Universiteit Leiden

Master Computer Science

Automating CUDA kernel optimizations: An LLM-driven framework for transforming naive CUDA kernels into optimized auto-tunable kernels

Name:	Nikita Zelenskis
Student ID:	2622157
Date:	20/08/2025

Specialisation:
Advanced Computing and Systems

1st supervisor:	Ben van Werkhoven
2nd supervisor:	Michiel van der Meer

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Achieving optimal performance on Graphics Processing Units (GPUs) often requires kernel tuning, a process of finding a set of parameters that have the highest possible performance for the given kernel and GPU architecture. However, the manual refactoring of standard, "naive" kernels into a form suitable for auto-tuning frameworks is a significant barrier, requiring considerable expertise and labor. This thesis investigates the potential of Large Language Models (LLMs) to automate this laborious task.

This thesis introduces a Python framework designed to make use of LLMs by generating optimized and auto-tunable CUDA kernels from the existing "naive" CUDA kernels. Three distinct strategies have been created for this framework and compared against each other: a "One-Prompt" approach, an "Explicit" strategy following a predefined sequence of optimizations, and a multi-step "Autonomous" strategy where the LLM first creates an optimization plan and then executes it iteratively. The effectiveness of these strategies is benchmarked across a suite of simple and complex CUDA kernels using various LLMs.

The results of experiments demonstrate that the iterative "Autonomous" strategy consistently resulted in higher performance gain, outperforming the "One-Prompt" and "Explicit" strategies. While LLMs can successfully apply standard CUDA optimizations, such as tiling and memory prefetching, their effectiveness is highly variable even within the same experiment. In particular, LLMs proved to be highly reliable for well-defined and straightforward tasks such as generating test data and extracting kernel metadata.

In conclusion, while LLMs are not yet a complete replacement for human expertise in High-Performance Computing, they can be a useful tool for implementing standard optimizations for kernels.

1 Introduction

Achieving maximum computational performance on modern hardware, particularly Graphics Processing Units (GPUs), is a common challenge in high-performance computing (HPC). A critical optimization technique is kernel tuning, where the code is adapted with specific parameters to match the underlying hardware architecture. While auto-tuning frameworks can discover these optimal parameters, the manual process of refactoring a standard, or "naive," kernel into an auto-tunable form presents a significant barrier, often requiring substantial time and expertise. This thesis investigates the potential of modern Large Language Models (LLMs) to bridge this gap. This thesis will explore the feasibility of automating the transformation of naive CUDA kernels into robust, auto-tunable code, examining different LLM-driven strategies to generate correct and efficient solutions.

1.1 Motivation

CUDA kernels can experience a significant boost in performance by setting the right parameters for specific hardware [RRB+08]. However, the search space can be rather large depending on the number of parameters. This makes manual exploration of the entire search space impractical, if not impossible. An auto-tuner can be used to automatically select the right parameters for the kernel. However, many kernels are still tuned by hand. This is because it can be a tedious and time-consuming task to rewrite a kernel to an auto-tunable kernel. Yet, it is practically impossible to find the optimal kernel without auto-tuning it.
Figure 1 shows an example of how a kernel could be rewritten to be able to tune `BLOCK_SIZE` and `UNROLL` parameters. After the code has been rewritten, the parameters need to be tuned. One of the easiest ways to do this would be to use an auto-tuner such as the "Kernel Tuner" [vW19]. Figure 2 shows an example Python code that uses Kernel Tuner to tune the `saxpy_tunable` kernel.

Naive kernel

```
// y[i] = a*x[i] + y[i]; block size fixed at 256
__global__ void saxpy(int n, float a, const float* x, float* y) {
    int i = blockIdx.x * 256 + threadIdx.x;
    if (i < n) y[i] = a * x[i] + y[i];
}
```

Auto-tunable kernel

```
__global__ void saxpy_tunable(int n, float a, const float* x, float* y) {
    int base = (blockIdx.x * BLOCK_SIZE + threadIdx.x) * UNROLL;
    #pragma unroll
    for (int k=0; k<UNROLL; ++k) {
        int idx = base + k * BLOCK_SIZE;
        if (idx < n) y[idx] = a * x[idx] + y[idx];
    }
}
```

Figure 1: Refactoring a naive CUDA kernel into an auto-tunable variant with tunable parameters (`BLOCK_SIZE`, `UNROLL`).

This example illustrates that, even for a very simple kernel, the effort required to tune it can be relatively high. Evidently, this effort increases significantly with more complex kernels. This

```python
import numpy as np
from kernel_tuner import tune_kernel

# Problem setup
n = 1 << 20
a = np.float32(2.5)
x = np.random.rand(n).astype(np.float32)
y_init = np.random.rand(n).astype(np.float32)

# Expected output for verification
y_ref = a * x + y_init

tune_params = {"BLOCK_SIZE": [32, 64, 128, 256, 512], "UNROLL": [1, 2, 4]}
grid_div_x = ["UNROLL"]
args = [n, a, x, y_init.copy()]
answer = [None, None, None, y_ref.astype(np.float32)]

results, env = tune_kernel(
    kernel_name="saxpy_tuned", kernel_source="saxpy_tunable_kernel.cu",
    problem_size=n, arguments=args, tune_params=tune_params,
    grid_div_x=grid_div_x, answer=answer, block_size_names=["BLOCK_SIZE"]
)
```

Figure 2: Kernel Tuner script exploring `BLOCK_SIZE` and `UNROLL`.

presents a challenge: while auto-tuning is essential for optimal performance, the manual effort required to enable it often leads to it being neglected.
Recent advances in LLMs have demonstrated their potential in code generation. This code generation might be useful for writing auto-tunable kernels. However, LLMs still lack correctness while (re)writing the code [LXWZ23][LLL+25].

1.2 Proposed solution

The main topic of this thesis is to explore the possibility of writing auto-tunable CUDA kernels from naive CUDA kernels using LLMs. This thesis will also explore the generation of correct and robust code with LLMs and test generation with LLMs. One promising approach to improve LLMs' performance in reasoning tasks is using chain of thought (CoT).
This work introduces a Python framework[1] that takes a naive kernel as input, iteratively modifies it, and provides a tuned kernel as output with the best parameters. This thesis will also look at different code generation strategies as well as which steps in the transformation and tuning process had the most impact.

1.3 Findings

Experiments indicate that:

- An iterative, multistep "autonomous" strategy, where an LLM first creates an optimization plan and then executes it step-by-step, yields significantly better performance gains than a single-prompt approach.
- LLMs can successfully apply standard CUDA optimizations such as tiling and memory prefetching, though their effectiveness is inconsistent and highly dependent on the

[1]The framework repository can be found at: `https://github.com/NikitaZelenskis/LLM-Kernel-Tuner`

specific kernel and LLM used.

- LLMs excel in simple, well-defined tasks such as generating test input data and extracting output variables from a CUDA kernel.

1.4 Thesis layout

This thesis is structured as follows:
Section 2 will provide some fundamental knowledge required for this work. It covers GPU programming, CUDA, kernel tuning, regression testing, some LLM concepts, and best practices.
Section 3 will discuss the current state-of-the-art in using LLMs for code generation, in particular for HPC applications. This section will highlight the gap this thesis aims to fill regarding automated kernel optimization.
Section 4 details the core contribution of this work: the design and implementation of the framework that was built for this thesis. It describes the high-level framework architecture, common challenges when working with LLMs, chosen solutions for these challenges, and the three chosen tuning strategies that will be evaluated.
Section 5 outlines the experimental setup used to assess the effectiveness of the proposed strategies. It specifies the hardware, the selected LLMs, the kernels that were benchmarked, and the metrics used to measure performance improvements.
Section 6 presents and analyzes the experimental results. It compares the performance of the different tuning strategies and LLMs across a set of benchmark kernels. This section will also go through the successful steps LLMs took while transforming the kernel.
Section 7 summarizes the key findings of the thesis, reflects on the effectiveness of using LLMs for kernel tuning, and discusses the overall contribution of this work.
Section 8 outlines potential directions for future research, including enhancements to the framework, more advanced tuning strategies, and alternative ways to leverage LLM capabilities for automatic kernel optimizations.

2 Background

This section will provide background knowledge for the main topics of this thesis. It will begin by establishing the fundamental concepts of GPU computing. We will use NVIDIA's CUDA to demonstrate how this is applied in practice, comparing a naive matrix multiplication kernel with an optimized version that uses shared memory. The challenge of achieving peak performance extends to kernel tuning, where parameters such as block and tile sizes must be optimized for specific hardware architectures. The vast search space makes manual tuning impractical, which has led to the development of automated tools, or "auto-tuners," to find these optimal configurations. Finally, this section will discuss some principles of using LLMs for complex tasks and how these techniques can be used to increase accuracy and robustness for code generation.

2.1 Kernel functions

A typical function on a CPU is expected to execute sequentially, instruction by instruction, as if it were run in a single thread.

To simplify, a CPU typically executes a single instruction per clock cycle on a single **word**, where a word refers to the fixed bit size of data that the processor can handle in one operation, such as 32 bits or 64 bits depending on the architecture.
This can be further improved with **Single Instruction Multiple Data (SIMD)**, which allows the execution of the same instruction on multiple words in the same clock cycle. A modern CPU has the ability to execute SIMD instructions on various data sizes such as 128 bits, 256 bits, or even 512 bits. This concept can be extended by executing the same instruction on even more words simultaneously. In principle, that is exactly what a GPU does. It executes the same instruction on thousands or even hundreds of thousands of words. A function that executes a large number of threads, each performing the same instruction in parallel on a GPU, is called a **kernel function**.
A **thread** that executes an instruction on a single word is known as a **kernel thread**. Multiple kernel threads form a **thread block**, and multiple thread blocks form a **grid**. This is illustrated in Figure 3, where the "add 2" instruction is executed in a single thread, in a single 4×1 block, and in a single 2×2 grid of 4×1 blocks.
Therefore, when launching a kernel, the number of threads must be specified by defining the **block size** and the **grid size**.

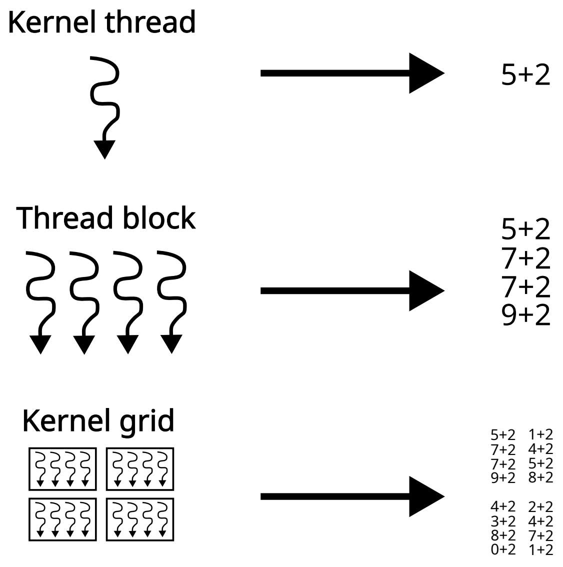


Figure 3: Illustration of how kernel threads form a thread block and thread blocks form kernel grids. The instruction being used is "add 2"

2.2 GPU programming

To be able to write GPU kernels, a specific language or a framework is used in almost all cases. At the moment, the most dominant language for writing GPU programs is CUDA. CUDA was developed by NVIDIA specifically for writing programs that could run directly on GPUs. CUDA is a C/C++ extension with a few added keywords specific for the GPU (e.g. `__global__`, `__shared__`).
Figure 4 shows an example of a naive matrix multiplication CUDA kernel. This kernel takes three matrix elements as arguments: A, B, and C. A and B are the input matrices and C is the

output matrix where the final result will be stored, all of these variables reside in the global memory of the GPU. The code in Figure 4 will launch BLOCK_SIZE × BLOCK_SIZE threads per block in parallel (in our case, $16 \times 16 = 256$ threads per block) and the total number of threads is determined by the grid dimensions (`blocksPerGrid`), where each thread is responsible for a single element of the output matrix C. Each thread accumulates the value in the variable `cValue` by looping over the common dimensions of A and B. After the loop, the computed `cValue` is stored in the corresponding element of the C matrix.

```
#define BLOCK_SIZE 16

__global__ void matMulNaiveKernel(Matrix A, Matrix B, Matrix C){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Each thread accumulates one element of C by accumulating results into cValue
    float cValue = 0;

    // C[i][j] = sum_k A[i][k] * B[k][j]
    // Iterates over common dimensions of A and B (k = A.width = B.height)
    if (row < A.height && col < B.width){
        for (int k = 0; k < A.width; ++k){
            cValue += A.elements[row * A.width + k] * B.elements[k * B.width + col];
        }
        C.elements[row * C.width + col] = cValue;
    }
}

                                ...

dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 blocksPerGrid((B.width + threadsPerBlock.x - 1) / threadsPerBlock.x,
                (A.height + threadsPerBlock.y - 1) / threadsPerBlock.y);
matMulNaiveKernel<<<blocksPerGrid, threadsPerBlock>>>(A, B, C);
```

Figure 4: Naive matrix multiplication kernel in CUDA [Kum24].

The `matMulNaiveKernel` CUDA kernel shown in Figure 4 is expected to compute much faster than an equivalent code for the CPU. However, `matMulNaiveKernel` could benefit from using some common optimization techniques, such as using the shared memory of the GPU.
The loop in the naive kernel iterates over the full row and full column. The key observation for optimization is that the elements of matrices A and B are read multiple times. In the naive implementation, these repeated reads access global memory, and specifically for matrix B, these accesses are non-coalesced, which is highly inefficient.
Figure 5 shows `matMulKernelSharedMem` CUDA kernel that utilizes shared memory of the GPU by implementing loop blocking (also known as tiling). Instead of computing the full dot product of each row and column, `matMulKernelSharedMem` computes the dot product of each tile first. The threads in the thread block cooperate to fetch all the elements needed by the thread block into shared memory once. This allows these elements to be accessed faster by all other threads in the thread block.
The shared memory of a GPU acts as a programmer-managed cache or scratchpad. This is why `matMulKernelSharedMem` is faster than `matMulNaiveKernel`, because it does not rely on the GPU's caching and prefetching and instead does the caching manually by using shared memory.
What is shown above is only one small optimization for a simple kernel. In reality, kernels can become hundreds of lines of code for relatively simple tasks. The (re)writing of performant

```
#define TILE_SIZE 16

__global__ void matMulKernelSharedMem(Matrix A, Matrix B, Matrix C){
    __shared__ float shared_A[TILE_SIZE][TILE_SIZE];
    __shared__ float shared_B[TILE_SIZE][TILE_SIZE];

    int globalRow = blockIdx.y * blockDim.y + threadIdx.y;
    int globalCol = blockIdx.x * blockDim.x + threadIdx.x;

    float Cvalue = 0.0f;

    int row = threadIdx.y;
    int col = threadIdx.x;

    for (int m = 0; m < (A.width + TILE_SIZE - 1) / TILE_SIZE; ++m){
        if (row < A.height && (m * TILE_SIZE + col) < A.width){
            shared_A[row][col] = A.elements[globalRow * A.width + m * TILE_SIZE + col];
        }else{
            shared_A[row][col] = 0.0f;
        }

        if (col < B.width && (m * TILE_SIZE + row) < B.height){
            shared_B[row][col] = B.elements[(m * TILE_SIZE + row) * B.width + globalCol];
        }else{
            shared_B[row][col] = 0.0f;
        }

        __syncthreads();

        for (int k = 0; k < TILE_SIZE; ++k)
            Cvalue += shared_A[row][k] * shared_B[k][col];

        __syncthreads();
    }

    if (globalRow < C.height && globalCol < C.width)
        C.elements[globalRow * C.width + globalCol] = Cvalue;
}

                             ...

dim3 blockDim(TILE_SIZE, TILE_SIZE);
dim3 gridDim((C.width + TILE_SIZE - 1) / TILE_SIZE, (C.height + TILE_SIZE - 1) / TILE_SIZE);
matMulKernelSharedMem<<<gridDim, blockDim>>>(A, B, C);
```

Figure 5: Matrix multiplication kernel that utilizes tiling and shared memory [Kum24].

kernels is a laborious task that requires a lot of knowledge of the software and hardware, as well as a lot of trial and error.
For example, "KernelBench: Can LLMs Write Efficient GPU Kernels?" [OGA+25] notes that it took 5 years from the release of the Transformer architecture to obtain performant kernels.

2.3 Kernel tuning and auto tuning

Both Figure 4 and Figure 5 have parameters that could be tuned for performance, `BLOCK_SIZE` and `TILE_SIZE` respectively. But every GPU has a different architecture. As a result of that, each kernel needs different parameters on different GPUs for optimal execution times. Having an optimal `BLOCK_SIZE` and `TILE_SIZE` ensures optimal performance of the kernel. The finding of optimal parameters for a kernel is called tuning.
Although auto-tuners exist, the process of refactoring a kernel into a tunable kernel remains largely manual, due to the high refactoring effort (see Section 1.1 for an example). The search

space can be extremely large and can grow exponentially. This makes it impractical to explore the entire search space manually.
Auto-tuners are designed to address this challenge of finding the optimal parameters for CUDA kernels.
As the name suggests, auto-tuners adjust the kernel parameters automatically, either by brute-forcing the whole search space or by using an optimization algorithm.
One such auto-tuner is "Kernel Tuner" [vW19], which will be used for all tests conducted in this thesis. Kernel Tuner was chosen because of its ease of use, flexibility, and because it is written in Python making it easy to work with LLMs as most frameworks and libraries are written for Python or have bindings for Python.

2.4 LLMs and CoT

There are many techniques that can improve the quality and accuracy of the code generated by LLMs. At the moment of writing, the most promising technique seems to be CoT [WWS+23]. CoT is a technique that tries to reason about a problem first before solving it. Often it does so by breaking a step into smaller steps first. This technique works because LLMs demonstrate greater proficiency in solving simpler problems, whereas they might struggle with larger problems. Furthermore, reasoning can sometimes be a way for LLMs to check their own correctness.

2.4.1 Manual CoT and context size accuracy

Most of the time, when a prompt is written for an LLM, it must adhere to multiple criteria, thus creating multiple instructions for the LLM to execute. Figure 6 shows an example of this prompt.

Prompt:
1. Give a step by step recipe for a pizza.
2. Use only products that are available in dutch super markets.
3. The price of the pizza should not exceed 10 euros.
4. Rate the difficulty of each step on a scale from 1 to 10.
5. At the end of each step use an emoji.
6. Return only a JSON string with steps for recipe, prices per ingredient, the amount of each ingredient needed and the difficulty per step.

Figure 6: An example prompt that has multiple instructions. The numbering is purely for the ease of readability and would not exist in the final prompt.

Although this is an extreme example, it illustrates how an LLM can be overwhelmed by the need to adhere to the whole prompt at once. Not only does each instruction increase the context size, which can reduce the accuracy of LLMs. However, it also has multiple instructions, which further reduces the accuracy of the models [MDD+25][LZLC24][YXJ+25][Lan24]. In addition, multiple distinct instructions within a single prompt can make it harder for the model to maintain consistent reasoning. Instead, a user who already knows the desired workflow can split the task into several smaller prompts, interleaving model calls with deterministic, non-LLM processing steps. This guided prompting approach allows the user to handle parts that do not require the LLM to reason while ensuring that each model call has a focused and well-scoped objective. This can be described as "manual" or "guided" CoT and has been shown to be even better than normal CoT in various tasks[WWS+23]. Figure 7 shows an example of how the previous prompt could be split and made into such a workflow.

Workflow steps

Step 1:

input:	Give a step by step recipe for pizza. Use only products that are available in dutch supermarket. The price of the pizza should not exceed 10 euro.
output:	Here is how you can make a great pizza ... Step 1: ... Step 2: ... ...

Step 2:

input:	Here are steps for making a pizza: """⟨output of step 1⟩""". Rate the difficulty of each step on a scale from 1 to 10
output:	Here is the breakdown of the difficulty for each step: Step 1: ... Step 2: ... ...

Step 3:

input:	Add a fitting emoji after each step: """⟨output of step 1⟩"""
output:	Step 1: ... :) Step 2: ... >:-)

Step 4:

input:	Return a JSON of the following format: """⟨json_schema⟩""" from this text: """⟨output form step 2⟩, ⟨output form step 3⟩"""
output:	{ "steps": [{ "step": "...", "difficulty": "..." }, ...], "ingredients": ["item": "...", "amount":{"units": "...", "measurement": "..."}] }

Figure 7: Prompt chain workflow with LLM.

For a more difficult or critical task, a verification step could be added. Consider the workflow shown in Figure 7 as an example. If it is critical that the price is not exceeded, a manual confirmation without LLM could be added that verifies the price does not exceed 10 euros. However, in some cases, this can be very difficult or even impossible. In the current example, without a database of all possible products, it is impossible to estimate the price of all ingredients. Therefore, there could be a verification or repair prompt that would ask LLM to estimate the cost of all the ingredients for a certain country.
The last step of the workflow in Figure 7 might seem redundant as most LLMs support structured output or function call natively. This however, is not the case for all LLMs, for example

DeepSeek models do not support structured output or function calling. Furthermore, structured output can reduce the performance of LLMs. [CCN24] [TWT+24]

2.5 Regression testing

Because of the LLM hallucinations it is crucial to have reliable tests. Most sophisticated testing techniques try to eliminate bugs in the original code. So in a sense, there is an assumption that there is a bug in the original code. In our case, we are going to modify the original kernel and assume that the original kernel code is always correct; thus, we cannot use this approach. Regression testing, on the other hand, is a testing method that tries to prevent the creation of future bugs as the code is modified. Even if the given kernel has a bug in it, the output of the new kernel needs to stay the same for all inputs throughout the tuning process. Having the assumption that the original code is always correct makes it easier to generate correct tests, as we can evaluate correctness of the input and the output pairs from the original kernel. However, generating all possible inputs of a kernel is practically impossible as the input space of a typical kernel is virtually infinite.
This means that even if the generated input and output pairs are the same, the original kernel and the newly generated kernel might still be different.
Consider the following example to illustrate how such a situation can arise in practice. Suppose a simple max function which takes an array of integers as input and returns the largest integer of the input array. Now imagine introducing a bug to this function where the first half of the input array is ignored. In that case, there is the possibility that our test will not find this bug if the largest integer happens to be in the second half of the input. This issue may arise from an incorrect thread index calculation. Here is an extreme code example of how this can occur: Instead of using `int idx = threadIdx.x + blockDim.x * blockIdx.x;` for the index, when `int idx = 1 + blockDim.x * blockIdx.x;` is used instead, it would mean that only one element inside the thread block is used. Regression testing would easily be able to mitigate this bug if the test input includes the largest element in the first half of the array, as the mismatch with the original kernel output would be detected.
However, this approach cannot guarantee that all such issues will be caught, as it is infeasible to test every possible input. This is a fundamental problem that cannot be eliminated. However, to maximize coverage, it is possible to generate multiple inputs as well as try to find the edge cases.

3 Related work

It appears that there are no peer-reviewed papersomes that would try to auto-tune kernels using LLMs. There are, however, some papers that are closely related. This section reviews some relevant literature within the topic of this thesis.

3.1 LLMs for writing GPU kernels

A paper that is the most closely related to our approach is by Ouyang et al. and describes a method that they used to improve GPU kernels using LLMs. The paper introduces Kernel-Bench [OGA+25]. A suite of 250 PyTorch tasks that focus on machine learning. The tasks in this suite are written in PyTorch and LLMs are expected to improve them by either rewriting

some of the PyTorch functions in the workflow or by fusing multiple PyTorch functions into one. For example, one of the tasks was the Softsign activation function that used the following PyTorch code: `x / (1 + torch.abs(x))`. When this function is called, in the backend three separate GPU kernels are being executed; first `abs` kernel then `add` kernel for adding one to the tensor and lastly `div` kernel. Claude-3.5 Sonnet was able to create its own Softsign kernel that fused these three kernels into one, which led to a 1.3x speedup. Even though this approach generates some GPU kernel code, the main focus is on improving the PyTorch workflow, not the underlying GPU kernel code. Furthermore, this approach does not perform any parameter tuning, which is a crucial step in optimizing GPU kernels. In theory, their approach could attempt to tune its generated kernels by iteratively trying multiple parameter configurations and measuring their performance. However, such a process is highly unlikely to emerge without explicit guidance and would be prohibitively expensive in terms of computational cost.
Lastly, tests for all tasks were already written, which in itself can be a tedious task.

3.2 LLMs in High Performance Computing

Several papers have looked at how LLMs can be utilized for use in HPC. One of such papers introduces ParEval [NDX+24], a benchmark that evaluates LLM's abilities of generating parallel code. It showed that LLMs are worse at writing parallel code compared to writing serial code. The highest success rate in writing the correct code (not necessarily a performant code) was from GPT-4 and had a success rate of 60% for the OMP code, $\approx 52\%$ for Kokkos, $\approx 23\%$ for MPI and $\approx 37\%$ for CUDA. However, ParEval also tested the ability of LLMs to "translate" code from one execution model to another. This approach had a higher success rate than writing code from scratch. Translating the code from serial to OMP had a success rate of $\approx 70\%$ and from serial to MPI $\approx 50\%$. Unfortunately, ParEval did not test the code translations to CUDA.
Another paper introduced HPC-GPT [DCE+23], a fine-tuned LLaMA model for specifically HPC usage. The main purpose of this fine-tuning was to detect race conditions. Their approach was to generate dataset by utilizing static and dynamic analysis tools to detect race conditions on existing code samples. That was then used as an input-output pair for training.
Furthermore, Bowen Cui et al. [CRHZ25] and William F. Godoy et al. [GVLT+24] have built systems that can automatically parallelize serial code using different prompting techniques and utilizing agent approach.
Some of the ideas described above can be used to generate fine-tunable code. However, none of those techniques focus on fine-tuning CUDA kernels using LLMs.

3.3 Code generation with LLMs

Prompt engineering is known to increase the correctness and robustness of code generation [DAKC23] [HV23] [WHF+23].
Some techniques like COCO [YCZ+23] and SymPrompt [RJS+24] that try to improve the performance of the LLMs by utilizing abstract syntax tree or control flow of the program. Other techniques like CodeT [CZN+22] and AceCoder [LZL+23] can improve the correctness and robustness of the code by having better examples or by generating tests for a given prompt. Lastly, there are techniques like RepoCoder [ZCZ+23] that look at the code in the underlying code base to provide a better context for the LLM to understand its task.

Although these techniques help with the performance of the code generation in the benchmarks. Ultimately, performance comes from the underlying performance of the LLMs in use. Although prompting techniques can enhance performance to some extent, the overall potential remains in the strength and limitations of the underlying model. Therefore, this will not be the main focus of the thesis.

3.4 Transpilation

Transpilation seems to be a close match for the problem at hand, where the original version of a kernel needs to be rewritten into an alternative version such that for all inputs, both versions produce the same output. Several papers have explored transpilation with LLMs, often focusing on ensuring the correctness of the translated code. For example, VERT [YTP+24] transpiles the code to rust. For correctness, it uses rust compiler errors as the first step to repair the code. As the authors have pointed out themselves, rust compiler errors are much more verbose than CUDA errors. Therefore, rust compiler errors are more useful to LLMs than CUDA compiler errors.
Batfix [RLM+24] also uses compiler errors in the first phase and in the second phase it uses control-flow to verify code correctness. This approach will not work for CUDA kernels because the control flow can change significantly between each optimization step.

4 Methodology

To address the research gap identified in the previous section, this thesis introduces a framework[2] designed to automate the transformation and tuning of CUDA kernels using LLMs. This framework was developed to support iterative LLM prompting with kernel auto-tuning in between code generation steps that evaluate the kernel. The goal of the framework is to make it easy to use different code and test generation techniques. To ease the future research of auto-tuning kernels with LLMs considerable effort was made to ensure that the framework remains maintainable and extensible.

4.1 Framework design

This section will describe some core framework design decisions, explain why the decisions were made, and discuss the pros and cons of those decisions.
One of the core philosophies throughout the whole framework is the idea of extending CoT and creating a manual CoT (see Section 2.4.1 for explanation).

4.1.1 Used frameworks

One of guiding principles in the framework's design was to minimize the number of external dependencies. However, re-implementing certain features from scratch would be impractical. Therefore, a select few established frameworks were chosen for their specific capabilities:

- **LangChain and LangGraph** – LangChain and LangGraph were used for interactions between the different components and LLM. LangChain already has integration with a

[2]The framework repository can be found at: `https://github.com/NikitaZelenskis/LLM-Kernel-Tuner`

wide variety of APIs. This makes it easy to switch between different LLMs.
Likewise the use of LangGraph allows for more flexible workflows while generating kernel code and test generation.

- **Clang** – To extract structural information about the kernel such as the name of the kernel, argument names, and their types, clang was used.
- **Kernel Tuner** – For the execution, correctness verification, and performance tuning of CUDA kernels, Kernel Tuner was used.

4.1.2 Context management

Instead of maintaining a running conversation history, each LLM call operates within a new context window, discarding previous conversation history except in the cases where an exception was raised. This strategy provides two main benefits. First, the context length, and thus the cost per invocation, will be reduced. Second, this allows LLMs to focus specifically on the current task, enhancing the accuracy of the task as described in Section 2.4.1. However, this method has two drawbacks; the first drawback is that the LLM cannot see previous interactions. This can lead to the generation of duplicate tasks as well as duplicate code. In the ideal scenario, the LLM would be able to see its own previous attempts and only create new tasks and new code; however, current LLMs lack the accuracy when the context size increases. The second drawback is that the KV cache cannot be utilized, which could increase the cost and time it takes to process each invocation. And last drawback is that the LLM will not be able to see the bigger picture, it will lack awareness of its previous optimization attempts and the rationale for choosing those specific optimizations. This can lead to performance degradation.

4.1.3 Fault management

By the nature of LLMs it is expected that they will generate code that might crash, code that might run in an infinite loop, or not generate any code at all. Therefore, safeguards must be in place to ensure that the main process does not crash and can continue to execute even if the code generated by an LLM is faulty. The following things have been implemented for this purpose:

1. Executing all generated code by LLMs in a separate process. This ensures that the main process does not crash. Even if some unrecoverable state has been reached, the subprocess should be able to be killed without any issues.
2. Timeouts were implemented to prevent processes from hanging indefinitely, especially when executing LLM-generated code that might contain infinite loops or be unexpectedly slow. If the execution of a generated code segment exceeds a predefined time limit, the subprocess is terminated. This ensures that the system remains responsive and can proceed, potentially by updating the state and reporting timeout to the LLM.
3. Retry mechanism in the form of a retry wrapper. The retry wrapper describes how the underlying function or the LangGraph workflow should be rerun, how the state should change between reruns, and how many times to retry. In the retry wrapper, each exception is mapped to a specific handler. A handler is a function that describes how the state needs to be changed before retrying. It is therefore expected to have

custom exceptions in place to be able to handle each exception separately. If the caught exception is not specified in the retry wrapper, the retry wrapper executes the default handler. Here is an example of how an retry policy might look like:

```
def timeout_handler(state: Dict[str, Any], error: Exception) -> Dict[str, Any]:
    #add a retry message to the current context
    state["messages"].append(HumanMessage(retry_prompts.timeout_prompt))
    return state

retry_policy = RetryPolicy(
max_retries=3,
handlers={
    TimeoutError: timeout_handler,
})
```

And this is how to wrap a function into a retry wrapper with the newly created retry policy:

```
def llm_invokation(state: State) -> State:
    # ...
    raise TimeoutError("Timeout reached")
    # ...

retry_llm_invokation = create_retry_wrapper(
    llm_invokation,
    retry_policy
)
retry_llm_invokation.invoke(...) # will call llm_invokation
```

4.1.4 Helper functionality

Structured output As described in Section 2.4.1 not all LLMs support structured JSON output, and forcing it can reduce performance. To address this, the framework provides two alternative methods. That can be invoked similar to the LangChain's built in `.with_structured_output()`.
The first is the "separate request" method, which uses two LLM invocations to generate structured output. The answer is generated by the LLM without structured output in the first invocation, and in the second invocation, the LLM is asked to jsonify the output into a certain JSON schema.
The second method is a "hybrid" approach and is similar to the first method where LLM is invoked multiple times, with the difference being that the second invocation is called with structured output enabled. Both of these methods remove the restriction of reasoning, which should increase the accuracy of the answer.
The first method is useful for when the model does not have a built-in structured output available such as DeepSeek-R1. The second method is useful for allowing the model to "think" in the first invocation and only then generate structured output in the second invocation. The second method uses fewer tokens and is therefore faster and cheaper.

Thinking stripper Thinking models produce `<think>` `</think>` tokens for their reasoning before giving the answer. For when these tokens are in the output of the LLM, thinking stripper can be used. When the thinking stripper is enabled, all requests will be sanitized with the thinking stripper, removing those thinking tokens and everything that is in between these tokens, leaving only the main output.

Test generation Test generation is a part of any transformation process, as there needs to be a guarantee that the code still produces the same output between transformations. For this purpose a general helper function called `get_test_from_code` exists in the `BaseTesting Strategy`. This function helps to generate test input and output pair for the kernel from the provided Python code by executing it in a separate thread and extracting the produced input.

Testing the kernel LLM generated code needs to be tested. For this purpose, the framework has a helper function called `_run_tests` inside the `BaseTuningStrategy` class. It tunes the first test to get the best tuning parameters for the provided kernel and compares the output of the remaining tests with those tuning parameters to save time. Additionally, a simple caching was added that skips the testing process if the provided kernel code and tuning parameters are exactly the same.

Tunable parameter restrictions Some kernels have restrictions on the combination of the tunable parameters with which the kernel can work. Figure 8 shows one of such kernels, `matmul_kernel` will have invalid memory accesses if the block size is not a square. Kernel Tuner has built-in functionality for this purpose, it can take restrictions in the form of a list of strings for example: `["block_size_x==block_size_y"]` for the `matmul_kernel`. As this is a common case for all tuning strategies, a helper function was built that would ask LLM to generate such restriction in the form of an array of strings.

```
__global__ void matmul_kernel(float *C, float *A, float *B) {
    __shared__ float sA[block_size_y][block_size_x];
    __shared__ float sB[block_size_y][block_size_x];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int x = blockIdx.x * block_size_x + tx;
    int y = blockIdx.y * block_size_y + ty;

    float sum = 0.0;
    int k,kb;

    for (k=0; k<WIDTH; k+=block_size_x) {
        __syncthreads();
        sA[ty][tx] = A[y*WIDTH+k+tx];
        sB[ty][tx] = B[(k+ty)*WIDTH+x];
        __syncthreads();

        for (kb=0; kb<block_size_x; kb++) {
            sum += sA[ty][kb] * sB[kb][tx];
        }

    }

    C[y*WIDTH+x] = sum;
}
```

Figure 8: CUDA kernel with restrictions on block size

4.1.5 Framework architecture

The transformation process, illustrated in Figure 9, is executed as a multistage LangGraph workflow. This workflow structure allows for a clear separation of concerns, with a shared state

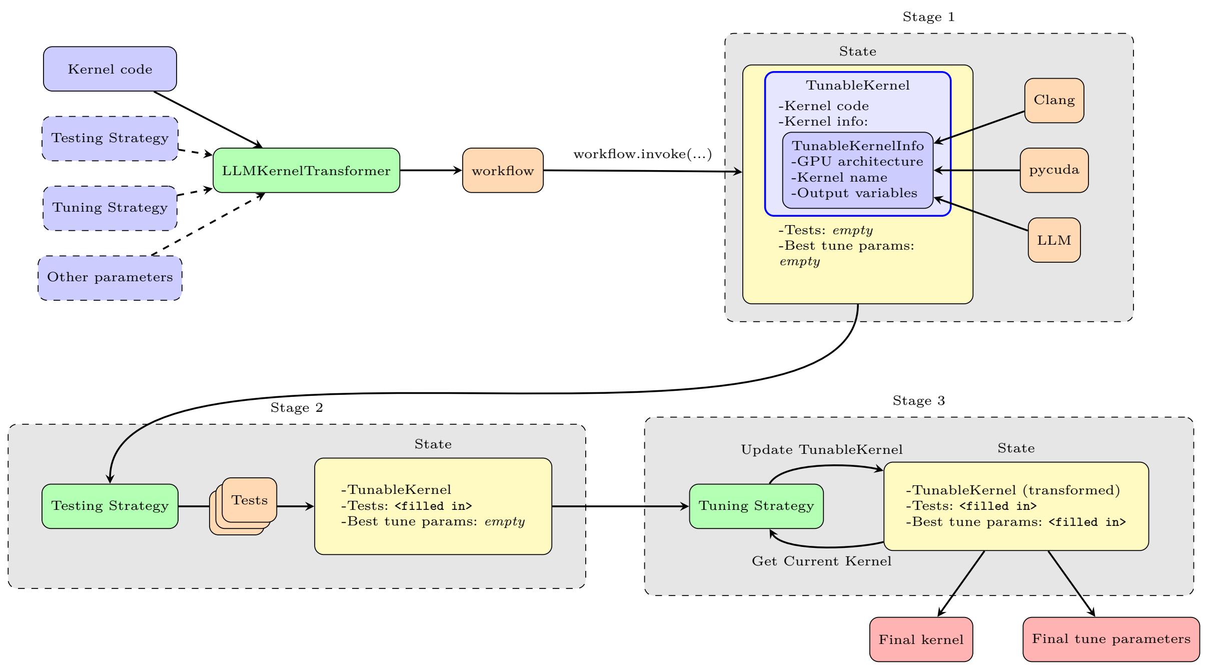


Figure 9: High-level overview of the framework's `LangGraph` workflow.
(Stage 1) Initial kernel code is analyzed using static tools (`clang`, `pycuda`) and an LLM to populate a `TunableKernel` object in the shared state.
(Stage 2) A `TestingStrategy` uses the initial kernel information to generate a suite of tests, which are added to the state.
(Stage 3) A `TuningStrategy` iteratively transforms and evaluates the kernel, using the tests for verification, until an optimal version is found. The final outputs are the optimized kernel and its tuning parameters.

being passed and modified between stages. The `LLMTransformer` class serves as the primary orchestrator, which takes user input such as the initial kernel code, a chosen `TestingStrategy`, and a `TuningStrategy`, and then instantiates and runs the workflow.

Stage 1: Static and Semantic Analysis The workflow begins by populating the kernel's metadata. As shown in Stage 1 of the figure, static analysis tools such as pycuda and clang are used to extract concrete information such as the target GPU architecture in which the test will be run and the kernel name. Concurrently, an LLM is engaged to extract more abstract, semantic information that is difficult to parse automatically, such as a natural language description of the kernel's purpose, the problem size variables (critical for tuning), and the primary output variables. All of this information is aggregated into a singular TunableKernelInfo object. This object is then referenced by the initial TunableKernel instance, which holds the kernel's code. Throughout the transformation process, new TunableKernel instances representing different code versions will be created, but they will all reference this same TunableKernelInfo object, ensuring metadata consistency without duplication. At the end of this stage, the state contains a fully annotated TunableKernel but lacks any tests or tuning results.

Stage 2: Test Generation The workflow then transitions to the Test Generation Phase. The state from Stage 1 is passed to a TestingStrategy component. The responsibility of this component is to generate a set of KernelTest objects. Each KernelTest defines a specific test case with concrete input data, expected output, and problem size. These tests are crucial for verifying the correctness of all subsequent kernel transformations. The generated tests are added to the workflow's state, which is then passed to the final stage.

Stage 3: Transformation and Tuning The final major stage is the Kernel Transformation and Tuning Phase, managed by the selected TuningStrategy. This stage operates as an iterative loop. In each iteration, the TuningStrategy applies a transformation to the current TunableKernel and discovers new tuning parameters. The transformed kernel is then compiled and evaluated against the test cases generated in Stage 2 to ensure correctness and measure performance. The results are used to update the state, which may include a new TunableKernel instance with the transformed code and its measured execution time.

Upon completion of the workflow, the final state yields the two primary artifacts: the best-performing, verified kernel code and the set of final tuning parameters that produced it.

4.2 Testing strategy

A testing strategy generates tests. As described in Section 2.5, most sophisticated testing techniques will not work. What is needed is a regression testing method that would ensure that the output stays the same for all inputs of the kernel between code transformations. This means that a test for our purposes should describe an input and an output pair. The input and output pairs will also be used for performance evaluation purposes. This is done by comparing the execution time of the kernel for a specific input. This means that the time to execute the input for a kernel should be large enough to be able to see the execution time difference, in most cases in a scale of a few million elements. Generating such large input and output data directly by an LLM is impractical and even impossible due to context length and costs. Therefore, a test is generated using Python code. Figure 10 shows an example code that is expected to be generated by an LLM for a simple add kernel. This code will be placed in a Python template and executed in a separate thread, and the input is then extracted by the main thread. After that, the input is passed to the original kernel to generate the output. The kernel is timed while it is generating the output. If the time to generate the output is too small, it cannot be used as a benchmark because the GPU will be underutilized. On the other hand, if the time to generate the output is too large, it would take a lot of time to run the parameter tuning. For this purpose, a minimum and maximum duration of a test can be set, as well as the data size of the test. If the test time or the data size exceeds these bounds, the LLM will be prompted to adjust the input size accordingly.

```
size = 10000000

a = np.random.randn(size).astype(np.float32)
b = np.random.randn(size).astype(np.float32)
c = np.zeros_like(a)
n = np.int32(size)

input_data = [c, a, b, n]
```

Figure 10: Example code that is expected to be generated by test generation strategy

By default, the current strategy generates 3 such tests by executing the LLM-generated Python code 3 times without any sophisticated techniques. This number was chosen as a trade-off: it ensures that multiple test cases are available to reduce the risk of coincidental correctness, while keeping execution time manageable. However, the framework does allow the user to change the number of test generated or to create a custom testing strategy implementation, and provides functionality to aid with test generation.

4.3 Chosen tuning strategies

The built framework provides three strategies for optimizing the kernel: a one-prompt strategy, the autonomous tuning strategy, and an explicit tuning strategy. These will also be used to test which of the strategies performs better.

4.3.1 One-Prompt strategy

The one-prompt strategy will use only a single LLM call to generate code, after which the generated code will be immediately tested and tuned. The prompts for this strategy are either zero-shot or have a few examples, making it few-shot prompts. This strategy can be seen as a baseline against the autonomous strategy and the explicit strategy.

4.3.2 Autonomous tuning strategy

As the name suggests, the autonomous tuning strategy is autonomous in the sense that it creates its own planning. It is inspired by Plan-and-solve idea [WXL$^+$23] where an LLM first generates a plan and then executes the generated plan. This forces LLM to generate a CoT for each task. Unlike the original Plan-and-Solve, which executes the whole plan in one LLM call, our method executes each step in a different LLM call. This is done because many steps generated by LLMs have errors or cannot be applied for tuning.
Figure 11 illustrates the decision tree of the autonomous strategy. Below is a more detailed explanation of the key components (numbers correspond to the nodes labeled in Figure 11):

1. The first step in the strategy is called "planning" as it plans out the steps that will be taken to optimize the kernel. This step is expected to generate a list of optimization steps that will improve the performance of the kernel.

2. The second step in the strategy looks at the remaining optimization steps and chooses the first optimization step in the array as the current optimization step to execute. If there are no optimization steps remaining, it goes to the re-planning step **(6)**.

3. Often the LLMs return a non-valid optimization step. For example, one of the optimization steps might include profiling or changing the host code. These optimization steps will be skipped as they are considered out of the scope of the framework.
   So this step effectively double checks the validity of the optimization step and filters out the list of planned optimizations. This strategy step is separated from the initial planning step to increase correctness and accuracy as described in Section 2.4.1.

4. This strategy step will ask LLM whether the current optimization step needs to be broken down into smaller optimization steps.
   If the optimization step is broken down then all new optimization steps are added to the

beginning of the optimization steps list. The first step of the optimization steps list is then chosen as the new current optimization step and we go back to **(3)** the verification step in the strategy.
If the current optimization step does not need to be broken down workflow proceeds to the next step **(5)**.

5. This strategy step generates a new kernel and potentially new tuning parameters by applying the current optimization step to the kernel. The output of the newly generated kernel is compared with the output of the previous kernel to ensure the correctness of the newly generated kernel.

    5.1. LLMs can have difficulty adhering to use tuning parameters or often introduce parameters with similar names (e.g. block_size and BLOCK_SIZE_X). Therefore, there is a step that "fixes" the tuning parameters to always have the tuning parameter in code and not introduce parameters with similar names.

6. This strategy step decides whether to attempt new optimization steps or not by analyzing the kernel that was generated and optimization steps that have been taken thus-far.
If new optimization steps are generated, workflow goes back to step **(2)**.
If no new optimization steps have been generated, the workflow is done.

Note: Breakdown **(4)** and replan **(6)** steps are optional and can be disabled before the execution of the strategy.
When the breakdown step **(4)** is disabled, the validation step **(3)** proceeds directly to the execution step **(5)** if the step is valid.
Likewise, when the replan step is disabled, "next step" **(2)** transitions to the end instead of the replan step if there are no more steps to execute.

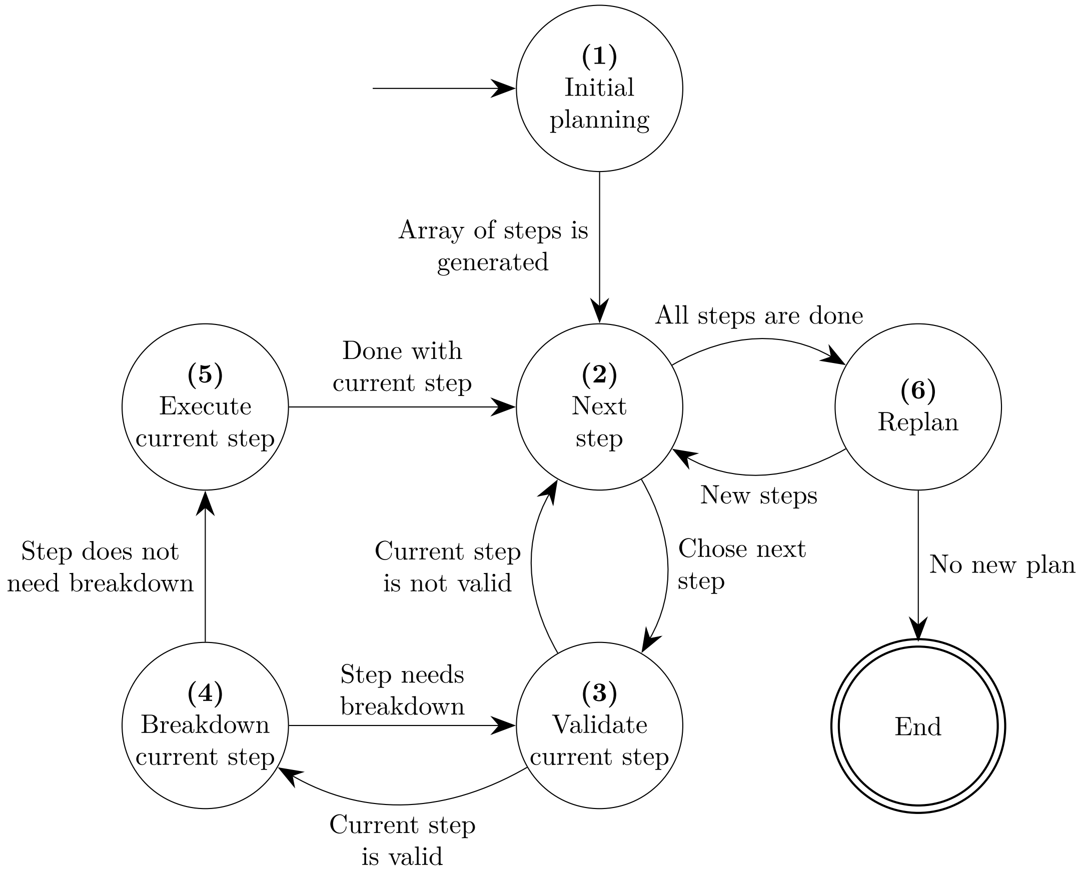


Figure 11: Overview of autonomous tuning strategy. Numbered nodes correspond to the detailed strategy steps described in the text above.

4.3.3 Explicit tuning strategy

Explicit tuning strategy has all the optimization steps predefined. An example of a step could be implementing loop blocking or implementing prefetching. The idea of the explicit tuning strategy is that it tries as many optimizations as possible and sees which of them improve the performance of the kernel. Each optimization can be seen as a step in a long sequence of CoT. The explicit tuning strategy keeps a list of optimization steps that need to be executed and executes them one after another while keeping track of the best-performing kernel thus far.
Some of the optimization steps can have dependencies between them; this allows for more granularity and a more navigable CoT (see Section 2.4.1).
For example, instead of asking an LLM to use n elements per thread immediately, it can be asked to first process 2 elements per thread, and after that it can be asked to process n elements per thread with a tunable parameter.
Before each step is executed, the necessity of the step is evaluated by asking LLM whether or not it is a good idea to implement the step. This can be disabled per step if it is deemed that a step is always necessary.

5 Evaluation

The main goal of the experiments is to find the strategy that produces the most efficient kernel functions. For this purpose, multiple strategies and multiple settings for these strategies have been tested.

5.1 Experimental setup

All experiments have been conducted on two clusters provided by LIACS called: "saronite" and "ceratanium". Both clusers are identical and have the following specifications: 2 x 32 Intel Xeon Gold 6438M cores @ 2.20GHz, 512GB of main memory and 4 x NVIDIA L40S GPU (48 GB memory each).
One of the main limitations of this thesis is the cost of running the models. Therefore, a small selection of handpicked proprietary and open-source models has been chosen from WebDev Arena created by LMArena [CZS+24]. The following models have been chosen:

- OpenAI – gpt-4.1-2025-04-14
- Google – gemini-2.5-pro [Goo25]
- DeepSeek – DeepSeek-R1-0528-UD-TQ1_0 [DA25] by unsloth
- Alibaba Cloud – Qwen3-235B-A22B-fp8-tput [YLY+25] (through Together.ai)

All of these models were released in mid-2025 and represent recent state-of-the-art LLMs. They differ in architecture, size, and licenses. Nonetheless, they provide a diverse set of models for evaluating LLM-assisted kernel tuning.
DeepSeek R1 was deployed directly onto the cluster, and all other models were used through an API. Initially, Qwen3-235B-A22B_UD-Q5_K_XL was deployed directly on the cluster and not through an API. However, this resulted in poor performance of the model. It would often result in syntax errors or would end up in degenerative text loop until the context window was full. This is due to the quantization of the model; therefore, the decision was made to run the model through the API as the 8fp or 16fp models do not fit into the cluster's VRAM.
The inference for DeepSeek R1 was done on each cluster separately through Llama.cpp as this is the best way to run GGUF at the moment.
All models were run with a maximum context length of 10000 and their default recommended inference settings such as temperature, top-p, and top-k. Although leaving these parameters at default introduces some stochasticity between runs, these configurations represent the vendor-recommended usage of the models and thus reflect their expected performance in realistic scenarios. To mitigate stochasticity, each experiment was repeated 3 times, and averaged results are reported. The models were run with the following structured output types (see 4.1.4):

1. GPT-4.1 – json_schema
2. Gemini-2.5-Pro – JSON output
3. DeepSeek-R1 – Separate request
4. Qwen3-235B-A22B – JSON schema

All experiments were run through SLURM with each job having a maximum of 260 minutes to run; if the execution was longer than 260 minutes, the job was terminated. API timeout was set to 15 minutes for each request; such a high timeout was set because some models take a long time to generate thinking output, with thinking time exceeding 10 minutes on some occasions. Initially, Qwen3-235B-A22B was run with hybrid mode structured output (see Section 4.1.4) but most experiments hit the SLURM time wall, therefore, it was decided to run Qwen3-235B-A22B with JSON schema mode instead. This significantly reduced the execution time of Qwen3-235B-A22B.

5.1.1 Evaluated kernels

Experiments were conducted on two sets of kernels, which were deliberately left unoptimized to observe how the LLMs would perform the optimization. The first list contains four "simple" kernels that are small general-purpose kernels that are commonly used, the selected kernels are `matrixAdd`, `matrixMultiply`, `matrixTranspose` and `sigmoidActivation`
The second list contains four "longer" kernels that are a bit more nuanced and are more specialized compared to "simple" kernels. The list consists of the following kernels: `assign_clusters`, `mandelbrot_kernel`, `game_of_life` and `verlet_integration`.
The idea is that LLMs have seen the "simple" kernels more often than the "longer" kernels, and thus might have better performance for the "simple" kernels. Moreover, the "simple" kernels are much simpler and should be easier to tune. Both lists were generated synthetically by LLMs and checked manually. Both "simple" and "longer" kernels can be found in the Appendixes A.1 and A.2, respectively.

5.1.2 Kernel performance measurement

To evaluate the performance of each generated kernel, the mean execution time has been chosen for the evaluation of the generated kernels. This metric provides a measurement to compare the kernels with each other.

To better understand the performance characteristics of the naive kernels and establish a baseline for potential improvement, the memory bandwidth and arithmetic intensity were measured on the naive version of the kernels.
It was not possible to measure these metrics through NVIDIA's Nsight Compute CLI as the security settings of the cluster do not allow for root access that is needed to profile CUDA binaries. Therefore, these metrics have been manually derived by running the kernels directly on the cluster and measuring their execution times using the following formulas: $\mathrm{GB/s} = \frac{\text{bytes}}{10^9\,\text{s}}$ and $\mathrm{TFLOP/s} = \frac{\text{FLOPs}}{10^{12}\,\text{s}}$ where s is the measured execution time in seconds. The maximum possible memory bandwidth claimed for Nvidia L40S is 864 GB/s and the claimed peak FP32 throughput is 91.6 TFLOP/s.
Table 1 shows the measured memory bandwidth in GB/s and the arithmetic intensity in TFLOP/s. Four of the tested kernels were near theoretical maximum memory bandwidth namely; `matrixAdd`, `sigmoidActivation`, `verlet_integration` and `game_of_life`. At the same time, none of the kernels were arithmetically bound. This indicates that the kernels that are near memory bandwidth bound will not get as much performance gain as the ones that are not bound. However, some slight performance gains should theoretically be possible.

5.1.3 Evaluating the tuning strategies

Before executing the main experiment, it was helpful to assess the importance of the optional parameters of the tuning strategies. Namely, the breaking down of a step and the replanning

Kernel	**Problem size**	**Achieved BW** (GB/s, % of peak)	**Achieved FP32** (TFLOP/s, % of peak)
Matrix Addition	16384×16384	665.92 (77.1%)	0.06 (0.1%)
Matrix Multiply	M=16384, K=4096, N=8192	[4.12, 19278.82]*	4.82 (5.3%)
Matrix Transpose	32768×16384	352.41 (40.8%)	Not meaningful (no FLOPs)
Sigmoid Activation	$n = 1.5 \times 10^9$	666.32 (77.1%)	Not meaningful (special ops)
K-means Assignment	32M pts, 128 clust., 32 dims	[10.84, 2691.93]*	1.01 (1.1%)
Mandelbrot	32768×16384, 2000 iters	42.10 (4.9%)	Not meaningful (iter-dependent)
Game of Life	32768×16384	[645.62, 3228.08]*	Not meaningful (logic ops)
Verlet Integration	100M particles	641.44 (74.2%)	0.26 (0.3%)

Table 1: Memory bandwidth and arithmetic intensity of the naive CUDA kernels on an NVIDIA L40S.

*For some kernels, bandwidth is shown as a range '[optimistic, pessimistic]'. The optimistic value assumes perfect data reuse in cache (reading data only once), while the pessimistic value assumes no cache reuse. The true performance lies between these bounds. Percentages over 100% reflect this pessimistic byte count.

step for the autonomous tuning strategy. And the step evaluation of explicit tuning strategy. It might turn out that they are redundant and do not improve or even hurt the performance. For this purpose a smaller scale experiment has been conducted on four hand-selected kernels: `matrixAdd` and `matrixMultiply` (see Section A.1), as well as `mandelbrot` and `verlet_integration` (see Section A.2). These experiments were carried out using ChatGPT 4.1, DeepSeek-R1-0528, and Qwen3-235B-A22B. Each configuration has been repeated three times to try to mitigate stochasticity. The number of repetitions was limited to three due to the high cost of running the models. For the same reason, Gemini-2.5-Pro was excluded from this experiment as it is significantly more expensive than other used models.
For the autonomous tuning strategy, the following configurations of the autonomous tuning strategy have been used:

1. Step breakdown on, replanning on
2. Step breakdown off, replanning on
3. Step breakdown on, replanning off
4. Step breakdown off, replanning off

Replanning was set to be a maximum of 3 replans, and step breakdown was set to be a maximum of 1 breakdown per step.
For the explicit tuning strategy, an experiment was conducted that would look at how often LLMs would choose to skip or include steps. This experiment was conducted because of an observation that was made during testing the framework to which certain LLMs would consistently answer only true or only false depending on the strictness of the prompt.

5.1.4 Main experiment

There are three different strategies that have been tested; the one-prompt strategy, the autonomous strategy, and the explicit strategy. These three strategies were compared against baseline, where baseline is the original CUDA kernel without any optimizations. The performance of each kernel was measured by taking the average execution time of a kernel through 7 iterations. Each strategy has been evaluated 3 times with a different LLM for each kernel.

6 Results

This section will provide a summary of all the experiments conducted. A more detailed overview of the results can be found in Appendix B.

It should be noted that for all experiments conducted in all kernels, a speedup of 0.5% was measured even when there was no change to the code or tuning parameters. This is most likely due to the reuse of the L2 cache at the GPU level, although operating system-level caching or other forms of caching may also contribute. This is a problem that cannot easily be solved without adding a significant amount of overhead to the transformation time.

6.1 Tuning strategies settings evaluation

Before conducting the main experiments, it was necessary to determine the optimal configuration for each tuning strategy. This subsection details the preliminary evaluation performed to identify the settings that offer the best trade-off between performance gain and transformation time. The analysis begins with exploration of various configurations of the autonomous tuning strategy, followed by an examination of LLM's ability to determine whether a tuning step is necessary that will be used in the explicit tuning strategy.

6.1.1 Autonomous tuning strategy

The results of different configurations of the autonomous tuning strategy are detailed in Table 2, Table 3, and Table 4 for GPT-4.1, DeepSeek-R1-0528, and Qwen3-235B-A22B, respectively. The tables show the average performance gain and the average transformation time for each combination of step breakdown and replanning settings.

For GPT 4.1, the base settings without breakdown and without replanning resulted in an average of 4.22 ± 4.22 performance gain that took 252 ± 38 seconds on average to transform the kernel. Replanning on average increased the performance gain with a minimal increase in transformation time. At the same time, breaking down the steps significantly increased the average transformation time while not providing much if any performance gain.

Settings (T)rue (F)alse	**Avg Performance Gain (percent)**	**Avg Transformation Time (seconds)**
Breakdown: F, Replan: F	4.22 ± 4.22	252 ± 38
Breakdown: F, Replan: T	6.58 ± 7.73	266 ± 90
Breakdown: T, Replan: F	1.96 ± 1.24	658 ± 343
Breakdown: T, Replan: T	7.23 ± 6.06	721 ± 321

Table 2: Performance and transformation time of Autonomous Tuning Strategy settings for GPT-4.1 averaged across all test kernels tested.

For DeepSeek-R1-0528 the breakdown of the steps and the replanning of the steps had a negative impact on the performance of the kernels and an increase in transformation time. This is most likely due to the fact that DeepSeek-R1-0528 is a thinking model. Moreover, with both breakdown and replanning enabled, DeepSeek-R1-0528 would sometimes hit the SLURM time wall that was set to 260 minutes.

Settings (T)rue (F)alse	**Avg Performance Gain (percent)**	**Avg Transformation Time (seconds)**
Breakdown: F, Replan: F	6.72 ± 11.37	5942 ± 1307
Breakdown: F, Replan: T	4.61 ± 6.54	6964 ± 1800
Breakdown: T, Replan: F	5.65 ± 5.55	9467 ± 3940
Breakdown: T, Replan: T	0.91 ± 1.02	9853 ± 4887

Table 3: Performance and transformation time of Autonomous Tuning Strategy settings for DeepSeek-R1-0528 averaged across all test kernels tested.

For Qwen3-235B-A22B both breakdown and replanning had a positive impact on the performance of the kernel. In particular, the replanning step had the most positive impact on performance. The combination of step breakdown and replanning both set to true had the highest performance gain. At the same time, this configuration increased the average transformation time more than six fold compared to the next-best configuration and led to a lot of SLURM timeouts.

Settings (T)rue (F)alse	**Avg Performance Gain (percent)**	**Avg Transformation Time (seconds)**
Breakdown: F, Replan: F	3.75 ± 3.99	567 ± 262
Breakdown: F, Replan: T	12.51 ± 18.70	1919 ± 1012
Breakdown: T, Replan: F	7.65 ± 8.24	2486 ± 1855
Breakdown: T, Replan: T	19.50 ± 17.79	11636 ± 3797

Table 4: Performance and transformation time of Autonomous Tuning Strategy settings for Qwen3-235B-A22B averaged across all test kernels tested.

It is important to note that the standard deviation was high, indicating a high variety in performance gain; this will be discussed later in Section 6.2.
In summary, replanning slightly increases the transformation time for GPT-4.1 and DeepSeek-R1 and by a factor of 3.4x to 4.7x for Qwen3-235B-A22B depending on whether step breakdown was enabled. Nonetheless, replanning was highly effective for GPT-4.1 and Qwen3-235B-A22B in terms of performance gain. In contrast, step breakdown sometimes increased the performance but sometimes made it worse while increasing transformation time multiple fold.

6.1.2 Explicit tuning strategy

Figure 12 shows how LLMs answered to the question "Is tuning step necessary?" for the explicit tuning strategy (see Section 4.3.3). It might seem like the answers from DeepSeek-R1 and GPT-4.1 are random, as the "yes" / "no" ratio is around 1.0. But on a closer look the consistency of the answers for the same question is $> 70\%$, which shows that the answers were not random.

Qwen3-235B-A22B, on the other hand, shows a higher bias towards answering "yes" compared to DeepSeek-R1 and GPT-4.1. This confirms that some models do indeed have a higher bias towards the same answer for slightly different questions. However, it is not 100% "yes" or 100% "no". This validation step is therefore a meaningful guardrail, preventing the framework from blindly applying every possible optimization or never applying any optimization at all. This makes the explicit strategy more intelligent than a simple hard-coded sequence of transformations.

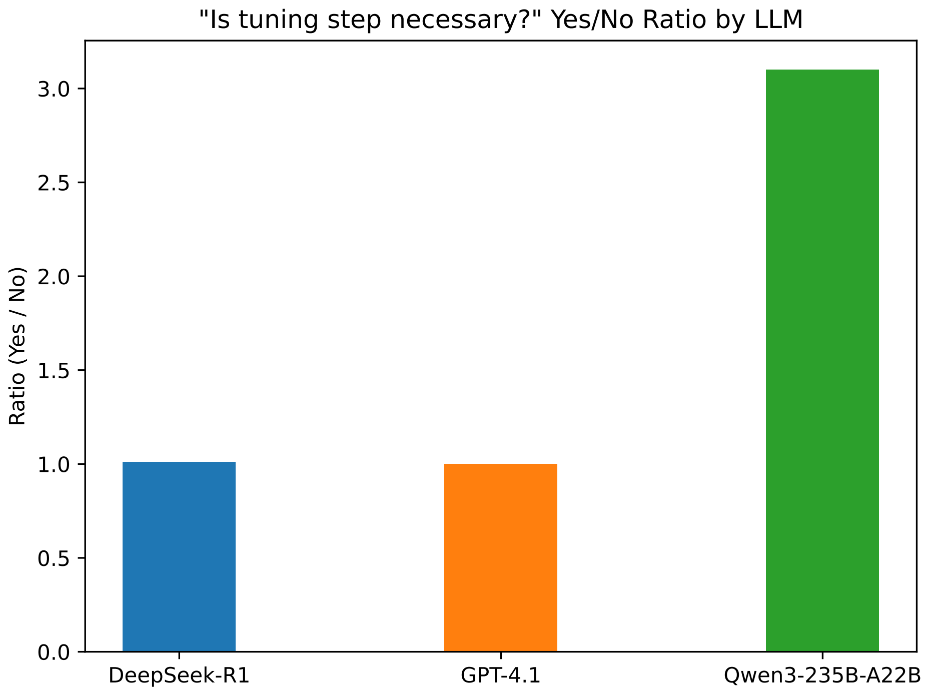


Figure 12: How often LLMs answer to question "Is tuning step necessary" with yes and with no.

6.2 Main experiment

From the previous experiment, the following settings have been chosen for autonomous tuning strategy: step breakdown set to false and replanning set to true for GPT-4.1, Qwen3-235B-A22B and Gemini-2.5-Pro. For DeepSeek-R1 both the breakdown of steps and the replanning were set to false. The choice to disable step breakdown for Gemini-2.5-Pro was made due to the high cost of enabling the breakdown. At the same time, replanning showed high effectiveness with little overhead for two out of three models tested; therefore, it was kept enabled.
Figure 13 shows a detailed overview of the performance gain per kernel for each strategy and LLM. The exact numbers and transformation times can be found in Appendix B.2
On average, the autonomous tuning strategy had the highest performance gain of $9.82 \pm 21.10\%$ followed by the explicit tuning strategy with $5.67 \pm 12.50\%$ followed by the one-prompt strategy with $1.39 \pm 7.78\%$.
Some kernels had a high performance gain, while others had near zero performance gain.
All LLMs could optimize the matrixMultiply kernel to get better performance with at least one of the strategies. This has two reasons; first, matrix multiplication is one of the most used

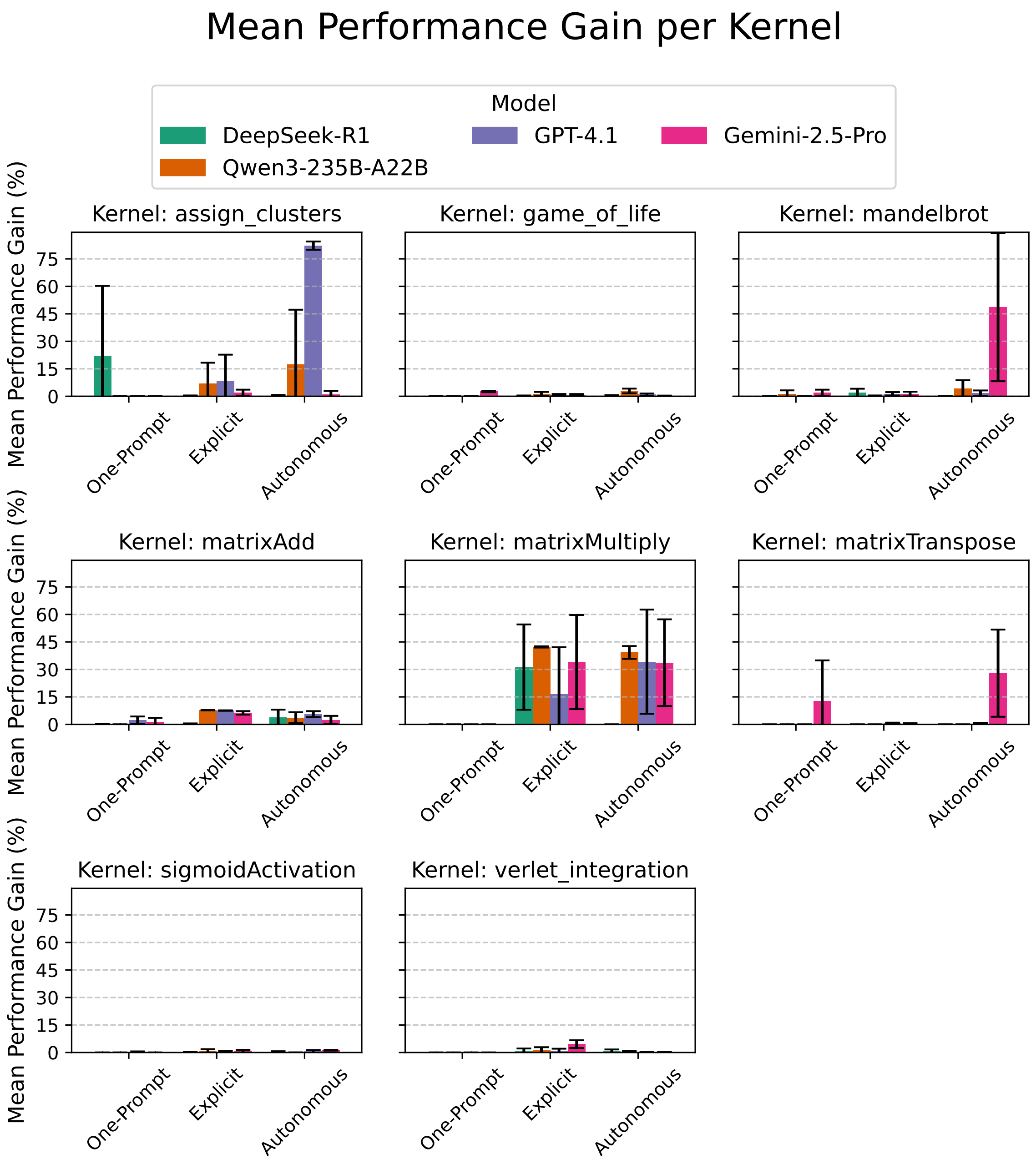


Figure 13: Mean Performance Gain per Kernel (averaged across three runs)

kernels to teach CUDA optimizations and therefore, is probably often included in the training set of LLMs. The second reason is that matrix multiplication benefits the most from block size tuning. In fact, all models successfully tuned the block size. More advanced optimizations were also observed; for instance, Qwen3-235B-A22B implemented prefetching in one of the runs while Gemini-2.5-Pro achieved a speedup through tiling in one run and vectorized memory operations by casting the global memory pointers to float4 in an other run.
The second best kernel that could be optimized by most LLMs was assign_clusters. All successful optimizations for assign_clusters included some form of prefetching or coalescing global memory access. What is noteworthy is that GPT-4.1 did this optimization consistently across

all runs for assign_clusters kernel.
Beyond these more common successes, Gemini-2.5-Pro was the only model capable of optimizing the mandelbrot and matrixTranspose kernels. Mandelbrot kernel was speeded up by Cardioid and period-2 Bulb checking optimizations in two of the three autonomous tuning strategy runs. And matrixTranspose was speedup using tiling for both one-prompt and autonomous strategies with nearly identical code.
The near-zero performance gain for `matrixAdd`, `sigmoidActivation`, `verlet_integration` and `game_of_life` kernels could be as a result of these kernels being close to the maximum memory bandwidth.
A recurring and critical observation across all experiments was that the standard deviation was high across all experiments, often in the same magnitude as the mean. It implies that a user cannot expect a successful optimization on every attempt; a single run might yield no improvement, while another might produce a significant speedup. This variance could partially be eliminated by changing LLM's parameters, such as temperature and top-k. This would make LLMs less stochastic and more deterministic; however, this is not necessarily positive for LLM's ability to find unique solutions, as it might cause the model to repeatedly generate a common but suboptimal solution, missing opportunities for more creative and impactful optimizations.

6.3 Discussion

Restrictions are an essential part of kernel tuning. Even though the method for choosing restrictions was very simple, it was highly effective for the kernels that needed it. However, it does have some small issues.
The first issue is that LLMs would try to use `sizeof(float)` function even though the prompt said not to use it. However, this would often be resolved in the retry after LLM received the error message.
The second is that most of the restrictions were useless, for example, it would often generate restriction $block\_size\_x <= 1024$ while $block\_size\_x = \{32, 64, 128, 256, 512\}$. While this does not have an impact on the execution time of the kernel, it could have an impact on the transformation time if there is a significant amount of constraints that need to be resolved.

Another essential part of kernel tuning is test generation. Like discussed in previous sections, no sophisticated test generation strategy has been created for the framework. It was uncertain whether this would be sufficient enough to generate comprehensive tests. However, for the tested kernels, the chosen test generation strategy was sophisticated enough to generate tests with high accuracy while not letting through any obvious bugs in the newly generated code. This is explained by the fact that it is sufficient to check the output of the generated kernel against the newly generated kernel for the same input.

7 Conclusion

This thesis presents a robust LLM-driven framework designed to automate the tuning and optimization of naive CUDA kernels into auto-tunable optimized CUDA code.
Three tuning strategies were created and tested against each other: autonomous tuning strategy, explicit tuning strategy, and one-prompt tuning strategy.
The autonomous tuning strategy, which first creates a plan and executes it, showed the highest ability to increase kernel performance. For the autonomous tuning strategy, replanning can be

costly in terms of time and compute, but crucial to increase performance. Breaking down of a step into even smaller steps does not always result in better performance but increases the cost several-fold.
Some LLMs do have a slight bias towards "yes" or "no" when asked whether they find a tuning step necessary; this however does not seem to be an issue when transforming the kernel.

No single LLM is universally superior; each LLM's performance is kernel-dependent, for example, GPT 4.1 could consistently increase the performance of assign_clusters kernel while Gemini-2.5-Pro was the only model that succeeded optimizing mandelbrot and matrixTranspose kernels.
However, LLMs are inherently nondeterministic, even kernels that had a speedup could not get the speedup consistently.

At the same time, tasks such as test creation, finding problem size, finding output variables, and creating restrictions were highly reliable and had a near 100% success rate with retries enabled for the tested kernels.

To conclude, LLMs are not yet autonomous HPC experts and lack consistency for CUDA code generation, but they can serve as an initial step in automating common, straightforward tasks such as test input and output generation or standard optimizations, including tuning block size, implementing loop unrolling, implementing tiling.

8 Further research and development

While this work successfully demonstrates the feasibility of the approach, several avenues for future work could enhance its capabilities. This section will provide some information about some ideas that were out of scope for this thesis and some other ideas that can be improved upon but have not been implemented for this thesis for various reasons.

8.1 Grid divisor

Grid divisor is a tuning parameter used to dynamically determine the size of the grid (i.e., the number of blocks) based on the problem size. Grid divisor has not been implemented into the framework. The main reason being the complexity it adds to the prompt or the workflow. Additionally, from a small test run, the LLMs sometimes had trouble understanding what a grid divisor is in terms of auto-tuning.

8.2 Kernel recompilation

With the current implementation, when the kernel is being tested, it is recompiled each time before the test. This is not needed, as the kernel stays the same and only the input of the kernel changes.
Depending on the number of tests and the complexity of the kernel, this compilation time can add up. What could be done instead is to compile the kernel once and run all tests for that kernel. To implement this, it would mean either changing the internals of Kernel Tuner or compiling and tuning the kernel manually without a framework. Therefore, this was not implemented as it was not a bottleneck.

8.3 Fully autonomous tuning strategy

The current autonomous strategy does not choose when to tune and test the kernel, it is done after each successful code transformation. The main reason for this architecture is that, at the start of the writing of the thesis, LLMs were not natively trained with tool usage. This meant that most LLMs hesitated to use the tools or did not use them properly. The best workaround at the time was to use Plan-and-solve [WXL+23] or ReAct [VBK24]. But the field of LLMs has improved a lot in a short amount of time to the point where LLMs can intelligently choose when and how to utilize the tools at hand. So, naturally, one of the ideas that could further be improved upon is the creation of a fully autonomous tuning strategy that would choose when to tune and test the kernel on its own and maybe even profile the kernel to determine what optimizations to focus on. The main advantage would be that it could be made aware of the environment in which it was running and given the full signature of the Kernel Tuner functions. Subsequently, some overhead might be removed as the LLM could choose when to tune on its own and not tune after every step.

8.4 More context aware strategy

The current implementation uses a new context window for each new request. For example, the new kernel code is generated by an LLM in the first context window, tunable parameters are generated with a second context window, and restrictions are generated with a third context window. This could be done within a single context window. For example, kernel code is generated in the first prompt of the context window, and subsequently after the kernel code generation, tunable parameters could be generated in the second prompt of the same context window, ect. This approach has its theoretical benefits and drawbacks; the benefit is that the LLM would be more aware of the previous interactions and the final goal. The drawbacks are the costs and a bigger context window, possibly making the LLM attention and thus the accuracy worse. It should therefore not be seen as an inherent improvement but as an idea for further research.

8.5 Multiple outputs per prompt

In almost all experiments conducted, the standard deviation was high, often higher than the mean. This shows that there is a high variance in performance and accuracy in LLMs. This comes from the fact that LLMs inherently are non-deterministic. This, however, is not a bad thing as it could be exploited to generate multiple outputs, testing all the outputs, and choosing the best performing one for the next step. This could even be done in parallel, eliminating overhead.
However, this approach would increase the cost multiple fold depending on the number of outputs generated.

References

[CCN24] KuanChao Chu, Yi-Pei Chen, and Hideki Nakayama. A better llm evaluator for text generation: The impact of prompt output sequencing and optimization. *Findings of the Association for Computational Linguistics: ACL 2024*, 2024. arXiv:2406.09972v1; accepted to ACL Findings 2024, updated metadata as of May 25, 2025.

[CRHZ25] Bowen Cui, Tejas Ramesh, Oscar Hernandez, and Keren Zhou. Do large language models understand performance optimization?, 2025.

[CZN^{+}22] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022.

[CZS^{+}24] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024.

[DA25] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[DAKC23] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. Piloting copilot and codex: Hot temperature, cold prompts, or black magic?, 2023.

[DCE^{+}23] Xianzhong Ding, Le Chen, Murali Emani, Chunhua Liao, Pei-Hung Lin, Tristan Vanderbruggen, Zhen Xie, Alberto Cerpa, and Wan Du. Hpc-gpt: Integrating large language model for high-performance computing. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W 2023, page 951–960. ACM, November 2023.

[Goo25] Google. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[GVLT^{+}24] William F. Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience*, 36(26):e8269, 2024.

[HV23] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing, 2023.

[Kum24] Harshit Kumar. Matrix multiplication in cuda. `https://kharshit.github.io/blog/2024/06/07/matrix-multiplication-cuda`, June 2024. Adapted under CC BY-NC 4.0.

[Lan24] LangChain. Multi needle in a haystack, 2024.

[LLL+25] Zike Li, Mingwei Liu, Anji Li, Kaifeng He, Yanlin Wang, Xin Peng, and Zibin Zheng. Enhancing the robustness of llm-generated code: Empirical study and framework, 2025.

[LXWZ23] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc., 2023.

[LZL+23] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. Acecoder: Utilizing existing code to enhance code generation, 2023.

[LZLC24] Mo Li, Songyang Zhang, Yunxin Liu, and Kai Chen. Needlebench: Can llms do retrieval and reasoning in 1 million context window?, 2024.

[MDD+25] Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, and Hinrich Schütze. Nolima: Long-context evaluation beyond literal matching, 2025.

[NDX+24] Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 281–294. ACM, June 2024.

[OGA+25] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.

[RJS+24] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[RLM+24] Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. Batfix: Repairing language model-based transpilation. *ACM Trans. Softw. Eng. Methodol.*, apr 2024. Just Accepted.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 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*, PPoPP '08, page 73–82, New York, NY, USA, 2008. Association for Computing Machinery.

[TWT+24] Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung yi Lee, and Yun-Nung Chen. Let me speak freely? a study on the impact of format restrictions on performance of large language models, 2024.

[VBK24] Mudit Verma, Siddhant Bhambri, and Subbarao Kambhampati. On the brittle foundations of react prompting for agentic large language models, 2024.

[vW19] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

[WHF$^+$23] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, 2023.

[WWS$^+$23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[WXL$^+$23] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models, 2023.

[YCZ$^+$23] Ming Yan, Junjie Chen, Jie M. Zhang, Xuejie Cao, Chen Yang, and Mark Harman. Coco: Testing code generation systems via concretized instructions, 2023.

[YLY$^+$25] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[YTP$^+$24] Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. Vert: Verified equivalent rust transpilation with few-shot learning, 2024.

[YXJ$^+$25] Yijiong Yu, Ma Xiufa, Fang Jianwei, Zhi Xu, Su Guangyao, Wang Jiancheng, Yongfeng Huang, Zhixiao Qi, Wei Wang, Weifeng Liu, Ran Chen, and Ji Pei. Long-context language models are not good at all retrieval tasks without sufficient steps, 2025.

[ZCZ$^+$23] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023.

Appendices

A Tested kernels

A.1 Simple kernels

```
// 1. Matrix Addition
__global__ void matrixAdd(float *A, float *B, float *C, int width, int height) {
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    if (col < width && row < height) {
        int idx = row * width + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

```
// 2. Matrix Multiplication
__global__ void matrixMultiply(float *A, float *B, float *C, int A_width, int A_height, int B_width) {
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    if (col < B_width && row < A_height) {
        float sum = 0;
        for (int k = 0; k < A_width; ++k) {
            sum += A[row * A_width + k] * B[k * B_width + col];
        }
        C[row * B_width + col] = sum;
    }
}
```

```
// 3. Matrix Transpose
__global__ void matrixTranspose(float *in, float *out, int width, int height) {
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    if (col < width && row < height) {
        int idx_in = row * width + col;
        int idx_out = col * height + row;
        out[idx_out] = in[idx_in];
    }
}
```

```
// 4. Element-wise Sigmoid Function
__global__ void sigmoidActivation(float *in, float *out, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        out[idx] = 1.0f / (1.0f + expf(-in[idx]));
    }
}
```

A.2 Longer kernels

```
// 1. K-Means Clustering Assignment Step
__global__ void assign_clusters(
    const float *data_points,  // Data points (num_points x dims)
    const float *centroids,    // Centroids (num_clusters x dims)
    int *labels,               // Output labels (num_points)
    int num_points,
    int num_clusters,
    int dims) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_points) {
        float min_dist = INFINITY;
        int min_idx = -1;
        for (int c = 0; c < num_clusters; ++c) {
            float dist = 0.0f;
            for (int d = 0; d < dims; ++d) {
                float diff = data_points[idx * dims + d] - centroids[c * dims + d];
                dist += diff * diff;
            }
            if (dist < min_dist) {
                min_dist = dist;
                min_idx = c;
            }
        }
        labels[idx] = min_idx;
    }
}
```

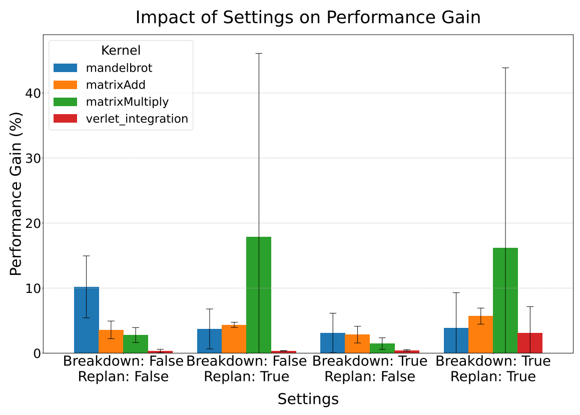
```
// 2. Mandelbrot Set Fractal Generation
__global__ void mandelbrot(
    int *output, int width, int height,
    float x_min, float x_max, float y_min, float y_max, int max_iter) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < width && idy < height) {
        float x0 = x_min + idx * (x_max - x_min) / width;
        float y0 = y_min + idy * (y_max - y_min) / height;
        float x = 0.0f;
        float y = 0.0f;
        int iter = 0;

        while (x * x + y * y <= 4.0f && iter < max_iter) {
            float x_temp = x * x - y * y + x0;
            y = 2.0f * x * y + y0;
            x = x_temp;
            iter++;
        }
        output[idy * width + idx] = iter;
    }
}
```

```
// 3. Cellular Automaton (Game of Life)
__global__ void game_of_life(
    const int *current_grid, int *next_grid,
    int width, int height) {

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        int count = 0;
        for (int dx = -1; dx <=1; ++dx) {
            for (int dy = -1; dy <=1; ++dy) {
                if (dx == 0 && dy == 0) continue;
                int nx = (x + dx + width) % width;
                int ny = (y + dy + height) % height;
                count += current_grid[ny * width + nx];
            }
        }
        int state = current_grid[y * width + x];
        if (state == 1 && (count < 2 || count > 3)) {
            next_grid[y * width + x] = 0;
        } else if (state == 0 && count == 3) {
            next_grid[y * width + x] = 1;
        } else {
            next_grid[y * width + x] = state;
        }
    }
}
```

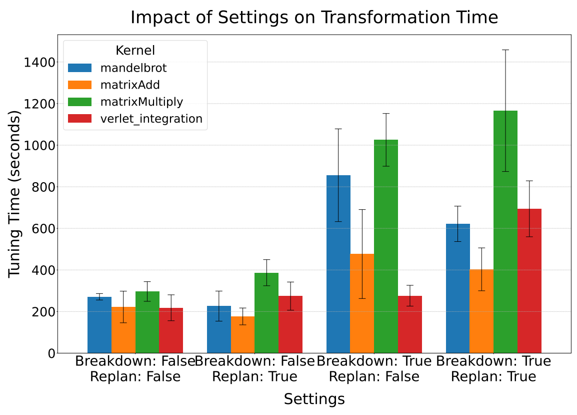
```
// 4. Particle Simulation with Verlet Integration
__global__ void verlet_integration(
    float3 *positions, float3 *velocities, float3 *accelerations,
    float dt, int num_particles) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num_particles) {
        float3 pos = positions[idx];
        float3 vel = velocities[idx];
        float3 acc = accelerations[idx];

        pos.x += vel.x * dt + 0.5f * acc.x * dt * dt;
        pos.y += vel.y * dt + 0.5f * acc.y * dt * dt;
        pos.z += vel.z * dt + 0.5f * acc.z * dt * dt;

        vel.x += acc.x * dt;
        vel.y += acc.y * dt;
        vel.z += acc.z * dt;

        positions[idx] = pos;
        velocities[idx] = vel;
    }
}
```

B Results

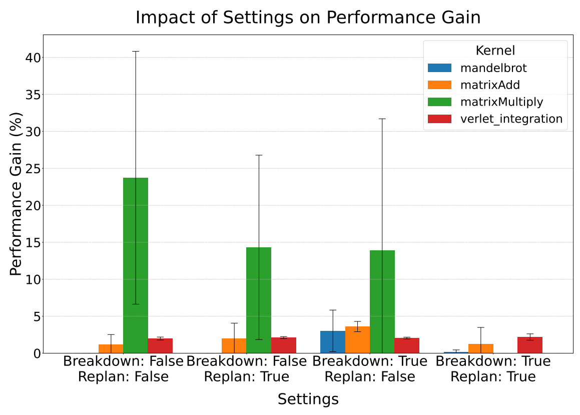
B.1 Tuning the tuning strategies



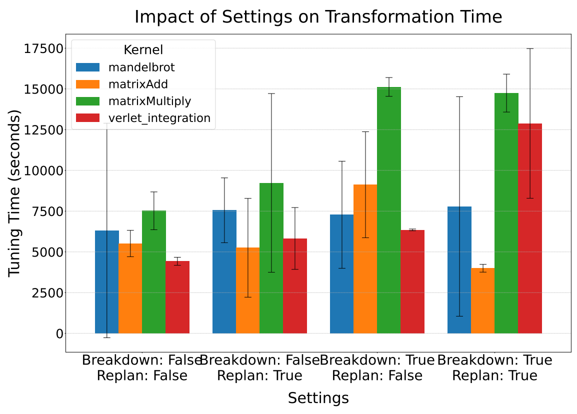
(a) Autonomous tuning strategy performance gain ChatGPT 4.1 with different settings



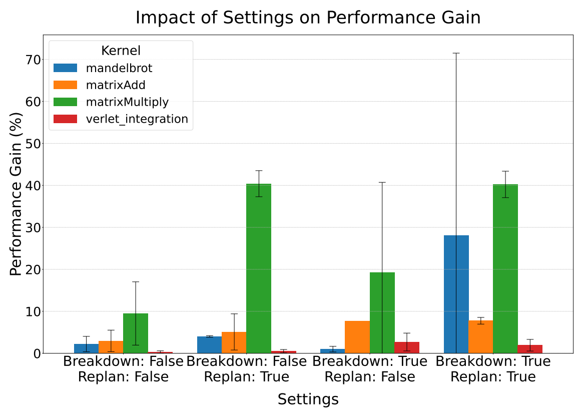
(b) Autonomous tuning strategy transformation time ChatGPT 4.1 with different settings



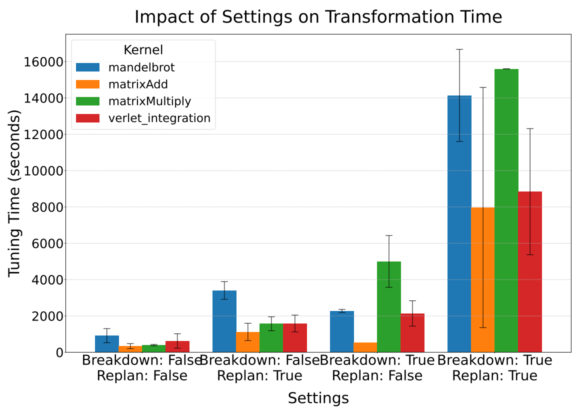
(c) Autonomous tuning strategy performance gain DeepSeek-R1-0528 with different settings



(d) Autonomous tuning strategy transformation time DeepSeek-R1-0528 with different settings



(e) Autonomous tuning strategy performance gain Qwen3-235B-A22B with different settings



(f) Autonomous tuning strategy transformation time Qwen3-235B-A22B with different settings

Figure 14: Autonomous tuning strategy with different settings and different LLM models

B.2 Main experiments

Table 5: Aggregated Results for **DeepSeek-R1**

Kernel	Strategy	Mean Gain (%)	Std Gain (%)	Mean Time (s)	Std Time (s)
assign_clusters	Autonomous	0.279568	0.484225	3556.733471	4419.612258
assign_clusters	Explicit	0.315399	0.146543	6378.461056	1176.486835
assign_clusters	One-Prompt	22.044356	38.181944	1523.187860	744.704308
game_of_life	Autonomous	0.235147	0.407287	7504.285347	1705.648829
game_of_life	Explicit	0.217583	0.228529	6735.184510	4682.661644
game_of_life	One-Prompt	0.000000	0.000000	1124.101291	637.139091
mandelbrot	Autonomous	0.000000	0.000000	1832.275001	120.867794
mandelbrot	Explicit	1.961112	2.174973	4342.997849	2365.577918
mandelbrot	One-Prompt	0.000000	0.000000	2077.394579	237.167338
matrixAdd	Autonomous	3.797245	4.252472	2163.660222	1297.699680
matrixAdd	Explicit	0.192768	0.333884	4663.707581	3626.968040
matrixAdd	One-Prompt	0.115671	0.200349	1715.489979	1016.726230
matrixMultiply	Autonomous	0.000000	0.000000	1433.725701	264.988450
matrixMultiply	Explicit	31.242265	23.283803	11508.723905	3094.056814
matrixMultiply	One-Prompt	0.000000	0.000000	1991.529615	468.057958
matrixTranspose	Autonomous	0.000000	0.000000	258.819719	3.123933
matrixTranspose	Explicit	0.000000	0.000000	258.214715	4.426743
matrixTranspose	One-Prompt	0.000000	0.000000	256.030666	1.375975
sigmoidActivation	Autonomous	0.273059	0.388099	3101.619619	1313.121549
sigmoidActivation	Explicit	0.074354	0.128785	4182.230334	778.414701
sigmoidActivation	One-Prompt	0.000000	0.000000	1396.966711	879.493077
verletIntegration	Autonomous	0.602489	1.043542	4317.669410	1988.222310
verletIntegration	Explicit	0.799958	1.385567	4538.189099	4701.925657
verletIntegration	One-Prompt	0.000000	0.000000	1592.374564	104.872715

Table 6: Aggregated Results for **GPT-4.1**

Kernel	Strategy	Mean Gain (%)	Std Gain (%)	Mean Time (s)	Std Time (s)
assign_clusters	Autonomous	82.214962	2.253149	570.584819	146.354120
assign_clusters	Explicit	8.492476	14.230472	170.388118	49.613140
assign_clusters	One-Prompt	0.000000	0.000000	62.071927	4.309950
game_of_life	Autonomous	0.926509	0.605984	486.273998	224.553352
game_of_life	Explicit	0.674148	0.590703	345.080341	89.400404
game_of_life	One-Prompt	0.000000	0.000000	101.820469	26.524840
mandelbrot	Autonomous	1.614902	1.577308	453.183856	195.858135
mandelbrot	Explicit	1.348476	0.935226	278.267788	249.655749
mandelbrot	One-Prompt	0.000000	0.000000	42.981476	4.198659
matrixAdd	Autonomous	5.625702	1.606441	475.678119	288.649559
matrixAdd	Explicit	7.537059	0.132157	523.676393	485.092864
matrixAdd	One-Prompt	2.295309	2.023325	136.582887	92.427575
matrixMultiply	Autonomous	34.211754	28.417527	251.377483	48.689702
matrixMultiply	Explicit	16.435990	25.644090	327.621312	13.425517
matrixMultiply	One-Prompt	0.000000	0.000000	74.059945	45.583129

(continued)

Kernel	Strategy	Mean Gain (%)	Std Gain (%)	Mean Time (s)	Std Time (s)
matrixTranspose	Autonomous	0.312058	0.540501	538.218976	341.592016
matrixTranspose	Explicit	0.837029	0.103545	765.409431	137.801622
matrixTranspose	One-Prompt	0.000000	0.000000	96.817200	21.674325
sigmoidActivation	Autonomous	0.570279	0.798098	222.814414	97.863986
sigmoidActivation	Explicit	0.277235	0.480185	164.888735	37.836029
sigmoidActivation	One-Prompt	0.209729	0.363261	94.804297	44.241062
verletIntegration	Autonomous	0.148150	0.031920	296.460523	75.789721
verletIntegration	Explicit	0.782440	1.273720	66.491377	32.488909
verletIntegration	One-Prompt	0.000000	0.000000	50.483573	10.461688

Table 7: Aggregated Results for **Gemini-2.5-Pro**

Kernel	Strategy	Mean Gain (%)	Std Gain (%)	Mean Time (s)	Std Time (s)
assign_clusters	Autonomous	1.069626	1.852647	2524.511306	1658.830870
assign_clusters	Explicit	1.980288	1.646518	1604.381334	200.129660
assign_clusters	One-Prompt	0.000000	0.000000	395.907880	205.458764
game_of_life	Autonomous	0.167636	0.224115	1692.468731	407.127153
game_of_life	Explicit	0.653606	0.594297	3412.463488	1834.312674
game_of_life	One-Prompt	2.633667	0.405741	319.546674	91.964855
mandelbrot	Autonomous	48.682679	40.461352	1641.358905	1144.379009
mandelbrot	Explicit	1.227757	1.288697	736.390801	282.091449
mandelbrot	One-Prompt	1.902900	1.721659	332.744497	139.141658
matrixAdd	Autonomous	2.458515	2.184966	1934.051095	1379.009273
matrixAdd	Explicit	6.279494	0.962079	636.142024	252.513581
matrixAdd	One-Prompt	1.317891	2.282655	444.267184	166.441315
matrixMultiply	Autonomous	33.656960	23.656926	3451.310371	1985.490147
matrixMultiply	Explicit	33.989001	25.689162	2142.964458	277.010955
matrixMultiply	One-Prompt	0.000000	0.000000	480.361210	62.322367
matrixTranspose	Autonomous	27.937517	23.777062	2002.734115	1072.755064
matrixTranspose	Explicit	0.243963	0.422556	1678.397802	480.900974
matrixTranspose	One-Prompt	12.780950	22.137255	444.426174	177.627470
sigmoidActivation	Autonomous	1.215179	0.187501	3563.343401	3317.702919
sigmoidActivation	Explicit	0.711584	0.727987	523.246186	69.317180
sigmoidActivation	One-Prompt	0.000000	0.000000	379.185996	95.391158
verletIntegration	Autonomous	0.112893	0.087530	2057.317174	414.322675
verletIntegration	Explicit	4.548874	2.113432	567.882943	73.384343
verletIntegration	One-Prompt	0.000000	0.000000	425.209929	46.457839

Table 8: Aggregated Results for **Qwen3-235B-A22B**

Kernel	Strategy	Mean Gain (%)	Std Gain (%)	Mean Time (s)	Std Time (s)
assign_clusters	Autonomous	17.453584	29.801164	1133.472087	878.288873

(continued)

Kernel	Strategy	Mean Gain (%)	Std Gain (%)	Mean Time (s)	Std Time (s)
assign_clusters	Explicit	6.994705	11.326297	476.818959	421.318361
assign_clusters	One-Prompt	0.000000	0.000000	88.188580	10.170280
game_of_life	Autonomous	2.927852	1.269779	3074.160420	540.222623
game_of_life	Explicit	1.243072	1.150984	716.289460	709.813217
game_of_life	One-Prompt	0.000000	0.000000	111.528347	15.212996
mandelbrot	Autonomous	4.325296	4.446058	1710.819749	880.003685
mandelbrot	Explicit	0.201489	0.261381	787.927830	458.258855
mandelbrot	One-Prompt	1.184812	2.052155	162.907589	147.249406
matrixAdd	Autonomous	3.645648	3.004099	1013.542307	434.535239
matrixAdd	Explicit	7.733767	0.075712	865.591007	305.159081
matrixAdd	One-Prompt	0.000000	0.000000	92.411257	12.192384
matrixMultiply	Autonomous	39.238767	3.485601	1277.991449	597.179345
matrixMultiply	Explicit	42.249553	0.328153	512.439309	84.900355
matrixMultiply	One-Prompt	0.000000	0.000000	78.648626	65.372053
matrixTranspose	Autonomous	0.000000	0.000000	1094.358655	269.410454
matrixTranspose	Explicit	0.000000	0.000000	427.191536	31.819225
matrixTranspose	One-Prompt	0.000000	0.000000	102.138937	5.666874
sigmoidActivation	Autonomous	0.013971	0.024198	665.211632	56.748214
sigmoidActivation	Explicit	0.895762	0.920193	365.223592	262.524793
sigmoidActivation	One-Prompt	0.000000	0.000000	88.317037	8.651471
verletIntegration	Autonomous	0.412190	0.390706	1140.291782	383.758958
verletIntegration	Explicit	1.358510	1.505074	478.593559	155.323905
verletIntegration	One-Prompt	0.000000	0.000000	69.870552	7.808637