs execution flow chart. (Mohamed, 2020).

Optimizing data transfer between the host CPU and the GPU device is another crucial aspect of GPU application optimization (Gelado & Garland, 2019). This can be achieved through techniques such as asynchronous data transfer (see figure 2.6), pinned memory allocation, and the use of unified memory in supported architectures. Furthermore, the implementation of stream processing and concurrent kernel execution can help to maximize

GPU utilization by overlapping computation and data transfer operations.

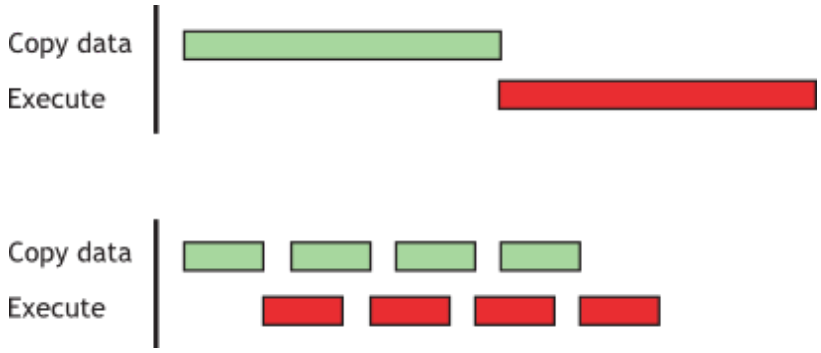

Figure 2.6: Timeline comparison between data copy and kernel execution (NVIDIA Corporation, 2024a).

Algorithmic optimizations tailored specifically for GPU architectures play a vital role in enhancing application performance (Rovere, Chen, Di Pilato, Pantaleo, & Seez, 2020). These may include techniques such as loop unrolling, instruction-level parallelism, and the use of intrinsic functions to leverage hardware-specific features. Moreover, the adoption of GPU-friendly data structures and algorithms, such as those that promote coalesced memory access and reduce thread divergence, can lead to substantial performance gains.

Profiling and performance analysis tools are indispensable in the optimization process (Ebrahim, Hammad, Zeki, & Alqaddoumi, 2021; Zhou et al., 2021). These tools provide insights into kernel execution times, memory access patterns, and potential bottlenecks, enabling developers to identify and address performance issues systematically. Through iterative profiling and optimization, significant improvements in GPU application performance can be realized.

With time, new optimization techniques emerge, and existing ones are refined (Reyes & de Sande, 2012). The field of GPU optimization remains dynamic, with ongoing research exploring novel approaches to harness the full potential of these powerful computing resources.

2.2.2 Performance Portability

Performance portability has become an increasingly critical concern in the field of high-performance computing (Neely, 2016). As computational architectures continue to diversify, the challenge of writing code that can efficiently execute across multiple platforms has grown more complex (Sedova, Eblen, Budiardja, Tharrington, & Smith, 2018; McIntosh-Smith, Boulton, Curran, & Price, 2014). To address this issue, various performance portability libraries and standards have been developed, each offering unique approaches to achieving portable performance (Ben-Nun, Gamblin, Hollman, Krishnan, & Newburn, 2020).

Higher-level performance portability libraries, such as RAJA (Beckingsale et al., 2019) and Kokkos (Trott et al., 2022), provide abstraction layers that allow programmers to write code once and run it efficiently on different architectures. RAJA, for instance, employs a series of loop transformations that are mapped to the underlying hardware, enabling parallelization without requiring extensive knowledge of the target architecture. Kokkos, on the other hand, offers a `parallel\_for` construct and algorithms based on `parallel\_for`, which can be customized using policies to map to various levels of parallelism (refer to listing (2.2.4)). These libraries aim to simplify the process of writing portable code by handling many of the low-level details of parallelization and optimization.

```
struct Functor {
    DataType _data;
    Functor (DataType data) : _data (data) {}
    void operator ()(...) const {...}
}
...
Functor functor(data);
Kokkos::parallel_for(numberOfDataPoints, functor);
...
```

Listing (2.2.4): Kokkos example.

In contrast, lower-level performance portability libraries like alpaka (Zenker et al., 2016) provide a different approach. Alpaka gives programmers the tools to write portable kernels, but it is the programmer's responsibility to implement the details of the parallel algorithm (refer to listing (2.2.5). This approach offers greater flexibility and control over the implementation, but it also requires a deeper understanding of parallel programming concepts and the target architectures.

```
struct Kernel
{
    // A template for the device type, CPU, GPU, FPGA, etc.
    template<typename TAcc>
    ALPAKA_FN_ACC auto operator()(TAcc const& acc) const -> void
    { /* Kernel Logic */}
};
...
// Launching the kernel, assuming everything needed is predefined.
alpaka::exec<Acc>(
        queue,
        workDiv,
        helloWorldKernel);
```

Listing (2.2.5): Alpaka example.

Standards such as OpenCL and SYCL have been developed to address performance portability at a broader level. These standards define a set of specifications that must be implemented by hardware vendors, ensuring a consistent programming interface across different platforms. OpenCL, for example, provides a framework for writing programs that execute across heterogeneous platforms, including CPUs, GPUs, and other processors. SYCL, building upon OpenCL, offers a higher-level abstraction that allows for single-source C++ programming for heterogeneous systems (see listing (2.1.3) from the previous section).

Another approach to performance portability is exemplified by OpenMP, which uses compiler directives to specify parallelism in the code (listing

(2.2.6)). These directives allow programmers to annotate their code with parallelization instructions, which are then interpreted by the compiler to generate appropriate parallel code for the target architecture. This approach can be particularly useful for incrementally parallelizing existing codebases.

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += i;
}
```

Listing (2.2.6): OpenMP example.

It is important to note that while these libraries and standards facilitate the development of portable code, achieving optimal performance across different architectures often requires additional tuning (Matthes et al., 2017). This tuning process may be performed manually by the programmer or automatically by the compiler or autotuners, depending on the specific library or standard being used. The goal of this tuning is to ensure maximum utilization of the available hardware resources on each target architecture, which is crucial for achieving true performance portability.

2.3 Autotuning

Autotuning became a necessity as soon as the complexity of hardware architectures and software systems began to outpace the ability of human experts to manually optimize code for diverse platforms (Whaley & Dongarra, 1998). As computational systems grew increasingly heterogeneous and complicated, the need for automated optimization techniques became paramount. Autotuning addresses this challenge by systematically exploring the vast configuration space of software parameters, employing various search algorithms and machine learning techniques to identify optimal or near-optimal configurations.

2.3.1 Concept and Importance

Autotuning, a critical process in software optimization, involves the automated adjustment of program parameters to enhance performance (Balaprakash et al., 2018). The search space, which encompasses all possible combinations of parameter values, is a fundamental concept in autotuning. It is often characterized by its dimensionality and the range of values each parameter can assume. As the search space grows larger, the complexity of finding optimal configurations increases exponentially, making efficient exploration strategies necessary.

The tuning interface serves as the bridge between the autotuner and the target application. It is typically implemented as a communication interface that allows the autotuner to modify parameters and measure the resulting performance (Balaprakash et al., 2018). This interface must be designed with care to ensure minimal overhead and accurate performance measurements. Tunable code, the portion of the application that can be modified by the autotuner, is prepared by developers to expose key parameters that significantly impact performance. These parameters may include algorithmic choices, data structure configurations, or hardware-specific settings.

Search algorithms form the core of autotuning systems, guiding the exploration of the search space to identify high-performing configurations. Various approaches have been developed, ranging from simple techniques like random search and grid search to more sophisticated methods such as genetic algorithms, simulated annealing, and Bayesian optimization (Seymour, You, & Dongarra, 2008a). The choice of search algorithm often depends on the characteristics of the search space and the available computational resources for tuning.

Different types of autotuners have emerged to address specific optimization challenges (Balaprakash et al., 2018). Offline autotuners perform the tuning process before the application is deployed, generating optimized con-

figurations for various scenarios. Online autotuners, in contrast, adapt the application during runtime, responding to changes in input data or execution environment. Hybrid approaches combine elements of both offline and online tuning to balance the benefits of pre-computed optimizations with runtime adaptability.

Performance metrics and objective functions play a crucial role in guiding the autotuning process. These metrics serve as quantitative measures of a configuration's effectiveness and are used to compare different parameter sets. Common performance metrics include execution time, throughput, energy consumption, and resource utilization. The choice of metric depends on the specific optimization goals and can significantly influence the tuning outcomes. Objective functions, which may combine multiple metrics, are formulated to provide a single value representing the overall quality of a configuration. These functions can be designed to balance conflicting objectives, such as maximizing performance while minimizing resource usage. The definition of appropriate objective functions is a challenging task that often requires domain expertise and careful consideration of the target application's requirements. Multi-objective optimization techniques are sometimes employed when multiple, potentially conflicting, performance goals must be addressed simultaneously (Balaprakash, Tiwari, & Wild, 2014).

Despite its potential benefits, autotuning faces several challenges and limitations that require consideration. One significant issue is the curse of dimensionality, where the search space grows exponentially with the number of tunable parameters, making exhaustive exploration infeasible for complex systems (Ashouri et al., 2018). Additionally, the cost of evaluating configurations can be prohibitively high, especially for large-scale applications or when real-world workloads are required for accurate performance assessment (Ashouri et al., 2018). Autotuners may also struggle with non-deterministic behaviour in target applications, where performance variations

due to factors such as system noise or data-dependent execution paths can obscure the true impact of parameter changes (Willemsen et al., 2024). The portability of tuned configurations across different hardware platforms or input datasets remains a challenge, as optimal settings for one environment may not translate well to another (Tørring et al., 2023). Furthermore, the interpretability of autotuning results can be limited, making it difficult for developers to gain insights into the underlying performance characteristics of their applications (Willemsen et al., 2024). Addressing these challenges requires ongoing research in areas such as dimensionality reduction techniques, adaptive sampling strategies, robust performance modelling, and explainable autotuning approaches.

The effectiveness of autotuning is influenced by factors such as the quality of performance models, the selection of representative workloads, and the ability to generalize findings across different execution environments. As autotuning techniques continue to evolve, researchers are exploring ways to handle increasingly complex search spaces, improve the scalability of tuning processes, and integrate machine learning approaches to enhance the efficiency and effectiveness of parameter optimization in software systems.

2.3.2 Autotuning for GPUs

The process of autotuning a GPU application involves the systematic adjustment of various tunable parameters to achieve optimal execution efficiency. These parameters may encompass a wide range of factors, including but not limited to thread block dimensions, cache usage, memory access patterns, and algorithmic choices (van Werkhoven, 2019). The complexity of modern GPU architectures necessitates a thorough exploration of the parameter space to identify configurations that yield the best performance for a given application.

Kernel tuning, a fundamental aspect of GPU application optimization,

focuses on refining individual computational kernels. This process often involves the manipulation of low-level parameters such as thread hierarchy, shared memory utilization, and register allocation. By fine-tuning these elements, developers can significantly enhance the efficiency of core computational units within their applications. The optimization of kernels can lead to substantial improvements in overall application performance, as these units typically constitute the most computationally intensive portions of GPU programs (van Werkhoven, 2019).

Full application tuning extends beyond individual kernels to encompass the entire GPU application (Guerreiro, Ilic, Roma, & Tomás, 2015). This holistic approach considers the interplay between different components of the software, including data transfer operations, synchronization points, and the orchestration of multiple kernels. The process may involve restructuring the application architecture, optimizing memory hierarchies, and balancing workloads across available resources. Full application tuning often requires a deep understanding of both the underlying hardware capabilities and the specific requirements of the application domain.

In scenarios involving multiprocess GPU applications, the complexity of autotuning increases significantly. The optimization process must account for the interactions between multiple concurrent processes, each competing for GPU resources. Considerations such as inter-process communication, resource contention, and load balancing become important. Effective multiprocess tuning strategies may involve dynamic workload distribution, cooperative scheduling algorithms, and intelligent resource allocation mechanisms to maximize overall system throughput whilst minimizing conflicts between competing processes (Guerreiro et al., 2015).

As GPU manufacturers introduce new features such as ray tracing cores, tensor cores for AI acceleration, and improved memory hierarchies, autotuning strategies must adapt to exploit these advancements effectively.

(Markidis, Chien, Laure, Peng, & Vetter, 2018) The increasing complexity of GPU architectures, including multi-GPU systems and GPUs with heterogeneous compute units, necessitates more sophisticated autotuning approaches. For instance, optimizing workloads for mixed-precision arithmetic operations, now common in AI and scientific computing, requires autotuners to consider precision-performance trade-offs (Gu & Becchi, 2020).

Additional aspects of GPU application autotuning may include the optimization of data layouts, the exploration of alternative algorithmic implementations, and the exploitation of hardware-specific features. The development of robust autotuning frameworks and tools has greatly facilitated this process, enabling automated exploration of vast parameter spaces and the discovery of non-intuitive optimizations.

This chapter has provided a comprehensive overview of GPU computing fundamentals, performance optimization techniques, and autotuning approaches. The discussion covered the essential aspects of GPU architecture, including its parallel processing capabilities and memory hierarchy, as well as various programming platforms available for GPU development. Performance optimization techniques, from memory access patterns to cross-platform portability solutions, were examined. The chapter also explored autotuning concepts and their application to GPU optimization, highlighting both kernel-level and full-application approaches. Table 2.1 summarizes the key topics and concepts discussed.

Table 2.1: Summary of Background Topics

Topic	**Key Points**
GPU Computing	• Parallel architecture optimized for throughput • SIMT execution model with SMs as building blocks • Specialized memory hierarchy • Programming platforms: CUDA, ROCm, OpenCL, SYCL
Performance Optimization	• Coalesced memory access patterns • Thread divergence mitigation • Data transfer optimization • Profiling and analysis tools • Performance portability approaches
Autotuning	• Search space exploration techniques • Tuning interface design • Multiple optimization objectives • Online vs offline tuning • GPU-specific kernel and application tuning

Chapter 3

Literature Review

An extensive review of the current state-of-the-art in performance auto-tuning for GPU applications is presented in this chapter. The scope of this review covers several key areas that are fundamental to the development of an effective auto-tuning framework. These areas include existing auto-tuning frameworks, the application of machine learning techniques in performance optimization, multi-fidelity optimization strategies, multiprocess performance autotuning, and benchmarking methodologies.

Throughout this review, the current gaps in the field are identified, and the foundation for the proposed performance auto-tuning framework is established.

3.1 Autotuning Frameworks

The literature on autotuning presents a rich and diverse landscape of tools and frameworks, each designed to address specific challenges in optimizing performance across various domains. Autotuning, the automated process

of optimizing parameters or configurations to enhance the performance of software or hardware systems, is crucial in a world where computational efficiency is of utmost importance, especially in HPC in general, and GPU-accelerated applications in particular.

One of the most widely used autotuning frameworks is OpenTuner (Ansel et al., 2014), which emphasizes the creation of domain-specific, multi-objective autotuners. OpenTuner is particularly notable for its flexibility, allowing users to define custom optimization problems tailored to the specific requirements of their applications. This framework supports a wide range of search techniques, including evolutionary algorithms, hill climbing, and simulated annealing, among others. The ability to integrate multiple search techniques makes OpenTuner a powerful tool for complex optimization tasks where the objective functions may be multifaceted, involving trade-offs between performance metrics such as execution time, memory usage, and energy consumption.

Another significant contribution to the autotuning ecosystem is Kernel Tuner (van Werkhoven, 2019), which, along with Kernel Tuner Toolkit (Petrovič et al., 2020), is specifically designed for tuning computational kernels. Computational kernels are the core components of many high-performance applications, particularly those leveraging the capabilities of GPUs. The efficiency of these kernels is critical, as they often represent the most computationally intensive parts of the code. Kernel Tuner allows developers to experiment with different configurations, such as thread block sizes and loop unrolling factors, to identify the optimal settings that maximize performance on a given hardware platform. The Kernel Tuner Toolkit extends this functionality by providing additional tools and utilities that facilitate the development and tuning of these kernels, making it easier for developers to achieve peak performance in HPC and GPU-accelerated environments.

CLTune (Nugteren & Codreanu, 2015) is another specialized autotuning tool that focuses on the optimization of OpenCL kernels. OpenCL is a framework for writing programs that execute across heterogeneous platforms, including CPUs, GPUs, and other processors. CLTune provides a user-friendly interface for exploring the vast search space of possible kernel configurations, enabling developers to quickly find the most efficient implementations. Its design emphasizes ease of use and integration with existing OpenCL projects, making it an attractive option for developers working in heterogeneous computing environments.

GPTune (Liu et al., 2021) takes a more advanced approach by incorporating transfer learning into the autotuning process. Transfer learning (Torrey & Shavlik, 2010), a technique commonly used in machine learning, involves applying knowledge gained from one problem to improve performance on a related problem. GPTune leverages this concept to accelerate the tuning process, particularly in scenarios where prior tuning data is available. By reusing information from previous tuning efforts, GPTune can reduce the number of evaluations needed to find optimal configurations, making it a highly efficient tool for applications where tuning time is a critical factor.

Optuna (Akiba, Sano, Yanase, Ohta, & Koyama, 2019) adds another dimension to the autotuning landscape by focusing on hyperparameter tuning in machine learning models. Hyperparameter tuning is a crucial step in the development of machine learning algorithms (Yang & Shami, 2020), as the choice of hyperparameters can significantly impact model performance. Optuna employs Bayesian optimization (BO), a probabilistic model-based approach that balances exploration and exploitation to efficiently search the hyperparameter space. This makes Optuna particularly well-suited for machine learning applications, where the tuning process can be both time-consuming and computationally expensive.

These autotuning frameworks illustrate the wide applicability and im-

portance of autotuning across different domains (see Table 3.1). From optimizing computational kernels in HPC and GPU-accelerated applications to fine-tuning hyperparameters in machine learning models, autotuning remains a critical tool for developers seeking to maximize performance. The diversity of approaches, ranging from domain-specific frameworks like OpenTuner and Kernel Tuner to advanced techniques like those employed by GPTune and Optuna, emphasizes the ongoing efforts in this field and the continued need for specialized tools capable of addressing the unique challenges of different applications.

Table 3.1: Comparison of Autotuning Frameworks and Their Features

Framework	**Primary Focus**	**Key Features**	**Search Methods**
OpenTuner (Ansel et al., 2014)	General-purpose autotuning	• Multi-objective optimization • Custom search space definition • Extensible architecture	• Evolutionary algorithms • Hill climbing • Simulated annealing • AUCBandit
Kernel Tuner (van Werkhoven, 2019)	GPU kernel optimization	• CUDA/OpenCL support • Thread block optimization • Loop unrolling tuning	• Random search • Bayesian optimization • Evolutionary algorithms • Basin Hopping
CLTune (Nugteren & Codreanu, 2015)	OpenCL kernel tuning	• OpenCL-specific optimizations • Easy integration • Cross-platform support	• Random search • Simulated annealing
GPTune (Liu et al., 2021)	Transfer learning-based tuning	• Knowledge transfer • Reduced evaluation count • Historical data utilization	• Bayesian optimization • Transfer learning
Optuna (Akiba et al., 2019)	Hyperparameter optimization	• ML-focused design • Distributed optimization • Pruning mechanisms	• Bayesian optimization • Tree-structured Parzen • Random search

3.2 Autotuning Benchmarking Methodologies

In the field of heterogeneous computing, benchmark suites play a crucial role in evaluating and comparing the performance of different systems and programming models. Over the years, several benchmark suites have been developed to address various aspects of heterogeneous computing, each with its own focus and strengths.

One of the earliest and most influential benchmark suites in this domain is Rodinia (Che et al., 2009). Developed to cover a wide range of parallel patterns and synchronization techniques, Rodinia has been a standard in the field for many years. Despite being somewhat outdated now, it continues to be a valuable resource for researchers and developers. Rodinia's strength lies in its comprehensive coverage of different computational patterns and its implementation support for multiple programming models, including CUDA, OpenCL, and OpenMP. This versatility has made it a go-to benchmark suite for comparing the performance of different heterogeneous computing platforms and programming approaches.

Building upon the foundation laid by Rodinia, newer benchmark suites have emerged to address the evolving needs of the heterogeneous computing landscape. One such suite is HeCBench (Jin & Vetter, 2023), which aims to improve the performance portability of the SYCL programming model. HeCBench extends the Rodinia suite and combines it with other benchmarks, providing a more comprehensive set of tests that reflect modern heterogeneous computing challenges. This suite is particularly useful for developers working with SYCL and those interested in performance portability across different hardware architectures.

As the field of heterogeneous computing continues to advance, particularly with the rise of deep learning and neural networks, new benchmark suites have been developed to address these emerging areas. Mirovia (Hu & Rossbach, 2019) is a prime example of this trend. This modern bench-

mark suite includes applications from previous suites while also incorporating new benchmarks focused on deep neural networks. By combining traditional heterogeneous computing tasks with cutting-edge deep learning applications, Mirovia provides a more holistic view of the performance characteristics of modern heterogeneous systems.

While some benchmark suites focus on specific programming models or application domains, others take a broader approach to assessing heterogeneous computing systems. The SHOC benchmark suite (Danalis et al., 2010), for instance, is designed to evaluate both the performance and stability of scalable heterogeneous computing systems that incorporate GPUs and multicore processors. This comprehensive approach makes SHOC particularly valuable for researchers and developers working on large-scale heterogeneous systems where stability is as critical as raw performance.

As energy efficiency becomes an increasingly important consideration in computing, benchmark suites that address power consumption have also emerged. EPPMiner is one such suite (Q. Wang, Xu, Zhang, & Chu, 2017), designed to evaluate not just performance but also power consumption and energy efficiency of heterogeneous systems. This multi-objective approach reflects the growing importance of energy considerations in modern computing environments, particularly in data centers and high-performance computing facilities.

Some benchmark suites focus on very specific aspects of heterogeneous computing. HeteroSync,(Sinclair, Alsop, & Adve, 2017) for example, is a specialized suite that measures fine-grained synchronization on tightly coupled CPU-GPU systems. While narrow in focus, such specialized suites provide invaluable insights into specific performance characteristics that can be critical for certain applications or system designs.

The challenge of objectively comparing autotuners and tuning algorithms has become increasingly important in the field of heterogeneous comput-

ing. As systems grow more complex, manual optimization becomes increasingly difficult, leading to a greater reliance on automatic performance tuning. However, evaluating the effectiveness of different autotuners and tuning algorithms presents its own set of challenges (van Werkhoven, 2019).

Recognizing this need, researchers like Werkhoven et al. have focused on developing methodologies and tools specifically designed for this purpose (Tørring et al., n.d.). Their work addresses the fundamental problem of how to fairly and accurately assess the performance of different autotuning approaches across a wide range of scenarios.

The development of "BAT: A Benchmark suite for AutoTuners" (Sund, Kirkhorn, Tørring, & Elster, n.d.) represents a significant step forward in this area. This suite provides a standardized set of problems and metrics specifically designed to test autotuners, allowing for more direct and meaningful comparisons between different approaches. By offering a common ground for evaluation, BAT helps researchers and developers to identify the strengths and weaknesses of various autotuning strategies more objectively.

However, the challenge extends beyond just having a suitable benchmark suite. The methodology for using such suites is equally crucial. Willemsen et al. (2024) work on a rigorous benchmarking methodology addresses this aspect, providing guidelines on how to conduct fair and reproducible comparisons of autotuners. This methodology takes into account factors such as the stochastic nature of many tuning algorithms, the impact of different hardware configurations, and the variability in problem characteristics.

By combining a specialized benchmark suite with a well-defined methodology, this approach aims to bring more objectivity and reliability to the evaluation of autotuners and tuning algorithms. This is particularly important in heterogeneous computing environments, where the interaction between different hardware components and the wide variety of potential optimizations make it difficult to predict which tuning approaches will be most

effective in any given scenario. Table 3.2 lists a summary of the benchmarks discussed in this section.

Table 3.2: Comparison of Heterogeneous Computing Benchmark Suites

Suite	Focus	Key Features
Rodinia (Che et al., 2009)	General parallel computing	• Multiple programming models • Pattern coverage • Platform comparison
HeCBench (Jin & Vetter, 2023)	SYCL performance portability	• Extended Rodinia suite • Modern heterogeneous computing • Cross-platform evaluation
Mirovia (Hu & Rossbach, 2019)	Modern GPU workloads	• Deep neural networks • Mixed benchmark types • Contemporary applications
SHOC (Danalis et al., 2010)	Scalable systems	• System stability testing • Multi-GPU performance • Multicore integration
EPPMiner (Q. Wang et al., 2017)	Energy efficiency	• Power consumption metrics • Performance-power trade-offs • Efficiency evaluation
HeteroSync (Sinclair et al., 2017)	Fine-grained sync	• CPU-GPU synchronization • Timing analysis • Specialized measurements
BAT (Sund et al., n.d.)	Autotuner evaluation	• Standardized test problems • Comparison metrics • Reproducible benchmarks

3.3 Optimization Techniques Used in Autotuning

Different autotuners employ a variety of search and optimization strategies to enhance performance, each with its own strengths and limitations. This

diversity in approaches reflects the complexity of the autotuning problem and the need to adapt to various scenarios.

Some autotuners use brute force algorithms, especially when dealing with smaller search spaces. For example, Kernel Tuner (van Werkhoven, 2019), as mentioned in the original text, employs this approach. Brute force methods systematically explore all possible combinations of parameter values within the defined search space. While this approach guarantees finding the optimal solution, it becomes impractical for larger search spaces due to the exponential growth in the number of combinations to evaluate.

For larger search spaces, more sophisticated algorithms are necessary. OpenTuner, for instance, uses a variety of techniques, including random search. Despite its simplicity, *random search* has proven to be very effective and often serves as a tough baseline for other methods to beat. This effectiveness has been noted in several studies, highlighting that sometimes simpler approaches can yield competitive results (Seymour, You, & Dongarra, 2008b; Tørring & Elster, 2022).

Bayesian optimization has gained significant traction in recent years due to its effectiveness in constructing surrogate models that guide the search process (Willemsen, van Nieuwpoort, & van Werkhoven, 2021; Victoria & Maragatham, 2021; Menon, Bhatele, & Gamblin, 2020). This approach uses probabilistic models to predict the performance of different parameter configurations, allowing it to make more informed decisions about which configurations to evaluate next. Bayesian optimization is particularly useful when evaluations are expensive or time-consuming, as it aims to minimize the number of evaluations needed to find a good solution.

Multi-fidelity autotuning is an advanced approach that optimizes the use of computational resources by integrating various levels of fidelity. Tools like SMAC3 (Lindauer et al., 2022) and GPTuneBand (Zhu, Liu, Ghysels, Bindel, & Li, 2022) fall into this category. The basic idea behind

multi-fidelity tuning is to use cheaper, lower-fidelity evaluations to guide the search in the early stages, and then progressively increase the fidelity as the search narrows down to promising regions of the parameter space. This approach can significantly reduce the overall computational cost of tuning, especially for problems where high-fidelity evaluations are very expensive.

Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) offer alternative approaches to autotuning. These are nature-inspired algorithms that mimic biological evolution and social behavior, respectively. GAs work by evolving a population of candidate solutions over multiple generations, using operations like selection, crossover, and mutation. PSO, on the other hand, simulates a swarm of particles moving through the search space, with each particle's movement influenced by its own best known position and the swarm's best known position. While these methods can be very effective for certain types of problems, they often come with higher computational costs compared to some other methods (van Werkhoven, 2019).

The choice of which autotuning strategy to use depends on various factors, including the size and nature of the search space (Tørring & Elster, 2022), the cost of evaluating each configuration (Lindauer et al., 2022; Zhu et al., 2022), the available computational resources, and the specific characteristics of the problem at hand. For instance, if evaluations are quick and the search space is small, a brute force approach might be feasible and guarantee finding the global optimum. For larger spaces with expensive evaluations, Bayesian optimization or multi-fidelity approaches might be more appropriate.

In practice (see Table 3.3), many modern autotuning systems use a combination of these strategies, often allowing users to choose or even combine different search algorithms. This flexibility allows autotuners to be applied to a wide range of problems and to adapt to the specific needs and constraints of different software systems and computing environments.

Table 3.3: Summary of Autotuning Search Techniques

Technique	Strengths	Best Use Cases
Brute Force	• Guaranteed optimal solution • Simple implementation • Complete coverage	• Small search spaces • Quick evaluations • Verification needs
Random Search	• Simple yet effective • Good baseline performance • Easy to implement	• Large search spaces • Unknown landscapes • Initial exploration
Bayesian Optimization	• Efficient sampling • Model-based guidance • Fewer evaluations needed	• Expensive evaluations • Complex landscapes
Multi-fidelity	• Resource efficient • Progressive refinement • Balanced exploration	• Limited compute budget • Scalable applications
Genetic Algorithms	• Handles complex spaces • Good for mixed types • Parallel evaluation	• Discrete parameters • Large populations • Non-smooth objectives
Particle Swarm	• Dynamic adaptation • Good local search • Natural parallelism	• Continuous spaces • Social learning problems • Dynamic environments

3.4 Summary and Research Gaps

This literature review has examined three key areas of GPU application autotuning: frameworks, benchmarking methodologies, and optimization techniques. Autotuning frameworks for GPU applications were found to range from specific to general, each with its own advantages and tradeoffs, reflecting the complexity of GPU architectures and diverse application needs. In benchmarking methodologies, new approaches have been proposed to improve evaluation techniques, though many remain theoretical and need practical implementation. For optimization techniques, a wide range of strategies was observed, each with a unique set of benefits and tradeoffs, but no definitive solution for the problem.

A significant gap identified in current autotuning approaches is the lack of consideration for overall system throughput. Existing autotuners typically focus on optimizing individual process performance without accounting for the impact on concurrent processes. This oversight can lead to sub-optimal system-wide performance, particularly in GPU-intensive workloads where resource contention is a critical factor (Aguilera, Morrow, & Kim, 2014; Chaudhary, Ramjee, Sivathanu, Kwatra, & Viswanatha, 2020; Z. Wang et al., 2016). The discrepancy between favorable individual process metrics and potentially degraded overall system throughput highlights a crucial area for improvement in autotuning research.

Future research directions should address this gap by developing autotuning strategies that consider system-wide performance metrics. This includes extending the current autotuners to work with multiple processes, implementing and validating new benchmarking methodologies that account for multiprocess scenarios, and exploring optimization techniques that can balance individual process performance with overall system throughput.

Chapter 4

Methodology

This chapter outlines the methodological approach employed in this study on performance auto-tuning for GPU applications. The research design, data collection methods, and analysis techniques are described to provide a comprehensive understanding of how the study was conducted.

4.1 Overview of Methodology Components

This chapter presents several interconnected components that form a comprehensive methodology for GPU kernel autotuning in CMSSW. Here's how these components relate to each other:

1. **Research Design**: Establishes the foundational approach, combining comparative analysis and case study methods to evaluate autotuning techniques in CMSSW.

2. **CMSSW Framework**: Serves as the case study platform, providing the real-world context where the autotuning techniques are applied

and evaluated.

3. **Data Collection**: Details how performance data is gathered from both individual GPU kernels and the overall system, feeding directly into the evaluation process.

4. **Evaluation Criteria**: Defines how the collected data is analysed to assess the effectiveness of different autotuning approaches across multiple dimensions.

5. **Proposed Framework**: Presents the practical implementation that brings together all previous components, showing how autotuning is actually performed within CMSSW.

6. **Experimental Setup**: Describes the hardware and software environment where all the above components are deployed and tested.

These components form a logical progression from theoretical foundation to practical implementation and evaluation. The research design guides the overall approach, while CMSSW provides the application context. Data collection methods feed into the evaluation criteria, which are then applied to assess the proposed framework's performance across different experimental setups.

4.2 Research Design

This study employed a hybrid research design, combining comparative and case-study approaches to investigate performance autotuning techniques. The research focused specifically on offline GPU kernel configurations autotuning for performance optimization.

The comparative aspect of the design allowed for a systematic evaluation of different autotuning techniques and their relative effectiveness in

optimizing GPU kernel performance. This approach facilitated the identification of the most promising autotuning strategies and the conditions under which they excel.

The case study component centred on the CMSSW (Compact Muon Solenoid Software) framework (CMS Collaboration, 2024), a complex software system used in high-energy physics experiments. By applying autotuning techniques to the GPU kernels within CMSSW, the study aimed to provide in-depth insights into the practical challenges and benefits of autotuning in a real-world, computationally intensive scientific application.

The offline nature of the autotuning process allowed for extensive exploration of the parameter space without the constraints of real-time performance requirements. This approach enabled the research to focus on finding optimal or near-optimal configurations that could be applied in subsequent runs of the software.

By combining these research design elements, the study sought to contribute both generalizable knowledge about GPU kernel autotuning techniques and specific, actionable insights for improving the performance of the CMSSW framework. This dual approach aimed to bridge the gap between theoretical advancements in autotuning and their practical application in complex scientific software systems.

4.3 CMSSW Framework

The Compact Muon Solenoid Software (CMSSW) framework forms the core of this study's case analysis in GPU kernel autotuning. CMSSW is a complex, event-processing software developed and maintained by the CMS Collaboration for use in high-energy physics experiments at the Large Hadron Collider (LHC) at CERN (CMS Collaboration, 2006, 2021). This framework plays a crucial role in processing and analysing the vast amounts of

data generated by particle collisions in the CMS detector. As the computational demands of particle physics research continue to grow (Fernandez Perez Tomei, 2022), there is an increasing need to optimize CMSSW's performance, particularly through GPU acceleration and efficient kernel configurations (Bocci et al., 2019).

4.3.1 Overview of CMSSW

The Compact Muon Solenoid Software (CMSSW) is an event-processing framework developed for the CMS experiment at the Large Hadron Collider (LHC). It plays a crucial role in both real-time data acquisition and offline analysis of particle collision data.

In the data acquisition chain, CMSSW forms a key component of the High-Level Trigger (HLT) system. The HLT receives input from the Low-Level Trigger, which deals with the raw signals directly from the CMS detector. CMSSW within the HLT performs rapid event reconstruction and initial physics analysis, making critical decisions about which events to record for further study.

Beyond its real-time applications, CMSSW is extensively used offline on various computing resources, from individual workstations to large-scale distributed grids. In this capacity, it processes recorded data, performing more detailed event reconstruction and enabling in-depth physics analyses.

The framework's primary functions span from initial data processing and event reconstruction to complex physics analysis and particle interaction simulation. Its modular, open-source architecture facilitates continuous improvement and adaptation to evolving research needs.

CMSSW's dual role in both online triggering and offline processing, combined with its flexibility and community-driven development, makes it an ideal candidate for performance optimization studies. This is particularly relevant in the context of GPU acceleration and autotuning, which have the

potential to enhance both real-time data handling and offline analysis capabilities.

4.3.2 Relevance to GPU Kernel Autotuning

CMSSW is highly suitable for GPU acceleration due to its parallel nature (Ramírez, Yzquierdo, & Hernández, 2016), processing numerous independent events simultaneously. This aligns well with GPU architecture, especially crucial for the High-Level Trigger's 200ms decision window per event (Fernandez Perez Tomei, 2022). GPU acceleration offers the potential to significantly reduce processing times while providing a cost-effective method to enhance computing power compared to CPU-only scaling (Evans et al., 2021; Bocci, 2023).

The CMS collaboration has made substantial progress in GPU integration (Bocci et al., 2019). CMSSW has been operational on NVIDIA GPUs using CUDA since LHC Run 3 (Bocci, 2023). Current efforts focus on porting CMSSW to Alpaka, a platform-independent programming model, to support GPUs from various vendors including AMD and Intel (Bocci, Czirkos, et al., 2023).

Performance improvements through GPU acceleration and autotuning can profoundly impact CMSSW's capabilities (Bocci, Jones, & Kortelainen, 2023). Enhanced processing power enables more sophisticated physics analyses, potentially leading to new discoveries or more precise measurements (Bocci, Kortelainen, Innocente, Pantaleo, & Rovere, 2020). Moreover, increased performance can lower power consumption, contributing to more sustainable scientific computing practices. These advancements in speed and efficiency make GPU acceleration and autotuning of CMSSW a promising avenue for advancing high-energy physics research while optimizing resource utilization.

4.3.3 Integration of Autotuning in CMSSW

The autotuning approach integrates seamlessly with the CMSSW framework by leveraging its existing configuration system. CMSSW uses configuration files written in a Python-like language to specify which modules to execute, input datasets, expected outputs, and other runtime parameters. This configuration mechanism was extended to include GPU kernel configurations, allowing them to be passed to CMSSW during runtime. This approach eliminates the need for recompiling the core framework each time parameters are adjusted, significantly reducing the overhead of the autotuning process.

The autotuner itself operates as a standalone script that modifies these configuration files and executes CMSSW. This design allows for efficient exploration of different kernel configurations without altering the core codebase.

To facilitate autotuning within CMSSW, an autotuning interface was implemented in C++, which is CMSSW's primary development language. This interface, implemented as a header file, serves two main functions. It parses the kernel configurations from the modified configuration files and stores these configurations in a data structure accessible to kernel developers. This data structure enables kernel developers to easily supply the autotuned configurations to the code responsible for launching the kernels.

By abstracting the autotuning mechanism behind this interface, the implementation minimizes changes to existing CMSSW code while providing a flexible framework for incorporating autotuned parameters. This integration strategy ensures that autotuning can be applied to CMSSW with minimal disruption to its existing architecture, while still allowing for comprehensive exploration of kernel performance optimizations.

4.4 Data Collection

The process of gathering performance data for GPU-accelerated CMSSW modules under various kernel configurations is outlined in this section. A data collection strategy was designed to capture key performance metrics across a wide range of settings, providing a comprehensive basis for the autotuning process.

The collected data serves two primary purposes: it is used as input for the autotuning algorithms to explore and optimize kernel configurations, and it is analysed to measure the performance of CMSSW modules under different GPU kernel settings. Through this approach, a thorough exploration of the parameter space is enabled, facilitating the identification of optimal configurations for each GPU kernel within CMSSW.

In the following subsections, the specific methods, tools, and procedures employed in the data collection process are detailed.

4.4.1 GPU Kernels

The study targeted all reconstruction GPU kernels within CMSSW that have been successfully ported to GPU architecture. These modules were selected because they represent the current state of GPU acceleration in CMSSW, implementing various essential algorithms for event reconstruction in high-energy physics data analysis, such as line fitting, energy clustering, and histogram generation.

The choice of these modules was primarily driven by their existing GPU implementations. While these kernels are already functional on GPUs, they require autotuning to ensure optimal performance on current and future hardware. This approach allows for the optimization of already-accelerated modules, potentially unlocking further performance gains and ensuring adaptability to evolving GPU architectures.

By focusing on these successfully ported modules, the study aims to enhance the efficiency of CMSSW's GPU-accelerated components, maximizing the benefits of GPU computation across different hardware configurations.

4.4.2 Parameter Space

The study explored a range of GPU kernel configurations, focusing on several key parameters. The primary parameters adjusted were the number of threads and the number of blocks. These are fundamental to GPU performance optimization, as they directly influence work distribution across the GPU's computational units.

When possible, the block size was configured to find the optimal balance between parallelism and resource utilization. The number of blocks was adjusted to determine the most effective work distribution across the GPU's streaming multiprocessors.

In some kernels, an additional parameter, the number of strides, was also configurable. This parameter sets the number of elements processed by each GPU thread, affecting the workload distribution and potentially impacting overall kernel efficiency.

Beyond the GPU-specific parameters, the study also considered the broader context of CMSSW's multiprocess execution environment. To this end, the number of CPU threads, the number of streams, and the number of processes were included in the parameter space. These parameters play a crucial role in determining how effectively the CPU and GPU resources are utilized, and how well the overall system scales across different hardware configurations.

By exploring this comprehensive parameter space, the study aimed to identify configurations that optimize individual GPU kernel performance while enhancing the overall efficiency of CMSSW in its multiprocess environment.

4.4.3 Performance Metrics

Two primary performance metrics were collected and analysed in this study: single kernel throughput and system throughput.

Single Kernel Throughput: This metric measures the performance of individual GPU kernels. It represents the rate at which a specific kernel processes data, typically expressed in events per second or a similar unit. The throughput for each kernel was computed from its recorded execution time, providing a direct measure of the kernel's processing efficiency under various configurations.

System Throughput: This metric captures the overall performance of CMSSW running on a single compute node. It records the aggregate throughput of all CMSSW processes executing simultaneously on the node. System throughput provides an overall view of performance, accounting for the interplay between multiple processes, CPU threads, and GPU kernels. Like the single kernel metric, system throughput was derived from recorded execution times.

Both metrics were calculated using the following formula:

$$Throughput = \frac{Number\ of\ Events\ Processed}{Total\ Execution\ Time}$$

These performance metrics were chosen for their ability to provide comprehensive insights into both fine-grained (individual kernel) and coarse-grained (whole system) performance characteristics. By analysing these metrics across different parameter configurations, the study aimed to identify optimal settings that maximize the system-level efficiency in CMSSW's GPU-accelerated modules.

4.4.4 Data Collection Methodology

The data collection process was designed to ensure accurate and significant measurements of both individual GPU kernel performance and overall system throughput. Two distinct approaches were used:

1. **Individual Kernel Studies:** Nvidia Nsight profiling tools were employed to collect detailed performance data on individual GPU kernel executions (NVIDIA Corporation, 2024b). These tools provided insights into the behaviour and efficiency of specific kernels under various configurations.

2. **System Throughput Measurement:** A custom script was developed to measure and analyse the overall system throughput (Ebrahim, 2024a). This script automated the following processes:

   (a) **I/O Throughput Check:** The script measured I/O throughput to ensure it was insignificant compared to the system throughput, verifying that I/O was not a bottleneck affecting the results.

   (b) **Cache Preheating:** Before each measurement, a dummy load was run to preheat the caches, minimizing the impact of cold-start effects on the performance measurements.

   (c) **Repeated Measurements:** For configurations identified as potentially optimal, the measurement process was repeated 10 times after cache preheating. The average of these runs was then reported as the final result, reducing the impact of run-to-run variations.

   (d) **Throughput Calculation:** The script calculated system throughput based on the total execution time and the number of events processed across all CMSSW processes running on the compute node.

For each test, CMSSW was configured by the autotuner with the specific parameter settings under investigation, including GPU kernel configurations and process-level parameters. The custom script then managed the execution of CMSSW and the collection of performance data.

This approach allowed for a comprehensive analysis of both fine-grained kernel performance and overall system efficiency under various configurations.

4.5 Evaluation Criteria for Autotuners

The evaluation criteria described in this section are derived from the work presented in the paper ”Towards a Benchmarking Suite for Kernel Tuners” by Tørring et al. (n.d.). Five key evaluation criteria are employed to assess and compare the performance of different autotuning techniques in this study:

- **Distribution of Configurations:** The spread of performance across different configurations in the search space is analysed. Histograms or cumulative distribution functions of performance for all tested configurations are examined. The overall landscape of possible solutions, the frequency of high-performing configurations, and the potential risks of poor configurations can be understood through this criterion.

- **Convergence Rate:** The speed at which high-quality solutions are identified by an autotuner is measured by this metric. The best configuration found over time is tracked, allowing comparisons to be made on how quickly optimal or near-optimal solutions are approached by different autotuners. This is crucial for understanding the efficiency of various optimization algorithms, especially in time-constrained scenarios.

- **Optimal Speed-up:** The maximum performance improvement achieved by the autotuner is evaluated by this criterion. The execution time of the best configuration found by the autotuner is compared against a baseline configuration (typically default or naive settings). The potential impact of autotuning on application performance is demonstrated by this metric.

- **Feature Importance:** This technique is employed to identify which tuning parameters have the most significant impact on performance. This analysis is conducted across different GPU architectures to understand how the importance of various parameters may be changed with hardware. Tuning efforts can be prioritized and the search space for future optimizations can potentially be reduced through feature importance analysis.

- **Performance Portability:** How well the optimal configurations found for one GPU architecture perform when transferred to different GPU architectures is assessed by this criterion. The relative performance of configurations across different GPUs is calculated to quantify the degree of performance portability and highlight the need for architecture-specific tuning.

These five criteria enable a detailed evaluation of autotuning techniques, assessing not only final performance but also search efficiency, parameter importance, cross-architecture applicability, and performance distribution. This multifaceted approach provides valuable insights into the strengths and weaknesses of different autotuning methods for both users and developers.

4.6 The Proposed Framework

In the following sections, the architecture and technical details of the proposed framework will be presented, with its key components and their interactions being explored. Through this section, a comprehensive understanding of the framework's design, implementation, and practical application in the context of CMSSW GPU kernel optimization is aimed to be provided.

4.6.1 Introduction to the Autotuning Framework

The proposed autotuning framework is built as an extension of OpenTuner (Ansel et al., 2014). OpenTuner was chosen because it is widely used, flexible, and can be easily extended. These qualities were considered important for dealing with the specific challenges of optimizing GPU kernels in CMSSW.

Several key additions have been made to OpenTuner to make it better suited for GPU kernel autotuning in CMSSW. First, a new interface for GPU kernels was created. This allows the framework to work with different types of GPU applications. It helps the autotuner adjust kernel settings and measure performance across various problems types.

Second, a special interface with CMSSW was added. This helps the autotuner integrate with CMSSW without major changes to its original codebase. In addition, it allows reduces the overhead of autotuning by eliminating the need to recompile the code every time the code changes.

New search techniques were introduced to build on those already available in OpenTuner. These new techniques are specifically designed to handle the complex search space that arises from CMSSW's multiprocess and multi-kernel nature. The framework needs to optimize across multiple GPU kernels that may be running simultaneously or in sequence, making the optimization problem particularly challenging.

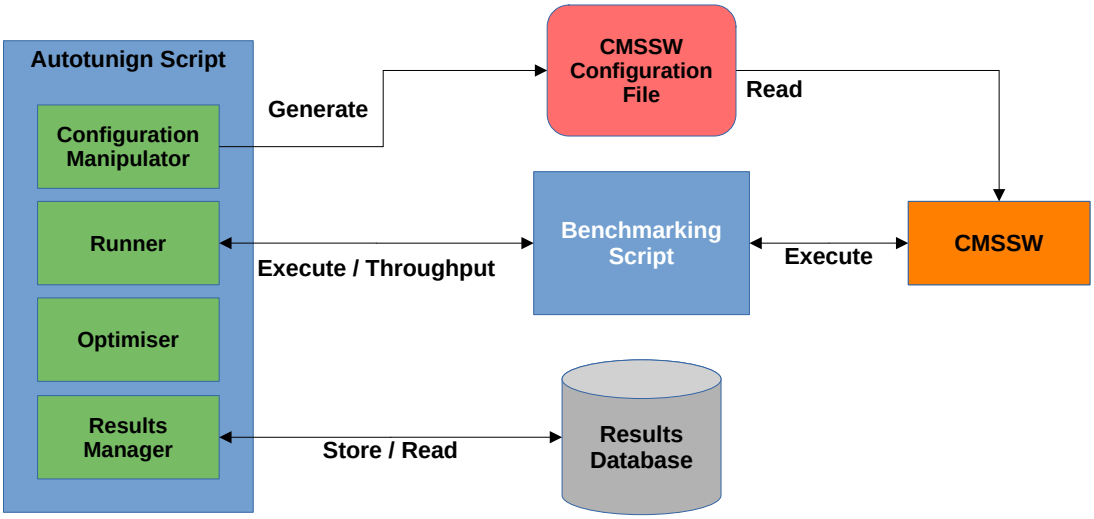

Figure 4.1: A diagram of the autotuning framework used in this study.

Lastly, some small improvements were made to make the framework easier to use. These changes help developers use the autotuner more easily when working with CMSSW.

By building on OpenTuner and adding these new features, the proposed framework is designed to handle the challenges of GPU kernel optimization in CMSSW. The rest of this chapter will explain these new features in more detail and show how well they work through a series of initial tests.

4.6.2 Framework Architecture

The autotuning framework consists of four primary components: the configuration manipulator, runner, optimizer, and results manager. A diagrammatic representation of this architecture is provided in Figure 4.1.

- **Configuration Manipulator:** The responsibility of generating CMSSW configuration files with varied parameters from the search space is assigned to the configuration manipulator. Parameters such as number

of threads, blocks, strides, etc. are systematically altered based on guidance provided by the search algorithm.

- **Runner:** The execution of CMSSW using the generated configurations is performed by the runner module, which also measures the resulting performance. To ensure accurate and significant measurment, a wrapper script (discussed in a previous chapter in section 4.4.4) around CMSSW is utilized to conduct multiple benchmark repetitions for each configuration. The initial few repetitions are discarded to account for caching effects, after which the average throughput across the remaining repetitions is reported.

- **Optimizer:** The complex search space is navigated by the optimizer with the aim of maximizing CMSSW throughput, utilizing the performance data collected by the runner. Different optimization algorithms are used, such as random search, Bayesian optimization, evolutionary algorithms, and others.

- **Results Manager:** All benchmarking data is stored and retrieved by the results' manager in a database, which is used to inform the optimizer. The tuning progress is also tracked to support analysis of the optimization trajectories taken by the autotuner.

The efficient autotuning of CMSSW reconstruction pipelines is enabled by the modular design and integration of these components. This architecture allows for flexible adaptation to the multiprocess and multi-kernel nature of CMSSW, addressing the complexities inherent in optimizing such a system.

4.6.3 Parameter Space Definition

The parameter space for the autotuning process is defined through a systematic approach that allows the developers to control the range of configurations to be explored. This methodology is implemented by the developers, who specify the parameter boundaries in terms of block and grid dimensions through the autotuning interface in CMSSW. A static method in C++ is employed to generate a list of parameters with constraints (refer to listing (4.6.1) for an example).

```
cms::KernelConfigurations::fillKernelDescriptions(desc,
{
    // kernel_name, threads, blocks, minThreads, maxThreads,
        minBlocks, maxBlocks
    {"RawToDigi_kernel", {512}, {0}, {32}, {1024}, {0}, {0}},
    {"CalibDigis", {256}, {0}, {32}, {1024}, {0}, {0}},
    {"CountModules", {256}, {0}, {32}, {1024}, {0}, {0}},
    {"FindClus", {384}, {0}, {32}, {1024}, {0}, {0}},
    {"ClusterChargeCut", {256}, {0}, {32}, {1024}, {0}, {0}}
});
```

Listing (4.6.1): Kerenl description example.

The list of parameters generated by this method is subsequently parsed by the autotuner and utilized to guide the search process. Listing (4.6.2) shows an example of how the kernel description is dumped by CMSSW. The autotuner interprets this list, creating an internal representation of the parameter space (refer to listing (4.6.3)).

```
Section 1.1.1.1 LoadTracks VPSet description:
    All elements will be validated using the PSet description in
       Section 1.1.1.1.1.
    The default VPSet has 1 element.
    [0]: see Section 1.1.1.1.2
Section 1.1.1.1.1 description of PSet used to validate elements of
   VPSet:
    device     string   ''
    threads    vuint32  (vector size = 1)
      [0]: 128
    blocks     vuint32  (vector size = 1)
      [0]: 0
    minThredas vuint32  (vector size = 1)
      [0]: 32
    maxThreads vuint32  (vector size = 1)
      [0]: 1024
    minBlocks  vuint32  (vector size = 1)
      [0]: 0
    maxBlocks  vuint32  (vector size = 1)
      [0]: 0
```

Listing (4.6.2): Kerenl description dump example.

```
manipulator.add_parameter(IntegerParameter("number_of_jobs", 1, 4))
manipulator.add_parameter(IntegerParameter("number_of_cpu_threads",
   2, 16))
manipulator.add_parameter(IntegerParameter("number_of_streams", 2,
   12))
manipulator.add_parameter(IntegerParameter("RawToDigi_kernel", 1,
   16))
manipulator.add_parameter(IntegerParameter("CalibDigis", 1, 16))
manipulator.add_parameter(IntegerParameter("CountModules", 1, 16))
manipulator.add_parameter(IntegerParameter("FindClus", 1, 16))
```

Listing (4.6.3): Configuration manipulator example.

The configuration generated by the autotuner is structured to accommodate multiple GPU configurations within the CMSSW framework. This approach allows for device-specific optimizations, with the appropriate parameters selected at runtime. The configuration is represented using the

CMSSW-specific Python syntax, which is then parsed and utilized by the framework.

In the configuration in listing (4.6.4), two kernels are defined: 'kernel_connect' and 'fishbone'. Each kernel is associated with a VPSet (Vector of Parameter Sets) that can contain multiple device-specific configurations. In the provided example, configurations for an NVIDIA T4 GPU are specified.

```
kernels = cms.PSet(
    kernel_connect = cms.VPSet(
        cms.PSet(
            device = cms.string('cuda/sm_75/T4'),
            threads = cms.vuint32(${kernel_connect_threads * 32}, ${
                kernel_connect_stride * 2}),
            blocks = cms.vuint32(0),
        ),
    ),
    fishbone = cms.VPSet(
        cms.PSet(
            device = cms.string('cuda/sm_75/T4'),
            threads = cms.vuint32(${fishbone_threads * 32}, ${
                fishbone_stride * 2}),
            blocks = cms.vuint32(0),
        ),
    )
)
```

Listing (4.6.4): CMSSW paramterized configuration file.

For each kernel, the device is identified using a string that includes the architecture and model information ('`cuda/sm_75/T4`'). The thread configuration is specified using a vuint32 (vector of unsigned 32-bit integers) with two elements. These elements are defined using placeholder variables (${...}) that the autotuner populates with specific values during the optimization process. For instance, '${kernel_connect_threads * 32}' and '${kernel_connect_stride * 2}' are used to set the thread dimensions for the 'kernel_connect' kernel.

The 'blocks' parameter is set to 0, which indicates that the block size will be determined automatically by the CMSSW framework based on the input data size and the thread configuration.

This flexibility enables the optimization process to account for the characteristics of different GPU architectures, potentially leading to better overall performance across a range of hardware configurations.

4.6.4 Search Techniques

In this thesis, a diverse array of search techniques are employed to navigate the complex parameter space of GPU kernel optimization in CMSSW. The selected techniques represent a combination of established methods and state-of-the-art approaches in the field of autotuning. The following search techniques are utilized:

- **Random Search**: This is a basic method that tries random settings. It's simple but often works well (Seymour et al., 2008b), especially when we don't know how settings affect performance. It's useful for searching large spaces where we can't try every option. OpenTuner already includes this method.

- **Greedy**: This straightforward approach iteratively selects the best immediate option available. While simple and fast, it can get stuck in local optima since it only considers one parameter at a time (Curtis, 2003). This method is included in OpenTuner and is particularly useful for quick initial optimization or when parameters are relatively independent.

- **Genetic Algorithms**: These methods copy how evolution works in nature. They create new solutions by mixing and changing existing

ones. Genetic Algorithms can handle many types of settings (Immanuel & Chakraborty, 2019) and are included in OpenTuner.

- **Particle Swarm Optimization (PSO)**: This method mimics how birds flock or fish swim in groups. It uses multiple ”particles” that move around, looking for the best solution. PSO is good for exploring complex settings(Kameyama, 2009) and is part of OpenTuner.

- **Differential Evolution**: This method tries to improve solutions over time. It works well with continuous settings and doesn't need the problem to be smooth, but it can be adapted to work with discrete values (Price, Storn, & Lampinen, 2014). OpenTuner has this technique built-in.

- **AUCBandit**: This method, first described in the OpenTuner paper (Ansel et al., 2014), chooses between different search techniques. It picks the ones that work best as it goes along. AUCBandit is the default method in OpenTuner because it works well in many situations.

- **Basin Hopping**: This method combines local searches with big jumps (Wales & Doye, 1997). It's good for problems with many local best points. It works by doing a series of local searches, each starting from a slightly changed version of the last best point found. This helps it explore many areas and potentially find the overall best solution. It's often used in chemistry and materials science, but isn't part of the OpenTuner package.

- **Bayesian Optimization**: A sophisticated method that constructs a probabilistic surrogate model (Gaussian Process) of how configuration parameters affect performance (van Werkhoven, 2019). The search space is progressively limited through the combination of performance

predictions and uncertainty estimates (Willemsen et al., 2021). An acquisition function balances exploration of uncertain regions with exploitation of promising areas (Victoria & Maragatham, 2021). Poor-performing regions are automatically excluded, while parameter correlations are identified to reduce dimensionality. This targeted approach is particularly valuable for GPU kernel optimization, where each configuration evaluation is computationally expensive. The method was implemented as an extension to OpenTuner specifically for this study.

The inclusion of these diverse search techniques allows for a comprehensive evaluation of different approaches to GPU kernel optimization in CMSSW. By comparing a wide range of methods, from the simple yet effective Random Search to sophisticated techniques like SMAC3 and Bayesian Optimization, this study aims to identify the most effective strategies for navigating the complex parameter space of CMSSW GPU kernels.

4.7 Experimental Setup

Our experiments were conducted on three distinct nodes with varying hardware configurations and a consistent software environment. This setup allows for a thorough evaluation of autotuning techniques across different architectures.

4.7.1 Hardware

Tables 4.1 and 4.2 detail the CPU and GPU specifications for each node, respectively.

Table 4.1: CPU Specifications

Node	CPU Model	Architecture	Specs
1	2x Intel Xeon Gold 6130	Skylake	32 cores total, 2.10 GHz base, 3.70 GHz turbo
2	AMD EPYC 7502P	Zen 2	32 cores, 2.5 GHz base, 3.35 GHz boost
3	2x AMD EPYC 9754	Zen 4	256 cores total, 2.25 GHz base, 3.1 3.1 GHz
4	2x AMD EPYC 9454	Zen 4	96 cores total, 2.75 GHz base, 3.7 GHz boost

Table 4.2: GPU Specifications

Node	1	2	3	4
GPU Model	NVIDIA Tesla T4	NVIDIA A10	NVIDIA L4	NVIDIA L40S
Architecture	Turing	Ampere	Ada Lovelace	Ada Lovelace
Cores	2560 CUDA	9216 CUDA	7168 CUDA	18176 CUDA
SM Count	40	72	60	142
Memory	16 GB GDDR6	24 GB GDDR6	24 GB GDDR6	48 GB GDDR6
Bus Width	256 bit	384 bit	192 bit	384 bit
L1 Cache (per SM)	64 KB	128 KB	128 KB	128 KB
L2 Cache	4 MB	6 MB	48 MB	48 MB
TDP	70 W	150 W	72 W	300 W

4.7.2 Software

The software environment was consistent across all nodes, with minor variations in the operating system version. Table 4.3 presents the software specifications.

Table 4.3: Software Environment

Component	Node 1	Node 2	Node 3
Operating System	Red Hat 8.10	Red Hat 8.7	Red Hat 8.9
CMSSW	14.0.14	14.0.14	14.0.14
NVIDIA Driver	550.54.15	550.54.15	550.54.15
CUDA Version	12.4	12.4	12.4

This setup provides a balanced mix of hardware configurations while maintaining a consistent software environment. It enables evaluation of autotuning techniques across different CPU and GPU architectures, allowing for insights into performance variations and optimization strategies across diverse computing platforms. The consistent NVIDIA driver and CUDA versions across all nodes with NVIDIA GPUs ensure comparability.

Chapter 5

Results and Discussion

In this chapter, the results of the experiments on the performance auto-tuning framework for GPU applications are presented and discussed. The experiments were designed to test different aspects of the auto-tuning process and to evaluate the effectiveness of the proposed methods.

The chapter begins with an examination of the baseline performance. This is done to understand how the chosen benchmark applications perform before any optimization is applied. Following this, the effects of using seeded configurations are analysed. These configurations, based on expert knowledge, are tested to determine if they can accelerate the tuning process.

Several experimental approaches are then explored. The application of boosted trees to reduce the search space is investigated to see how effectively it can narrow down the optimization parameters. The implementation of multi-fidelity autotuning techniques is studied to find a balance between tuning speed and optimization quality. Additionally, the evaluation of performance portability across different GPU architectures is conducted

to assess how well the optimized configurations transfer to various GPU hardware.

For each experiment, the results are presented, and their implications are discussed. The strengths and limitations of each method are analysed, and their effectiveness is compared. The significance of these results for GPU application optimization and auto-tuning in general is also considered.

This chapter consolidates all the research work, demonstrating the effectiveness of the auto-tuning framework and contributing to the ongoing research in optimizing GPU application performance.

5.1 Baseline Performance Analysis

In this section, the baseline performance of the autotuning framework applied to CMSSW (CMS Software) using OpenTuner is presented. The focus is placed exclusively on throughput as the primary performance metric. Various predefined search techniques available in OpenTuner, along with several newly implemented search techniques, are compared. The baseline experiments are designed to establish a performance benchmark for CMSSW autotuning using OpenTuner, compare the effectiveness of different search techniques in optimizing CMSSW throughput, and identify the most promising search strategies for further improvement. The search techniques are evaluated based on maximum throughput achieved, time taken to reach the best results, and the distribution of search configurations explored. Through this analysis, insights into the behaviour of various search strategies when applied to CMSSW are gained, providing a foundation for the improvements introduced in subsequent sections.

5.1.1 Maximum Throughput

The initial analysis compares the performance of different autotuning techniques across several GPUs. In Table 5.1, the maximum throughput achieved

by each technique, measured in events per second, is listed alongside the baseline performance for each GPU.

Table 5.1: Maximum throughput (events/second) achieved by different auto-tuning techniques across GPU models

Technique	T4	A10	L4	L40S
Baseline	241.74	430.03	526.10	903.05
Random Search	246.33	452.28	523.34	915.84
Greedy Search	245.65	460.60	535.65	922.34
Genetic Algorithm	239.85	**461.62**	**537.03**	920.22
Particle Swarm Optimization	246.14	455.13	530.22	918.36
Differential Evolution	245.23	457.52	523.62	919.71
AUC Bandit	**247.16**	459.45	535.76	**922.42**
Basin Hopping	241.68	451.79	528.49	916.51
Bayesian Optimization	243.19	450.25	521.27	919.54

On the T4 GPU, only modest improvements are yielded by the auto-tuning techniques, with a maximum throughput gain of 2.2% achieved by the AUCBandit technique (247.16 events/second compared to the baseline of 241.74 events/second). This limited improvement suggests that the T4's baseline configuration, which was manually optimized within CMSSW, is already near-optimal for this architecture. Therefore, there is less room for auto-tuning to enhance performance further.

Despite being a newer generation GPU, similar behaviour to the T4 is exhibited by the L4, with modest improvements from auto-tuning. A maximum throughput gain of 2.1% is achieved by the Genetic Algorithm (537.03 events/second compared to the baseline of 526.10 events/second). The similarity in results between the L4 and T4 GPUs can be attributed to their shared architectural characteristics. Both GPUs are designed as low-profile, inference-focused processors with similar power constraints (70W for T4 and 72W for L4). As a result, configurations optimized for the T4 are transferable and perform well on the L4, leaving less opportunity for autotuning

to find significant improvements.

In contrast, substantially larger gains from auto-tuning are demonstrated by the A10 GPU. A maximum throughput of 461.62 events/second is achieved by the Genetic Algorithm, representing a 7.3% improvement over the baseline of 430.03 events/second. This significant increase indicates that the T4-optimized baseline configuration is less suitable for the A10's architecture, providing more room for autotuning to identify better-performing configurations.

Similarly, meaningful improvements are shown by the L40S GPU, with the AUCBandit technique increasing throughput by 2.1% (from 903.05 to 922.42 events/second). Although the percentage gain is similar to that of the L4, the absolute increase in events per second is more substantial due to the higher baseline performance.

These larger improvements on the A10 and L40S GPUs highlight the effectiveness of auto-tuning when the baseline configurations are not fully optimized for a given architecture. The T4-optimized configurations do not fully exploit the capabilities of these GPUs, allowing auto-tuning techniques to discover configurations that better align with their architectural features.

5.1.2 Distribution of Configurations

The distribution patterns in Figure 5.1 can be analysed from both the perspective of search techniques and GPU architectures. Each GPU's unique characteristics and configuration space influence how different search techniques perform.

The T4 GPU, being the target architecture for CMSSW's manual optimization, shows interesting distribution patterns across all search techniques. Most notably, all techniques produce relatively narrow distributions, clustered around the 230-245 events/second range. The baseline performance of 241.74 events/second often lies at the upper end of these distri-

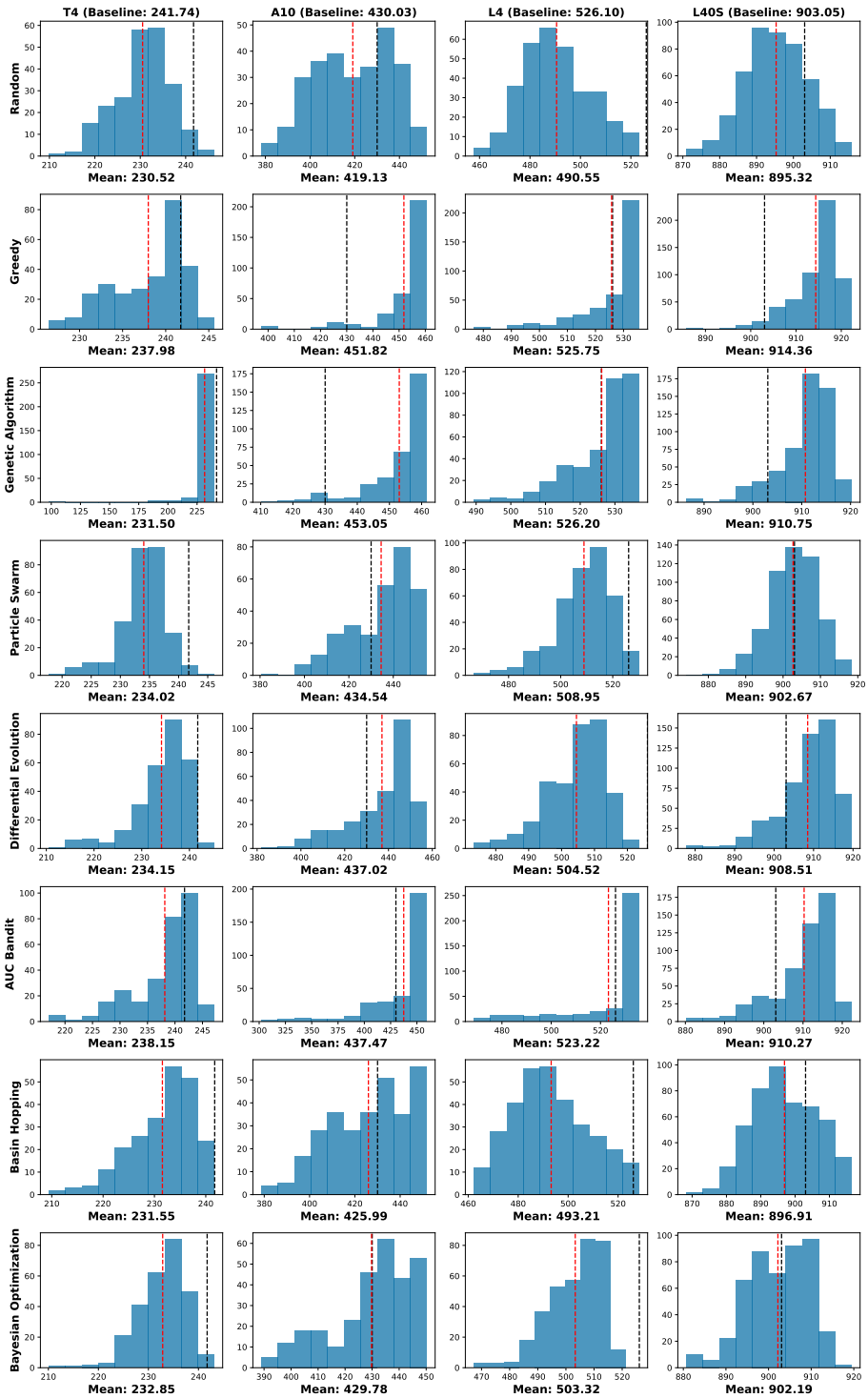

Figure 5.1: Distribution of configuration throughputs for different autotuning techniques across GPU architectures. The x-axis represent the frequency and y-axis is the throughput. The vertical black dashed line indicate the baseline performance for that GPU. The red dashed line is the distribution mean.

butions, indicating that search techniques struggle to surpass the manually optimized configuration when starting from random points. This suggests that the T4's configuration space may have a relatively well-defined optimal region, which aligns with its design as a low-profile, inference-focused GPU with a 70W power cap that inherently constrains the viable configuration space.

The L4 distributions shows patterns similar to the T4, which is expected given their architectural similarities and similar power constraints (72W for L4). However, the L4 shows slightly broader distributions. The baseline of 526.10 events/second, despite being optimized for T4, performs well relative to the tested configurations. This suggests that optimal configuration parameters may transfer well between architecturally similar GPUs, even when power-constrained.

The A10 GPU demonstrates different distribution patterns compared to the T4 and L4. All search techniques show wider distributions with clear right-skew, and consistently find configurations that outperform the baseline of 430.03 events/second. This wider spread suggests the A10's configuration space offers more opportunities for optimization, likely due to its higher power limit and different architectural characteristics. The right-skewed distributions indicate that while many random configurations perform poorly, there are opportunities for performance improvements through parameter tuning.

The L40S, despite having the highest absolute performance, shows distribution patterns that reveal the complexity of its configuration space. While all techniques achieve improvements over the baseline of 903.05 events/second, the distributions are wider and more varied across different search techniques. This suggests that the L40S's higher computational capabilities and power budget create a more complex optimization landscape. The variation in distribution shapes across different search techniques, particularly evident

in the contrast between AUCBandit's right-skewed distribution and Basin Hopping's more symmetric spread, indicates that different search strategies may be exploring distinct regions of the L40S's larger configuration space.

The L40S distributions, while showing the highest absolute throughput, reveal interesting patterns in how search techniques explore its configuration space. Starting from random configurations, the techniques manage to find solutions that outperform the baseline of 903.05 events/second. The AUCBandit technique shows the best distribution, with a mean of 910.27 events/second and a higher concentration of configurations in the upper performance range. This indicates that despite the T4-optimized baseline's reasonable performance on the L40S, the search techniques can effectively discover better configurations suited to the L40S's architecture.

These distribution patterns highlight how different search techniques explore and exploit the configuration space when starting from random configurations. The fact that many techniques independently discover configurations near or exceeding the manually optimized baseline demonstrates the effectiveness of autotuning in finding good configurations, particularly when the target architecture differs from the one for which manual optimizations were developed.

Random search establishes a baseline for comparison, producing approximately normal distributions across all architectures. These distributions reveal the natural landscape of the configuration space, with means typically below the manually optimized baseline but occasionally discovering high-performing configurations. The spread of these distributions is particularly informative, showing wider variance on the A10, L4, and L40S compared to T4, suggesting these architectures have more diverse performance characteristics across their configuration spaces.

The Greedy algorithm shows strongly right-skewed distributions, clearly showing in the A10 and L4 results. This skew reflects the algorithm's fun-

damental characteristic of immediately pursuing any improvement found, leading to clusters of configurations in higher-performing regions. However, the little spread in the distributions can be attributed to some parameters that affect the performance significantly more than the others.

The Genetic Algorithm produces notably different distributions across architectures. For the T4, it shows an unusually concentrated distribution near the baseline performance, while maintaining broader distributions on other architectures. This varying behaviour might suggest differences in the fitness landscape across architectures, though further investigation would be needed to definitively attribute this to issues of population diversity or other factors (Leung, Gao, & Xu, 1997). The broader distributions seen on A10 and L40S indicate more extensive exploration of these architectures' configuration spaces.

Particle Swarm Optimization demonstrates relatively consistent distribution patterns across different architectures, characterized by moderate spread and slight right-skew. This behaviour reflects PSO's social learning mechanism (Cheng & Jin, 2015), where particles explore individually while being influenced by the swarm's collective knowledge. The distributions suggest PSO maintains a balanced approach between exploring new configurations and exploiting known good regions.

Differential Evolution shows similar characteristics to PSO, but with tighter distributions, more noticeable in the L4 and L40S results. This pattern suggests DE's population-based search narrows in on promising regions of the configuration space while maintaining enough exploration to avoid convergence to local optima.

AUCBandit produces consistently right-skewed distributions across all architectures, most notably on the A10 and L4. This pattern shows the effectiveness of its adaptive technique selection strategy, allowing it to combine exploratory and exploitative approaches based on their observed per-

formance. The algorithm shows particular success in identifying and focusing on high-performing regions while maintaining sufficient exploration to discover new promising configurations.

Basin Hopping exhibits more symmetric distributions compared to other techniques, indicating its characteristic pattern of local optimization combined with periodic jumps to escape local optima. The distributions show consistent spread across all architectures, suggesting BH maintains reliable exploration throughout the search process regardless of the underlying hardware characteristics.

Bayesian Optimization shows interesting variations in its distribution shapes across architectures. The distributions tend to be more concentrated on T4 and L4, while showing more exploration on A10 and L40S. This pattern suggests the algorithm's surrogate model adapts its uncertainty estimates based on the observed performance landscape, leading to different exploration-exploitation trade-offs across different architectures.

These distributions show how each technique's search strategies function in practice when exploring GPU configuration spaces. The variations across architectures indicate that the effectiveness of each approach depends not only on the algorithm's characteristics but also on the underlying hardware architecture and its associated configuration space.

5.1.3 Autotuning Convergence

The convergence behaviour of different search techniques across GPU architectures provides insights into their efficiency and effectiveness. Figure 5.2 presents a grid of scatter plots showing the throughput of tested configurations over time, with horizontal lines indicating the baseline performance for each GPU.

The T4 GPU presents a challenging optimization scenario, with a baseline of 241.74 events/second that proves difficult to surpass. This reflects the

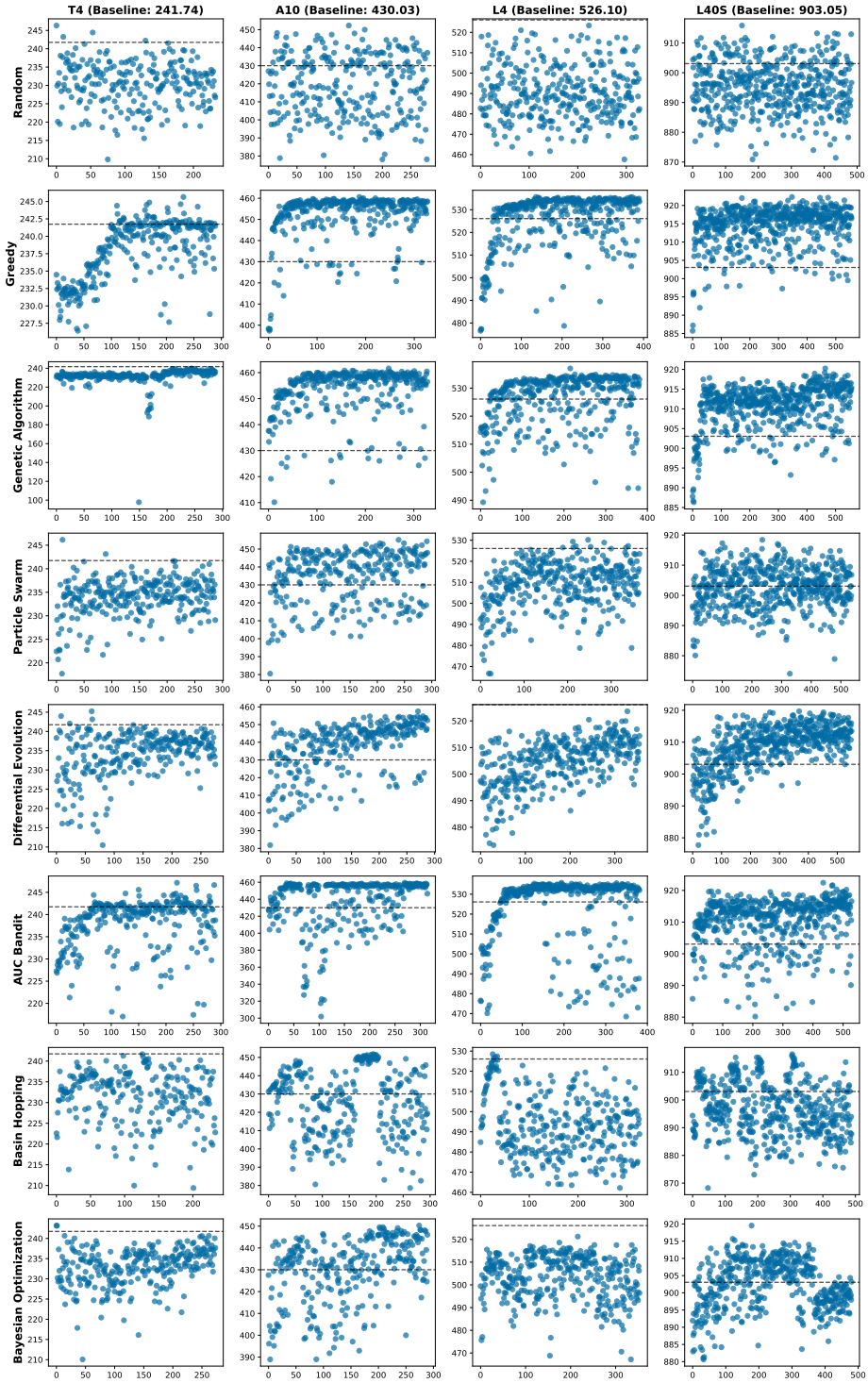

Figure 5.2: Convergence patterns of different autotuning techniques across GPU architectures. The x-axis represent the throughput and y-axis is the time. The horizontal dashed line indicate the baseline performance for that GPU.

effectiveness of CMSSW's manual optimization for this architecture. Most search techniques struggle to consistently exceed this baseline, though some might discover configurations with slightly better performance. AUCBandit and Greedy algorithms show the most promising convergence patterns, reaching near-baseline performance relatively quickly and occasionally finding superior configurations.

The L4 GPU shows similar patterns to the T4, with its baseline of 526.10 events/second being difficult to exceed consistently. This similarity in convergence behaviour confirms the architectural relationship between these GPUs. However, the L4 shows slightly more variance in configuration performance over time, suggesting a somewhat broader viable configuration space despite similar power constraints.

The A10 GPU demonstrates different convergence characteristics. With a baseline of 430.03 events/second, all search techniques successfully discover superior configurations. The Genetic Algorithm and AUCBandit show effective convergence patterns, quickly identifying configurations that outperform the baseline and maintaining this improved performance level throughout the search process.

The L40S presents noticeable improvements over baseline (903.05 events/second). Search techniques consistently discover better configurations, with most achieving significant improvements within the first 100-150 iterations. This rapid convergence to better configurations suggests that the T4-optimized baseline is substantially suboptimal for the L40S's architecture.

Random search shows no clear convergence pattern, as expected, but provides valuable information about the configuration space by revealing the range of achievable performance. Its scattered pattern across all architectures serves as a useful baseline for evaluating other techniques' convergence behaviour.

Greedy search demonstrates rapid initial convergence, particularly evi-

dent in its A10 and L40S results. This quick improvement aligns with the algorithm's nature of immediately exploiting better configurations. However, the technique sometimes shows performance plateaus, suggesting it may become trapped in local optima.

The Genetic Algorithm demonstrates distinct convergence behaviours across architectures. On the T4, it quickly finds and converges to a configuration with performance close to baseline, showing limited exploration afterward. In contrast, for the A10, it maintains broader exploration throughout the search process, consistently finding configurations that exceed the baseline. The L4 results show wider exploration but struggle to surpass the challenging baseline, while the L40S case demonstrates steady improvement with gradually reducing variation in performance over time.

Particle Swarm Optimization shows steady convergence characteristics across all architectures, with gradual improvement over time. Its convergence pattern suggests a balanced exploration-exploitation approach, though it sometimes takes longer to reach peak performance compared to more aggressive techniques.

Differential Evolution demonstrates relatively consistent convergence behaviour, showing steady improvement over time rather than dramatic early gains. This pattern suggests effective exploration of the configuration space, though perhaps at the cost of slower initial convergence.

AUCBandit shows among the most reliable convergence patterns across all architectures. It consistently achieves good performance early in the search process and maintains steady improvement. This behaviour validates its adaptive approach of selecting between different search strategies.

Basin Hopping's convergence pattern shows characteristic jumps in performance, reflecting its design of combining local optimization with periodic perturbations. While this leads to effective exploration, it sometimes results in less stable convergence compared to other techniques.

Bayesian Optimization exhibits distinct convergence behaviour characterized by initial exploration followed by more focused search phases. The technique shows concentrated sampling in regions it predicts as promising, visible in the L40S results where configurations cluster within specific performance bands. The technique appears particularly sensitive to its initial random sampling phase, which is evident in the L4 results where early poor-performing configurations may have influenced the surrogate model's predictions, leading to exploration of suboptimal regions and failure to exceed the baseline.

These convergence patterns reveal how different search strategies balance exploration and exploitation across varying architectural contexts. The results particularly highlight the challenge of optimizing for power-constrained architectures like T4 and L4, while demonstrating the potential for significant improvements on GPUs like A10 and L40S.

5.1.4 Configurations Validity

The percentage of valid versus invalid configurations generated by each search technique provides crucial insights into how well the algorithms learn and adapt to hardware constraints.

The T4 GPU (Figure 5.3) shows the highest proportion of invalid configurations across all techniques, reflecting its strict hardware constraints. With limited power budget (70W) and computational resources, many configurations fail due to excessive thread counts or resource requirements exceeding hardware capabilities. Random search performs particularly poorly on T4, generating the highest percentage of invalid configurations, as it has no mechanism to learn from or adapt to these hardware limitations.

The A10 (Figure 5.4) demonstrates the second-highest proportion of invalid configurations, despite its higher computational capabilities than the L4. This can be attributed to the significantly smaller L2 cache (6 MB)

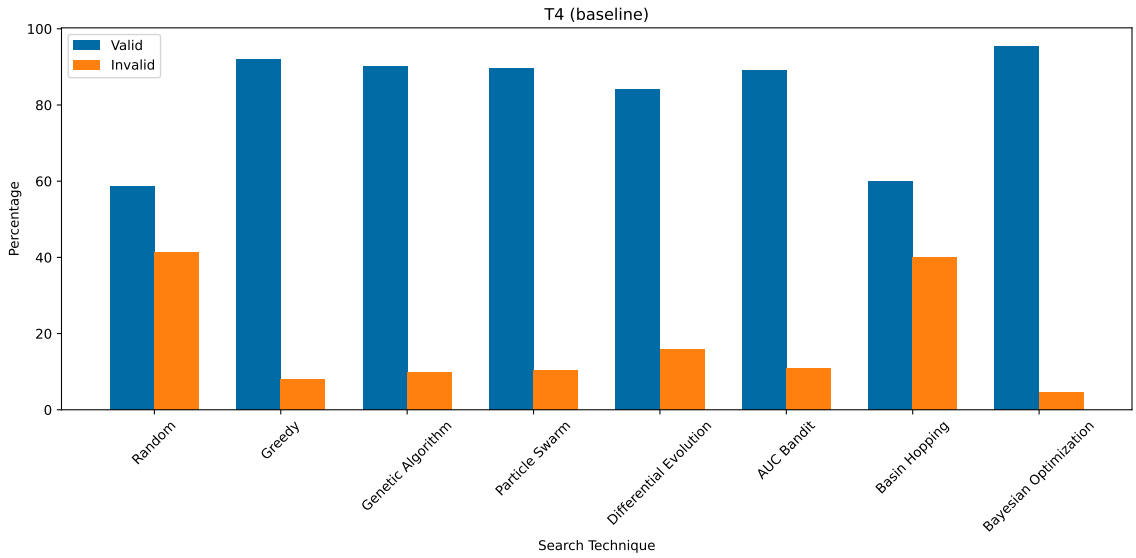

Figure 5.3: Percentage of valid and invalid configurations generated by different search techniques for the T4 GPU.

compared to the L4 (48 MB).

The L4 (Figure 5.5), despite sharing similar power constraints with T4 (72W), demonstrates a lower percentage of invalid configurations. This improved validity rate can be attributed to its larger register memory resources, allowing more configurations to execute successfully even with higher thread counts.

The L40S (Figure 5.6) show better validity rates across most techniques, reflecting its more hardware resources and power budgets. This GPU can accommodate a broader range of configurations, though Random search still generates the highest proportion of invalid configurations among all techniques.

Random search produces the highest percentage of invalid configurations across all architectures, showing the importance of intelligent search strategies in autotuning. Its blind sampling approach provides no mechanism to learn from previous failures or adapt to hardware constraints, resulting in repeated exploration of invalid configuration regions.

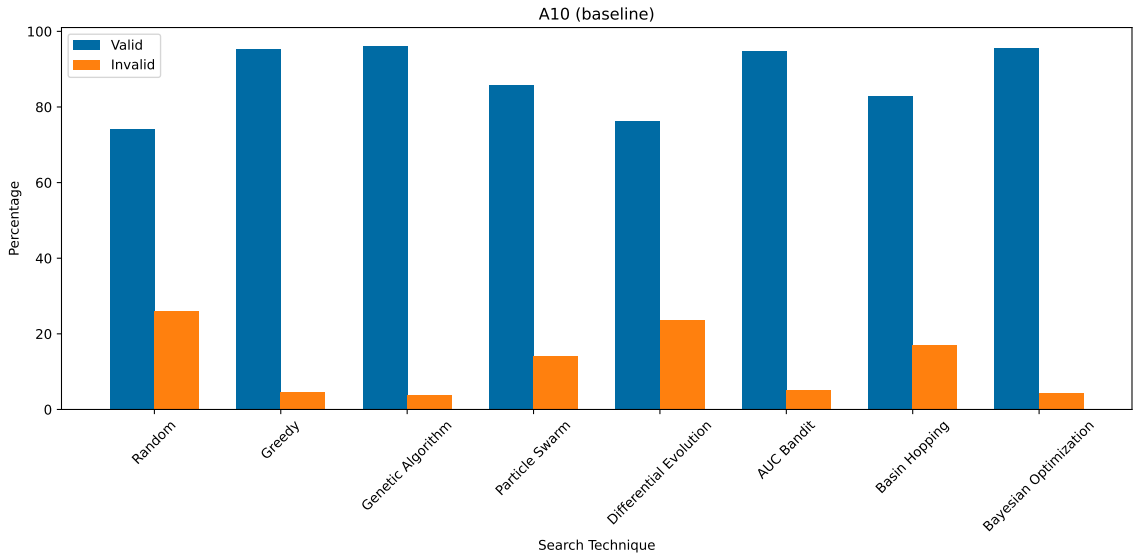

Figure 5.4: Percentage of valid and invalid configurations generated by different search techniques for the A10 GPU.

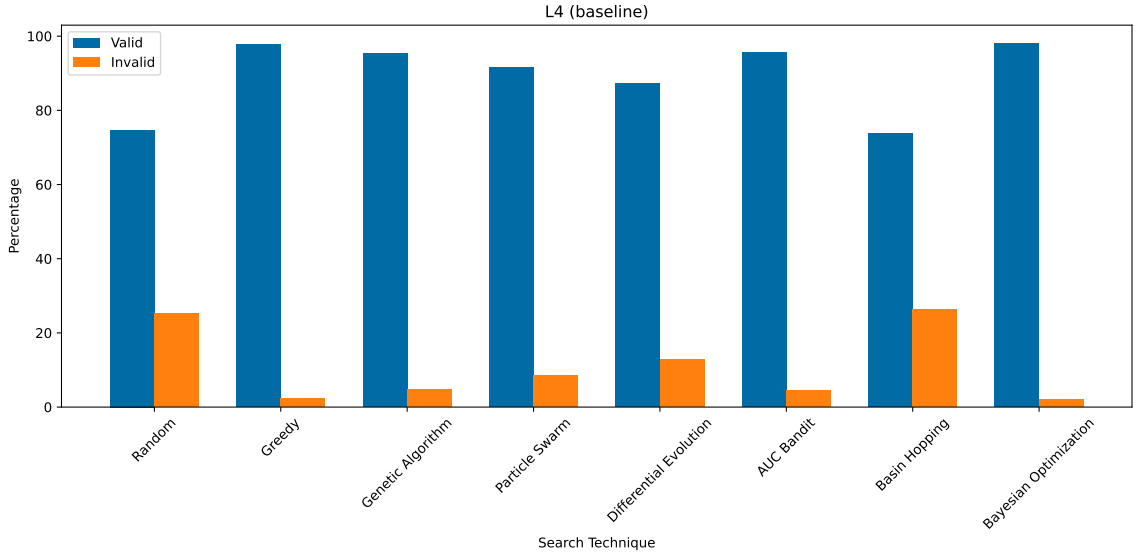

Figure 5.5: Percentage of valid and invalid configurations generated by different search techniques for the L4 GPU.

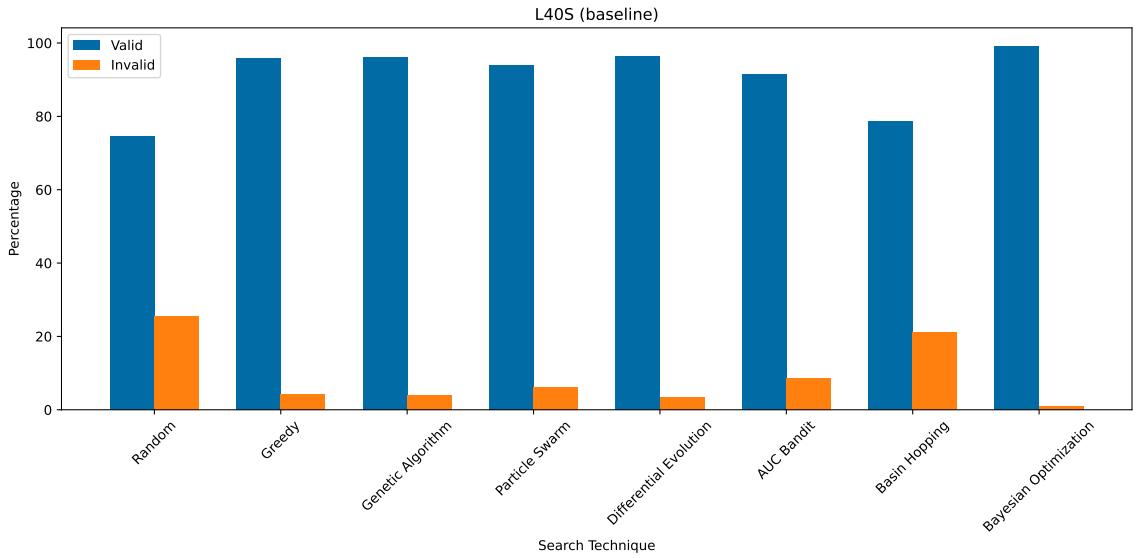

Figure 5.6: Percentage of valid and invalid configurations generated by different search techniques for the L40S GPU.

Basin Hopping also generates a high percentage of invalid configurations, second only to Random search. Its strategy of making periodic jumps in the configuration space often leads to exploring resource-intensive configurations that exceed hardware limitations.

AUCBandit and Genetic Algorithm demonstrate better adaptation to hardware constraints, maintaining higher validity rates, especially after initial exploration phases. This suggests these techniques effectively learn the bounds of acceptable configurations.

These validity patterns highlight both the challenges of constrained hardware environments and the importance of search strategies that can learn and adapt to these constraints.

5.2 Seed Configurations

The second experiment evaluates the effect of seeding search algorithms with expert knowledge, specifically using the manually-tuned T4 baseline

configuration as a seed configuration. This approach aims to leverage existing expertise while allowing algorithms to further optimize for each architecture. Table 5.2 presents the maximum throughput achieved by each technique across different GPUs.

5.2.1 Maximum Throughput

Table 5.2: Maximum throughput (events/second) achieved by different search techniques with seeded initialization across GPU architectures. The seeded configuration was manually optimized for T4. Bold values indicate the best performance for each GPU.

Technique	T4	A10	L4	L40S
Baseline	241.740	430.030	526.100	903.050
Exp. 1 Best	**247.160**	**461.620**	537.030	922.420
Greedy	246.652	460.872	532.255	923.702
Genetic Algorithm	245.881	459.791	532.113	921.404
Particle Swarm	242.078	455.394	525.956	916.254
Differential Evolution	241.809	459.876	528.877	920.302
AUC Bandit	245.555	461.021	**553.986**	**924.129**
Basin Hopping	240.975	451.651	524.990	913.910
Bayesian Optimization	241.470	457.834	525.845	915.919

Random initialization from the first experiment achieved the highest performance for both T4 (247.160 events/second) and A10 (461.620 events/second), surpassing the seeded search results. This suggests that for these architectures, the seed may have constrained the search space exploration, possibly trapping the algorithms in a local optimum near the baseline configuration.

However, the seeded approach shows its strength with the L4 and L40S architectures. On the L4, AUC Bandit achieved 553.986 events/second, significantly outperforming both the baseline (526.100 events/second) and the best random initialization result (537.030 events/second). This represents a

5.3% improvement over the baseline and a 3.2% improvement over random initialization's best performance.

Similarly, for the L40S, AUC Bandit reached 924.129 events/second, showing improvements over both the baseline (903.050 events/second, +2.3%) and the random initialization best result (922.420 events/second, +0.2%). While the margin of improvement is smaller for the L40S, the consistency of results across search techniques suggests that the seeded approach leads to more stable optimization outcomes.

AUC Bandit demonstrates better performance with seed configurations, particularly on L4 and L40S architectures, while Greedy search maintains strong, consistent performance across all platforms. The performance variation between techniques is notably reduced compared to random initialization. Basin Hopping and Particle Swarm techniques show relatively lower performance when initialized with seed configurations, suggesting their exploration strategies may be less suitable for refining pre-optimized starting points.

5.2.2 Distribution of Configurations

Figure 5.7 presents the distribution of throughput values achieved across different search techniques and GPU architectures when initialized with the T4-optimized seed configuration. Compared to random initialization, the distributions show distinct characteristics that reveal how seed configurations influences the search behaviour of different algorithms.

The seeded experiment produces notably narrower distributions across all techniques, indicating more concentrated exploration around the seed configuration. However, this concentration does not necessarily translate to improved performance, as mean throughput values often fall below those achieved with random initialization. Greedy search maintains the strongest mean performance across architectures (T4: 240.83, A10: 452.48, L4: 524.16,

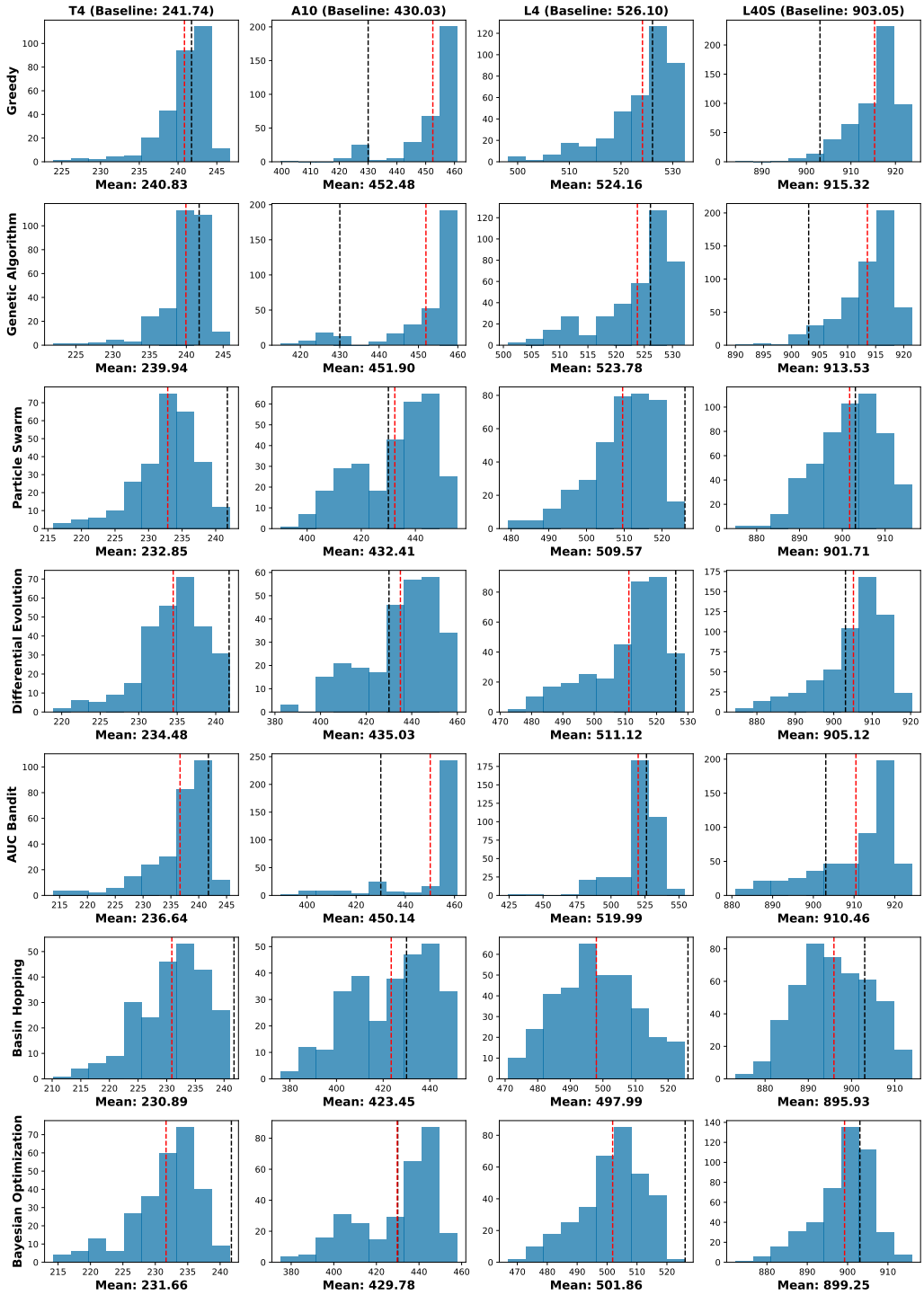

Figure 5.7: Distribution of throughput values achieved by different search techniques across GPU architectures when seeded with the T4-optimized baseline configuration. The black dashed line indicates the baseline performance, while the red dashed line shows the mean throughput for each distribution.

L40S: 915.32), effectively leveraging the seed configuration. The Genetic Algorithm shows increased exploitation behaviour on the T4, evidenced by a tighter distribution around the baseline configuration, suggesting strong influence from the seed configuration on its search pattern.

Particle Swarm Optimization and Differential Evolution maintain mean performances comparable to their random initialization results, but with slightly narrower distributions concentrated around the seeded configuration. AUC Bandit shows particularly interesting behaviour, while it successfully identified and exploited the optimal region on the A10, its mean performance on T4, L4, and L40S falls below the random initialization results. Basin Hopping and Bayesian Optimization consistently show lower mean performance compared to random initialization, likely due to their search patterns. These techniques initially focus on exploiting the region around the seed configuration before transitioning to exploration phases, for instance, Basin Hopping's exploitation period is constrained by its temperature parameter, after which it defaults to random search. This behaviour suggests that the seed configuration may actually restrict their ability to effectively explore the full configuration space.

5.2.3 Autotuning Convergence

The convergence patterns in Figure 5.8 show how different algorithms utilize the seed configuration across GPU architectures. Exploitation-focused techniques (Greedy, Genetic Algorithm, Differential Evolution, and AUC Bandit) exhibit similar behavioural patterns that vary by architecture. On T4 and L4, where the seed configuration is already near-optimal, these algorithms show limited exploitation without the gradual performance improvements seen in the baseline experiment. This suggests that the seed configuration effectively narrowed their search space. The A10 results show that these algorithms find the seed configuration suboptimal and quickly jump

to a better performance region before fine-tuning. This shows that the T4-optimized baseline configuration is less suitable for A10. On L40S, they demonstrate gradual improvement but with faster convergence compared to random initialization, indicating that the seed configuration provided a better starting point for optimization.

Particle Swarm Optimization shows different behaviour, appearing to perform random exploration in the close to the baseline configuration without clear convergence patterns. Basin Hopping and Bayesian Optimization show a two-phase pattern: initial exploitation around the seed configuration, followed by broader exploration. This transition is evident in their convergence curves, which show early clustering around the baseline performance before diverging into wider search patterns. This behaviour reflects their algorithmic design, where exploitation duration is controlled by parameters like temperature in Basin Hopping, after which they transition to more exploratory search strategies.

5.2.4 Configurations Validity

The seeded configuration experiment shows a general reduction in invalid configurations compared to random initialization across most search techniques (Figures 5.9, 5.10, 5.11, and 5.12). This improvement can be attributed to starting from an expert-provided valid configuration, which helps guide the search within reasonable regions of the configuration space. Exploitation-focused techniques like Greedy and Genetic Algorithm demonstrate particularly high validity rates, as they tend to make incremental modifications to the seed configuration.

However, techniques with stronger exploration drive show more variable validity rates. Basin Hopping, in particular, exhibits fluctuating validity percentages across different GPUs, reflecting its transition between exploitation and exploration phases. When in exploration mode, it generates

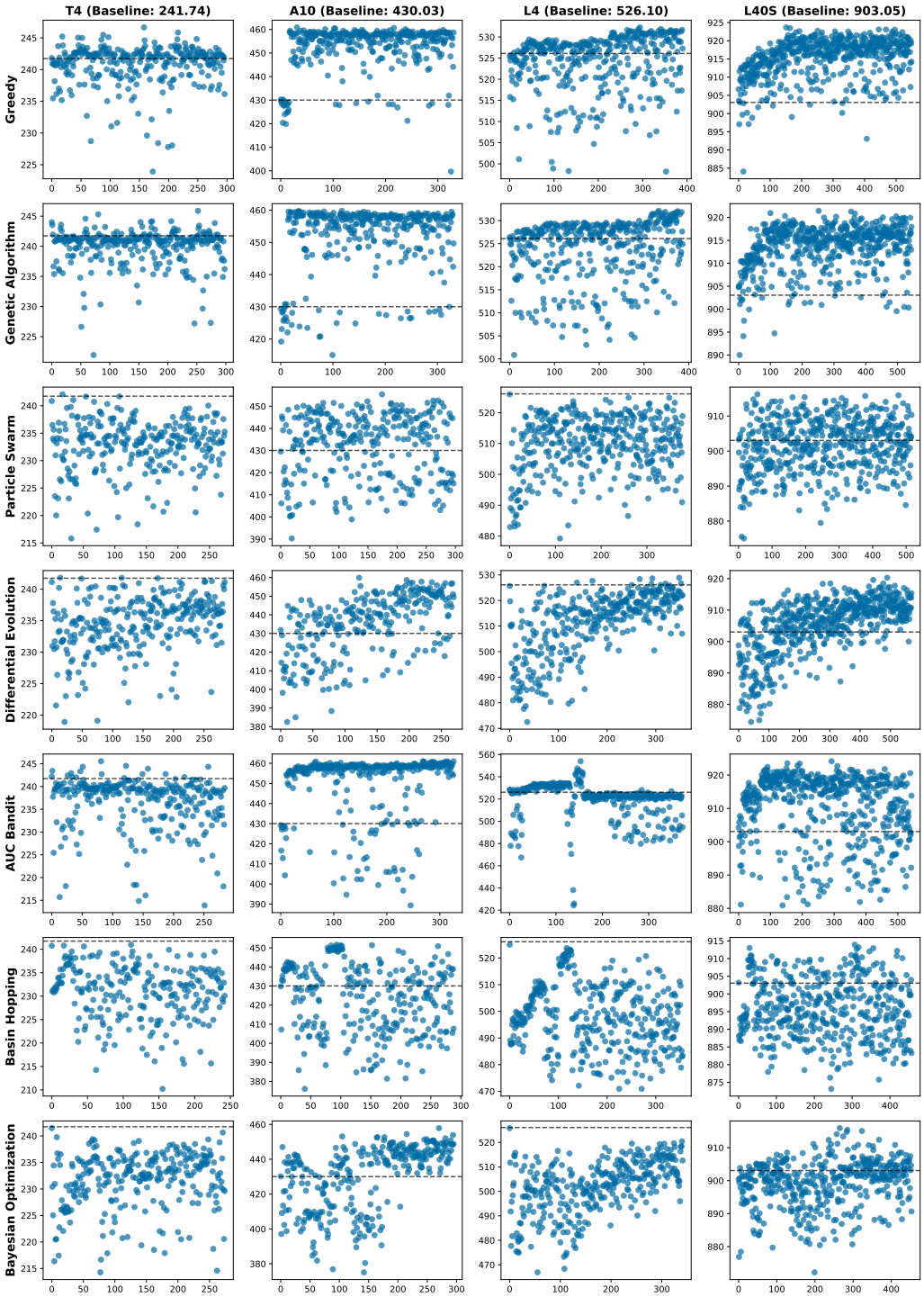

Figure 5.8: Convergence behaviour of different search techniques across GPU architectures when initialized with a seed configuration. Each plot shows the evolution of throughput over iterations, with the black dashed line indicating the baseline performance.

more invalid configurations as it moves away from the seed configuration's region. This pattern illustrates the inherent trade-off between maintaining high validity rates through exploitation of the seed configuration and potentially discovering better configurations through broader exploration of the search space.

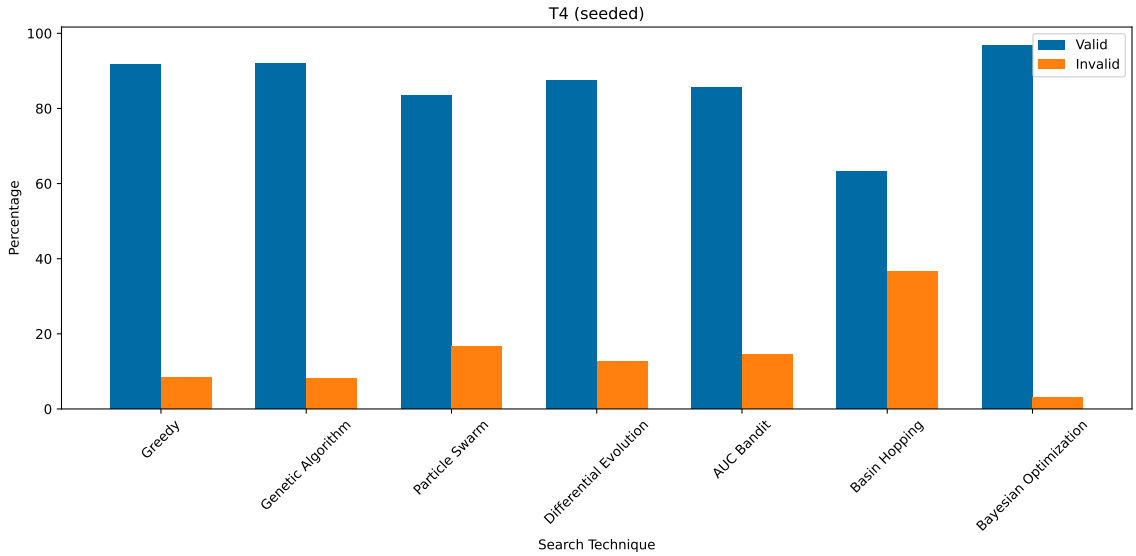

Figure 5.9: Percentage of valid and invalid configurations generated by different search techniques on T4 when initialized with a seed onfiguration.

5.3 Reducing Search Space using Boosted Trees

The application of boosted trees as a method to reduce the search space in the autotuning framework is investigated in this section. The implementation of the boosted trees algorithm is described, and its effectiveness in identifying the most influential optimization parameters for CMSSW is presented. The results demonstrate how this approach affects the speed and quality of the autotuning process compared to the baseline search methods. The potential of boosted trees to guide the search process towards more promising regions of the configuration space is analysed, and the implica-

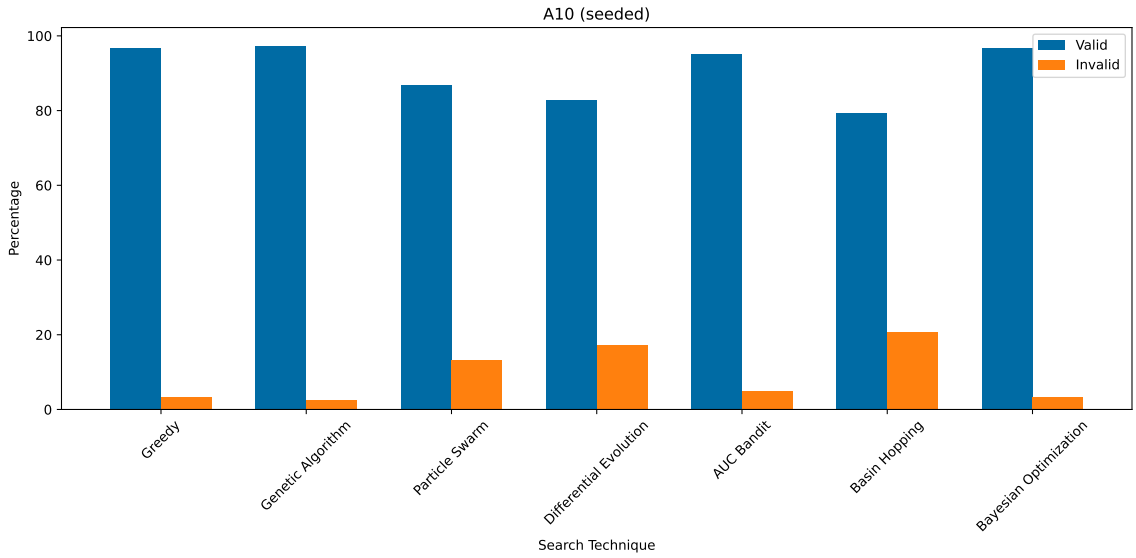

Figure 5.10: Percentage of valid and invalid configurations generated by different search techniques on A10 when initialized with a seed configuration.

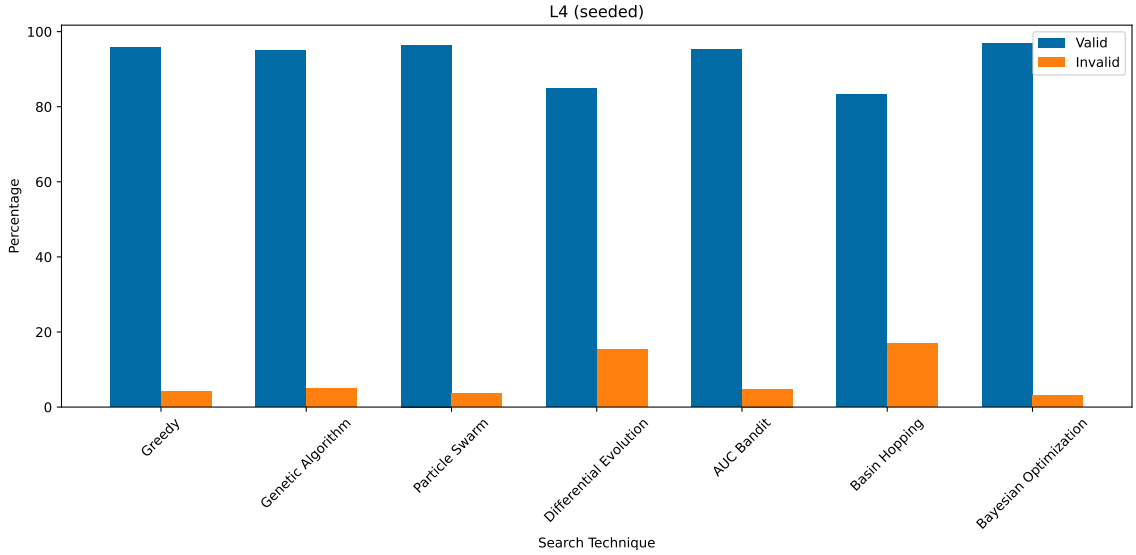

Figure 5.11: Percentage of valid and invalid configurations generated by different search techniques on L4 when initialized with a seed configuration.

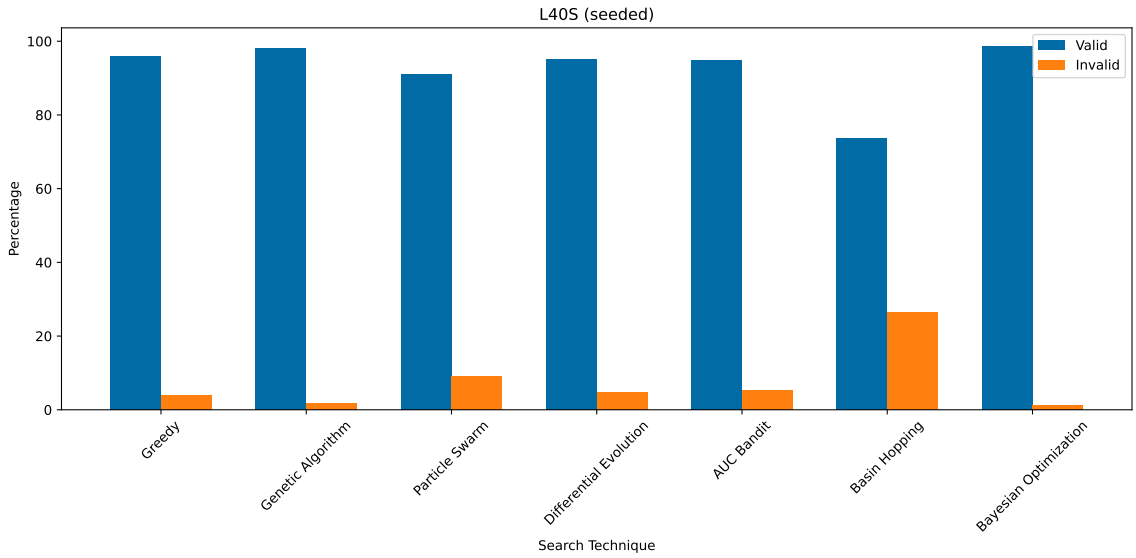

Figure 5.12: Percentage of valid and invalid configurations generated by different search techniques on L40S when initialized with a seed configuration.

tions for improving autotuning efficiency are discussed.

5.3.1 XGBoost for Feature Selection

XGBoost (Chen & Guestrin, 2016), a gradient boosting framework, is employed to learn the relationship between configuration parameters and performance outcomes from the previous experiments. This choice is supported by successful applications of XGBoost in similar optimization problems across the literature (H. Wang, Liang, Hancock, & Khoshgoftaar, 2024), where it has been effectively used for performance prediction and search space reduction in system configuration tasks. Through XGBoost's feature importance analysis, the number of tuning parameters is reduced from 29 to the 10 most influential parameters, significantly decreasing the dimensionality of the search space.

XGBoost was selected for parameter importance analysis based on its established advantages over alternative methods (Chen & Guestrin, 2016; H. Wang et al., 2024; Wu, Chen, Zhou, Wang, & Fan, 2020):

Advantages over Decision Trees

- High variance and overfitting issues in single decision trees are addressed by XGBoost's gradient boosting approach.
- Complex interactions between GPU configuration parameters are better captured through sequential tree building, where errors from previous trees are corrected.

Advantages over Random Forests

- Better prediction accuracy is achieved with fewer trees, which is crucial when performance measurements are computationally expensive.
- Imbalanced performance data, where optimal configurations are rare, is handled more effectively through gradient boosting.
- Overfitting is controlled more precisely through built-in regularization parameters.

Key Benefits

- Training speed and memory usage are optimized for large performance datasets.
- Multiple parameter importance metrics (feature importance and SHAP values) are provided.
- Strong empirical performance in heterogeneous computing systems is documented in the literature.

While Random Forests and Decision Trees were considered, XGBoost was determined to be the most suitable choice for GPU kernel autotuning due to its superior accuracy, efficiency, and robustness to noisy data.

The seed configuration is maintained as a starting point, but the search is now conducted over a reduced parameter space identified by the XGBoost model. This combination of machine learning-guided dimension reduction and expert knowledge aims to focus the search algorithms on the most impactful parameters while starting from known-good configurations, potentially leading to more efficient exploration of the configuration space.

For reproducibility purposes, the XGBoost model's configuration and dataset characteristics are detailed in Table 5.3. A moderate learning rate of 0.1 and max depth of 5 were chosen to prevent overfitting while maintaining model accuracy. The number of estimators was set to 100, providing sufficient model complexity while keeping computational requirements reasonable. Feature importance was calculated using the 'weight' metric, which measures the frequency of feature appearances in the decision trees, providing a straightforward measure of each parameter's influence on the model's decisions.

The datasets contain both valid and invalid configurations from previous experiments, with sizes reflecting each GPU's computational capacity within the fixed tuning time budget. The L40S, being the most powerful GPU, could evaluate the most configurations (7,821), while the T4 evaluated the fewest (4,133).

The feature importance analysis shows distinct patterns of influential parameters across different GPU architectures (Figures 5.13, 5.14, 5.15, and 5.16). This difference in important features suggests that optimal configuration strategies should be tailored to specific GPU architectures.

Some parameters demonstrate consistent importance across all architectures. Most notably, *FindClus*, which handles clustering hits based on energy deposits, appears as a critical parameter across all GPUs. This consistency aligns with the kernel's high computational demands.

However, the relative importance of other parameters varies significantly

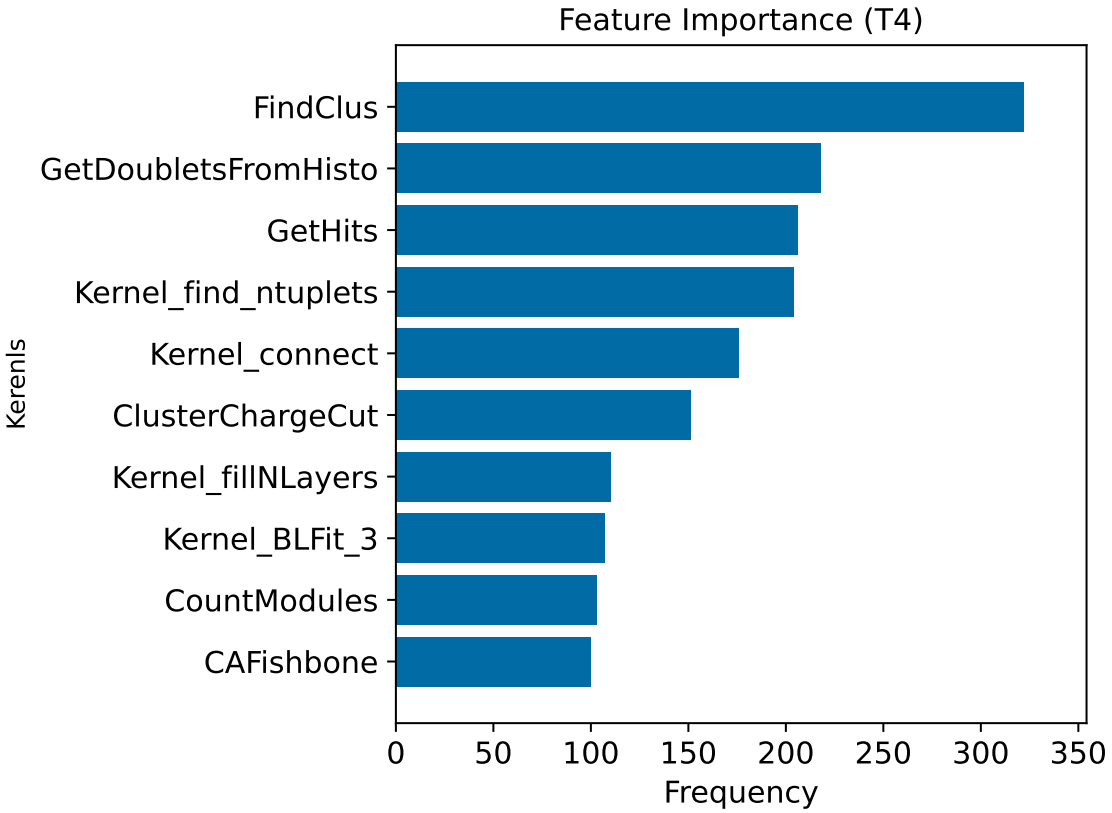

Figure 5.13: Feature importance ranking for T4 GPU based on XGBoost weight metric.

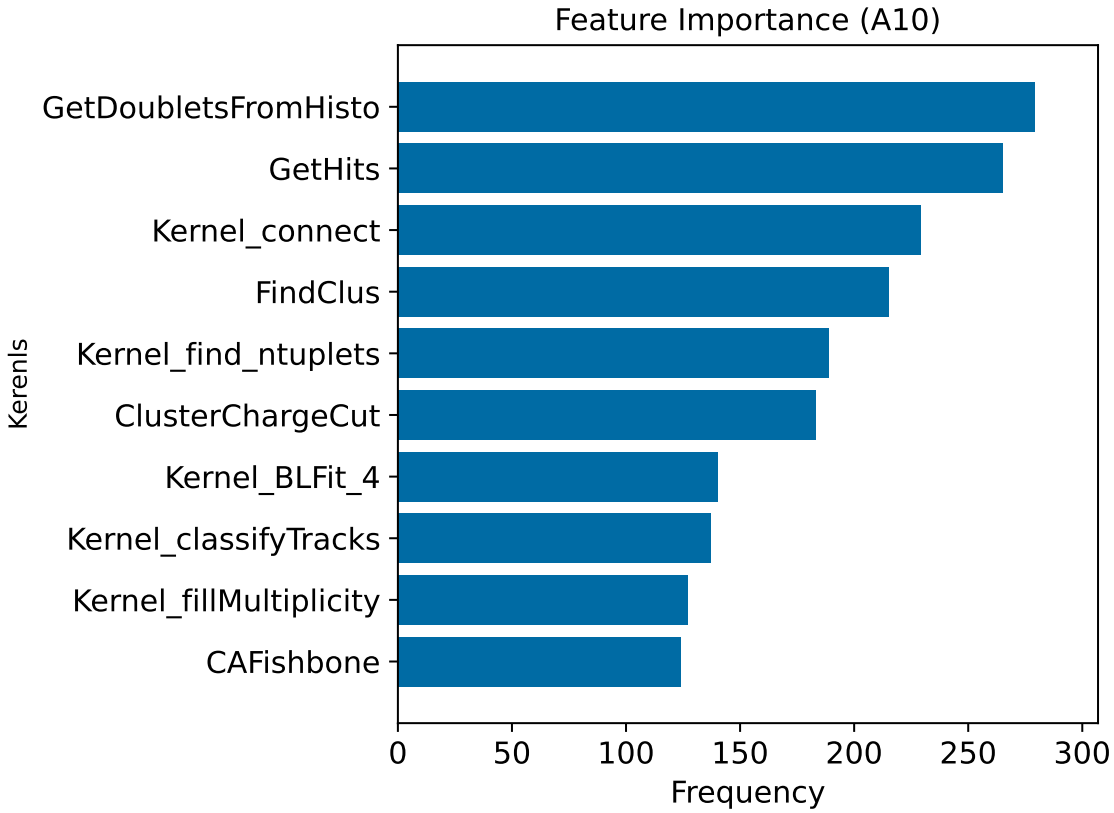

Figure 5.14: Feature importance ranking for A10 GPU based on XGBoost weight metric.

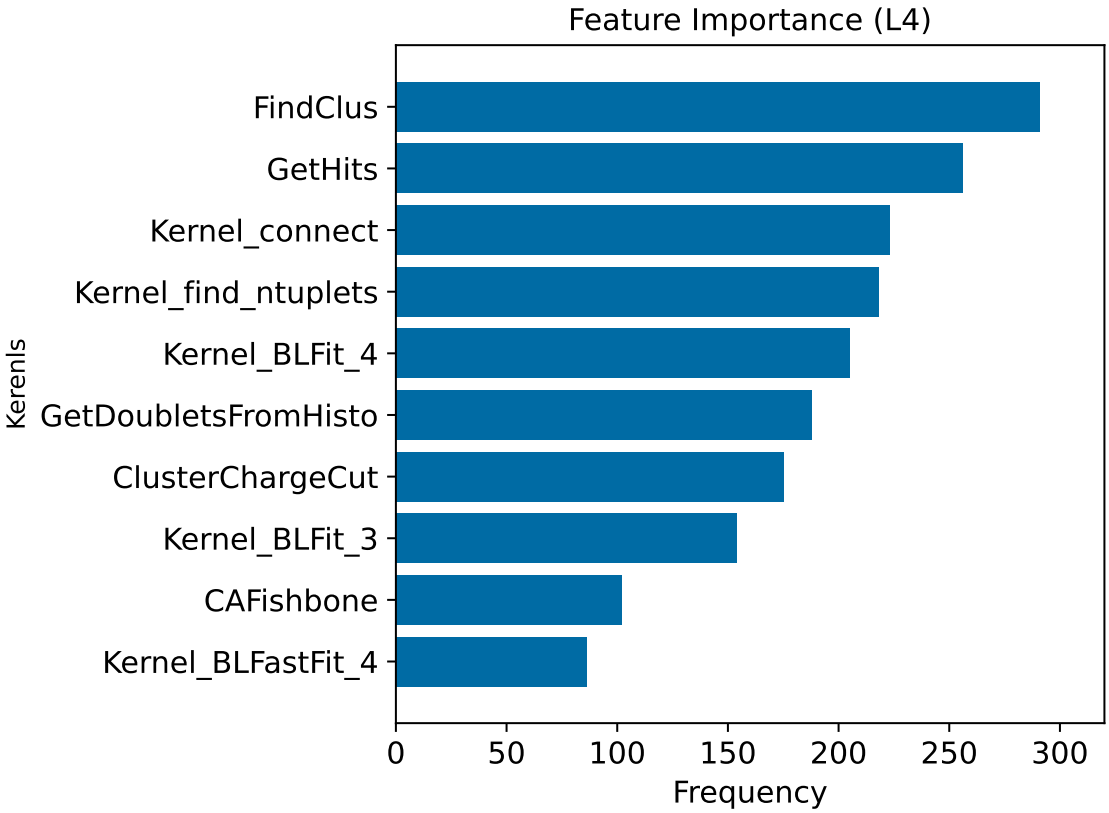

Figure 5.15: Feature importance ranking for L4 GPU based on XGBoost weight metric.

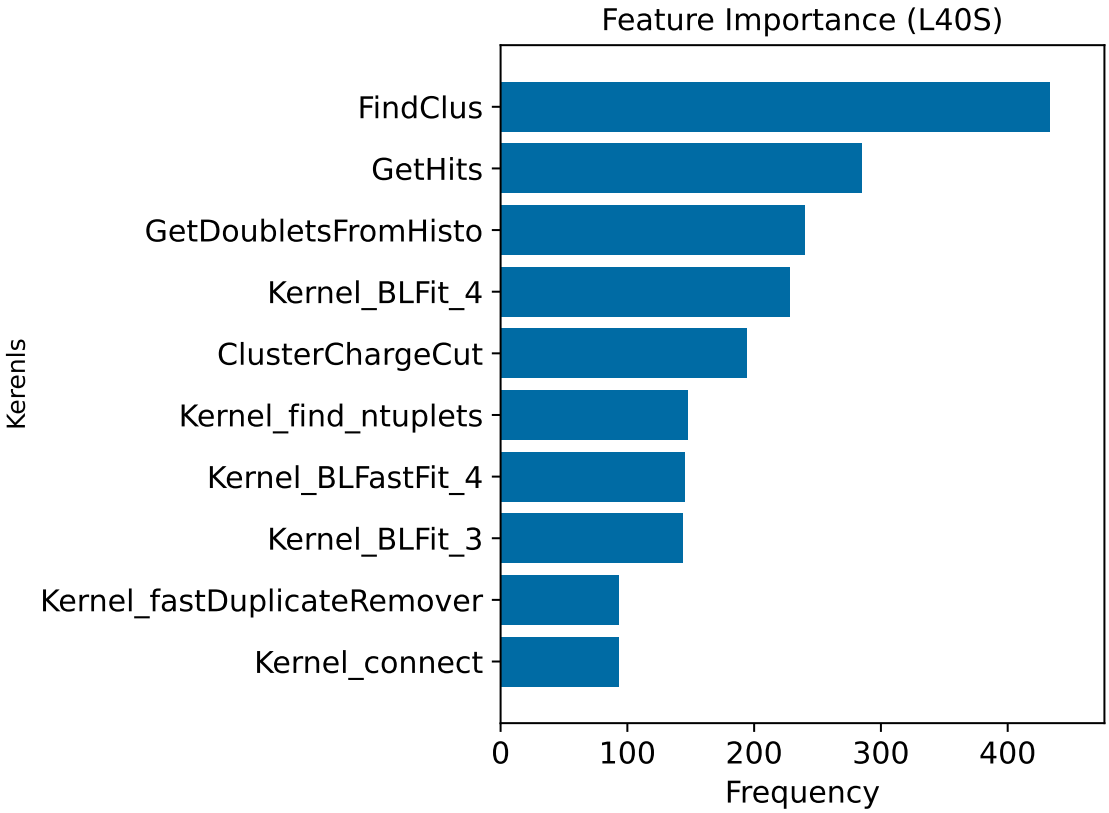

Figure 5.16: Feature importance ranking for L40S GPU based on XGBoost weight metric.

Table 5.3: XGBoost Model Configuration and Dataset Details

Model Parameters	
XGBoost Version	2.1.2
Random State	42
Number of Estimators	100
Learning Rate	0.1
Max Depth	5
Importance Type	weight
Max Number of Features	10
Dataset Details	
T4 Records	4,133
A10 Records	4,562
L4 Records	5,471
L40S Records	7,821
Training Split	90%
Test Split	10%

by architecture. For instance, $Kernel_connect$ appears as a top feature for A10 and L4 but is less significant for L40S. These variations likely reflect the different hardware characteristics and computational capabilities of each GPU architecture, suggesting that the impact of configuration parameters is heavily influenced by the underlying hardware.

5.3.2 Maximum Throughput

Table 5.4 presents the results of XGBoost-guided search compared to previous experiments. The XGBoost-guided approach with reduced parameter space shows mixed results across different GPU architectures. While it achieves the best performance on the A10 (463.478 events/second with Genetic Algorithm), it falls slightly short of the previous best results on other architectures.

Additionally, on the L4 and L40S, the seeded search from Experiment

Table 5.4: Maximum throughput (events/second) achieved by different search techniques with reduced parameter space using XGBoost across GPU architectures. Bold values indicate the best performance for each GPU.

Technique	T4	A10	L4	L40S
Baseline	241.740	430.030	526.100	903.050
Exp. 1 Best	**247.160**	461.620	537.030	922.420
Exp. 2 Best	246.652	461.021	**553.986**	**924.129**
Greedy	245.351	462.942	535.189	919.287
Genetic Algorithm	245.622	**463.478**	535.103	923.617
Particle Swarm	242.542	455.394	532.992	916.674
Differential Evolution	242.878	455.955	533.305	918.189
AUC Bandit	245.519	459.897	536.279	923.617
Basin Hopping	243.760	459.474	531.491	916.842
Bayesian Optimization	241.092	458.779	531.604	915.416

2 remains most effective (553.986 and 924.129 events/second respectively), while the random initialization from Experiment 1 maintains the best performance on T4 (247.160 events/second). Despite operating in a reduced parameter space (10 parameters instead of 29), the XGBoost-guided approach maintains competitive performance, with most algorithms achieving within 1-2% of the best known results. This suggests that the parameter reduction effectively preserved the most influential tuning parameters while simplifying the search process.

5.3.3 Distribution of Configurations

The distribution of throughput values (Figure 5.17) shows interesting patterns about the effectiveness of different search techniques under XGBoost-guided parameter reduction. Basin Hopping and Bayesian Optimization show improved mean performance compared to their seeded configuration results, suggesting that the reduced parameter space helps these algorithms

find better configurations more consistently.

A10 and L4 GPUs demonstrate slightly better mean throughput values across most search techniques, indicating that the XGBoost-guided parameter selection maybe effective for these architectures. Other search techniques maintain similar mean performance to their seeded configuration results, suggesting that the parameter reduction preserved the most important tuning factors.

5.3.4 Autotuning Convergence

The convergence behaviour of different search techniques under XGBoost-guided parameter reduction (Figure 5.18) reveals slight changes in exploration-exploitation patterns. Exploitative techniques such as Greedy and Genetic Algorithm now display broader exploration, suggesting that the reduced parameter space enables them to explore more configurations while maintaining their performance levels.

Basin Hopping notably demonstrates increased exploitation bands compared to previous experiments, which explains its improved mean performance. The tighter bands indicate more consistent discovery of good configurations within the reduced search space, suggesting that the XGBoost-guided parameter selection has helped focus the algorithm's search in promising regions.

5.3.5 Configurations Validity

The reduction in parameter space through XGBoost-guided selection has led to notably improved validity rates across all search techniques and GPU architectures (Figures 5.19, 5.20, 5.21, and 5.22). This improvement is particularly evident when compared to previous experiments, suggesting that the XGBoost model effectively identified parameters that contribute to con-

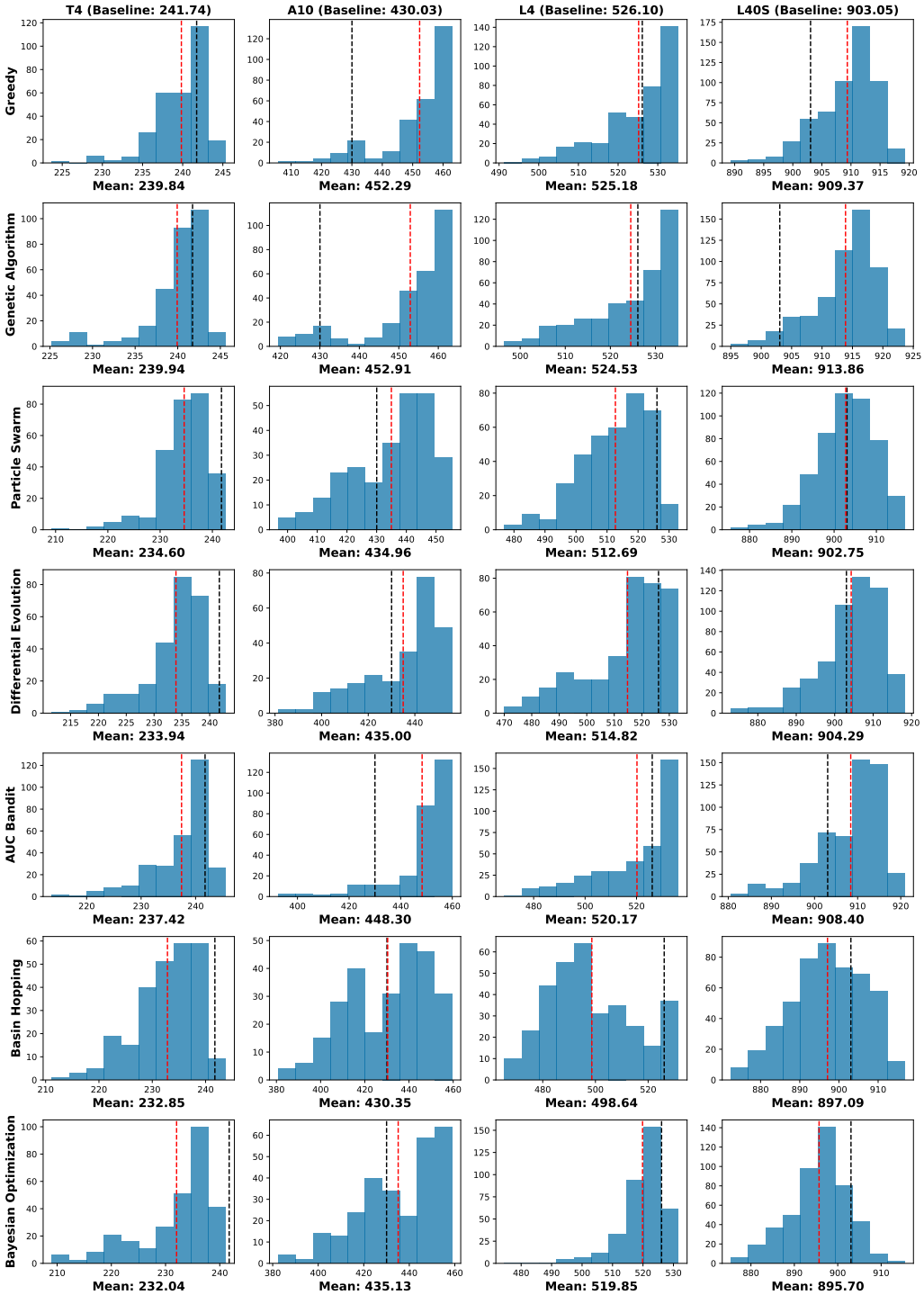

Figure 5.17: Distribution of throughput values achieved by different search techniques using XGBoost-guided search across GPU architectures. The black dashed line indicates the baseline performance, while the red dashed line shows the mean throughput for each distribution.

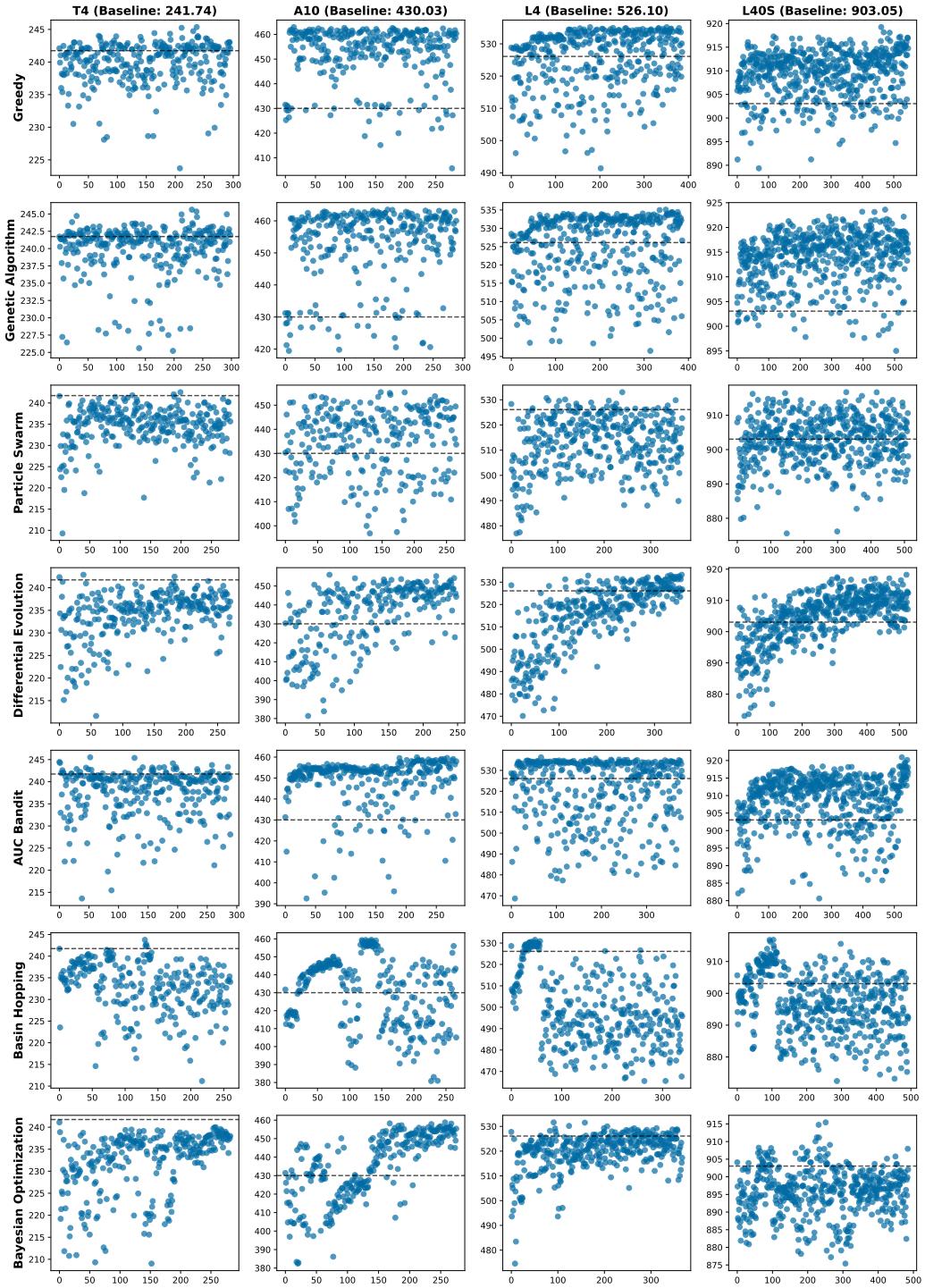

Figure 5.18: Convergence plots of different search techniques using XGBoost-guided search across GPU architectures. Each plot shows the evolution of throughput over iterations, with the black dashed line indicating the baseline performance.

figuration validity.

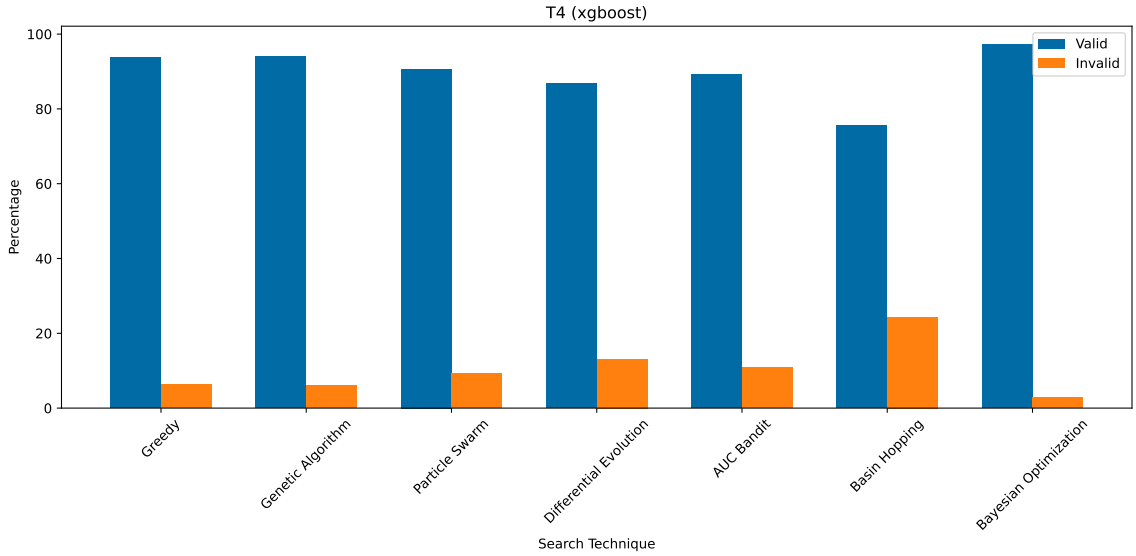

Figure 5.19: Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on T4 GPU.

The validity rates are consistence across different search techniques within each GPU architecture. Greedy and Genetic Algorithm maintain high validity rates across all GPUs, while even typically more exploratory algorithms like Particle Swarm and Basin Hopping demonstrate increased validity rates. This consistency suggests that the reduced parameter space limits the search to more viable regions of the configuration space.

The improved validity rates also explain the enhanced mean performance observed in the distribution analysis, particularly for Basin Hopping and Bayesian Optimization. By spending less time evaluating invalid configurations, these algorithms can focus their search effort on optimizing within valid regions of the parameter space. This efficiency gain is particularly valuable given the computational cost of evaluating invalid configurations during the tuning process.

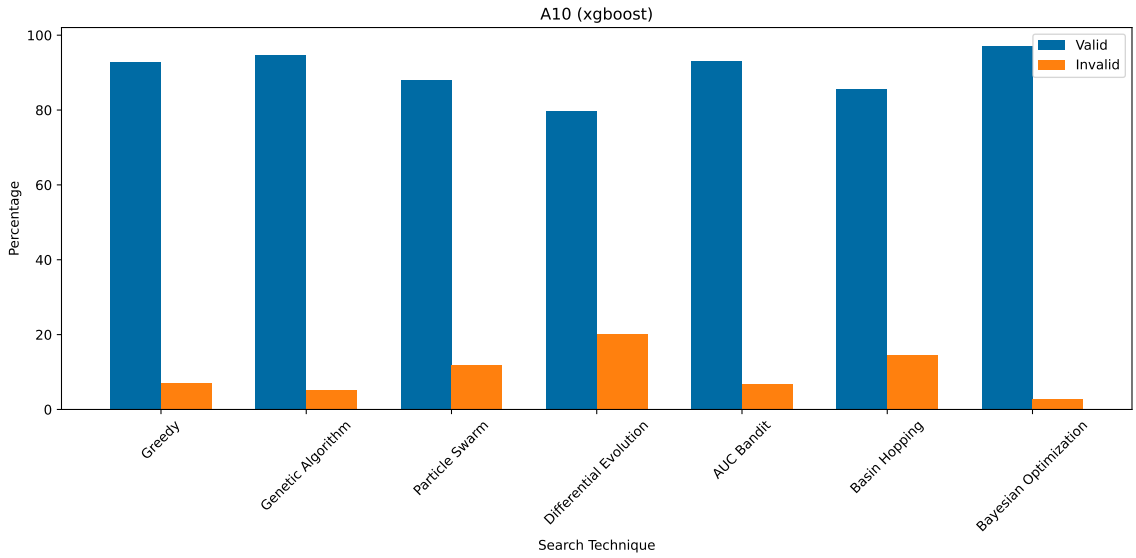

Figure 5.20: Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on A10 GPU.

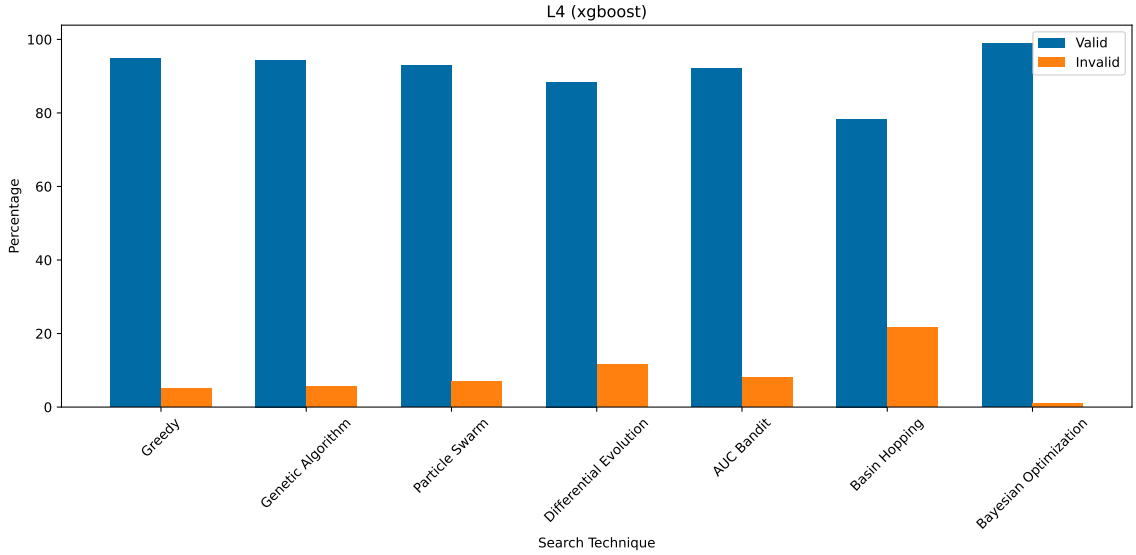

Figure 5.21: Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on L4 GPU.

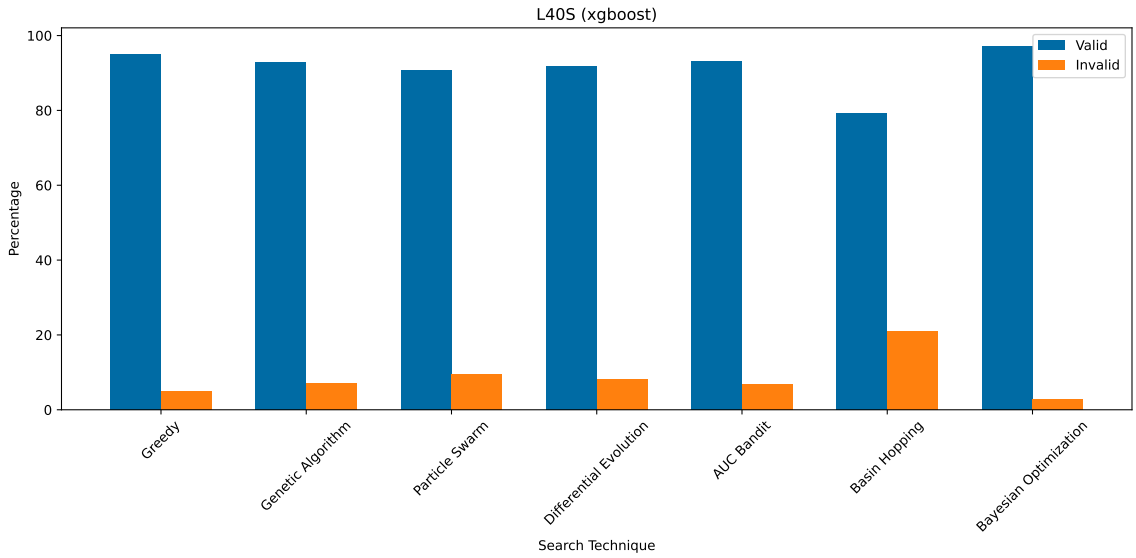

Figure 5.22: Valid and invalid configuration percentages for different search techniques using XGBoost-guided parameter reduction on L40S GPU.

5.4 Multi-fidelity Autotuning

The implementation and results of multi-fidelity autotuning techniques are examined in this section. The approach to balancing tuning speed and optimization quality by using lower-fidelity approximations in the early stages of tuning is presented. A comparison of different fidelity levels and their impact on the final optimization results is included, as well as an analysis of the trade-offs between tuning time and performance gains. The potential of multi-fidelity techniques to accelerate the autotuning process for CMSSW without significantly compromising the quality of the final solution is evaluated.

Two complementary multi-fidelity approaches have been implemented to maximize exploration efficiency during the autotuning process: reduced events count and simplified framework execution.

5.4.1 Events Count Reduction

In standard CMSSW benchmarking, performance measurements are processed using 10,000 events to ensure accurate significant measurements. As shown in Figure 5.23, when the events count is reduced to 1,000 during the exploration phase, the configuration evaluation rate is increased by 2-3x across all GPU architectures. The throughput measurements obtained with reduced events show moderate correlation with full-event evaluations, with performance differences ranging from 0.2% to 3.5% across different GPU architectures (Table 5.5).

Table 5.5: Maximum throughput (events/second) achieved using different fidelity levels

GPU	Original	1K Events	Pixeltrack
T4	**247.160**	245.2	241.8
A10	**461.620**	460.6	459.6
L4	**553.986**	534.4	528.6
L40S	**924.129**	919.5	915.6

5.4.2 Pixeltrack Standalone Framework

The second approach is implemented through a minimal CMSSW framework called pixeltrack standalone (Ebrahim, 2024b), where non-essential components have been stripped away while preserving core elements required for pixel tracking GPU kernel execution. Through this simplified framework, configuration evaluation rates are increased by 5-8x compared to the original framework, enabling broader exploration of the parameter space. As demonstrated in Table 5.5, the performance measurements obtained through the pixeltrack standalone framework show consistent trends with the full framework, though with slightly lower absolute throughput values, particularly notable on the L4 GPU where the difference reaches 4.6%.

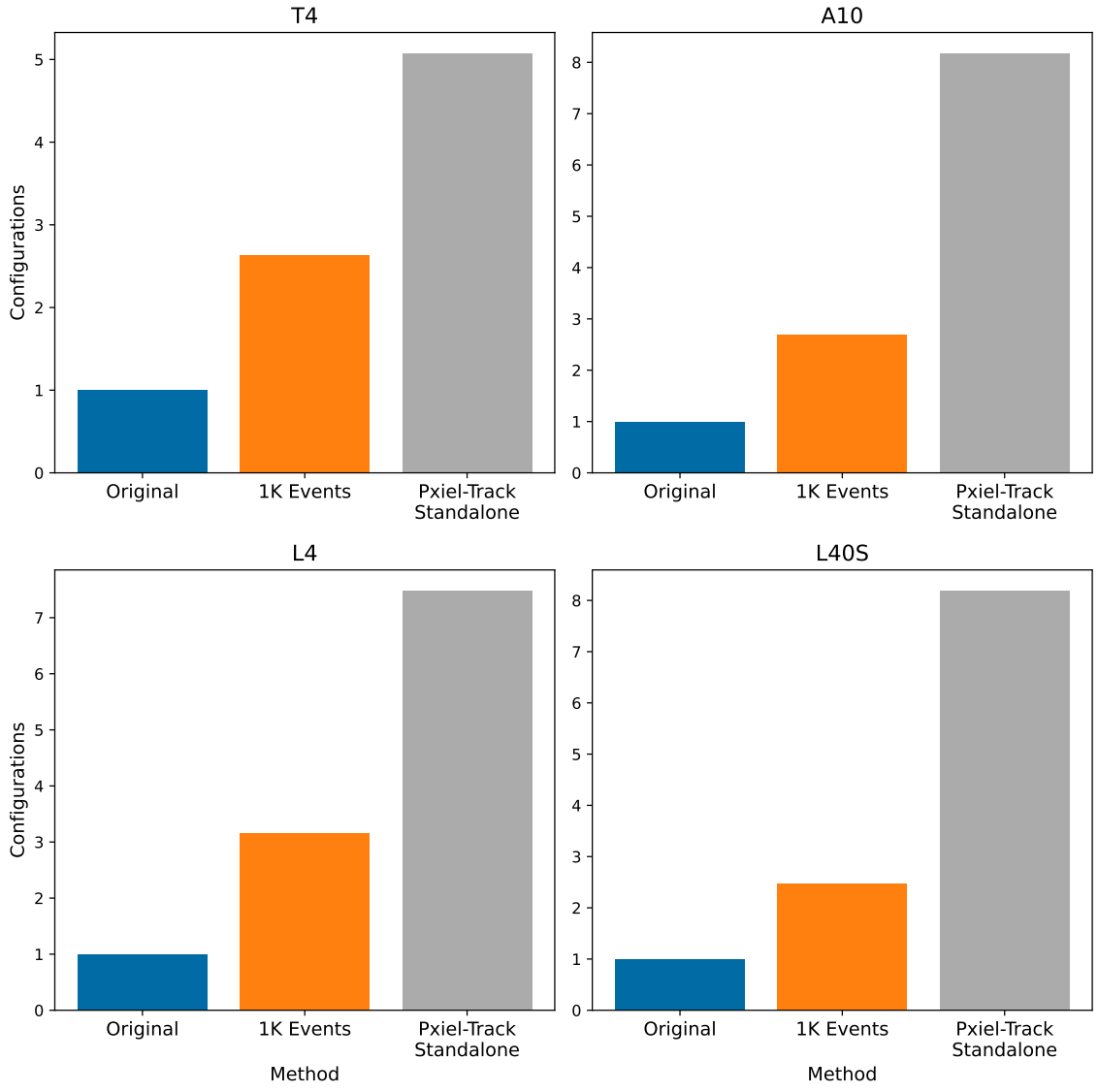

Figure 5.23: Configuration evaluation rates across different GPUs using original CMSSW (10,000 events), reduced events (1,000), and pixeltrack standalone framework approaches.

5.4.3 Integration with Full Framework Autotuning

The configurations discovered through these low-fidelity approaches can be leveraged in two ways for full framework optimization. Similar to Experiment 2, high-performing configurations from the low-fidelity evaluations can be used as seeds for full framework autotuning, providing better starting points for the search process. Alternatively, as demonstrated in Experiment 3, these configurations can be used to train an XGBoost model to identify important parameters and reduce the search space size. Both approaches benefit from the increased exploration capacity of low-fidelity evaluation while maintaining the accuracy of full framework tuning.

5.5 Performance Portability

In this final section, the performance portability of the autotuned configurations across different GPU architectures is evaluated. The results of applying optimized configurations from one GPU to others are presented, and the degree of performance maintained is analysed. The implications of these findings for developing more generalized optimization strategies that can work effectively across various hardware platforms are discussed. The challenges and opportunities in creating portable autotuning solutions for CMSSW are explored, with a focus on balancing architecture-specific optimizations with broader applicability.

5.5.1 Performance Portability Analysis

The performance portability of autotuned configurations across different GPU architectures has been evaluated using the metric proposed by Pennycook et al. (Pennycook, Sewall, & Lee, 2016). This application efficiency metric measures achieved performance as a fraction of the best observed

performance for each platform.

As shown in Figure 5.24, several key patterns emerge in the cross-architecture performance:

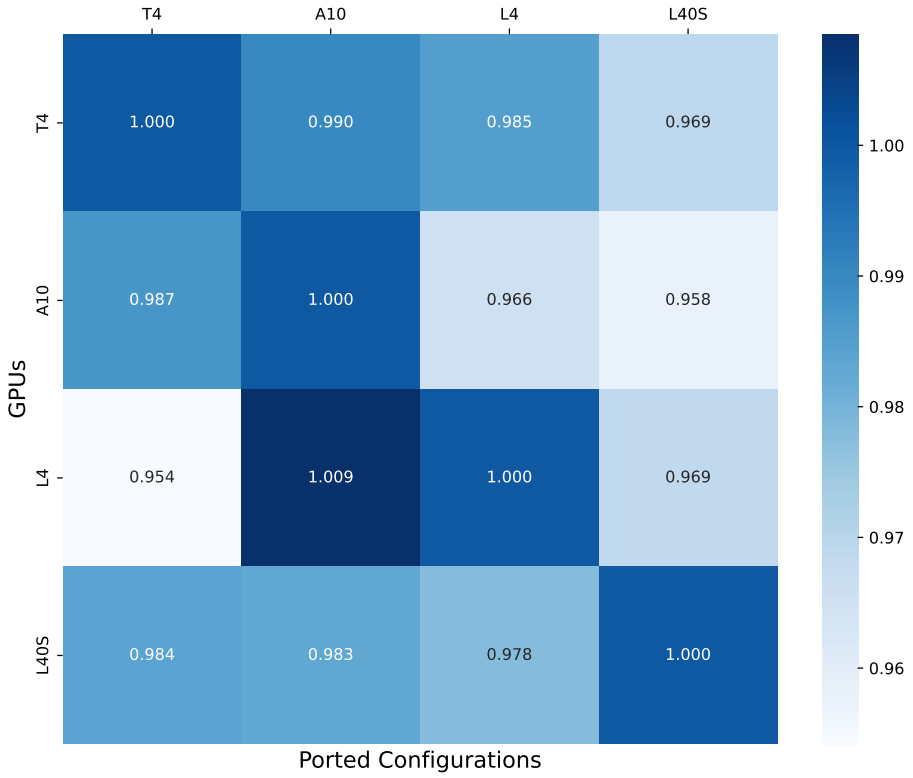

Figure 5.24: Performance portability matrix showing application efficiency when running different GPUs (rows) using configurations optimized for specific architectures (columns). Diagonal values represent the maximum achievable performance for each GPU using its respective optimized configuration.

On the T4 GPU (first row), near-optimal performance is achieved with A10-optimized settings (99%). However, reduced efficiency is observed with configurations optimized for newer architectures - 98.5% for L4-optimized and 96.9% for L40S-optimized configurations.

For the A10 (second row), decreased efficiency is seen when configurations optimized for newer architectures are used. Performance is reduced to

96.6% with L4-optimized settings and 95.8% with L40S-optimized configurations, which can be attributed to its smaller L2 cache (6MB) compared to these newer GPUs.

In the L4 results (third row), configurations optimized for the A10 are found to outperform the L4's own optimized configuration by 0.9% (100.9% efficiency). However, reduced efficiency (95.4%) is observed when T4-optimized configurations are used. This is particularly notable as good performance was achieved with the baseline manually-tuned T4 configuration, but not with the autotuning-optimized T4 configuration. Lower performance (96.9%) is also seen with L40S-optimized configurations.

The most consistent cross-architecture performance is demonstrated by the L40S (fourth row), where efficiency between 97.8% and 98.4% is maintained when configurations optimized for other architectures are used. This consistency can be attributed to the L40S's advanced architecture and larger resources.

A10-optimized configurations are found to be the most portable across all platforms, with high performance being maintained even on newer architectures. Conversely, the poorest portability is observed with L40S-optimized configurations, likely due to its advanced architecture and larger resource that does not exist in older or smaller GPUs.

These results highlight the challenges that are encountered in developing portable optimizations, especially when hardware characteristics vary significantly across architectures. While good cross-architecture performance can be achieved with some configurations, architecture-specific tuning is often required for optimal performance.

Chapter 6

Conclusion and Future Work

The advancement of heterogeneous computing architectures, particularly GPUs, has increased the need for efficient and portable optimization techniques for GPU kernels. This thesis aimed to enhance the performance and portability of GPU kernels within the CMS Software (CMSSW) framework by addressing four main research objectives. Several contributions were made to the field of auto-tuning and performance optimization.

6.1 Achieving the Research Objectives

The research objectives outlined in Chapter 1 have been successfully addressed through the contributions of this thesis:

Objective 1: Develop Efficient Search Techniques This objective was achieved through the implementation of multiple optimization strategies. The search space was effectively reduced through machine learning approaches using boosted trees, while computational overhead was minimized using multi-fidelity optimization techniques. The integration of Basin Hop-

ping and Bayesian Optimization further enhanced the framework's ability to find optimal configurations efficiently.

Objective 2: Integrate the Autotuning Framework with CMSSW This objective was fulfilled through the successful development and integration of an autotuning system within CMSSW. The framework was designed to handle the unique challenges of CMSSW, particularly the concurrent execution of multiple GPU kernels across different processes. This integration demonstrates the practical applicability of the developed solutions in a real-world high-energy physics environment.

Objective 3: Evaluate the Framework The framework's effectiveness was thoroughly evaluated through cross-architecture performance analysis. Performance improvements were demonstrated across multiple GPU architectures, validating the framework's capability to achieve performance portability in heterogeneous computing environments. The evaluation was particularly focused on scenarios where multiple kernels are executed simultaneously, reflecting real-world usage patterns in CMSSW.

Objective 4: Provide Insights into Autotuners Benchmarking Methodologies This objective was addressed through the implementation of established benchmarking methodologies from literature. The research contributed to the field by demonstrating how to ensure reproducible results through controlled testing environments, multiple measurement repetitions, and detailed documentation of experimental conditions. These practices provide a foundation for future research in GPU kernel autotuning.

The contributions of this thesis not only met but exceeded the initial research objectives by addressing previously unexplored areas, particularly in the context of multiprocess GPU kernel optimization. The developed framework provides a comprehensive solution for the high-energy physics community while establishing methodologies that can be applied in broader contexts. Table 6.1.

Table 6.1: Mapping of Research Objectives to Thesis Contributions

Research Objectives	Related Contributions

Objective 1	• Enhanced Search Space Optimization Techniques:
	  – Machine learning with boosted trees
	  – Multi-fidelity optimization
	  – Basin Hopping and Bayesian Optimization
	• Multi-Process GPU Kernel Autotuning Framework

Objective 2	• CMSSW GPU Optimization Framework • Multi-Process GPU Kernel Autotuning Framework
Objective 3	• Cross-Architecture Performance Analysis • Reproducible Benchmarking Implementation
Objective 4	• Reproducible Benchmarking Implementation • Cross-Architecture Performance Analysis

6.2 Implications of the Research

The contributions made in this thesis have significant practical implications across multiple domains and stakeholder groups:

High-Energy Physics Community

- **Immediate Performance Benefits**: The optimization of CMSSW GPU kernels directly impacts data processing capabilities at the LHC, potentially enabling more efficient particle detection and analysis.
- **Resource Utilization**: Enhanced GPU performance through autotuning helps maximize the return on investment in computing infrastructure, particularly important given the significant costs associated with high-energy physics facilities.
- **Future Scalability**: The framework provides a foundation for handling increasing data volumes expected in future LHC runs, helping ensure computational resources can keep pace with experimental capabilities.

HPC and Scientific Computing

- **Framework Adaptability**: The enhanced OpenTuner framework can be applied to other complex scientific applications beyond CMSSW, offering a template for GPU optimization in other domains.
- **Cost Efficiency**: Improved GPU utilization through autotuning can reduce the need for hardware upgrades, leading to significant cost savings in large-scale computing facilities.
- **Energy Efficiency**: Optimized GPU kernels typically result in better energy efficiency, contributing to more sustainable scientific computing practices.

Software Development Community

- **Development Practices**: The integration methodology demonstrated with CMSSW provides a model for incorporating autotuning into large, complex software systems without major architectural changes.
- **Performance Portability**: The framework's ability to optimize across different GPU architectures helps developers maintain performance as hardware evolves.
- **Optimization Workflows**: The automated nature of the framework reduces the manual effort required for performance optimization, allowing developers to focus on algorithmic improvements.

Research and Innovation

- **Machine Learning Integration**: The successful application of machine learning techniques in autotuning opens new avenues for AI-driven performance optimization.
- **Methodology Advancement**: The benchmarking and evaluation approaches developed provide a foundation for future research in autotuning and performance optimization.
- **Cross-Domain Applications**: The principles and techniques developed could be adapted for other domains requiring high-performance computing, such as climate modeling, computational biology, or financial modeling.

These implications extend beyond theoretical contributions, offering practical benefits that can be realized in current and future high-performance computing applications. The framework's successful integration with CMSSW demonstrates its viability in production environments, while its design principles provide a blueprint for similar implementations in other domains.

6.3 Limitations

While the research achieved its objectives, certain limitations were identified:

- **Limited Hardware Diversity:** The evaluations were conducted on a specific set of GPU architectures. The results may not generalize to all possible hardware platforms, and performance on other architectures remains to be validated.
- **Specific Machine Learning Models:** The search space reduction utilized boosted trees algorithms. Other machine learning models, such as deep learning or reinforcement learning techniques, were not explored and might offer different benefits.
- **Benchmarking Methodology Scope:** The benchmarking methodology employed was based on approaches proposed by other researchers. Alternative benchmarking methods were not investigated, which could provide additional insights.
- **Software Package Focus:** The autotuner was primarily used with CMSSW. Its effectiveness with other software packages was not tested, so its general applicability to different systems is not fully established.
- **Static Tuning Approach:** The tuning process was performed offline, and dynamic tuning mechanisms that adapt to changing workloads or system conditions were not implemented.
- **Single-Objective Tuning Focus:** The auto-tuning process focused primarily on optimizing for performance metrics such as execution time and throughput. Other objectives, such as power efficiency and energy consumption, were not explicitly considered. Multi-objective tuning, which involves balancing multiple objectives like performance

and power efficiency, was not explored. This limitation reduces the applicability of the framework in contexts where factors like power consumption are critical.

These limitations highlight areas where the research could be extended to enhance the applicability and robustness of the auto-tuning framework.

6.4 Future Work

Several areas remain for further exploration:

1. **Expansion to Other Hardware Architectures:** Extending the auto-tuning framework to support additional types of accelerators, such as Field-Programmable Gate Arrays (FPGAs) or emerging GPU architectures, could enhance its applicability.

2. **Experiment with Advanced Machine Learning and Reinforcement Learning Techniques:** Investigating other machine learning models, such as deep neural networks or reinforcement learning, might improve the predictive capabilities of search space reduction and optimization.

3. **Use of the Autotuner on Different Software Packages:** Applying the autotuner to other software systems would test its versatility and provide insights into its generalizability across different applications.

4. **Dynamic Tuning:** Developing dynamic tuning mechanisms that can adapt to changing workloads or system conditions in real-time could increase the framework's utility in dynamic computing environments.

5. **Real-Time Feature Selection:** The current batch XGBoost model could be replaced with streaming variants that enable continuous model

updates based on runtime performance data without requiring full retraining (Montiel et al., 2020).

6. **User-friendly Interfaces and Visualization Tools:** Enhancing the user interface and providing visualization tools for the tuning process could make the framework more accessible to users.

6.5 Concluding Remarks

This thesis addressed the key objectives of developing efficient search techniques, integrating an auto-tuning framework with CMSSW in a loosely coupled manner, evaluating its effectiveness, and providing insights into autotuner benchmarking methodologies. The contributions advance the understanding and practice of GPU kernel optimization, offering practical solutions to challenges in high-performance computing.

By bridging the gap between theoretical optimization techniques and practical implementation within a complex framework like CMSSW, this work demonstrates the benefits of auto-tuning in enhancing performance and portability. The insights and tools developed lay a foundation for future research and development, contributing to ongoing efforts to improve computational efficiency in high-energy physics and related fields.

As computational demands continue to grow and hardware architectures evolve, the importance of efficient and adaptable optimization techniques increases. This research represents a step toward meeting these challenges, providing advancements and practical tools that will benefit the scientific community and contribute to future innovations in the field.

References

Aguilera, P., Morrow, K., & Kim, N. S. (2014). Fair share: Allocation of gpu resources for both performance and fairness. In *2014 ieee 32nd international conference on computer design (iccd)* (pp. 440–447). doi: 10.1109/ICCD.2014.6974717

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th acm sigkdd international conference on knowledge discovery & data mining* (p. 2623–2631). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3292500.3330701` doi: 10.1145/3292500.3330701

AMD, Inc. (2024). *HIP programming model.* Online. Retrieved from `https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.0/understand/programming_model.html` (HIP 6.2.41133 Documentation)

Anderson, J. A., Lorenz, C. D., & Travesset, A. (2008). General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, *227*(10), 5342–5359. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0021999108000818` doi: https://doi.org/10.1016/j.jcp.2008.01.047

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J.,

O'Reilly, U.-M., & Amarasinghe, S. (2014). Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd international conference on parallel architectures and compilation* (p. 303–316). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2628071.2628092` doi: 10.1145/2628071.2628092

Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., & Silvano, C. (2018, sep). A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, *51*(5). Retrieved from `https://doi.org/10.1145/3197978` doi: 10.1145/3197978

Balaprakash, P., Dongarra, J., Gamblin, T., Hall, M., Hollingsworth, J. K., Norris, B., & Vuduc, R. (2018). Autotuning in high-performance computing applications. *Proceedings of the IEEE*, *106*(11), 2068–2083. doi: 10.1109/JPROC.2018.2841200

Balaprakash, P., Tiwari, A., & Wild, S. M. (2014). Multi objective optimization of hpc kernels for performance, power, and energy. In S. A. Jarvis, S. A. Wright, & S. D. Hammond (Eds.), *High performance computing systems. performance modeling, benchmarking and simulation* (pp. 239–260). Cham: Springer International Publishing.

Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., ... Scogland, T. R. (2019). Raja: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)* (pp. 71–81). doi: 10.1109/P3HPC49587.2019.00012

Ben-Nun, T., Gamblin, T., Hollman, D. S., Krishnan, H., & Newburn, C. J. (2020). Workflows are the new applications: Challenges in performance, portability, and productivity. In *2020 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)* (pp. 57–69). doi: 10.1109/P3HPC51967.2020.00011

Bocci, A. (2023). Cms high level trigger performance comparison on cpus and gpus. *Journal of Physics: Conference Series*, *2438*. Retrieved from https://api.semanticscholar.org/CorpusID:256897623

Bocci, A., Czirkos, A., Pilato, A. D., Pantaleo, F., Hugo, G., Kortelainen, M. J., & Redjeb, W. (2023). Performance portability for the cms reconstruction with alpaka. *Journal of Physics: Conference Series*, *2438*. Retrieved from https://api.semanticscholar.org/CorpusID:256897589

Bocci, A., Dagenhart, D., Innocente, V., Jones, C., Kortelainen, M. J., Pantaleo, F., & Rovere, M. (2019). Bringing heterogeneity to the cms software framework. *EPJ Web of Conferences*. Retrieved from https://api.semanticscholar.org/CorpusID:215548458

Bocci, A., Innocente, V., Kortelainen, M., Pantaleo, F., & Rovere, M. (2020). Heterogeneous reconstruction of tracks and primary vertices with the cms pixel tracker. *Frontiers in Big Data*, *3*. Retrieved from https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2020.601728 doi: 10.3389/fdata.2020.601728

Bocci, A., Jones, C., & Kortelainen, M. J. (2023). Performance of heterogeneous algorithm scheduling in cmssw. *Performance of Heterogeneous Algorithm Scheduling in CMSSW*. Retrieved from https://api.semanticscholar.org/CorpusID:258793923

Bocci, A., Kortelainen, M. J., Innocente, V., Pantaleo, F., & Rovere, M. (2020). Heterogeneous reconstruction of tracks and primary vertices with the cms pixel tracker. *Frontiers in Big Data*, *3*. Retrieved from https://api.semanticscholar.org/CorpusID:221377257

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., & Hanrahan, P. (2004). Brook for gpus: stream computing on

graphics hardware. In *Acm siggraph 2004 papers* (p. 777–786). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1186562.1015800 doi: 10.1145/1186562.1015800

Chaudhary, S., Ramjee, R., Sivathanu, M., Kwatra, N., & Viswanatha, S. (2020). Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the fifteenth european conference on computer systems.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3342195.3387555 doi: 10.1145/3342195.3387555

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 44–54. Retrieved from https://api.semanticscholar.org/CorpusID:206915521

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (p. 785–794). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2939672.2939785 doi: 10.1145/2939672.2939785

Cheng, R., & Jin, Y. (2015). A social learning particle swarm optimization algorithm for scalable optimization. *Information Sciences*, *291*, 43–60. Retrieved from https://www.sciencedirect.com/science/article/pii/S0020025514008366 doi: https://doi.org/10.1016/j.ins.2014.08.039

CMS Collaboration. (2006). *CMS Physics: Technical Design Report Volume 1: Detector Performance and Software*. Geneva: CERN. Retrieved from https://cds.cern.ch/record/922757 (There is an

error on cover due to a technical problem for some items)

CMS Collaboration. (2021). *The Phase-2 Upgrade of the CMS Data Acquisition and High Level Trigger* (Tech. Rep.). Geneva: CERN. Retrieved from https://cds.cern.ch/record/2759072 (This is the final version of the document, approved by the LHCC)

CMS Collaboration. (2024). *Cms offline software.* https://cms-sw.github.io/. CERN. Retrieved from https://cms-sw.github.io/ (Documentation for CMS Software (CMSSW))

Curtis, S. A. (2003). The classification of greedy algorithms. *Sci. Comput. Program.*, *49*, 125–157. Retrieved from https://api.semanticscholar.org/CorpusID:41336427

Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., ... Vetter, J. S. (2010). The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units* (p. 63–74). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1735688.1735702 doi: 10.1145/1735688.1735702

Demeshko, I., Maruyama, N., Tomita, H., & Matsuoka, S. (2013). Multi-gpu implementation of the nicam atmospheric model. In I. Caragiannis et al. (Eds.), *Euro-par 2012: Parallel processing workshops* (pp. 175–184). Berlin, Heidelberg: Springer Berlin Heidelberg.

Ebrahim, A. (2024a). *patatrack-scripts: Scripts for benchmarking cmsrun-based patatrack jobs.* GitHub. Retrieved from https://github.com/asubah/patatrack-scripts (Accessed: [02-09-2024])

Ebrahim, A. (2024b). *pixeltrack-standalone.* https://github.com/asubah/pixeltrack-standalone/tree/autotuning. GitHub. Retrieved from https://github.com/asubah/pixeltrack-standalone (Fork of cms-patatrack/pixeltrack-

standalone, branch: autotuning)

Ebrahim, A., Hammad, M., Zeki, A., & Alqaddoumi, A. (2021). City-based approach for gpu kernel execution trace visualisation. In *4th smart cities symposium (scs 2021)* (Vol. 2021, pp. 301–306). doi: 10.1049/icp.2022.0360

Evans, R. T., Cawood, M., Harrell, S. L., Huang, L., Liu, S., Lu, C.-Y., … Zhang, Z. (2021). Optimizing gpu-enhanced hpc system and cloud procurements for scientific workloads. In *Information security conference.* Retrieved from `https://api.semanticscholar.org/CorpusID:235466296`

Fernandez Perez Tomei, T. R. (2022). *The High-Level Trigger for the CMS Phase-2 Upgrade* (Tech. Rep.). CMS. Retrieved from `https://cds.cern.ch/record/2847440` doi: 10.22323/1.414.0209

Fung, J., & Mann, S. (2005). Openvidia: parallel gpu computer vision. In *Proceedings of the 13th annual acm international conference on multimedia* (p. 849–852). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1101149.1101334` doi: 10.1145/1101149.1101334

Gelado, I., & Garland, M. (2019). Throughput-oriented gpu memory allocation. In *Proceedings of the 24th symposium on principles and practice of parallel programming* (p. 27–37). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3293883.3295727` doi: 10.1145/3293883.3295727

Gu, R., & Becchi, M. (2020). Gpu-fptuner: Mixed-precision auto-tuning for floating-point applications on gpu. In *2020 ieee 27th international conference on high performance computing, data, and analytics (hipc)* (pp. 294–304). doi: 10.1109/HiPC50609.2020.00043

Guerreiro, J., Ilic, A., Roma, N., & Tomás, P. (2015). Multi-kernel auto-tuning on gpus: Performance and energy-aware optimization.

In *2015 23rd euromicro international conference on parallel, distributed, and network-based processing* (pp. 438–445). doi: 10.1109/PDP.2015.44

Hu, B., & Rossbach, C. J. (2019). Mirovia: A benchmarking suite for modern heterogeneous computing. *ArXiv*, *abs/1906.10347*. Retrieved from `https://api.semanticscholar.org/CorpusID:13710490`

Immanuel, S. D., & Chakraborty, U. K. (2019). Genetic algorithm: An approach on optimization. In *2019 international conference on communication and electronics systems (icces)* (pp. 701–708). doi: 10.1109/ICCES45898.2019.9002372

Jin, Z., & Vetter, J. S. (2023). A benchmark suite for improving performance portability of the sycl programming model. *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 325–327. Retrieved from `https://api.semanticscholar.org/CorpusID:259236283`

Kameyama, K. (2009). Particle swarm optimization - a survey. *IEICE Trans. Inf. Syst.*, *92-D*, 1354–1361. Retrieved from `https://api.semanticscholar.org/CorpusID:39332633`

Khronos, S. W. G. (2020). *SYCL 2020 Specification* (Specification). Khronos Group. Retrieved from `https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html` (Revision 9)

Leung, Y., Gao, Y., & Xu, Z.-B. (1997). Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, *8*(5), 1165–1176. doi: 10.1109/72.623217

Li, B., Wei, J., Sun, J., Annavaram, M., & Kim, N. S. (2019, jun). An efficient gpu cache architecture for applications with irregular mem-

ory access patterns. *ACM Trans. Archit. Code Optim.*, *16*(3). Retrieved from https://doi.org/10.1145/3322127 doi: 10.1145/3322127

Lin, H., & Wang, C.-L. (2020). On-gpu thread-data remapping for nested branch divergence. *Journal of Parallel and Distributed Computing*, *139*, 75–86. Retrieved from https://www.sciencedirect.com/science/article/pii/S0743731518308967 doi: https://doi.org/10.1016/j.jpdc.2020.02.003

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., ... Hutter, F. (2022, jan). Smac3: a versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, *23*(1).

Liu, Y., Sid-Lakhdar, W. M., Marques, O., Zhu, X., Meng, C., Demmel, J. W., & Li, X. S. (2021). Gptune: multitask learning for autotuning exascale applications. In *Proceedings of the 26th acm sigplan symposium on principles and practice of parallel programming* (p. 234–246). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3437801.3441621 doi: 10.1145/3437801.3441621

Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., ... Lefohn, A. (2004). Gpgpu: general purpose computation on graphics hardware. In *Acm siggraph 2004 course notes* (p. 33–es). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1103900.1103933 doi: 10.1145/1103900.1103933

Markidis, S., Chien, S. W. D., Laure, E., Peng, I. B., & Vetter, J. S. (2018). NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 ieee international parallel and distributed processing symposium workshops (ipdpsw)* (pp. 522–531). doi: 10.1109/

IPDPSW.2018.00091

Matthes, A., Widera, R., Zenker, E., Worpitz, B., Huebl, A., & Bussmann, M. (2017). Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library. In J. M. Kunkel, R. Yokota, M. Taufer, & J. Shalf (Eds.), *High performance computing* (pp. 496–514). Cham: Springer International Publishing.

McIntosh-Smith, S., Boulton, M., Curran, D., & Price, J. (2014). On the performance portability of structured grid codes on many-core computer architectures. In J. M. Kunkel, T. Ludwig, & H. W. Meuer (Eds.), *Supercomputing* (pp. 53–75). Cham: Springer International Publishing.

Menon, H., Bhatele, A., & Gamblin, T. (2020). Auto-tuning parameter choices in hpc applications using bayesian optimization. In *2020 ieee international parallel and distributed processing symposium (ipdps)* (pp. 831–840). doi: 10.1109/IPDPS47924.2020.00090

Mohamed, A. (2020). *Enhancing cms daq systems performance using performance profiling of parallel programs on gpgpus* (Unpublished master's thesis). University of Bahrain.

Montiel, J., Mitchell, R., Frank, E., Pfahringer, B., Abdessalem, T., & Bifet, A. (2020). Adaptive xgboost for evolving data streams. In *2020 international joint conference on neural networks (ijcnn)* (pp. 1–8). doi: 10.1109/IJCNN48605.2020.9207555

Neely, J. R. (2016, 4). *Doe centers of excellence performance portability meeting* (Tech. Rep.). Retrieved from `https://www.osti.gov/biblio/1332474` doi: 10.2172/1332474

Nugteren, C., & Codreanu, V. (2015). Cltune: A generic auto-tuner for opencl kernels. In *2015 ieee 9th international symposium on embedded multicore/many-core systems-on-chip* (pp. 195–202). doi: 10.1109/MCSoC.2015.10

NVIDIA Corporation. (2024a). CUDA C++ Programming Guide [Computer software manual]. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Accessed: [31-08-2024])

NVIDIA Corporation. (2024b). User guide — nsight-systems 2024.5 documentation [Computer software manual]. Retrieved from https://docs.nvidia.com/nsight-systems/UserGuide/index.html (Accessed: [02-09-2024])

Pennycook, S. J., Sewall, J. D., & Lee, V. W. (2016). A metric for performance portability. *arXiv preprint arXiv:1611.07409*.

Petrovič, F., Střelák, D., Hozzová, J., Ol'ha, J., Trembecký, R., Benkner, S., & Filipovič, J. (2020). A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, *108*, 161–177. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167739X19327360 doi: https://doi.org/10.1016/j.future.2020.02.069

Price, K. V., Storn, R., & Lampinen, J. (2014). Differential evolution: A practical approach to global optimization.. Retrieved from https://api.semanticscholar.org/CorpusID:118963641

Ramírez, J. E., Yzquierdo, A. P.-C., & Hernández, J. M. (2016). Exploiting multicore compute resources in the cms experiment. *Journal of Physics: Conference Series*, *762*. Retrieved from https://api.semanticscholar.org/CorpusID:63840434

Reyes, R., & de Sande, F. (2012). Optimization strategies in different cuda architectures using llcomp. *Microprocess. Microsystems*, *36*, 78-87.

Rovere, M., Chen, Z., Di Pilato, A., Pantaleo, F., & Seez, C. (2020). Clue: a fast parallel clustering algorithm for high granularity calorimeters in high-energy physics. *Frontiers in big Data*, *3*, 591315.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th acm sigplan symposium on principles and practice of parallel programming* (p. 73–82). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1345206.1345220` doi: 10.1145/1345206.1345220

Schmidt, B., Gonzalez-Dominguez, J., Hundt, C., & Schlarb, M. (2018). Chap-ter 8-advanced cuda programming. In *Parallel programming* (pp. 287–313). Morgan Kaufmann.

Sedova, A., Eblen, J. D., Budiardja, R., Tharrington, A., & Smith, J. C. (2018). High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability. In *2018 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)* (pp. 1–13). doi: 10.1109/P3HPC.2018.00004

Seymour, K., You, H., & Dongarra, J. (2008a). A comparison of search heuristics for empirical code optimization. In *2008 ieee international conference on cluster computing* (pp. 421–429). doi: 10.1109/CLUSTR.2008.4663803

Seymour, K., You, H., & Dongarra, J. (2008b). A comparison of search heuristics for empirical code optimization. In *2008 ieee international conference on cluster computing* (pp. 421–429). doi: 10.1109/CLUSTR.2008.4663803

Sinclair, M. D., Alsop, J., & Adve, S. V. (2017). Heterosync: A benchmark suite for fine-grained synchronization on tightly coupled gpus. In *2017 ieee international symposium on workload characterization (iiswc)* (pp. 239–249). doi: 10.1109/IISWC.2017.8167781

Steinkraus, D., Buck, I., & Simard, P. (2005). Using gpus for machine

learning algorithms. In *Eighth international conference on document analysis and recognition (icdar'05)* (pp. 1115–1120 Vol. 2). doi: 10.1109/ICDAR.2005.251

Sund, I., Kirkhorn, K. A., Tørring, J. O., & Elster, A. C. (n.d.). BAT: A benchmark suite for AutoTuners. (1), 44–57. Retrieved 2021-12-10, from `https://ojs.bibsys.no/index.php/NIK/article/view/915` (Number: 1)

Tillmann, M., Karcher, T., Dachsbacher, C., & Tichy, W. F. (2013). Application-independent autotuning for gpus. In *International conference on parallel computing.*

Torrey, L., & Shavlik, J. (2010). Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques* (pp. 242–264). IGI global.

Tørring, J. O., & Elster, A. C. (2022). Analyzing search techniques for autotuning image-based gpu kernels: The impact of sample sizes. *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 972–981. Retrieved from `https://api.semanticscholar.org/CorpusID:247748759`

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., ... Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, *33*(4), 805–817. doi: 10.1109/TPDS.2021.3097283

Tørring, J. O., van Werkhoven, B., Petrovč, F., Willemsen, F.-J., Filipovič, J., & Elster, A. C. (n.d.). Towards a benchmarking suite for kernel tuners. In *2023 IEEE international parallel and distributed processing symposium workshops (IPDPSW)* (pp. 724–733). Retrieved 2024-01-24, from `https://ieeexplore.ieee.org/abstract/document/10196663` doi: 10.1109/

IPDPSW59300.2023.00124

Tørring, J. O., van Werkhoven, B., Petrovč, F., Willemsen, F.-J., Filipovič, J., & Elster, A. C. (2023). Towards a benchmarking suite for kernel tuners. In *2023 ieee international parallel and distributed processing symposium workshops (ipdpsw)* (pp. 724–733). doi: 10.1109/IPDPSW59300.2023.00124

van Werkhoven, B. (2019). Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, *90*, 347–358. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0167739X18313359` doi: https://doi.org/10.1016/j.future.2018.08.004

Victoria, A. H., & Maragatham, G. (2021). Automatic tuning of hyperparameters using bayesian optimization. *Evolving Systems*, *12*, 217–223.

Wales, D. J., & Doye, J. P. K. (1997). Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *Journal of Physical Chemistry A*, *101*, 5111–5116. Retrieved from `https://api.semanticscholar.org/CorpusID:28539701`

Wang, H., Liang, Q., Hancock, J. T., & Khoshgoftaar, T. M. (2024). Feature selection strategies: a comparative analysis of shap-value and importance-based methods. *Journal of Big Data*, *11*(1), 44.

Wang, Q., Xu, P., Zhang, Y., & Chu, X. (2017). Eppminer: An extended benchmark suite for energy, power and performance characterization of heterogeneous architecture. In *Proceedings of the eighth international conference on future energy systems* (p. 23–33). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3077839.3077858` doi: 10.1145/3077839.3077858

Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., & Guo, M. (2016). Simultaneous multikernel: Fine-grained sharing of gpus. *IEEE Computer Architecture Letters*, *15*(2), 113–116. doi: 10.1109/LCA.2015.2477405

Whaley, R., & Dongarra, J. (1998). Automatically tuned linear algebra software. In *Sc '98: Proceedings of the 1998 acm/ieee conference on supercomputing* (pp. 38–38). doi: 10.1109/SC.1998.10004

Willemsen, F.-J., Schoonhoven, R., Filipovič, J., Tørring, J. O., van Nieuwpoort, R., & van Werkhoven, B. (2024). A methodology for comparing optimization algorithms for auto-tuning. *Future Generation Computer Systems*, *159*, 489–504. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0167739X24002498` doi: https://doi.org/10.1016/j.future.2024.05.021

Willemsen, F.-J., van Nieuwpoort, R., & van Werkhoven, B. (2021). Bayesian optimization for auto-tuning gpu kernels. In *2021 international workshop on performance modeling, benchmarking and simulation of high performance computer systems (pmbs)* (pp. 106–117). doi: 10.1109/PMBS54543.2021.00017

Wu, J., Chen, S., Zhou, W., Wang, N., & Fan, Z. (2020). Evaluation of feature selection methods using bagging and boosting ensemble techniques on high throughput biological data. In *Proceedings of the 2020 10th international conference on biomedical engineering and technology* (p. 170–175). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3397391.3397403` doi: 10.1145/3397391.3397403

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316. Retrieved from `https://www.sciencedirect.com/`

science/article/pii/S0925231220311693 doi: https://doi.org/10.1016/j.neucom.2020.07.061

Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., ... Bussmann, M. (2016). Alpaka – an abstraction library for parallel kernel acceleration. In *2016 ieee international parallel and distributed processing symposium workshops (ipdpsw)* (pp. 631–640). doi: 10.1109/IPDPSW.2016.50

Zhou, K., Adhianto, L., Anderson, J., Cherian, A., Grubisic, D., Krentel, M., ... Mellor-Crummey, J. (2021, dec). Measurement and analysis of gpu-accelerated applications with hpctoolkit. *Parallel Comput.*, *108*(C). Retrieved from https://doi.org/10.1016/j.parco.2021.102837 doi: 10.1016/j.parco.2021.102837

Zhu, X., Liu, Y., Ghysels, P., Bindel, D., & Li, X. S. (2022). Gptuneband: Multi-task and multi-fidelity autotuning for large-scale high performance computing applications. In *Proceedings of the 2022 siam conference on parallel processing for scientific computing (pp)* (pp. 1–13). Retrieved from https://epubs.siam.org/doi/abs/10.1137/1.9781611977141.1 doi: 10.1137/1.9781611977141.1

ملخص الدراسة

من أهم الصعاب التي تواجه مطوري البرمجيات التي تستخدم معالجات الرُّسُوم، هي التنوع الكثيف في هذه المعالجات. حيث إن التطور الدائم في معماريات المعالجات، يصعّب من مهمة تطوير برامج عالية الأداء مضبوطة ضبطا دقيقًا يؤدي إلى الاستخدام الأمثل لقدرات هذه المعالجات. لذلك، في هذه الأطروحة نقدم إطار عمل لضبط البرمجيات التي تستخدم معالجات الرُّسُوم ضبطا آليا يهدف إلى تحسين أداء هذه البرمجيات على مختلف أنواع معالجات الرُّسُوم.

في هذه الأطروحة، طورت تقنيات بحث بغرض خفض التكلفة الحسابية لعملية الضبط الآلي لبرمجيات معالجات الرُّسُوم. أضيفت هذه التقنيات إلى أحد برامج الضبط الآلي المعروف بOpenTuner. كما طور البرنامج OpenTuner ليتوافق مع برمجيات معالجات الرُّسُوم. تقنيات البحث التي أضيفت إلى OpenTuner في هذه الأطروحة هي خوارزمية Basin Hopping و خوارزمية Bayesian Optimization. كما أضيفت تقنية أخرى مبنية على استخدام تعلم الآلة هدفها تقليص حجم فضاء البحث في الضبط الآلي. كما طورت تقنية قائمة على تعدد مستويات دقة الضبط، لتسريع عملية الضبط الآلي كذلك.

استخدمت هذه التقنيات المستحدثة لضبط برمجيات الرُّسُوم في المتضمنة لبرمجيات كاشف الميون العملاق (CMSSW) المستخدمة في تجارب فيزياء الطاقة العالية، وذلك بتطوير واجهة ربط بين برمجيات CMS وإطار العمل المقدم في هذه الأطروحة. اختبر إطار العمل المقترح على برمجيات الرُّسُوم في CMSSW، وذلك باتباع منهجيات قياس أداء مقترحة من قبل الباحثين في هذا المجال، وذلك بغرض مقارنة التقنيات المقترحة مع التقنيات الأخرى المعروفة في الوسط العلمي. ونتج من هذه التجارب والاختبارات، أن البرمجيات المضبوطة آليا بواسطة الإطار المقترح، تفوقت على الضبط اليدوي، مما زاد سرعة البرمجيات وقلل كلفة استهلاكها للموارد.

مما قيد هذه التجارب، عدم توفر عدد أكبر من معالجات الرُّسُوم الممكن استخدامها في التجارب، عدم استخدام تقنيات أكثر من مجال تعلم الآلة، عدم ضبط البرمجيات لأهداف أخرى غير زيادة الأداء، كتخفيف استخدام الطاقة. في المستقبل يمكن استخدام عدد أكبر من المعالجات ذات المعماريات المختلفة، كما يمكن تجربة بعض تقنيات تعلم الآلة المتقدمة كالتعلم العميق والتعلم المعزز، كما يمكن استخدام إطار الضبط الآلي مع برمجيات مختلفة، كما يمكن استخدام الضبط الفعال وتحسين واجهة المستخدم.

جامعة البحرين
كلية تقنية المعلومات

إطار عمل للضبط الآلي لأداء برمجيات معالجات الرُّسوم

أطروحة مقدمة كجزء من متطلبات الحصول على
درجة الدكتوراة في علوم الحوسبة والمعلومات

إعداد

عبد الله إبراهيم علي صباح

٢٠١٠٣٠٧٩

إشراف

د. وائل محمد المدني

(أستاذ مشارك)

د. هشام العمال

(أستاذ مساعد)

جامعة البحرين

مملكة البحرين

أكتوبر ٢٠٢٤