Universiteit Leiden
The Netherlands

Opleiding Informatica

Designing an Auto-Scalable Benchmark Suite for GPU Kernels

Jeppe Vonk

Supervisors:
Ben van Werkhoven & Skip Thijssen

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/08/2025

Abstract

Identifying the saturation point of GPU kernels, the input size beyond which performance no longer scales proportionally, is essential for optimizing high-performance computing workloads. Manual benchmarking is often required for this task, but it is time-consuming and error-prone.

This thesis introduces `Autobench`, a framework that automates benchmarking and saturation point detection. It combines filtering techniques with knee detection algorithms, specifically the Triangle method and the Kneedle algorithm. The framework was evaluated on two representative workloads, a bandwidth-bound Vector Addition and a compute-bound General Matrix Multiplication (GEMM), using NVIDIA Ada GPUs.

The results show that `Autobench` reliably detects saturation points across both kernels. For GEMM, the detection is very stable and yields identical results across repeated runs and devices. For Vector Addition, detection remains robust despite small run-to-run variations, reflecting the sensitivity of memory-bound workloads to system fluctuations. These findings demonstrate that saturation point detection can be automated in a practical and interpretable way, reducing the need for manual analysis. Future work includes extending the framework to various kernels and architectures, and integrating adaptive tuning and multi-method detection to improve generality and robustness.

Contents

1 Introduction

In recent years, Graphics Processing Units (GPUs) have become essential components in high-performance computing. Their massively parallel architecture makes them ideally suited for accelerating data-parallel workloads in fields such as machine learning, scientific simulations, and image processing. Compared to traditional Central Processing Units (CPUs), GPUs offer significantly higher computational throughput and energy efficiency for many types of compute- and memory-intensive tasks.
At the heart of GPU-accelerated applications are *GPU kernels*, functions executed across thousands of threads in parallel. Optimizing these kernels is crucial for achieving efficient, scalable performance. To this end, benchmark suites play a central role. They provide standardized, controlled methods for evaluating kernel behavior under various conditions [HP17]. Popular suites such as Rodinia [CBM+09], SHOC [DMM+10], and Parboil [SRS+12] are widely used to compare hardware platforms, detect bottlenecks, and guide optimization efforts.
However, most existing GPU benchmark suites are constrained by static configurations, often requiring manual reconfiguration to target different devices. This rigidity limits their usefulness in modern computing environments, where GPU architectures vary significantly in capabilities, memory bandwidth, and processing power. As a result, fixed-size benchmarks may either underutilize or oversaturate hardware, leading to misleading performance assessments and inefficient use of energy and resources.

1.1 Motivation and Problem Statement

As GPUs become increasingly heterogeneous, the need for adaptive and intelligent benchmarking tools grows. Traditional benchmarks fail to detect *saturation points*, the thresholds beyond which increasing problem sizes yield diminishing performance returns. They also lack the ability to dynamically tune workloads in response to system feedback, which hampers automated performance studies and cross-platform comparison.
A benchmark suite that can automatically adapt its workload to the performance characteristics of the GPU it runs on, scaling problem sizes up or down as needed, would address this challenge. Such a system could provide more meaningful performance insights, reduce unnecessary energy consumption, and support scalable evaluation across diverse GPU platforms.

1.2 Research Aim and Question

The central goal of this thesis is to design and implement an **Auto-Scalable Benchmark Suite for GPU Kernels**, a flexible benchmarking framework that dynamically adjusts workload sizes based on device characteristics and observed performance behavior.

"Can we design an Auto-Scalable Benchmark Suite for GPU Kernels?"

To explore this question, the thesis will pursue the following sub-goals:

- Develop a set of representative GPU benchmarks, categorized into bandwidth-bound and compute-bound workloads.

- Implement auto-scaling mechanisms that incrementally adjust problem sizes during benchmarking.
- Detect GPU throughput saturation points to avoid inefficient configurations.
- Record device specifications and associate them with benchmark results.
- Use the `KernelTuner` framework [van19] to execute and tune kernels.
- Evaluate the framework across multiple GPUs and analyze performance data.

1.3 Contributions

This thesis offers the following contributions:

- A prototype framework for building and running auto-scalable GPU benchmarks.
- A methodology for detecting and recording performance saturation points.
- A reusable dataset that maps GPU models to optimized benchmark configurations.
- An evaluation of scalability behavior across different GPU architectures.

1.4 Thesis Overview

This thesis is organized as follows. Section 2 presents the background theory on GPU architecture, kernel execution, and benchmarking, and discusses related work including Rodinia, Parboil, and SHOC. Next, Section 3 details the design and implementation of the auto-scalable benchmark suite, including benchmark selection, scaling mechanisms, and the integration with KernelTuner. Section 4 presents the outcomes of running the suite across different GPU architectures, analyzing throughput and saturation points. Finally, Sections 5 and 6 reflect on the findings, discuss the strengths and limitations of the approach, suggest directions for future work, and summarize the main contributions and conclusions of the thesis.

This bachelor thesis is part of the Computer Science program at the *Leiden Institute of Advanced Computer Science (LIACS)*, supervised by Dr. Ben van Werkhoven and PhD candidate Skip Thijssen.

2 Background and Related Work

Section 2 provides a detailed overview of GPU architecture, performance benchmarking methodologies, and representative benchmark suites. The section is structured as follows. Section 2.1 introduces core GPU architecture and kernel execution concepts. Section 2.2 and Section 2.3 discuss how benchmarks are used to evaluate GPU performance and highlight several established benchmark suites. Section 2.4 outlines their key limitations, while Section 2.5 introduces Kernel Tuner, a tool for automated tuning. Finally, Section 2.6 motivates the need for a dynamic, scalable benchmarking approach.

2.1 GPU Architecture and Kernel Execution

Modern Graphics Processing Units (GPUs) are purpose-built for high-throughput, data-parallel workloads. In contrast to Central Processing Units (CPUs), which are optimized for low-latency, sequential execution, GPUs consist of thousands of lightweight cores organized into multiprocessor units. The terminology for these units varies by vendor: for instance, NVIDIA refers to them as *Streaming Multiprocessors* (SMs), while AMD uses the term *Multithreaded SIMD Processor* [HP17]. These cores operate under the *Single Instruction, Multiple Threads* (SIMT) model, where groups of threads, often referred to as warps (NVIDIA) or wavefronts (AMD), execute the same instruction concurrently on different data elements. This model enables massive parallelism while also handling control flow divergence efficiently.

A central concept in GPU programming is the *kernel*, a parallel function written in languages such as CUDA [NBGS08, Lue08] or OpenCL [Mun09] and launched across a grid of thread blocks. Each thread executes the same code independently on a subset of data, mapped to the underlying SMs. Kernels are typically compact yet performance-critical components of larger, real-world applications [HP17].

To achieve high performance, developers must carefully optimize kernel characteristics, including memory access patterns, thread block size, and instruction-level parallelism. These optimizations also involve tuning occupancy, how efficiently hardware resources are utilized, often requiring device-specific adjustments. As a result, performance tuning for GPU applications is both complex and architecture-dependent.

2.2 Benchmarking for GPU Performance Evaluation

Benchmarking is a foundational technique for evaluating and optimizing GPU performance. By executing kernels under controlled workloads, benchmarks help expose how architectural features, such as memory hierarchy, thread scheduling, and compute density, influence throughput, latency, and overall resource utilization.

However, individual benchmark results can be misleading due to design-specific biases or dependencies on hardware and compiler configurations [HP17]. To address these limitations, benchmark *suites* aggregate a diverse set of tests, enabling broader and more reliable performance characterization across different workloads and architectures.

GPU benchmark suites generally fall into three categories:

- **Synthetic benchmarks** (or microbenchmarks) isolate specific architectural aspects, such as memory bandwidth or instruction throughput.

- **Application benchmarks** mimic real-world workloads drawn from domains like scientific computing, imaging, or machine learning.
- **Benchmark suites** combine both synthetic and application-style benchmarks to provide comprehensive coverage of performance dimensions.

These benchmarks also target different kernel bottlenecks:

- **Bandwidth-bound kernels** are limited by memory access speed.
- **Compute-bound kernels** are constrained by arithmetic throughput.
- **Latency-bound kernels** are affected by instruction dispatch delays or memory access latency.
- **Contention-bound kernels** suffer from synchronization overhead or frequent use of atomic operations.

2.3 Existing GPU Benchmark Suites

Several GPU benchmark suites have been widely adopted for performance evaluation, architecture research, and compiler optimization. Among the most prominent are Rodinia, Parboil, and SHOC. These suites provide curated workloads that span different application domains and parallelism styles, but each exhibits limitations in adaptability and automated tuning.

2.3.1 Rodinia

Rodinia [CBM+09] is designed to evaluate heterogeneous platforms, including multicore CPUs and GPUs. Its workloads are selected based on the Berkeley Motifs taxonomy (originally referred to as "Dwarfs"), covering diverse domains such as medical imaging, physics simulation, and machine learning. The suite emphasizes a variety of parallelism patterns, including data, task, and pipeline parallelism, and offers implementations in CUDA and OpenMP [DM98].
Rodinia is grounded in real-world applications, providing realistic performance behavior. However, its workloads typically use fixed input sizes and require manual parameter tuning, which limits scalability across architectures with different resource constraints.

2.3.2 Parboil

The Parboil benchmark suite [SRS+12] targets throughput-oriented computing and includes workloads such as MRI-Q, stencil computations, SGEMM, and sparse matrix-vector multiplication (SpMV). It categorizes benchmarks by performance bottlenecks, such as being compute- or memory-bound, and supports multiple backends, including CUDA, OpenCL, and OpenMP.
A distinguishing feature of Parboil is the inclusion of multiple versions per benchmark:

- A **baseline version** that ensures correctness but is not optimized.
- An **architecture-optimized version** that showcases performance tuning for specific hardware.

- A **pedagogical version** with readable code for teaching and experimentation.

While Parboil provides standardized input datasets to support reproducibility, it does not include mechanisms for dynamic input scaling or automated performance tuning, requiring manual adaptation for each new target architecture.

2.3.3 SHOC

SHOC (Scalable HeterOgeneous Computing) [DMM+10] includes both low-level microbenchmarks and higher-level application-style kernels. It features tests such as FFT, GEMM, and reductions, and supports CUDA and OpenCL backends. Notably, SHOC can be deployed on multi-node systems using MPI, making it suitable for evaluating performance across heterogeneous clusters.
Despite its versatility and wide hardware compatibility, SHOC benchmarks are configured with fixed problem sizes and do not incorporate dynamic workload scaling or saturation detection. As with Rodinia and Parboil, this limits its out-of-the-box adaptability for performance studies across modern, diverse GPU platforms.

2.4 Limitations of Static Benchmark Suites

Despite their widespread use, benchmark suites such as Rodinia, Parboil, and SHOC exhibit several critical limitations that hinder their effectiveness on modern, heterogeneous GPU architectures:

- **Static input sizes** limit adaptability across GPUs with varying compute and memory capabilities, making it difficult to evaluate scaling behavior or utilize available resources effectively.
- **Manual tuning requirements** reduce reproducibility and increase development overhead, particularly when targeting new hardware or comparing across architectures.
- **Lack of saturation detection** prevents identifying the point at which a larger problem size no longer yields meaningful performance improvements, potentially leading to misleading conclusions due to underutilization (not fully utilizing hardware capacity) or overutilization (exceeding capacity and causing throttling).

These constraints reduce the representativeness and portability of benchmark results. As GPU hardware becomes increasingly diverse and specialized, there is a growing need for dynamic, adaptive benchmarking methodologies that automatically respond to architectural differences and runtime behavior. Such approaches not only improve evaluation accuracy but also help make benchmarking frameworks more *future-proof*, allowing them to remain effective across generations of evolving hardware platforms.

2.5 Auto-Tuning with Kernel Tuner

Rather than manually determining suitable parameters for every new GPU, tools such as `Kernel Tuner` [van19] were developed to automate this laborious process. Kernel Tuner is a Python-based framework for automated tuning of GPU kernels, supporting CUDA, OpenCL, and C backends. It

allows developers to define parameterized kernels and empirically search for optimal configurations based on metrics such as execution time, occupancy, memory usage, or energy efficiency.
Kernel Tuner facilitates:

- Exploration of tuning parameters such as thread block size, memory layout, loop unrolling, and tiling factors.
- Execution of kernels across varying input sizes to profile performance under different workload scenarios.
- Integration with scientific workflows for automated and reproducible performance tuning.

While Kernel Tuner is not a benchmarking suite, as it lacks standardized kernels or datasets, it serves as a powerful tool for dynamic performance tuning. Its flexibility makes it well-suited for embedding within larger benchmarking or optimization frameworks, enabling fine-grained, architecture-aware performance exploration.

2.6 Motivation for Auto-Scalable Benchmarking

This thesis builds upon established GPU benchmark suites and tuning frameworks by proposing an auto-scalable, device-aware benchmarking framework. The approach integrates three key elements:

- The workload diversity and domain realism of Rodinia, Parboil, and SHOC.
- The adaptive parameter tuning capabilities of Kernel Tuner.
- A novel mechanism for dynamic input scaling and saturation point detection.

By incrementally increasing problem sizes and monitoring key performance metrics such as throughput, the framework identifies optimal workload configurations tailored to the capabilities of each target GPU. This avoids both underutilization and oversaturation, enabling efficient and fair performance evaluation across heterogeneous architectures.
In doing so, this work addresses a key gap in current GPU benchmarking practices by advancing the goals of scalability, reproducibility, and hardware adaptivity, critical for evaluating modern and future compute platforms.

3 Methodology

The methodology of this project is structured as a three-phase benchmarking pipeline, designed to be reproducible, automated, and hardware-agnostic. The phases are as follows: (1) *Configuration*, where the benchmarking scenario is defined and prepared; (2) *Autoscaling*, where benchmarks are executed at increasing problem sizes until a saturation point is detected; and (3) *Exporting*, where collected results are structured, stored, and visualized. This pipeline is implemented in the `Autobench` framework, which was developed as part of this project. Each phase addresses a key component of benchmarking GPU kernels in a scalable and automated way, while abstracting away unnecessary boilerplate and infrastructure details. The remainder of this section elaborates on the design decisions and mechanisms underpinning each of these phases, starting with the configuration stage.

3.1 Phase I: Configuration

The configuration phase defines the benchmarking scenario and prepares all required inputs for execution. This includes specifying which GPU kernel to benchmark, how to generate inputs for varying problem sizes, and how performance should be measured and interpreted. The goal is to decouple the benchmark definition from the execution logic, enabling users to easily swap in new benchmarks with minimal effort.
In `Autobench`, benchmark configuration is centered around a user-defined subclass of a base class called `BenchmarkConfig`. This configuration object encapsulates all the information necessary to run a benchmark: kernel source, kernel arguments, tunable parameters, correctness checks, and performance metrics. Configuration classes are loaded dynamically at runtime and must implement a minimal interface to allow full automation in later phases.

3.1.1 Benchmark Configuration Interface

The configuration interface is designed as a single abstract base class, `BenchmarkConfig`, which defines the contract that any user-defined benchmark must fulfill. This object acts as the bridge between the user-provided kernel code and the benchmarking infrastructure provided by `Autobench`. At a high level, the interface provides three key responsibilities:

- **Build configuration**: For a given problem size, prepare the arguments and internal state needed to execute the benchmark. This is done through the abstract method `build(problem_size)`.
- **Result interpretation**: After running a benchmark, the method `analyze_results(results)` is used to extract and compute key metrics, such as execution time or memory bandwidth.
- **Kernel source resolution and tunable parameters**: The configuration holds information about the kernel source, tuning parameters (e.g., thread block sizes), device options, optional verification functions, and many other options that are supported by Kernel Tuner's `tune_kernel(...)` function.

By requiring only a few concrete implementations from the user, this interface allows the benchmarking tool to remain generic and extensible. For instance, switching from benchmarking a matrix

multiplication kernel to a convolution kernel typically only involves creating a new configuration class that implements `build` and `analyze_results` accordingly.
Internally, the configuration is represented using Python `@dataclass` constructs, which provides a declarative and type-safe way of organizing benchmark parameters. Optional fields such as shared memory arguments, device/platform selection, and compiler flags are included with sensible defaults, ensuring flexibility without overwhelming the user.

3.1.2 Dynamic Loading of Benchmark Definitions

To ensure modularity and ease of use, benchmark definitions are loaded dynamically from Python files provided by the user. This allows benchmarks to be defined outside of the main benchmarking tool and encourages reuse and separation of concerns.
The entry point for dynamic loading is the class method `from_file(...)`, which takes a path to a user-defined Python module and a starting problem size. The expected convention is that this file defines a function name `get_config(problem_size)`, which returns an instance of a class that implements the `BenchmarkConfig` interface.
The loading logic performs the following steps:

1. Uses Python's `importlib` to load the specified module in isolation.
2. Ensures that the module contains the required `get_config` function.
3. Calls `get_config(problem_size)` to retrieve a configuration instance.
4. Validates that the returned object conforms to the expected protocol.

This approach enables complete decoupling between the benchmarking engine and the user-defined benchmark, making it easy to plug in new configurations without modifying core code. Additionally, by reusing the same entry point for all benchmarks, automated tooling (e.g., testing harnesses or visualization tools) can treat all configurations uniformly.

3.1.3 Kernel and Argument Handling

Once a benchmark configuration is loaded, the kernel source and arguments must be resolved in a way that is compatible with the underlying benchmarking backend, in this case, Kernel Tuner. The method `get_kernel_source()` is used to prepare kernel source files or callable source generators in the format expected by Kernel Tuner. It handles flexible input types, including strings, file paths, and dynamic code generators.
For each problem size in the autoscaling phase, the method `build(problem_size)` is invoked. This method regenerates all kernel arguments, such as input/output arrays, based on the new size. This ensures correctness and consistency across benchmarks at different scales. For example, if the kernel processes arrays of length N, the arguments will be reallocated and repopulated with fresh random values or zeros accordingly.
Once all data is prepared, the `to_kwargs()` method converts the configuration into a dictionary of keyword arguments that can be directly passed to Kernel Tuner. This includes both required fields (e.g., kernel source, arguments, tuning parameters) and optional fields (e.g., metric functions, platform selection).

3.1.4 Performance Metrics and Analysis

Each benchmark is responsible for defining its own performance metrics by implementing the `analyze_results(results)` method. The `results` dictionary, returned by Kernel Tuner after execution, typically includes raw timing measurements and any additional statistics captured during benchmarking.
The user-defined analysis function extracts the relevant values from the result and computes two return values:

- A primary metric, usually kernel execution time in milliseconds.
- A secondary metric, which could be domain-specific performance indicators such as throughput (e.g., GB/s or GFLOP/s), or energy usage.

To facilitate this, `BenchmarkConfig` allows users to define a custom dictionary of named metrics via the `metrics` field. Each entry in this dictionary maps a metric name to either a lambda function (computed from the raw results) or a string indicating which result field to extract.
This design allows for flexible and expressive performance evaluations while preserving compatibility with Kernel Tuner's built-in metric recording.

3.1.5 Configuration Example: Vector Addition

To illustrate the configuration phase, consider the example of a simple vector addition kernel. The kernel takes two input arrays `A` and `B`, and produces a third array `C`, where each element is computed as `C[i] = A[i] + B[i]`. This is a memory-bound operation, and its performance is often measured in effective memory bandwidth (GB/s).
In the `build` method of the configuration, random input vectors of a given problem size are generated using NumPy, and the output vector is initialized to zeros. The kernel arguments list is then populated with these arrays and the scalar size parameter.
The `analyze_results` method computes memory bandwidth by assuming two memory reads (for `A` and `B`) and one write (for `C`) per element, each of which is a 4-byte float. The resulting throughput in GB/s is calculated using the formula:

$$\text{GB/s} = \frac{(2 \text{ reads} + 1 \text{ write}) \times \texttt{sizeof(float)} \times \texttt{problem\_size}}{\texttt{time\_s}}$$

Although the vector addition kernel is simple, it demonstrates the flexibility of the configuration interface. By changing only the kernel string and adapting `build` and `analyze_results`, a new benchmark scenario can be created with minimal effort.

Summary and Transition In summary, the configuration phase establishes the foundation of the benchmarking workflow. It defines what is to be benchmarked, how to generate inputs for it, and how to measure performance meaningfully. With this infrastructure in place, the next phase focuses on systematically exploring how the benchmark behaves across a range of problem sizes and identifying performance trends.

3.2 Phase II: Autoscaling

The second phase of the benchmarking pipeline involves an automated, iterative scaling of the benchmark problem size. The purpose of this phase is to systematically evaluate how the performance of a kernel evolves as the input size increases, with the ultimate goal of identifying a *saturation point*, a region in the input space where performance improvements begin to plateau or regress. This information is crucial for understanding the limitations of the kernel and the underlying hardware. Autoscaling begins with a small input size and incrementally increases it according to a scale factor. After each run, performance metrics are collected and stored. A saturation detection algorithm monitors the performance curve and signals when the scaling process can terminate. This process is designed to be both hardware-aware and adaptive, stopping when further increases in problem size are unlikely to yield meaningful insights.

3.2.1 Initialization and Maximum Size Estimation

Before the scaling loop begins, several key components must be initialized. First, the autoscaler constructs the benchmark configuration using the dynamic configuration loader discussed in Phase I. This ensures that all kernel-specific logic is encapsulated and ready for repeated execution.
Three main variables are initialized at this point:

- `results_history`: a list used to store tuples of the form `(problem_size, (primary_metric, secondary_metric))`.
- `current_size`: the problem size used for the first benchmark run, typically set to 1 or a small constant.
- `current_scale_factor`: a float (e.g., 1.1) representing the multiplicative increment applied to `current_size` after each iteration. This value may also be estimated heuristically based on the performance trend of previous iterations.

To prevent the device from scaling beyond its capacity, a hardware-aware upper bound is calculated using the function `estimate_max_problem_size`. This function first queries the available memory on the target device and applies a safety factor to prevent over-allocation. Next, the function builds two small benchmark configurations (e.g., problem sizes 128 and 256) and measures the memory footprint of all kernel arguments. Assuming that memory usage scales linearly with problem size, the function estimates the per-element memory consumption and divides the usable memory by this estimate to determine the largest feasible problem size.
Although this method provides only an approximate bound, as its accuracy depends on the linear scaling assumption, the small sample size, and and the current state of the device memory, it is generally conservative. By predetermining this limit, the autoscaler prevents allocation errors and can focus on input sizes that provide meaningful performance insights.

3.2.2 The Scaling Loop

The autoscaling loop is the core of this phase. At each iteration, the current problem size is benchmarked, analyzed, stored, and evaluated for signs of performance saturation. This loop

continues until one of two termination conditions is met: either the saturation detection algorithm indicates a reliable plateau, or the estimated maximum input size is reached.
Each iteration of the loop consists of the following steps:

Step 1: Build Benchmark Using the `build(current_size)` method from the configuration, the benchmark input is regenerated for the new problem size. This step prepares the kernel arguments, reinitializes arrays, and sets the problem-specific parameters. It ensures correctness and consistency across different input sizes.

Step 2: Run Benchmark Next, the kernel is benchmarked using Kernel Tuner's `tune_kernel` function. The configuration object's `to_kwargs()` method transforms the benchmark configuration into a dictionary of arguments compatible with Kernel Tuner's tuning backend. This step may result in either a successful benchmark run or a failure (e.g., due to memory limits or invalid kernel parameters). Failures are caught using a `try-except` clause; if an exception occurs, the autoscaling process terminates early, assuming that the maximum practical problem size has been reached.

Step 3: Analyze Results Once a benchmark run completes, the result is passed to the configuration's `analyze_results` function. This function computes the primary and secondary performance metrics (e.g., runtime, bandwidth, throughput), returning them as a tuple. These metrics are used to monitor performance trends over time and to detect saturation.

Step 4: Record History The problem size and associated metrics are added to the `results_history`. This growing dataset forms the basis for performance curve analysis and visualization in later phases.

Step 5: Check for Saturation To determine whether further scaling is necessary, a saturation detection function is invoked. This function takes the full `results_history` and the current scale factor as input, and returns a tuple:

$$(\texttt{knee\_index\_or\_scale}, \texttt{saturated})$$

If `saturated` is `True`, it indicates that the performance curve has reached a region of diminishing returns. Otherwise, the autoscaler continues using the returned scale factor to update the problem size. This factor may be determined heuristically to adapt to changing performance trends during execution.

3.2.3 Saturation Detection Algorithms

Detecting saturation points in performance curves is a key component of Autobench's autoscaling mechanism, and therefore remains an active area of development, with several strategies being evaluated. Two notable approaches are: the *Triangle method*, a simple geometric heuristic related to the elbow method, and the *Kneedle algorithm*, a well-known curve analysis technique.

Triangle Method The Triangle method is a distance-based heuristic for knee detection. It is closely related to the "elbow method" used in clustering analysis [Tho53], and a similar approach was previously used in the Kernel Tuner framework [Ker22a], as part of energy efficiency detection [SVVWB22], before being replaced by a curve-fitting method [Ker22b]. The idea is to approximate the location of the maximum deviation from linear growth in a concave curve.
The method works by constructing a line between the first and last points on the performance curve (e.g., problem size vs. throughput, bandwidth, or runtime). For each intermediate point, the perpendicular (unsigned) distance to this line is calculated. The point with the greatest distance is selected as the candidate knee, marking the transition from rapid improvement to plateauing performance. In some cases, this point may also correspond to an inflection point where performance begins to decrease; In Autobench, the first strong knee is considered the saturation point.
In terms of implementation, the method requires only a linear pass through the data ($O(n)$), making it computationally inexpensive. Autobench applies safeguards to prevent false detections: at least five data points must be collected, the latest measurement must exceed a minimum runtime (default: 1 ms) to reduce noise, and the perpendicular distance must exceed a threshold before a knee is accepted. These thresholds are chosen heuristically and were found to improve robustness during experiments, though further evaluation could provide quantitative evidence.

Kneedle Algorithm The Kneedle algorithm, introduced by Satopää et al. [SAIR11] and available in the Python package `kneed` [Arv23], is a well-known method for detecting knees in curves. It identifies the knee as the point of maximum deviation between the normalized difference curve and a straight-line baseline, making it suitable for concave, monotonically increasing functions where the elbow may not always be sharply defined.
In Autobench, Kneedle is applied after at least five measurements are available, with the same runtime threshold as in the Triangle method to mitigate noise. This ensures that only stable data is considered. The sensitivity parameter S is adjusted relative to the number of data points, balancing responsiveness to subtle changes with robustness to measurement variability, ensuring the method remains flexible across different experimental setups.

Comparison and Usage Both methods focus on identifying the point at which performance improvements begin to decline. The Triangle method is fast, requires only a few data points, and is well-suited for early detection, though it may be sensitive to noise and ambiguous when both knees and inflections are present. The Kneedle algorithm, on the other hand, can handle smoother transitions and produces consistent results when curves do not exhibit a sharp elbow. The choice of which method to apply is left to the user, who selects the preferred detection algorithm during Autobench execution.
It should be noted that Autobench currently always applies knee detection to the *secondary metric* (the domain-specific performance indicator, such as throughput or bandwidth), as this most directly reflects diminishing returns with scaling. The *primary metric* (often execution time) is retained to enforce the minimum runtime safeguard and for inclusion in the exported results, but it does not directly affect the knee computation.

3.2.4 Post-Saturation Continuation and Termination

To reduce the risk of misidentifying a local maximum as the global saturation point, the autoscaler continues for a fixed number of *scale loop iterations* (currently set to three) after detecting a knee point. These additional iterations allow the system to observe whether performance starts to improve again, which would indicate that the initially detected knee was premature. The choice of three iterations has empirically proven to provide a good balance between confirming saturation and avoiding excessive measurement overhead.
In addition, several strategies can be considered to further refine the knee location:

- **Local sampling:** Taking additional measurements in the immediate neighborhood of the detected knee (e.g., one step before and after) could help confirm whether the knee is stable or simply an artifact of noise.
- **Binary or adaptive search:** Instead of linearly increasing problem sizes, a targeted search could more efficiently refine the precise saturation point more efficiently by repeatedly halving the search space or adjusting step sizes based on observed gradients.
- **Optimization-based scoring:** Defining a score function that quantifies how well a point matches the knee hypothesis (e.g., maximizing curvature or minimizing relative slope) would enable the use of numerical optimization methods, such as those available in `scipy.optimize`, to refine the estimate more systematically.

While these techniques are not part of the current prototype, they represent promising directions for future work to improve the accuracy and robustness of the autoscaler.

If no saturation is detected before the maximum problem size is reached, the autoscaler terminates and returns the full performance history. In this case, the maximum problem size is selected as the optimal workload configuration, as it represents the largest input that can be executed within the available hardware and mainly memory constraints.

In either case, the output of this phase is a detailed performance curve that maps input sizes to measured metrics, ready for export and visualization.

Summary and Transition The autoscaling phase enables fully automated performance profiling across a range of problem sizes. By adapting to memory constraints and incorporating saturation detection, it produces informative, reproducible benchmarks. In the next and final phase, the collected data will be structured, saved, and visualized to facilitate interpretation and comparison.

3.3 Phase III: Exporting

The third and final phase of the methodology focuses on exporting the results from the autoscaling benchmark process. This stage supports both immediate interpretability, through the visualization of performance trends and saturation points, and long-term utility via the structured export of collected data for reproducibility, documentation, or further analysis. Although less computationally intensive than previous phases, this step is crucial for making results accessible, interpretable, and ready for external review or future processing.

3.3.1 Data Visualization

Once the autoscaling loop has completed, the collected performance data, including execution times and a custom performance metric, can be visualized using a built-in plotting function. This visualization helps to clearly illustrate how performance evolves as the problem size increases, and it is especially useful for identifying saturation points where further scaling yields limited benefit. The system generates a two-axis plot: one axis shows the execution time, and the other shows the custom metric (for example, throughput or efficiency). Both curves are plotted against the problem size. If a saturation point (or "knee") has been detected, it is highlighted on the graph with a visual marker and an optional label.
Several options are available to tailor the plot, such as linear or logarithmic scaling for either axis, and the ability to either display the plot interactively or save it as an image. These customizations help users adapt the visualization to the scale and nature of their workloads.

3.3.2 Data Export

In addition to visual output, the benchmarking results are also written to a CSV file. Each row in the file contains the tested problem size, the corresponding execution time, the measured performance metric, and an indicator specifying whether that row represents the detected saturation point.
To ensure reproducibility and prevent accidental overwrites, all exports are saved in timestamped directories. This allows users to keep a historical record of benchmark runs, which is especially useful when comparing performance across devices or configurations.
This final phase thus ensures that all benchmarking output is both human-readable and machine-processable, closing the loop on a fully automated and informative tuning pipeline.

3.4 Summary

This section presented a complete three-phase methodology for adaptive GPU benchmarking using a novel autoscaling approach:

- **Phase I: Configuration** established a flexible and modular system for defining benchmarking tasks, enabling dynamic setup of kernels, devices, and tunable parameters.
- **Phase II: Autoscaling** implemented a dynamic scaling loop that incrementally increased problem size, used runtime performance feedback, and applied saturation detection methods, such as the Triangle and Kneedle algorithms, to identify optimal computation limits.
- **Phase III: Exporting** focused on making results accessible through visualizations and structured exports (e.g., CSV files), ensuring reproducibility and interpretability.

Together, these phases form an integrated and automated benchmarking pipeline that minimizes manual tuning, intelligently adapts to hardware limits, and provides structured performance data that can serve as the basis for further insights into GPU behavior. The next section presents experimental results generated using this methodology across diverse workloads and configurations.

4 Results

This section presents the outcomes of benchmarking two representative GPU kernels, a memory-bound **Vector Addition** and a compute-bound **General Matrix Multiplication (GEMM)**, using the `Autobench` framework. Both were tested on NVIDIA RTX 4000 Ada and RTX 5000 Ada GPUs, with performance saturation points detected by the *Triangle* and *Kneedle* algorithms. The following subsections first outline the experimental setup and then detail the results for each kernel, GPU, and detection configuration.

4.1 Experimental Setup

All experiments were performed using the `Autobench` benchmarking framework on the DAS-6 cluster [BEdL+16] at the Leiden University (LU) site. Two GPU models were evaluated: an NVIDIA RTX 4000 Ada and an NVIDIA RTX 5000 Ada, both hosted in LU cluster nodes equipped with dual-socket AMD EPYC 7282 CPUs. Each job was submitted via SLURM [JW23] and executed on a single GPU; with `Autobench` handling the configuration, execution, saturation detection, and result export.

Benchmarking Procedure For every {kernel, GPU, detection method} combination, `Autobench` performed five independent autoscaling runs. Each run produced a single plot that represents three key metrics: execution time in milliseconds (left y-axis, linear scale), throughput in either GB/s for Vector Addition or GFLOP/s for GEMM (right y-axis, logarithmic scale), and the detected saturation (knee) point. The x-axis represents problem size on a logarithmic scale. Two saturation detection algorithms were compared: the **Triangle** method, which relies on threshold-based distance analysis, and the **Kneedle** method, which detects knees based on curvature analysis. Both methods are described in detail in Section 3.2.3.

Benchmark Configurations The benchmarks themselves were defined through standalone configuration files that `Autobench` can interpret because they follow the configuration interface described in Section 3.1. These files specify all the necessary information for each benchmark, such as the kernel source, arguments, performance metrics, and execution parameters. For this evaluation, configurations were prepared specifically for the two chosen kernels.

Vector Addition Benchmark For the **Vector Addition** benchmark, two randomly generated `float32` input vectors were added element-wise into an output vector initialized to zero. This configuration is based on the example in Section 3.1.5, with minor modifications for this evaluation. Each problem size was tested with 33 iterations, so that after removing the first iteration, 32 iterations remained for averaging, ensuring consistency with the GEMM benchmark.
The first iteration was removed to prevent cold-start effects, i.e., the temporary performance drop that can occur before the GPU warms up caches, allocates resources, or reaches optimal clock speeds. Between iterations, the GPU's memory state was explicitly reset to simulate fully independent, cold-start conditions. This differs from typical warm-up benchmarking, where data remains in the cache and subsequent runs benefit from previous executions. By resetting the memory at each iteration, we ensure that each run measures performance on a fresh allocation, providing

reproducible and isolated performance data. The average of the remaining 32 iterations was reported as the final performance value.

General Matrix Multiplication (GEMM) Benchmark For the **General Matrix Multiplication (GEMM)** benchmark, the kernel was originally an OpenCL implementation from the CLBlast [Nug18] and CLTune [NC15] frameworks. It was translated into CUDA by including a specialized header file in front of the OpenCL source, also taken from the CLBlast/CLTune frameworks, enabling direct CUDA compilation. These source and header files, with slight modifications, are available as examples for the Kernel Tuner framework in [LHSvW24], and a merged variant of that example was used here. The benchmark's configuration file specified all relevant information about the kernel, including kernel arguments and tunable parameters. In this setup, matrices of increasing size were multiplied with a fixed inner dimension $k = 256$. Performance was measured in GFLOP/s. Unlike Vector Addition, GEMM showed no noticable cold-start effects, and therefore each problem size was tested with 32 iterations directly, with the average performance reported.

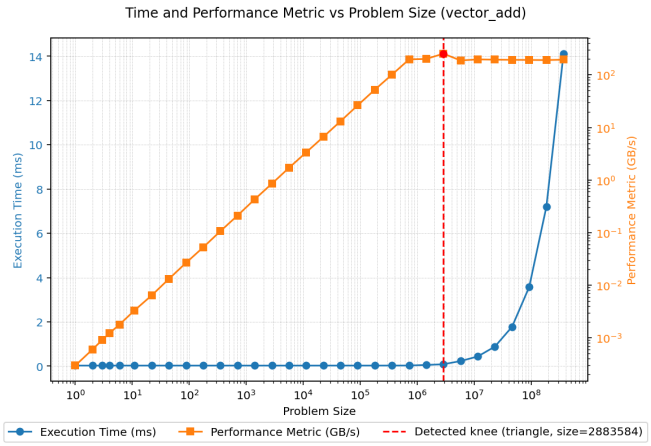
Consistency and Variation This setup ensured that every configuration was tested under a consistent methodology, while allowing for natural run-to-run variation. Such variation is not an error, but rather an important aspect of the evaluation, as it provides insight into how reliably each saturation detection algorithm can identify performance limits across multiple independent runs.

4.2 Vector Addition

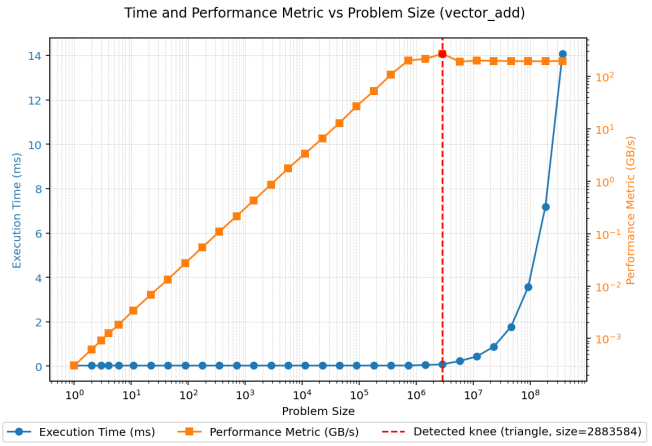
As described in Section 4.1, the Vector Addition kernel is a memory-bound operation tested on both GPUs with both detection algorithms.

4.2.1 NVIDIA RTX 4000 Ada

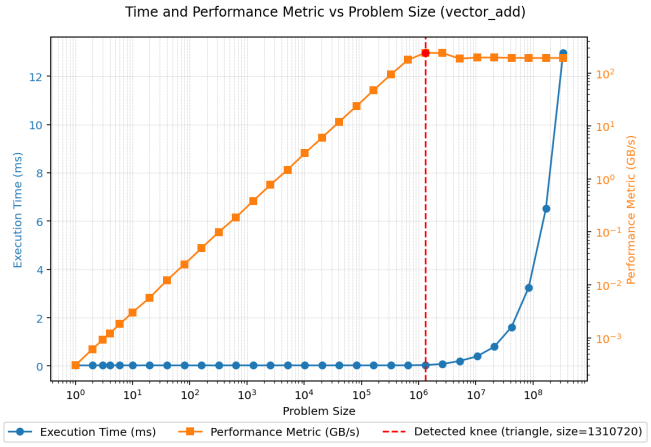
Triangle Algorithm Five autoscaling runs were executed using the Triangle detection algorithm (Figures 1a–1e).



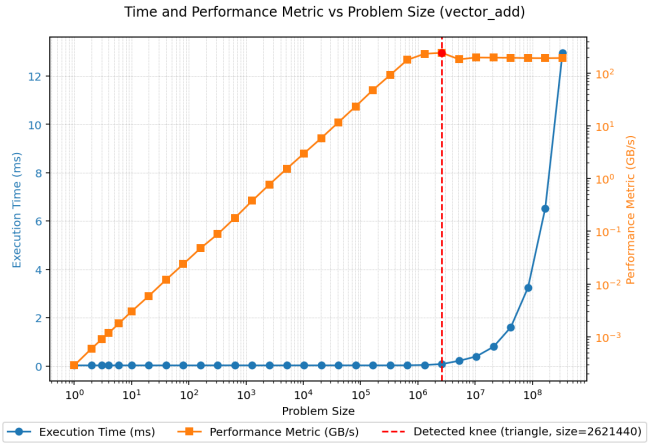
(a) Run 1



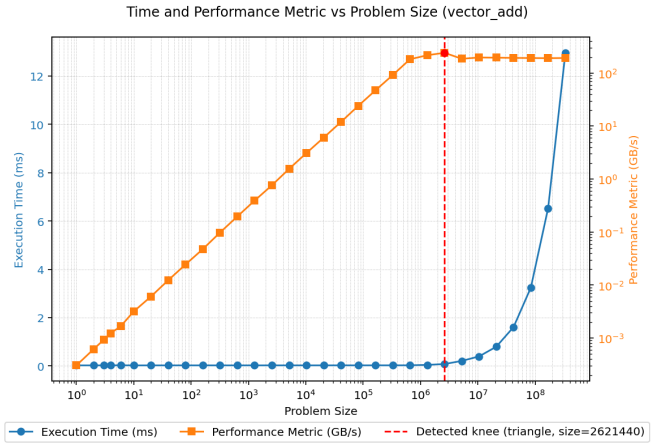
(b) Run 2



(c) Run 3



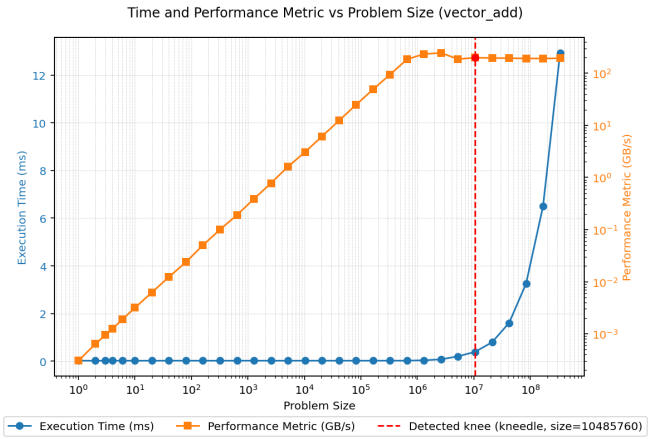
(d) Run 4



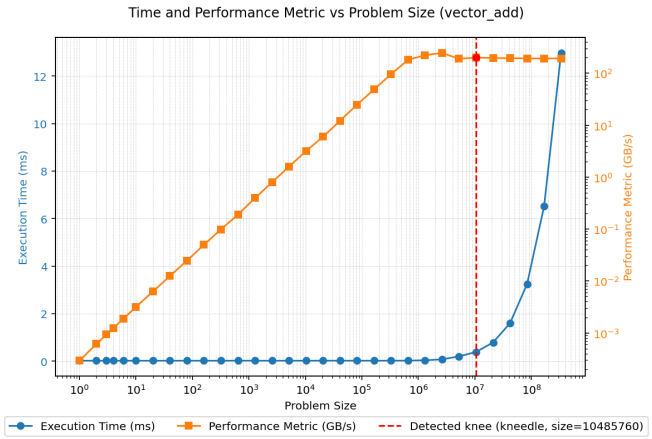
(e) Run 5

Figure 1: Vector Addition on RTX 4000 Ada using Triangle method (five runs)

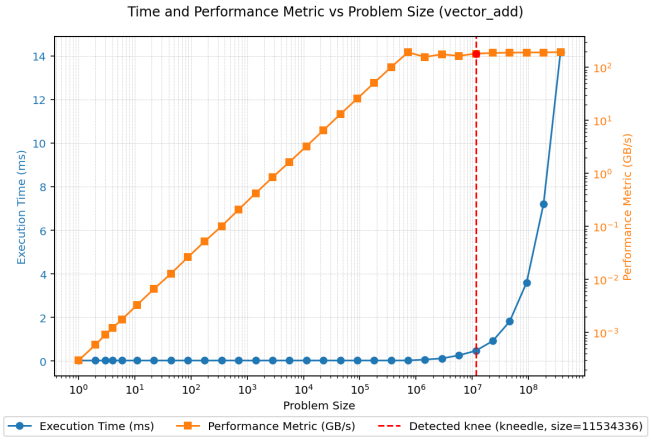
Kneedle Algorithm The Kneedle algorithm results are shown in Figures 2a–2e. The benchmarking procedure and visualization follow the same structure as for the Triangle method.



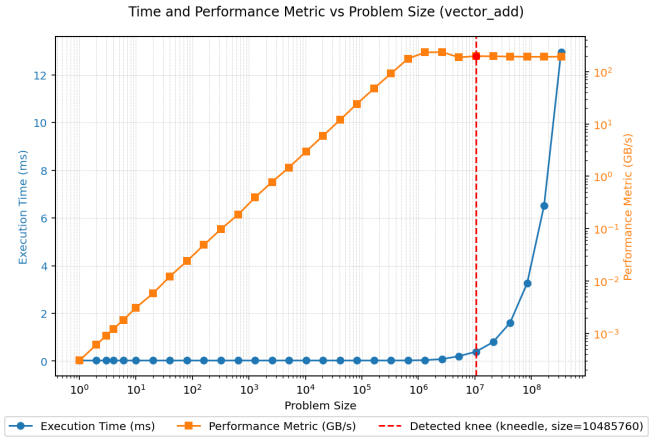
(a) Run 1



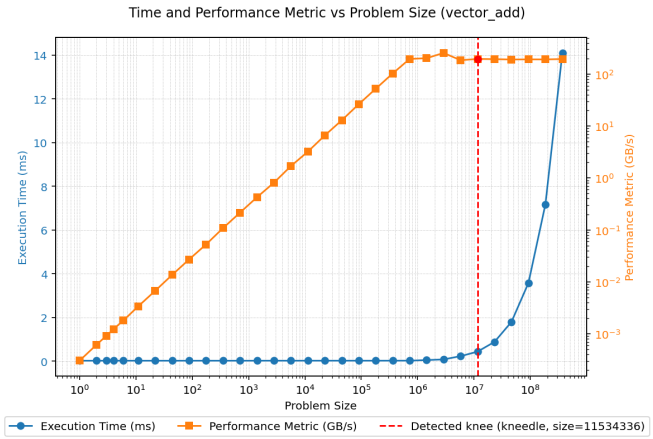
(b) Run 2



(c) Run 3



(d) Run 4

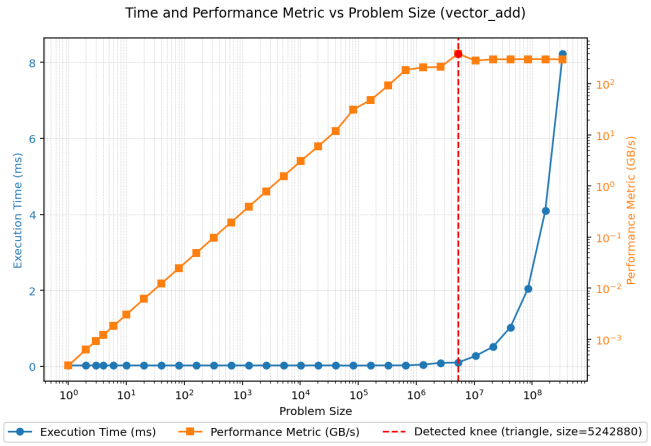


(e) Run 5

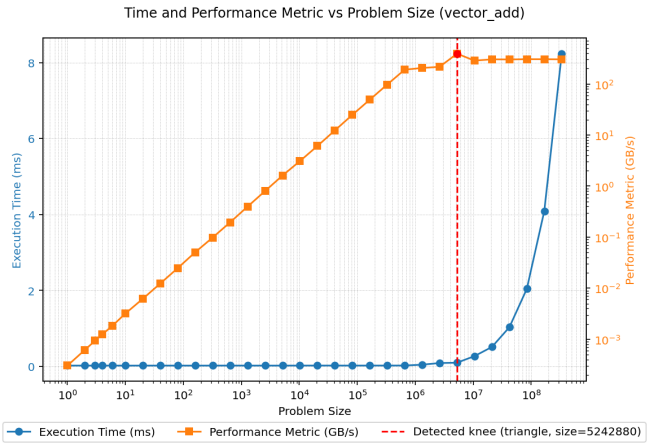
Figure 2: Vector Addition on RTX 4000 Ada using Kneedle method (five runs)

4.2.2 NVIDIA RTX 5000 Ada

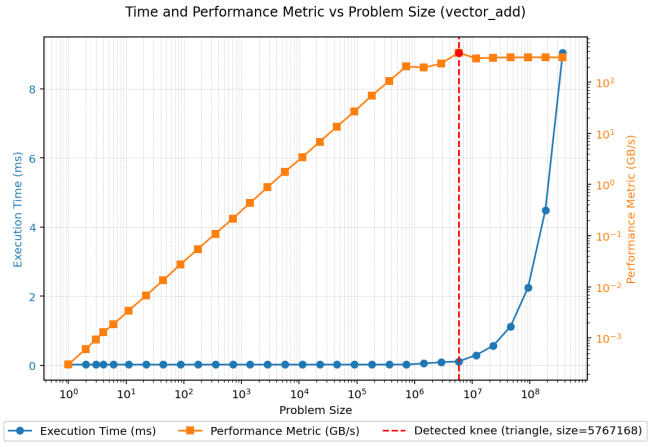
Triangle Algorithm Five Triangle-based runs were executed (Figures 3a–3e).



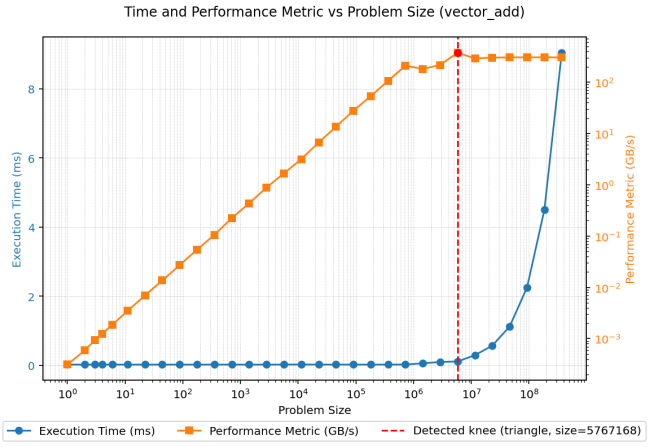
(a) Run 1



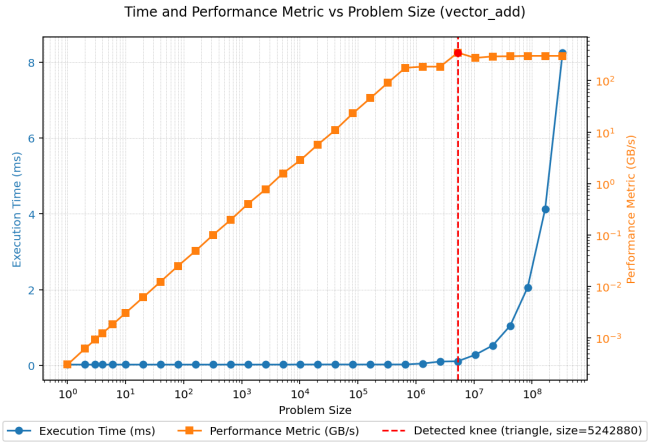
(b) Run 2



(c) Run 3



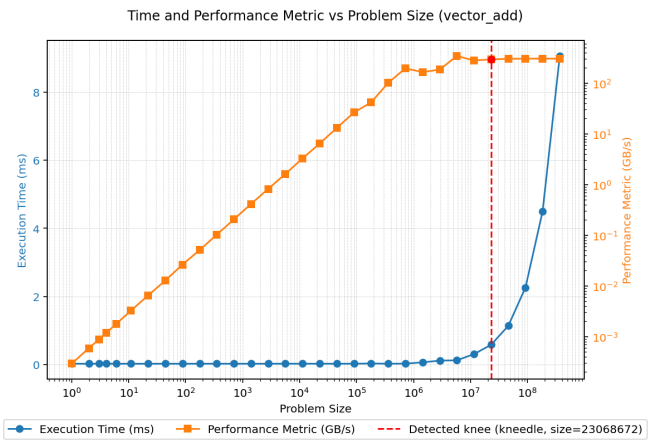
(d) Run 4



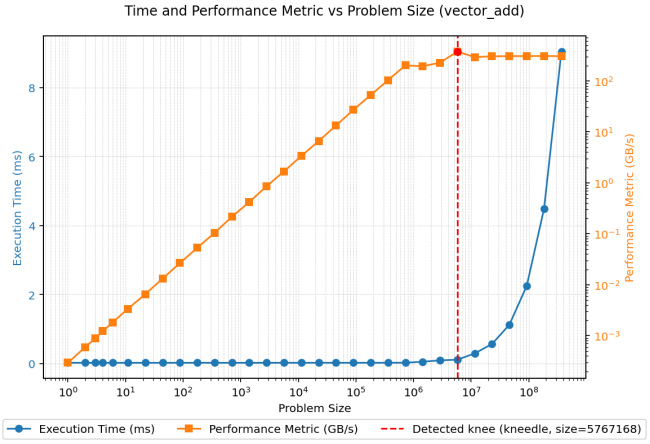
(e) Run 5

Figure 3: Vector Addition on RTX 5000 Ada using Triangle method (five runs)

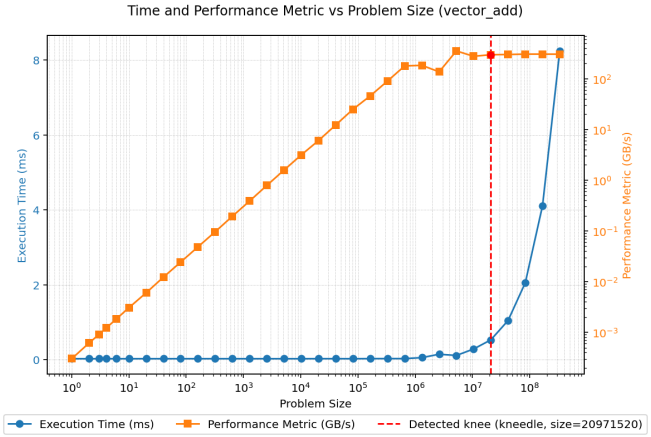
Kneedle Algorithm Kneedle algorithm results are given in Figures 4a–4e.



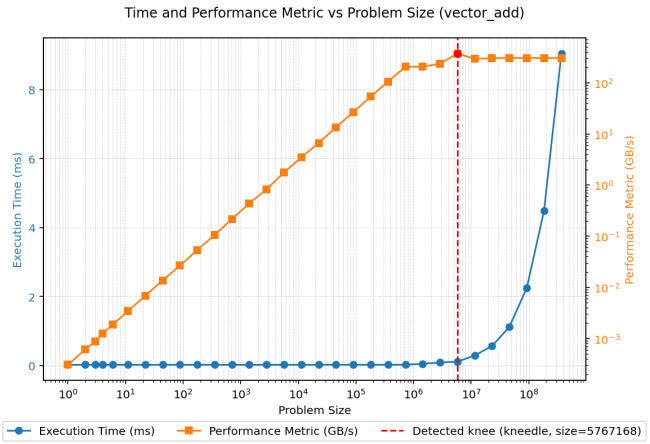
(a) Run 1



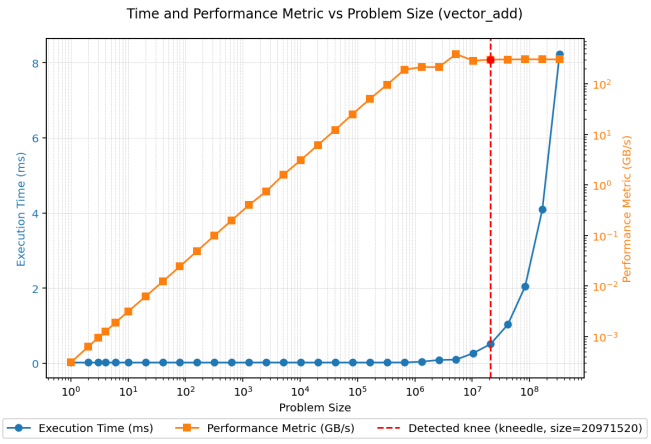
(b) Run 2



(c) Run 3



(d) Run 4



(e) Run 5

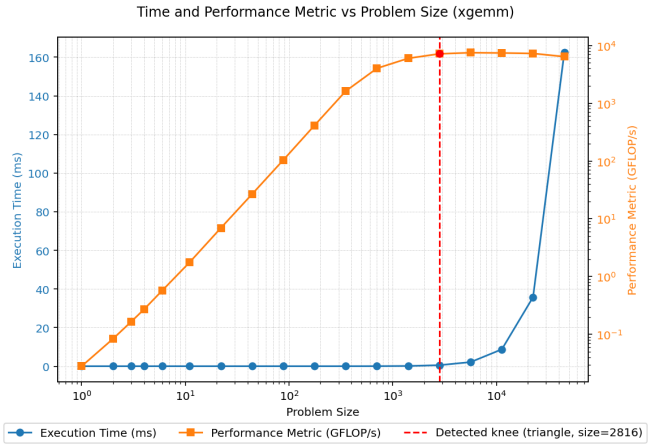
Figure 4: Vector Addition on RTX 5000 Ada using Kneedle method (five runs)

4.3 General Matrix Multiplication

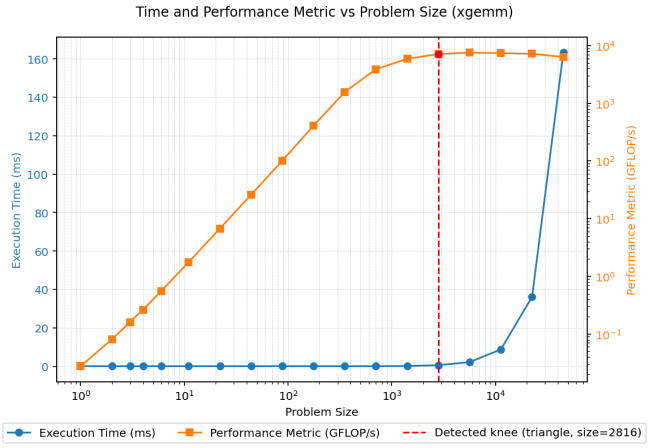
The GEMM kernel is a compute-bound benchmark, configured as described in Section 4.1.

4.3.1 NVIDIA RTX 4000 Ada

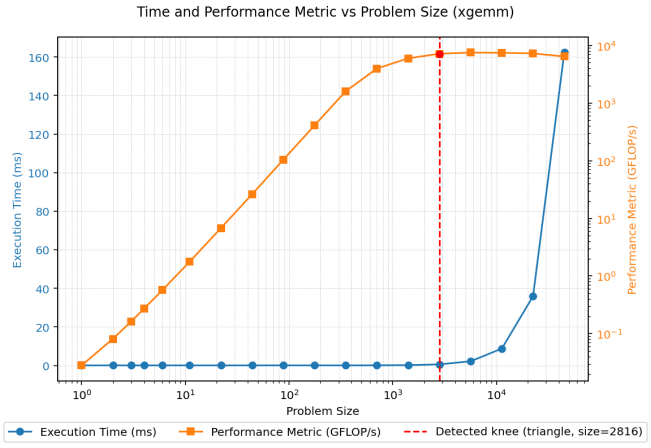
Triangle Algorithm Five Triangle-based autoscaling runs were executed (Figures 5a–5e).



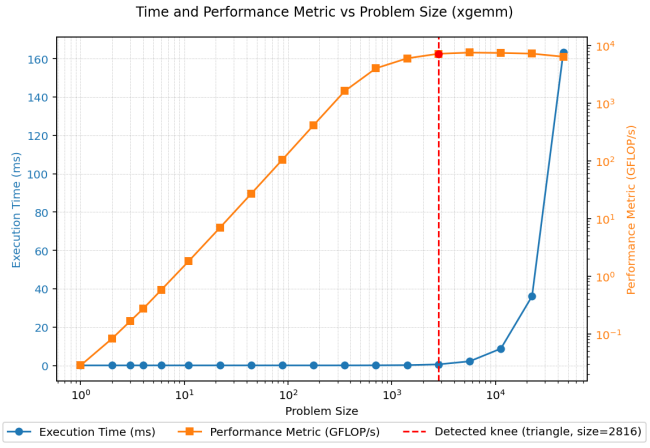
(a) Run 1



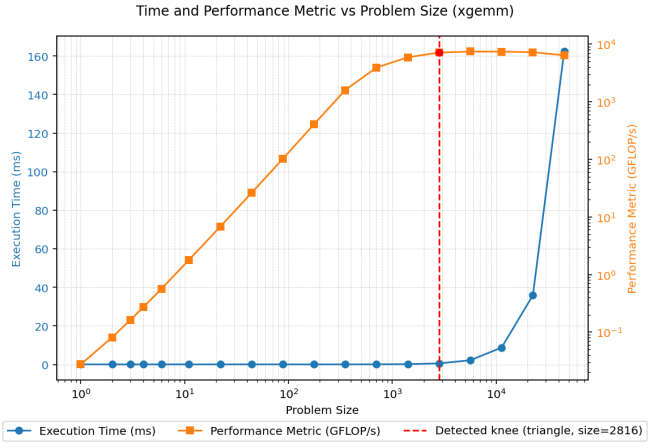
(b) Run 2



(c) Run 3



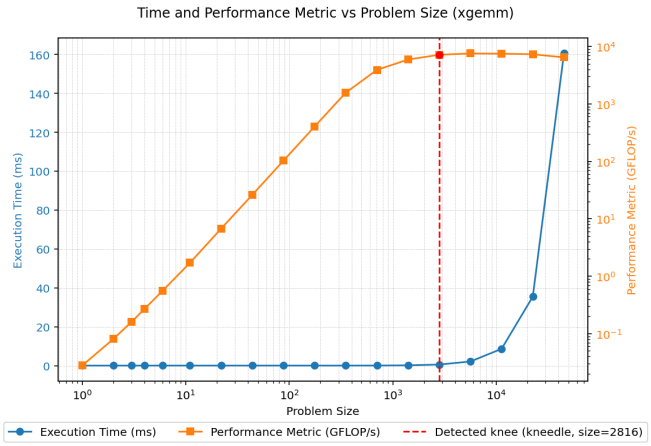
(d) Run 4



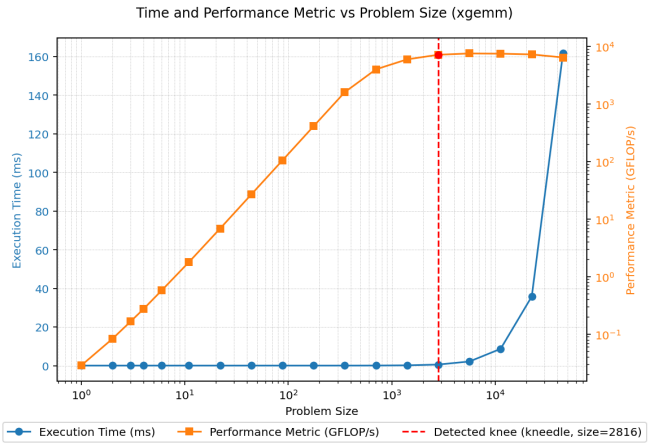
(e) Run 5

Figure 5: GEMM on RTX 4000 Ada using Triangle method (five runs)

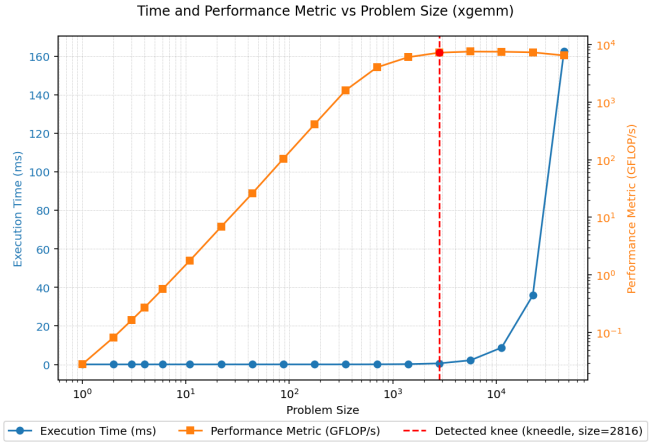
Kneedle Algorithm Results for Kneedle are shown in Figures 6a–6e.



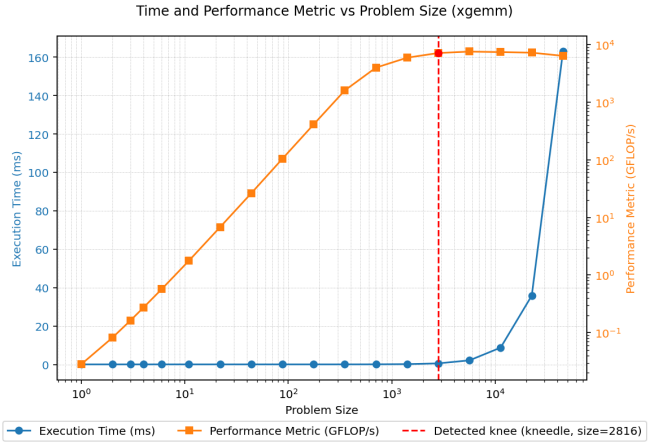
(a) Run 1



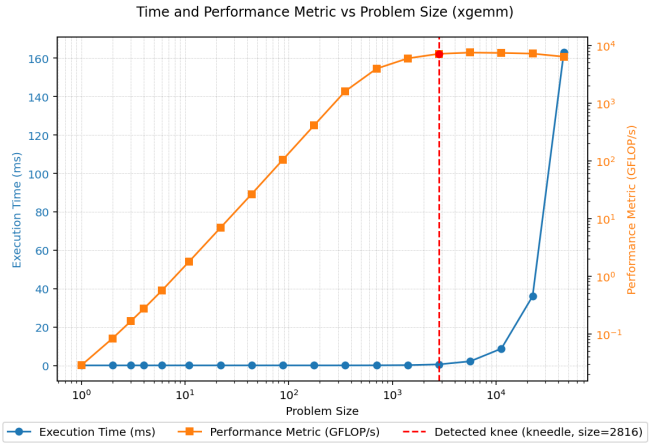
(b) Run 2



(c) Run 3



(d) Run 4

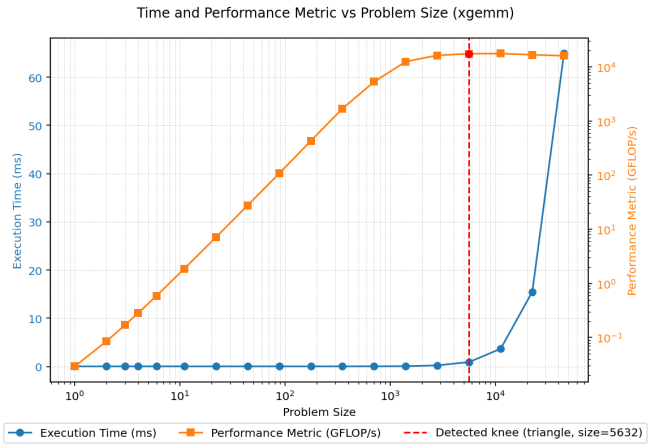


(e) Run 5

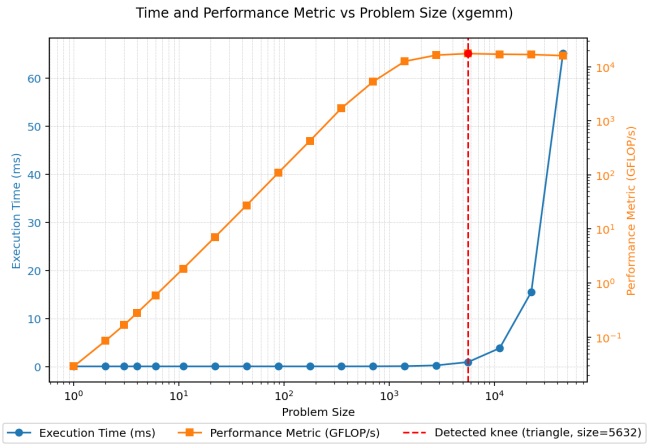
Figure 6: GEMM on RTX 4000 Ada using Kneedle method (five runs)

4.3.2 NVIDIA RTX 5000 Ada

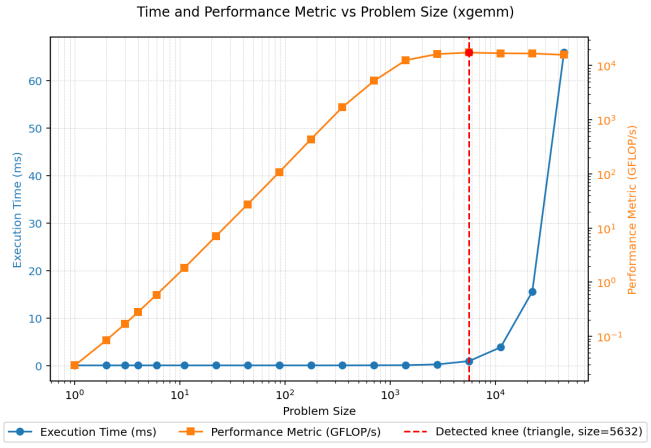
Triangle Algorithm Figures 7a–7e show the Triangle method results.



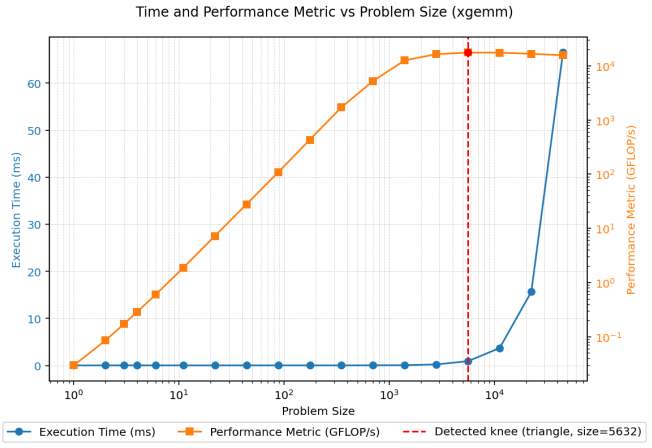
(a) Run 1



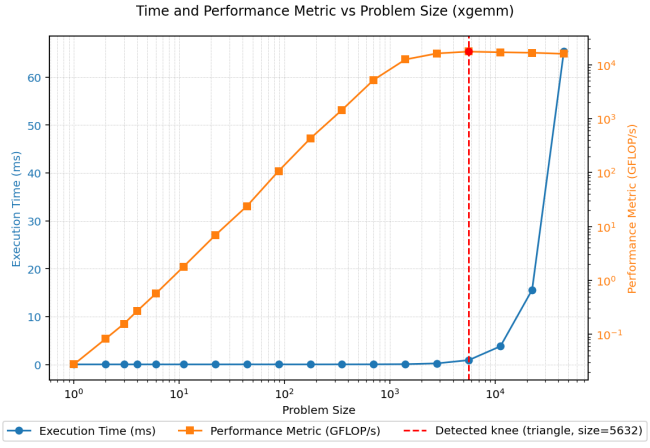
(b) Run 2



(c) Run 3



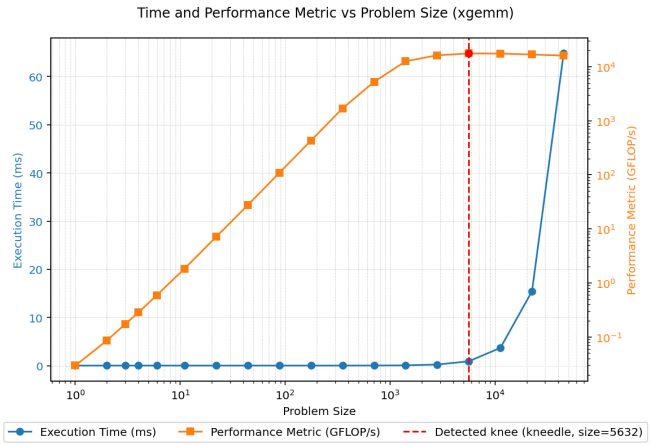
(d) Run 4



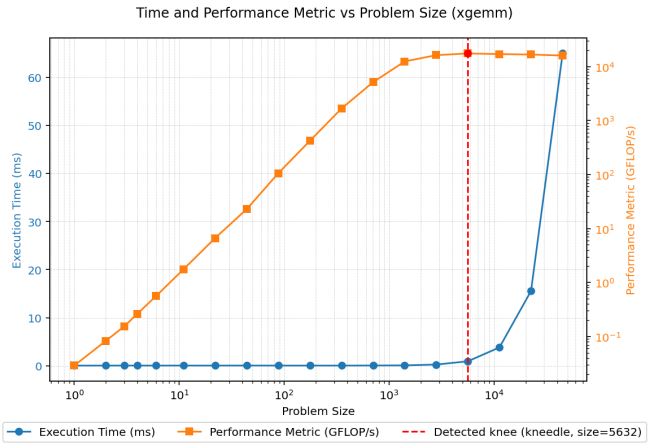
(e) Run 5

Figure 7: GEMM on RTX 5000 Ada using Triangle method (five runs)

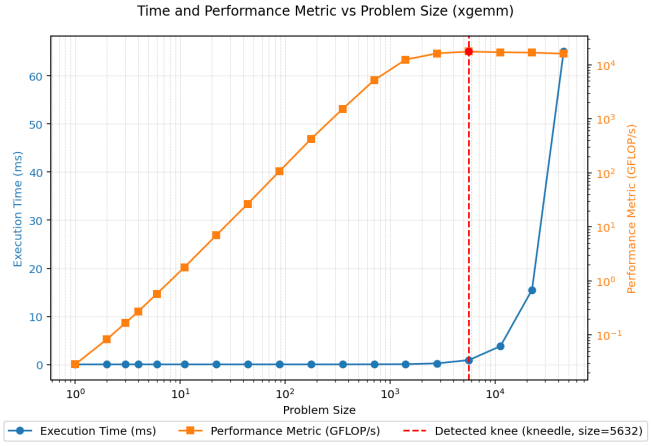
Kneedle Algorithm Figures 8a–8e show the Kneedle method results.



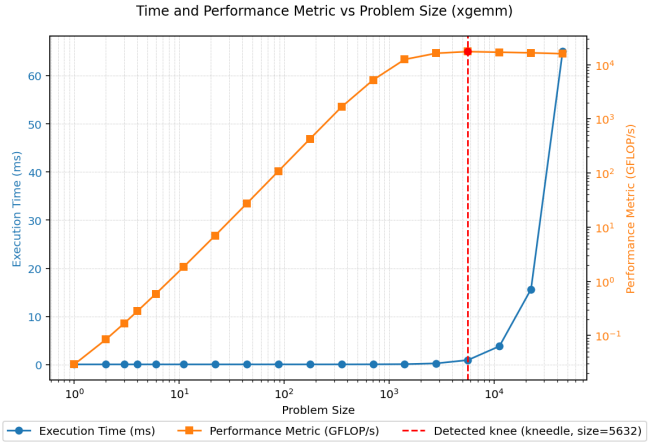
(a) Run 1



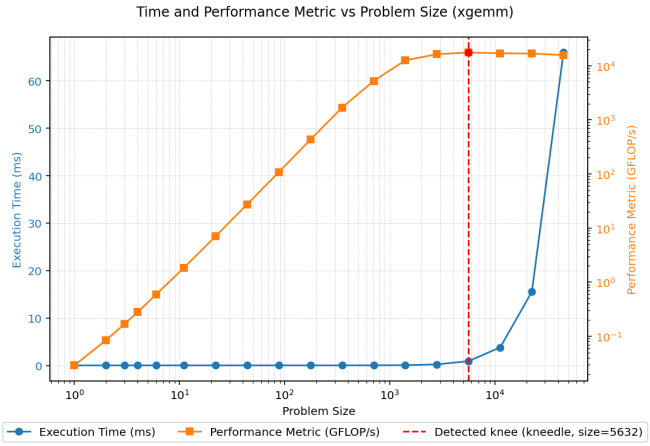
(b) Run 2



(c) Run 3



(d) Run 4



(e) Run 5

Figure 8: GEMM on RTX 5000 Ada using Kneedle method (five runs)

In summary, the `Autobench` framework successfully executed and visualized five independent

autoscaling runs for each kernel, device, and detection configuration. For both memory-bound (Vector Addition) and compute-bound (GEMM) kernels, the pipeline consistently produced interpretable results and saturation point detections. The next section discusses these outcomes, highlights trends, and interprets the observed patterns in performance scaling and saturation detection behavior.

5 Discussion

This section discusses the results of the `Autobench` benchmarking framework in the context of the research question. It reflects on the practical implications of the findings, identifies remaining limitations, and outlines promising directions for future work.
The aim of this thesis is to design a framework capable of automatically detecting the saturation point of GPU kernels, a point beyond which increasing input size no longer yields proportional performance gains. The results show that `Autobench` reliably produces interpretable, automated insights into kernel scaling behavior across both memory-bound and compute-bound workloads.
Across all experiments, both detection methods successfully identify a clear saturation point once the framework's filtering steps, such as the minimum execution time threshold, have been applied. For the compute-bound GEMM kernel, the detected saturation point is very stable: each run on both GPUs converges to the same problem size index, demonstrating that the methods are robust to noise in this setting. The memory-bound Vector Addition kernel, on the other hand, exhibits slightly more variability over repeated runs, with detected indices occasionally shifting a few steps earlier or later than the average. This variability is consistent with the greater sensitivity of memory-bound workloads to temporal bandwidth fluctuations, but the detected points nevertheless remain close to the practical saturation region and thus still provide reliable guidance.
A related observation is that, especially in the Vector Addition benchmark, the figures sometimes continue for more than three data points after a saturation point is detected. At first glance, this might seem like the framework is executing unnecessary continuation runs. However, this behavior is primarily caused by a small delay in the detection itself. In particular, a conservative setting of the Triangle method's `distance_threshold` or Kneedle's sensitivity parameter `S` can delay knee detection. Because the framework always executes three additional iterations after the first detected point to avoid premature termination, a delayed trigger results in what appears to be an extended continuation phase. In practice, this does not indicate a false detection, but rather that the knee (in this case) is less sharply defined in the memory-bound kernel compared to compute-bound kernel.
Taken together, these results suggest that both the Triangle method and the Kneedle algorithm are effective for automated knee point detection under the tested conditions. The slightly higher stability observed for GEMM reflects the smoother scaling curves of compute-bound kernels, while memory-bound kernels benefit more from the framework's filtering. Overall, the consistency between runs indicates that the filtering approach is a generalizable improvement and that the framework can reliably guide performance exploration without manual intervention.

5.1 Limitations

While the framework delivers stable results for the tested kernels and devices, several limitations remain.
First, the detection algorithms still rely on parameters such as the Triangle method's `distance_threshold` and Kneedle's sensitivity parameter `S`. These parameters are currently chosen manually and may need to be adjusted for kernels with unusual scaling patterns. As discussed above for the Vector Addition case, this sensitivity can sometimes delay detection of the saturation point, subsequently resulting in more continuation runs being executed than intended. Furthermore, the current execution time threshold is also fixed. While it effectively suppresses early noise in the tested scenarios, the optimal value can vary depending on the hardware, kernel characteristics, or

execution environment. Therefore, a static threshold may not be applicable in all cases.
Second, the experimental scope is limited. Only two kernels, Vector Addition and GEMM, were evaluated, and the experiments were conducted on two GPUs of the same vendor and architecture family (NVIDIA Ada). While these choices capture both memory-bound and compute-bound behavior, they do not cover the full spectrum of GPU workloads or hardware diversity. The framework's behavior for irregular, multi-phase, or application-level kernels remains untested.
Last, some planned features, such as local optimization refinements after initial detection, remain unimplemented. These could further refine saturation point accuracy and stability, and adapt to subtle performance curve variations.

5.2 Future Research

Several promising directions emerge for future work.
First, adaptive parameter tuning would help generalize the framework to a wider range of kernels and devices. For example, the execution time threshold could be scaled dynamically based on observed timing variance, or detection parameters could be auto-optimized using calibration runs.
Second, hybrid strategies that combine the strengths of both detection methods could be explored. For instance, Kneedle could provide an initial estimate, while Triangle validates and adjusts the point to guard against false positives.
Third, the planned local optimization pass should be implemented to allow refinement of initially detected saturation points. This could help recover from early misclassifications or detect multiple saturation stages in multi-phase kernels.
Fourth, more advanced search procedures such as binary or adaptive search could be used to refine knee detection more efficiently. These approaches would allow the autoscaler to narrow in on the precise saturation point faster than incremental scaling, reducing total benchmarking time.
Fifth, the recently published open-source library *Kneeliverse* [AEB+25], which includes a wide range of classical and state-of-the-art knee detection algorithms (e.g., Kneedle, L-method, Menger, Z-method, DFDT), offers a promising avenue for extending the current framework. Its recursive multi-knee detection capabilities and built-in pre/post-processing routines could help address several of the limitations encountered with parameter sensitivity and noise robustness in this work.
Last, broadening the scope of benchmarks is essential. Testing the framework on a diverse set of kernels, including those with irregular memory access patterns, synchronization barriers, or varying control flow, as well as on GPUs from different vendors and architectures, would provide a more comprehensive assessment of its strengths and weaknesses.

6 Conclusion

This thesis introduced `Autobench`, a flexible and extensible framework for autoscaling GPU kernel benchmarks. By combining automated saturation detection with detailed performance visualization, the system enables developers to better understand kernel scaling behavior across diverse GPU architectures.

After incorporating a minimum execution time threshold, the framework produced stable and accurate saturation detections for both memory-bound and compute-bound kernels, across all tested devices and algorithms. The Triangle and Kneedle methods, previously prone to early misclassification in certain cases, now also consistently deliver correct and reproducible results.

While opportunities remain for adaptive tuning, expanded kernel coverage, and refinement of detection strategies, `Autobench` in its current form already provides a reliable, modular, and interpretable tool for GPU performance analysis.

Overall, this project contributes a modular and interpretable approach to GPU benchmarking, offering both practical tooling and deeper insight into performance scaling behavior.

References

[AEB+25] Mário Antunes, Tyler Estro, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Kneeliverse: A universal knee-detection library for performance curves. *SoftwareX*, 30:102161, 2025.

[Arv23] Kevin Arvai. kneed (version v0.8.5), jul 2023. Software archived on Zenodo, accessed 2025-08-11.

[BEdL+16] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[DM98] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[DMM+10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[JW23] Morris A. Jette and Tim Wickberg. Architecture of the slurm workload manager. In Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 3–23, Cham, 2023. Springer Nature Switzerland.

[Ker22a] KernelTuner. First implementation of helper methods for energy tuning. https://github.com/KernelTuner/kernel_tuner/commit/84ce4900729c83e40c2d8528be9d7b1bead5f944, oct 2022. Accessed: 2025-08-22.

[Ker22b] KernelTuner. Initial version of model-steered clock frequency suggestions. https://github.com/KernelTuner/kernel_tuner/commit/083455a4e9aec032c7e9e0ef572e6ac535549064, oct 2022. Accessed: 2025-08-22.

[LHSvW24] Milo Lurati, Stijn Heldens, Alessio Sclocco, and Ben van Werkhoven. Bringing auto-tuning to hip: Analysis of tuning impact and difficulty on amd and nvidia gpus. In Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber, editors, *Euro-Par 2024: Parallel Processing*, pages 91–106, Cham, 2024. Springer, Springer Nature Switzerland.

[Lue08] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 836–838, 2008.

[Mun09] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314, 2009.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, March 2008.

[NC15] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202, 2015.

[Nug18] Cedric Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, New York, NY, USA, 2018. Association for Computing Machinery.

[SAIR11] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011.

[SRS+12] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127(7.2), 2012.

[SVVWB22] Richard Schoonhoven, Bram Veenboer, Ben Van Werkhoven, and K. Joost Batenburg. Going green: optimizing gpus for energy efficiency through model-steered auto-tuning. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 48–59, 2022.

[Tho53] Robert L. Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, 1953.

[van19] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.