Universiteit Leiden

Master Computer Science

Advanced Autotuning for Triton Kernels with Kernel Tuner

Name:	Andrija Kuzmanov
Student ID:	s3766780
Date:	06.09.2024
Specialisation:	Advanced Computing and Systems
1st supervisor:	Ben van Werkhoven

Master's Thesis in Computer Science

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

Contents

Chapter 1

Introduction

The continuous progress of high-performance computing (HPC) and artificial intelligence, especially in the domain of large language models and deep learning applications, has created an increased demand for computing power. Graphics Processing Units (GPUs), originally designed for graphics rendering, which requires high parallel processing power, have become a key component for accelerating these demanding workloads. However, harnessing the full potential of GPUs is a challenging task for software developers. Writing efficient GPU code, also known as kernels, requires a deep understanding of the underlying hardware architecture, especially the memory hierarchy and the management of thousands of parallel threads.

Until recently, NVIDIA's CUDA was the de facto standard for writing highly performant GPU code. While it is a versatile and powerful tool, it requires expertise in a low-level programming language like C or C++ and a good understanding of the hardware intricacies of the GPU [1]. This significantly raises the barrier to entry and increases the development time for projects. For this reason, many vendors have released their own libraries (e.g., cuBLAS, cuDNN, etc.) for utilizing GPUs for common operations such as matrix multiplication, neural network training, and more. These libraries work great for the designed use case, however, they lack the flexibility and customizability that is often required by cutting-edge research or specialized deep learning. Furthermore, they also contribute to vendor lock-in, as the code is specific to the vendor's hardware.

To address these limitations, OpenAI adopted the Triton compiler, developed at Harvard [2], and integrated it with Python. With this, they created a promising open-source programming language for writing high-performance GPU kernels within Python. Because it is written in Python's own syntax, and it is embedded into Python code as a library, it lowers the barrier to entry for developers and researchers who otherwise may not have been comfortable with low-level programming languages like C or C++. Furthermore, it also abstracts away the memory management on the GPU. It enforces operations on tiles (or tensors) of data, which allows the compiler to efficiently store and load data from the appropriate memory space while optimizing for memory coalescing [3]. Triton comes with its own Just-In-Time (JIT) compiler, which translates the code written in Python into efficient GPU instructions via the intermediate representation called Triton IR. Because of this, it can also be extended to work with different GPU vendors and models [2]. Currently it supports NVIDIA and AMD GPUs, which is a great advantage for users who want to take advantage of the latest hardware. Moreover, its close integration with the popular framework PyTorch further increases its appeal.

Despite the many advantages, achieving optimal performance, even for simple kernels in Triton, is not a straightforward task. The performance of a kernel varies greatly based on a multitude of configuration parameters, such as the thread block dimensions, tiling factors, memory access strategies, and more [4]. There is no one size fits all solution, and the optimal parameters vary depending on the specific GPU architecture, input data size, kernel's workload, etc. Manually finding the optimal parameters is usually not possible due to the large search space that needs to be covered. This is where auto-tuning comes into play. It is a process of automatically searching for the best performing configuration parameters for a given kernel and environment.

While Triton does come with its own auto-tuning feature, it is quite limited in terms of the search space exploration strategies and the way that the parameter spaces are defined. Furthermore, while Triton is open source, it is, at the time of writing, lacking in terms of documentation and examples. This would make it difficult for us to improve Triton's auto-tuning feature directly. Moreover, due to the less modular nature of Triton's codebase, we would have to implement many of the features that are already present in existing auto-tuning frameworks.

To overcome these limitations, this project integrates Triton support into Kernel Tuner, a mature and flexible auto-tuning framework. The primary goal is to provide developers with a powerful tool set to easily tune their Triton kernels. By leveraging Kernel Tuner's advanced search strategies and hardware-specific tuning capabilities, this work demonstrates that significant performance improvements can be unlocked for various critical kernels, such as matrix multiplication, attention, and convolutions. The results presented herein showcase not only substantial speed-ups over Triton's built-in auto-tuner but also reveal that tuned Triton code can achieve performance competitive with, and in some cases even surpass, highly optimized vendor libraries like cuBLAS and hipBLAS (which themselves are a collection of tuned kernels), affirming the viability of high-level GPU abstractions combined with sophisticated auto-tuning. The document is organized in the following way. Chapter 2 provides the necessary background information for the reader to understand the context of this work and the motivation for the project. Chapter 3 goes through and analyzes the state of the field of auto-tuning and compares previous works with our project, and explains the differences between our work and the previous works. Chapter 4 presents how we added support for the Triton programming language within Kernel Tuner and the challenges we faced. Chapter 5 evaluates the performance of the tuned kernels and compares them to the performance of the baseline kernels and the vendor libraries. Furthermore, we also analyze the impact of the different possible search strategies and argue for the benefits of using a non-brute force approach. Chapter 6 draws conclusions from the results and discusses the future work.

In our work we make the following contributions:

- We adapt Kernel Tuner to support tuning of Triton kernels.
- Comprehensive performance evaluation comparing Triton kernels tuned using this integrated framework against baseline Triton implementations (using its built-in autotuner) and highly optimized vendor libraries (cuBLAS/hipBLAS) across different hardware architectures.
- An analysis and comparison of different search strategies within Kernel Tuner to understand the trade-offs between tuning time and achieved performance for Triton kernels.

Chapter 2

Background

2.1 Graphics Processing Units

Graphics Processing Units (GPUs) have become a cornerstone of modern computing. Originally developed to accelerate rendering of 3D graphics, GPUs have developed into a powerful co-processor capable of handling highly parallel workloads across a wide range of domains, such as scientific computing, machine learning, and high performance computing.

In recent years, because of the rise in deep learning and the use of large language models, the demand and the performance of the GPUs have increased significantly. Nevertheless, from the perspective of software developers, the core architecture of GPUs has remained stable. They are composed of thousands of small processing cores, which are first grouped into warps that execute the same instruction in lockstep. Threads are then grouped with registers and shared L2 cache into a unit called a Streaming Multiprocessor (SM). GPUs are composed of multiple SMs and a memory hierarchy that includes the registers and the shared memory of the SMs, the L2 cache, the texture memory, the constant memory, and the global memory.

The key to achieving high performance on GPUs is to exploit as much parallelism as possible and to minimize the latency of memory accesses. Given the complex memory hierarchy of GPUs, and the need to manage parallelism, writing efficient GPU code can be challenging. For this reason, platforms like CUDA, HIP, OpenCL, and Triton have been developed to provide a high level interface to the GPU hardware.

2.2 Cuda

CUDA (Compute Unified Device Architecture), developed by NVIDIA, is one of the first parallel computing platforms and APIs to enable general-purpose programming on GPUs. This revolutionized the world of computing by allowing developers to process large blocks of data in parallel, which unlocked the true potential of high-performance computing and deep learning. CUDA provides a high-level API that is used as an extension of C and C++ programming languages, which allows developers to write parallel code for GPUs.

The general flow of a GPU program written in CUDA (and most other GPU programming frameworks) is as follows:

1. Allocate memory on the GPU for input and output data.

2. Copy the input data from the main memory (RAM) to the GPU's global memory.
3. Launch the function that will be executed on the GPU (called a kernel).
4. Copy the output data from the GPU back to the main memory.
5. Free the memory on the GPU.

CUDA provides the necessary tools and libraries to go through this process, and compile the code (kernels) to run on the GPU. Nevertheless, while CUDA offers high performance and control, utilizing it effectively requires, as mentioned in Section 2.1, a good understanding of the GPU architecture.

One of the critical challenges in CUDA programming is managing the different types of memory available on a GPU. CUDA exposes multiple memory spaces, each with distinct access speeds, lifetimes, and uses. These memory types include:

1. Global Memory: The largest memory space, but also the slowest. It is the space where the input and output of GPU kernels are stored by the CPU. However, it should be used with care, as it has much higher latency compared to other memory types.
2. Shared Memory: Much faster than global memory, shared memory is accessible to all threads within a thread block, and is the main way of communication between threads in the same thread block. It allows threads to share data without going through slower global memory, but its limited size requires careful management to ensure efficient use.
3. Constant Memory: Optimized for read-only data that is shared across all threads. Accesses to constant memory are cached, making it faster than global memory in some cases, but it is limited in size.

Effectively utilizing different memory types is crucial for achieving maximum performance on the GPU. While seemingly straightforward, managing memory in CUDA can be challenging even for experienced developers. In addition to memory management, CUDA developers are restricted to writing code in a lower level language, such as C or C++. While they offer fine-grained control, they are also more complex and error-prone, especially for newer developers. Furthermore, developing applications with these languages requires more time and effort compared to higher-level languages such as Python.

Moreover, an essential concept in CUDA programming is how the work is divided among the threads. CUDA kernels are executed on threads, which are grouped into blocks, which are grouped into grids. The number of threads per block and the number of blocks per grid can influence the performance of a kernel. Each block of threads is executed on a Streaming Multiprocessor (SM) of the GPU (Section 2.1). Furthermore, threads within a block can communicate with each other through shared memory. Choosing the right number of threads per block is important in order to fully utilize the resources of the GPU. The developers must find a balance between thread-level parallelism and resource usage.

To illustrate some of the previously mentioned concepts, consider the following example of a simple vector addition kernel in CUDA:

In the example Listing 2.1, the kernel `vector_addition` takes three arrays `a`, `b`, and `c`, and the size of the arrays `n` as input arguments. The goal of the kernel is to add the elements of arrays `a` and `b` and store the result in array `c`. The `__global__` keyword indicates that the

```
__global__ void vector_addition(float *a, float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

Listing 2.1: Vector addition kernel in CUDA

function is a kernel and will be executed on the GPU. The `blockIdx.x` and `threadIdx.x` variables are built-in variables that provide the index of the block and thread within the block, respectively. The `blockDim.x` variable provides the number of threads in a block. In case we have more threads than elements in the arrays, we need to check if the current thread is within the bounds of the arrays. Finally, using the index `i`, the kernel adds the elements of arrays `a` and `b` and stores the result in array `c`.

This example only showcases a simple kernel with a 1-dimensional grid of blocks and threads. However, we also need to write the code that will allocate the space and transfer the data to the GPU, define how many blocks and threads we will use, and finally launch the kernel. This adds to the complexity and also the amount of code that needs to be written in order to run a simple kernel on the GPU.

This fine-grained control, can be both a blessing and a curse. On one hand, it allows developers to squeeze the maximum amount of performance from the GPU; on the other hand, it significantly raises the barrier to entry, is more time-consuming, and requires a good understanding of the underlying hardware to be effective.

2.2.1 Vendor libraries

To help with writing efficient GPU code, NVIDIA provides libraries optimized for certain GPU workloads. These make it easier for developers to write code that gets executed on the GPU without having to know how to write CUDA code. Some of the most popular libraries are:

- cuBLAS: GPU-accelerated BLAS library for linear algebra operations [5].
- cuFFT: GPU-accelerated FFT library for signal processing and scientific computing [6].
- cuDNN: NVIDIA's CUDA Deep Neural Network library for optimized deep learning primitives [7].
- cuSPARSE: GPU-accelerated library for sparse matrix operations [8].

These libraries provide high performance implementations of common operations, however, they are often proprietary and can not be easily customized to accommodate the needs of modern deep learning workloads.

2.3 Triton

Triton is an open-source programming language based on Python that aims to enable researchers to write efficient (most of the time on par with CUDA) GPU code without having any CUDA experience.

It achieves this by providing a high-level interface on top of Python that uses the Triton just-in-time compiler to compile kernel code to run on the GPU. The compiler is able to generate efficient GPU code by first converting Triton code into a Triton intermediate representation (IR). This IR is then optimized and transformed into LLVM IR, which is then further optimized by the LLVM compiler. Finally, the LLVM IR is compiled to PTX code, which is the assembly language for NVIDIA GPUs [2].

One of the main advantages of Triton is the abstraction of memory management. From the developer's perspective, there is only one memory space, which is the global memory of the GPU. Triton automatically manages which parts of the data should be stored in which memory type (registers, shared memory, global memory) [2].

It is able to do this because of a different programming model compared to CUDA. In Triton, each thread is usually responsible for multiple elements, which are grouped into a block. This is in contrast to CUDA, where, in most cases, each thread is responsible for a single element. This allows Triton to optimize memory access patterns by trying to fit as much of the block data as possible in the shared memory of the SMs. While this can be achieved in CUDA as well, it requires manual management of the shared memory, which can be error-prone and time-consuming [3].

Furthermore, because Triton functions mostly operate on blocks of data, which they refer to as tiles, and because manual memory management does not exist, it also abstracts away the implementation of reductions by providing methods for common reductions. Moreover, it also provides a `reduce` function that can be used to implement custom reductions by passing a custom (lambda) function.

What is more, it also provides a range of functions for working with tensors. These range from shape manipulation, such as `reshape` and `broadcast`, to mathematical operations such as `dot` and `softmax`. These allow developers who have worked with libraries like PyTorch or TensorFlow to have an easier transition to Triton.

Moreover, Triton is also closely integrated with PyTorch, one of the most popular deep learning frameworks. PyTorch handles the memory allocation and the data transfer between the host and the GPU, while Triton is responsible for the computation on the GPU. This also allows developers to optimize performance-critical parts of their PyTorch code using Triton, while retaining the same code base and minimal changes to the code.

Another notable advantage of Triton is the possibility to include other GPU vendors in the future. Because it converts the high-level interface into different Intermediate Representations (IRs [9]) and because it executes optimizations on the IRs themselves, it is possible and encouraged by the OpenAI team to include support for other GPUs in the future. One notable example of this is the recently added support for AMD GPUs [10]. This is a huge advantage, as it allows for code that is not vendor-locked and can be used on different hardware if the need arises.

Because of this, Triton has quickly gained popularity in deep learning and scientific computing communities. It is used in a wide range of applications, from optimizing LLM inference to accelerating scientific simulations.

2.4 Autotuning

Nowadays, writing efficient data parallel code is a challenging task. For example, while developing CUDA programs, developers must decide on the number of threads per block, the number of blocks per grid, memory access patterns, and many other parameters. However, they might not have the necessary knowledge and intuition to make the best decisions. Moreover, the optimal configuration might change depending on the input data, GPU architecture, or workload.

For NVIDIA GPUs, the same configuration may not work well across different GPU models. Each GPU has an SM (Section 2.1) with a finite amount of registers, shared memory, and threads. Variations in thread and block sizes produce different occupancy of these hardware resources. A small number of threads per block may not be enough for the GPU to fully utilize all of the warps in the SM, while a large number may lead to threads competing for memory resources, thus increasing latency.

Moreover, there can be many other possible code optimizations. Optimizations such as tiling factor, loop unrolling, memory access patterns, different floating-point precisions, and many others can have a significant impact on code performance.

Taking all of this into account, for many programs, there are many possible configurations that provide correct results. However, there is usually only a small subset of possible configurations that provide substantially better performance. The process of automatically finding the best configuration for a given program is called autotuning [4].

2.5 Kernel Tuner

Kernel Tuner is an open-source auto-tuner designed to create highly optimized and tuned GPU applications. It simplifies the process of optimizing parameters such as thread and block sizes, tiling factors, and many more. Currently, it supports CUDA, OpenCL, and HIP, but it is designed to be easily extendable to other GPU programming languages.

To illustrate how Kernel Tuner works, let us look at an example of tuning a simple CUDA kernel.

The example in Listing 2.2 shows how a simple CUDA kernel for adding two vectors is tuned using Kernel Tuner. The kernel is first specified as a Python string in this case (the file name alongside the name of the kernel is also supported). Afterwards, using Numpy, the input data is generated, `n` which tells us the length of the arrays, the two arrays `a` and `b`, and the output array `c`. These are the arguments that will be used to run the kernel. Next we define the tunable parameters. For CUDA, `block_size_x` is a predefined parameter that Kernel Tuner will use to specify the number of threads per block in the first dimension. Similarly, `block_size_y` or `block_size_z` can be used to tell Kernel Tuner how many threads per block are needed for the given dimension. By default, the search space will be a Cartesian product between all possible values of each tunable parameter. Nevertheless, this can be limited by imposing restrictions, for example, restricting the block sizes to be equal in both dimensions (`block_size_x == block_size_y`).

Finally, we pass the name of the kernel, the source (in this case a string), the parameter `n` which in this case serves as the problem size, the kernel arguments, and the tunable parameters into the `tune_kernel` function. Kernel Tuner uses the problem size parameter to determine the grid size. In this case, the number of blocks per grid is calculated by dividing the problem

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

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}
"""

n = np.int32(10000000)

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

args = [c, a, b, n]

tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

Listing 2.2: Tuning a Cuda kernel with Kernel Tuner

size by the number of threads per block. However, this too can be customized by using grid divisors.

The `tune_kernel` function will return the best configuration for the given kernel along with the performance results. And just like that, we have tuned a kernel with fewer than 10 lines of code.

Now that we have had a brief glimpse of how Kernel Tuner works, let us look at its architecture in more depth. Through this, we explain how Kernel Tuner is designed and why it is a suitable tool for implementing autotuning for Triton.

The whole architecture of Kernel Tuner is shown in Figure 2.1. We will proceed by explaining

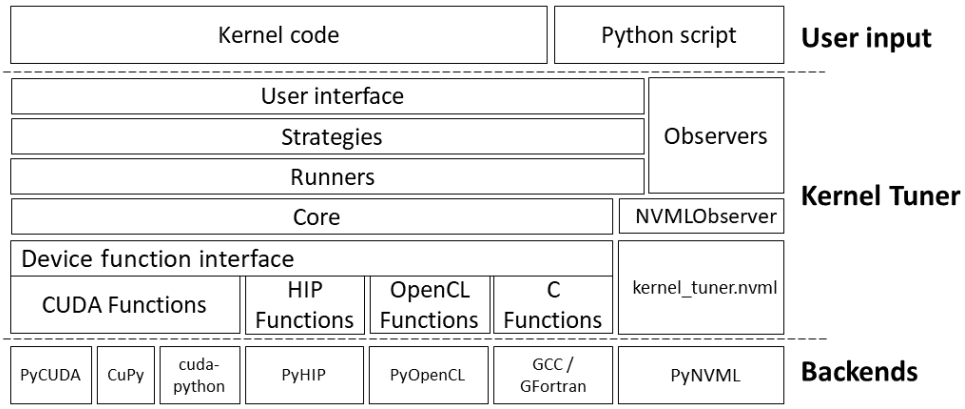

Figure 2.1: Kernel Tuner architecture

the important components from top to bottom.

2.5.1 Interface Component

The Interface component of Kernel Tuner is the part that the end user interacts with. It hosts two main functions: `tune_kernel` and `run_kernel`. In this section, we will focus on the former since it is the more complex of the two and also incorporates the logic of the latter for tuning.

The `tune_kernel` function tells Kernel Tuner how and what to tune. It provides a lot of flexibility to the user, allowing them to specify 33 different arguments.

We will not cover all of the arguments in detail, but we will focus on the most important ones. For starters, the `kernel_name` and `kernel_source` arguments are mandatory and are used to specify the name and source code of the kernel to be tuned. The `problem_size` argument is used to specify the total size of the problem in the form of a tuple. It is used internally to calculate the number of blocks in the grid by dividing the problem size by the block size. The `arguments` argument is used to provide the necessary list of arguments for the kernel to run. The `tune_params` argument is used to specify which parameters to tune and their possible values. If the user does not want to divide the grid evenly, they can use the `grid_div_x`, `grid_div_y`, and `grid_div_z` arguments to specify how to divide the grid. The `restrictions` argument is used to specify restrictions on the parameters to be tuned. For example, the user can specify that the block size must be a multiple of 32. Finally, the `lang` argument is used to specify the language of the kernel source code. This can usually be inferred automatically from the source code itself; however, in some cases, it needs to be specified manually. The rest of the arguments are used for various purposes; however, the ones mentioned are almost always used to tune a kernel. For this reason, we will not cover other arguments in detail.

Internally, `tune_kernel` first validates the provided kernel source and tuning parameters. It then selects a strategy (like the default brute-force or a user-specified one) to determine which parameter configurations to evaluate from the defined search space. A runner (such as the default `SequentialRunner`) is also chosen, which is responsible for executing the compilation and benchmarking tasks for the configurations selected by the strategy. The core tuning process involves the strategy iteratively directing the runner to benchmark configurations. This loop continues until the strategy's completion criteria are met (e.g., the search space is exhausted or a set number of iterations is reached). Finally, the function returns the best-performing configuration found.

2.5.2 Strategy

Strategies are the components responsible for selecting parameter configurations to evaluate from the defined search space. The strategies can be very simple, such as the brute-force strategy, or more complex, such as the Bayesian optimization strategy. The simplest strategy, the brute-force strategy, explores all possible combinations of parameters. It uses the search space object to generate all possible configurations and just passes them to the runner. It then waits for the runner to finish and returns the best configuration.

On the other hand, more complex strategies usually have some sort of model and cost function that they try to optimize. This way, they prepare a subset of configurations to run (which may also be just a single configuration), and they use the runner again to evaluate the performance

of the configurations. Once the runner finishes, the strategy updates the model and cost function and selects the next configuration to run. We will cover some of the strategies in more detail in Chapter 5. Nevertheless, if the strategy you are looking for is not implemented, it is easily extendable by the user.

2.5.3 Runner

Runners are components responsible for evaluating the performance of configurations selected by the strategy. The simplest runner, the `SequentialRunner`, just takes the list of configurations and runs them one after another in a sequential manner. There are also more complex runners, such as the `ParallelRunner`, which runs configurations in parallel using either multiple GPUs on the same machine or multiple nodes in a cluster. Both of these measure the performance of each configuration and, in the end, return the best-performing one. For interacting with the GPU (compiling and benchmarking), Runners use the Device Interface component.

2.5.4 Device Interface

Since all of the implemented GPU languages have similar flows of execution, the DeviceInterface class is used to abstract differences between the languages. It contains the high-level logic that is used to update the kernel source with given parameters, compile the kernel, run the kernel, and measure performance. However, the actual implementation of compilation, running the kernel, and measuring performance is left to the specific language. Instead of having all of the logic for each language in the DeviceInterface, Kernel Tuner uses a modular approach. Each language has its own `Backend` class that implements the necessary functions for that language.

2.5.5 Backends

In order for the `DeviceInterface` to work with different languages, some of the common logic needs to be abstracted away. For this reason, the `Backend` class is used. A backend class is used to provide the necessary functions to tune a kernel in a specific language. Let us look at the functionality that the backend class needs to provide in order to work with Kernel Tuner.

The backend class needs to implement the following functions:

- `ready_argument_list` - used to prepare the list of arguments and push them to the device.
- `compile` - used to compile the kernel instance and return the compiled kernel.
- `start_event` - used to record the starting event.
- `stop_event` - used to record the ending event.
- `run_kernel` - used to run the kernel with specified grid and block sizes.
- `synchronize` - used to synchronize the device with the host.
- `memset` - used to allocate memory on the device.
- `memcpy_htod` - used to copy memory from the host to the device.
- `memcpy_dtoh` - used to copy memory from the device to the host.

- `copy_constant_memory_args` - used to copy constant memory arguments to the device.
- `copy_texture_memory_args` - used to copy texture memory arguments to the device.
- `copy_shared_memory_args` - used to copy shared memory arguments to the device.

If a class implements the following functions, it can be used as a backend for Kernel Tuner. This way, the language/device-specific logic is kept in a single place in the codebase, making it easier to maintain and extend. No other part of the codebase needs to know how the kernel is compiled or run on the device. Furthermore, because of such isolation, adding support for a new language can be as simple as adding a new backend class.

2.5.6 Why Kernel Tuner?

The modularity we have described in the previous sections is what makes Kernel Tuner a great candidate for implementing autotuning for Triton. Most of the components can be reused as they are not tied to a specific backend or device. The core logic is abstract enough to be used with different backends, and the user interface is flexible enough to be used with different search strategies and runners. For this reason, adding support for Triton should be relatively simple. In theory, we just need to create a new backend class that implements the necessary functions for Triton. Since Triton works with PyTorch, we can use PyTorch's functions and tensor objects for implementing the necessary memory management operations. The compilation and running of the kernel can be done using Triton's JIT compiler.

Chapter 3

Field Overview

To understand the key contributions of this work and how it differs from existing work, this chapter will first provide an overview of the broader field of GPU programming, followed by a detailed overview of the field of GPU autotuning.

3.1 The landscape of GPU programming models

The emergence of high-performance computing and more advanced scientific computing applications has led to the development of a diverse ecosystem of GPU programming models. Each of the models has its own strengths and weaknesses, which are often related to trade-offs between performance, programmer productivity, and portability [11]. These models can be broadly categorized as follows:

- **Directive-Based models**: These models offer a low barrier to entry for programmers by allowing them to offload computations to the GPU with a simple set of directives. The developers annotate the code with directives that instruct the compiler to generate the appropriate GPU code. This approach would usually be used on functions that perform computationally intensive loops, such as matrix multiplications. These models also minimize the amount of code modifications and do not affect the code if the necessary SDKs or hardware are not available. However, the downside of these models is that they are not as flexible as the other models, and they may not always yield the highest possible performance. The most notable examples of such models are OpenACC [12] and, more recently, OpenMP [13].
- **Non-Portable, Kernel-Based**: These models allow you to write low-level code that directly interacts with the GPU and its hardware (called kernels). While they offer fine-grained control over how the code is executed, they also require a deep understanding of the GPU architecture, its memory hierarchy, and the programming model. This approach is the most complex, but it's also the one that can yield the highest performance. Furthermore, the code is not portable, meaning that it can, in most cases, run only on a specific set of GPUs (usually provided by one vendor). The best known and most used example of this approach is CUDA (Section 2.2).
- **Portable, Kernel-Based**: These models are the same as the non-portable, kernel-based models, but they are designed to be portable, meaning that the same code can be run on

different GPUs. The most notable examples of this approach are OpenCL [14], HIP [15], SYCL [16], Kokkos [17].

- **High-Level Language Support**: These models do not have a common programming model; instead, they are libraries that provide a high-level interface to the GPU. Some of the notable examples are Numba [18], which is an open-source JIT (Just-In-Time) compiler that translates a subset of Python and NumPy code to run on the GPU. Another example is CuPy [19], which is a library aimed at data science applications. It has a similar interface to SciPy and NumPy, and it utilizes the GPU for its computations, making it easy for data scientists to accelerate their workflows. Then there are also wrapper libraries like CUDA Python [20], PyCUDA [21], and HIP Python [22], which offer Python bindings to the CUDA and HIP APIs. However, in most cases, prior knowledge of CUDA or HIP is required to use these libraries.

In this work, we use Triton (Section 2.3), which falls into the portable, kernel-based category. However, compared to other languages in that category, Triton is unique in that it is a much higher-level language, yet it retains most of the performance benefits of the lower-level languages.

3.2 The landscape of GPU autotuning

While the programming models described above provide the means to execute code on a GPU, they do not automatically guarantee optimal performance. The mapping of a parallel algorithm to a specific GPU architecture involves numerous tunable parameters, such as block size, grid dimensions, and memory access patterns. Finding the best combination of these parameters is a complex task that is critical for performance. This challenge motivates the field of software auto-tuning.

The field of auto-tuning can be split into two main paradigms: (1) auto-tuning compiler-generated code optimizations and (2) software auto-tuning [23]. The former is focused on enabling the right compiler optimizations for a given code [24]. However, in this work, we are interested in the latter, which is focused on tuning user-defined parameters.

There have been some very domain-specific autotuners, such as ATLAS for dense linear algebra, OSKI for sparse linear algebra, FFTW and SPIRAL for signal processing, etc. [25]. However, these, as the name suggests, can only work with the code for a specific domain. In this work, we are interested in creating a generic autotuner that can be used for any GPU kernel. In recent years, there have been several generic autotuners that can work on any domain.

One of the first ones was Opentuner, which allowed users to specify a search space and provided an ensemble of search techniques to attempt to find the best configuration more effectively [26]. However, it did not have support for tuning GPU kernels. GPTune was one of the first tuning frameworks for MPI-based applications with support for parallel tuning. CLTune was the first autotuner that was specifically designed for tuning GPU kernels, in this case those written using OpenCL [27]. Auto-tuning Framework (ATF) constructs search spaces in a more efficient way by using a chain of trees (search space structure) [28]. In our work, we build upon a generic autotuner (Kernel Tuner, see Section 2.5) that also has support for efficient search space exploration.

When the performance of the kernel varies greatly depending on the input data, it is possible to use online autotuning. In this case, the autotuner is integrated into the application and can,

during runtime, adapt the necessary parameters for the best performance. One such example is Kernel Tuning Toolkit (KTT), which is a C++ library developed to support online autotuning of GPU kernels [29]. In our work, we focus on offline autotuning, as we are interested in finding the best configuration for Triton kernels that is independent of the input data.

There are also some autotuners that specialize in floating-point tuning, like GPUMixer [30]. Because different floating-point operations can have different performance characteristics on different hardware, changing the precision can have a big impact on the performance of the code. For this reason, libraries like Kernel Float [31] have been developed to make it easier to work with and tune different floating-point precisions. Since Triton allows you to specify the floating-point precision (e.g., `tl.float16`), we could tune the floating-point precisions with Kernel Tuner by extracting the data types into variables that can be tuned.

Since autotuned parameters are not portable, i.e., they are specific to the hardware and application, some work has been done to create portable autotuners. An example of that is the Kernel Launcher [23], which has a runtime kernel selection that can select the best kernel for the given hardware. It captures and later replays the kernel launches with different parameters in order to generate different parameter configurations. The idea of capturing and replaying the kernel is also used in the record-replay autotuner [32]. There, a Bayesian optimization algorithm is used on the captured instances to find the right parameters for the kernel execution.

Triton, the language we are tuning in this work (Section 2.3), also has built-in autotuning capabilities. It provides the option to specify multiple configurations, which will be executed in parallel, and the best one will be selected at runtime. However, specifying the configurations manually can be cumbersome, and it does not provide any search space exploration techniques. Furthermore, the autotuning is executed at runtime, which means that the first execution of the kernel will be slower, and because of that, in most cases, we would not want to specify numerous configurations.

In Listing 3.1, we can see how Triton's built-in autotuner works. The `@triton.autotune` decorator specifies three different configurations, each with different values for `BLOCK_SIZE` and `num_warps`. The `key=['n_elements']` parameter tells Triton to cache the best configuration based on the problem size, meaning that for each unique value of `n_elements`, Triton will remember which configuration performed best. When the kernel is first called with a specific problem size, Triton will benchmark all three configurations and select the fastest one for future invocations with the same problem size. This example illustrates the main limitations of Triton's autotuner: the configurations must be manually specified, there is no automatic search space exploration, and adding many configurations can significantly slow down the first execution.

In our work, we address these shortcomings by using Kernel Tuner [4] (see Section 2.5), a framework designed for automatic offline software tuning of GPU kernels.

Kernel Tuner encompasses many of the features of the previously mentioned autotuners; for example, it has support for different search space exploration techniques and can be used with different GPU languages. Recently, Kernel Tuner has been extended to support AMD GPUs through the addition of HIP support. The results obtained by tuning HIP kernels have been shown to improve performance on AMD GPUs up to 10x [33]. Our work is most similar to this, as we also use Kernel Tuner to add support for tuning kernels in a new language—in this case, Triton.

```python
import triton
import triton.language as tl

@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE': 128}, num_warps=4),
        triton.Config({'BLOCK_SIZE': 256}, num_warps=8),
        triton.Config({'BLOCK_SIZE': 512}, num_warps=16),
    ],
    key=['n_elements'],
)
@triton.jit
def vector_add_kernel(x_ptr, y_ptr, output_ptr, n_elements, BLOCK_SIZE:
    tl.constexpr):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)
```

Listing 3.1: Triton vector addition kernel with built-in autotuner

Chapter 4

Implementation

In this chapter, we will describe how we added support for Triton to Kernel Tuner. As Section 2.5 showcased the inner workings of Kernel Tuner, we will focus on the changes that were made to the existing codebase. Unlike the existing languages, Triton is the only one that is written in Python, and its compiler expects a runnable Python function as input. This presents a unique challenge, as the existing backends all work with file-based source code. Nevertheless, since Triton is a new language, a great way to start the implementation is to create its own backend class.

4.1 Triton Backend

Due to the modular design of Kernel Tuner, implementing a new backend is fairly straightforward. The new backend class needs to implement the functions mentioned in Section 2.5.5. After that, the whole framework should work with the new language without any changes to the existing codebase.

So, let us take a look at how the necessary functions were implemented for Triton. The function descriptions are available in Section 2.5.5.

4.1.1 Ready Argument List

For the ready argument list function, we need to do preprocessing of the arguments if needed and push the arguments to the device. As mentioned in Section 2.3, Triton uses PyTorch to handle the memory management. For this reason, we needed to convert the incoming arguments to PyTorch tensors first. Luckily for us, Kernel Tuner only allows NumPy and torch tensors as arguments. If the arguments are already torch tensors, we just push them to the device. Otherwise, the two libraries play nicely together and provide the helper functions to convert between the two (for example, `torch.from_numpy`). Finally, pushing the arguments to the device with PyTorch is as simple as calling the `to` function on the tensor, with the device as an argument.

4.1.2 Compile

In this function, we will see the first major difference between the existing backends and the Triton backend. The usual process flow for compilation is to take a source file containing the

kernel code, compile it, and return either a wrapper function or a reference to the compiled kernel file. After that, the constant, shared, and texture memory arguments are copied to the device. And finally, the kernel is run by calling the compiled kernel, providing the grid and block sizes, and passing the arguments.

However, Triton does not work like this, and its JIT (Just-in-Time compile) function is quite different from the other languages. Firstly, the JIT function expects a Python function as input, not a file or a source code string. Secondly, the JIT function will not compile the kernel until the kernel needs to be executed.

The first problem is easily solvable by just passing the Python function instead of the file to the backend. However, the second problem carries additional complexity. While the Triton codebase does have a compile function available, it is not at all documented, and we believe it is not meant to be used by the end user. Even if we were to use it, the future versions of Triton might change how it is used and break the compatibility with Kernel Tuner. For this reason, we decided to use the JIT function since it is the only documented way to compile Triton code.

Since Triton handles advanced memory management on its own, it needs to know the arguments that will be passed and the grid and block sizes before the kernel is compiled. It uses this information to try to squeeze out as much performance as possible by trying to squeeze as much data as possible into the shared memory. For our backend, this means we need to extend the compile function to take the arguments alongside the kernel instance.

Furthermore, the JIT function also accepts some special parameters that can greatly impact performance. Because we want to tune those parameters as well, we need to extract them and pass them to the JIT function. We will currently support the following special parameters [34]:

- `num_ctas` - controls the number of blocks that are run in parallel.
- `num_warps` - controls the number of warps per block.
- `num_stages` - refers to the number of pipeline stages that are used for shared memory and data prefetching.

In order to prepare the kernel for compilation, we first call the jit function to create a `JITFunction` wrapper object. This object contains the metadata about the kernel that is necessary for compilation. Finally, we call the JITFunction object, passing the grid and block sizes, the arguments, and the special parameters. This will not only compile the kernel but also run it.

While we can pass an argument called `warmup` to the JIT function, which will just compile the kernel without running it, we lose one important feature. That is, we will not know whether the kernel is actually viable to run before we run it once. The way Triton's JIT is currently implemented, it will only check whether the device has sufficient memory resources to run the kernel when the kernel is run. For this reason, we avoid using the `warmup` argument and just run the kernel once.

This has two implications. Firstly, the exact compilation time will never be known since the kernel is run as soon as it is compiled. This is not a big issue since we are not trying to compare or improve compilation times, as there is only one Triton compiler. Furthermore, even if we were to use the warmup parameter, we would have to alter the abstract logic for computing compilation time within Kernel Tuner.

Secondly, we need to consider whether the kernel is always going to be compiled before running. This could be a huge issue since we do not want to include the compilation time in the performance measurements. Longer compile times may mean that the kernel was better optimized and will run faster.

Fortunately, Triton caches the compiled kernels, meaning that subsequent invocations of the same kernel will not recompile the kernel. This means that we do not receive the compiled version of the kernel from the JIT function, but calling the same function with the same parameters will result in a cached version of the kernel being used.

While this may seem like an unorthodox way of compiling and running the kernel, it is also used by the Triton developers in their autotuner implementation.

4.1.3 Run Kernel

Apart from setting a stream before executing the kernel, the `run_kernel` function works the same way as the compile function described in Section 4.1.2. Only this time, the kernel has already been compiled, and when we run the kernel, we are only measuring performance. The reason these two functions are separated in this case is because there may be some extra flags that can be passed to the JIT function that we only want to use during the compilation phase. For example, the `warmup` parameter was used to tell the JIT function to only compile the kernel without running it.

4.1.4 Start and Stop Event

For recording the start and stop events, we use the PyTorch `torch.cuda.Event` class. When initializing the backend, we create two events that are used to record the start and stop events. We also enable timing on the events in order for time to be accurately measured. The time is then obtained by using the `elapsed_time` function on the start event and passing the stop event as an argument.

4.2 Parsing and Creating Triton Kernels

In the previous section, we mentioned that the Triton backend expects a Python function as input. This means that the user needs to provide a Python function that will be used as the kernel. In contrast, the other languages either provide a string containing the kernel code or a file name containing the kernel code. Regardless of the input, other languages always end up being read and updated as strings.

Furthermore, the templating system used in Kernel Tuner is based on string manipulation to insert values into the preprocessed source code. This means that for the Triton integration to work, we would need to convert the Python function to a string, update the parameters, and then convert it back to a Python function. However, this approach presents several challenges that we address by using AST (Abstract Syntax Tree) manipulation.

4.2.1 Problem statement

While operating on strings is a valid approach, having a Python callable object (function) as kernel source has its advantages. For example, developers do not need to parse the arguments

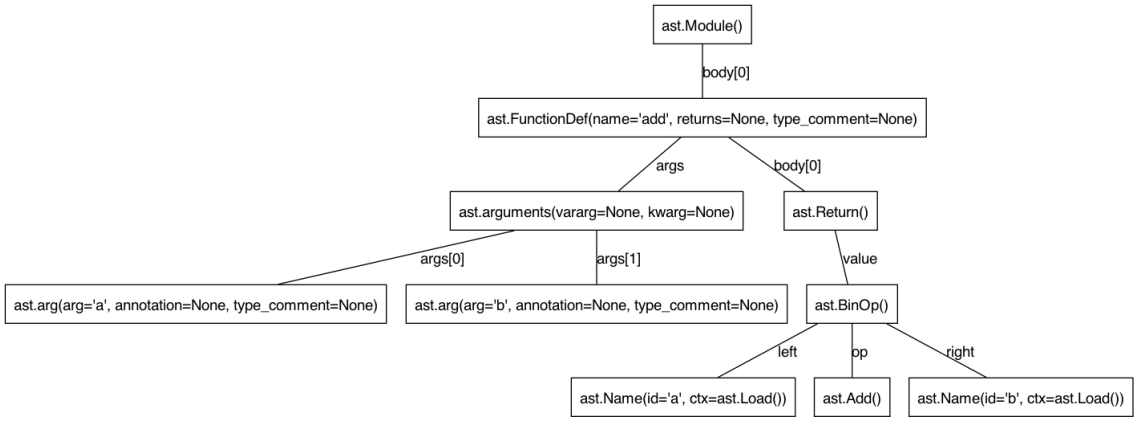

Figure 4.1: AST representation of the code in Listing 4.1

of the function to check whether they match user-provided arguments. This is already available in the Python callable object. Furthermore, editing the kernel source can be done by using AST (Abstract Syntax Tree) manipulation, which is a more robust way of manipulating code.

For this reason, we decided to create a new method of applying parameters to the kernel source. In the case of Triton and potentially other languages that use Python functions as kernels, we will update parameters by using AST manipulation.

4.2.2 Abstract Syntax Trees

Before we explain how we used AST (Abstract Syntax Tree) to update the kernel source, we will provide a brief explanation of what AST is. AST is a tree representation of source code where each node in a tree represents a construct in the source code. Let us look at an example to better understand how it works.

```
1 def add(a, b):
2     return a + b
```

Listing 4.1: Example Python code for AST

From Figure 4.1, we can see what the tree structure of the code in Listing 4.1 looks like. The root of the tree is always the module, followed by one or usually more nodes that represent various constructs in the code. In this case, we only have one function declaration node that has two arguments and a return statement. The arguments and the return statement are, in this case, children of the function declaration node. Finally, the return statement consists of a binary operation node, where the operator is the addition and the left and right children of the node are the variables a and b. With this knowledge, we can now change the code by updating the nodes in the tree.

4.2.3 Applying Parameters to Triton Kernels

Finally, we can apply parameters to the Triton kernel by using AST manipulation. In order to do so, we use the `ast` module that is part of the Python standard library. It allows us to parse Python code into an AST that can be traversed and modified as needed.

To specify how the nodes should be updated, we use the `ast.NodeTransformer` class. Extending this class allows us to override the `visit_Name` method, which is called when a node of type `ast.Name` is visited. Those would be the nodes that contain some sort of identity (variable, function name, etc.) in the form of a string (called name). For us, the important ones are the nodes that represent the variables within the kernel function. We want to replace those variables with the actual values that the user provided. With this, we apply tunable parameters to the kernel source and generate a new kernel.

Moreover, we need to be careful not to replace the variables that are being initialized (for example, `a = 2`). For this, we need to look at the context of the node. In Python's AST, each node has a context attribute that indicates how the variable is being used.

- `ast.Load` - the variable is being read (e.g., `x + y`)
- `ast.Store` - the variable is being assigned to (e.g., `x = 5`)
- `ast.Del` - the variable is being deleted (e.g., `del x`)

If the node is being assigned a value or deleted, we do not want to replace it. However, by checking that the node's context is of type `ast.Load`, we can be sure that the node is simply reading the value of the variable and not updating it.

With this in mind, we can pass our parameters to our transformer object and update the AST in those places where the parameter name matches the node name and the context is of type `ast.Load`.

Now that we have a transformer class defined, we first need to create a deep copy of the kernel function. This is because the `ast` module works with references, and we do not want to modify the original kernel function. Once that is done, we apply our transformer to the copied function and obtain the updated AST.

From this point on, we have two options for converting the AST back to a Python function. The first option is to use the `ast.unparse` function to convert it back to a string. Followed by using the `compile` and `exec` functions to convert the string to a Python function. The second option is to again convert the AST to a string, but this time write it to a temporary file. After that, use the `importlib` module to import the function from the file.

The first option is simpler and faster; however, the Triton JIT function uses the `inspect` module to get the source code of the function. This means that it looks for the source code in the file where the function is defined. If we use the first option, we will not have the source code in the file, and the JIT function will not work. For this reason, we decided to use the second option, as it is easier to debug and it works with the JIT function.

Creating and using temporary files is pretty simple in Python. We use the `tempfile` module to create a temporary file, write the updated AST to the file, and then import the function from the file. In the end, we also include the name of the file in the `KernelInstance` object to make sure that we delete it once the kernel is done running.

4.3 Handling Kernel Dependencies

Apart from applying parameters to the kernel, we also need to make sure that kernel dependencies are present in the generated kernel. Currently, these can only be the `triton` and `triton.language` modules, as well as the other kernels that may be used within the kernel.

For the two Triton modules, we create a new AST node that we append to the body of our kernel's AST. For the latter, we need to first determine which other kernels are used within the provided kernel. This is done by traversing the original kernel AST and collecting the function names that are used within the kernel. Once we have the function names, we can traverse the function definitions of the entire module and collect the functions that have the `triton.jit` decorator. Since we will not be editing the dependency functions, we can simply copy them to the new AST before adding the updated kernel.

This way, we can test a more complex kernel that uses other kernels without having to edit the original kernel source.

4.4 Small changes to Kernel Tuner

To make the Triton backend work with Kernel Tuner, we needed to make a few small changes to the existing codebase. First of all, we needed to improve support for passing and receiving PyTorch tensors as arguments. Initially, you could pass PyTorch tensors as arguments, but the results would always be returned as NumPy objects. Furthermore, the results were not properly copied to NumPy arrays because they stayed on the device (GPU). This was fixed by allowing the results to be returned as PyTorch tensors or converted to NumPy arrays if needed.

Secondly, we needed to extend the `compile_kernel` function to accept GPU arguments in order to compile the Triton kernel. Thirdly, we created new classes to handle different kernel sources. In front of it, we added a kernel source factory that is used to create the correct kernel source object based on the language. These would then have the correct functions to parse and apply parameters to the provided kernel.

4.5 Parallel compilation

One of the big disadvantages of using Triton alongside autotuning is compilation time. Triton compiles the kernel using a single core; furthermore, in order to know whether the kernel is viable to run, we need to compile it once and run it. On top of that, compilation time can take up to 30 minutes for more complex kernels with multiple stages. This is a big bottleneck if you want to explore a large search space. Moreover, long compilation time means that the GPU will most likely cool down before running the kernel. This will produce less consistent performance measurements, potentially leading to incorrect results.

Because we wanted to explore the entire search space for some problems that are already very well optimized, like matrix multiplication, we needed to address this issue. The result is a new module within Kernel Tuner called `paralleltritoncompiler`.

The key idea is to precompile as much of the search space as possible using multiple cores and possibly multiple nodes in a cluster. As mentioned there, Triton does offer a kernel compiler module that is not documented, but if examined closely, it is possible to use it. It requires a Python function, its signature, the necessary constants, target architecture (of the GPU), and the special parameters that control the number of CTAs, warps, and stages (see Section 5.1.3).

This seems like a great solution since it is not tied to a specific GPU or a specific node in a cluster. Furthermore, we do not need to run the kernel in order to compile it. Unfortunately, this approach has one key problem. Compiling the kernel produces a hash, which is used to store it in Triton's cache, such that subsequent invocations of the same kernel will not recompile it. The big issue is that the hash generated by Triton's compile module produces a different hash

than the hash produced by the JIT function. This means that the JIT function, which is used to run the kernel, will not be able to find the compiled kernel in the cache and will recompile it.

To solve this, we need to generate the same hash as the JIT function. However, understanding and properly implementing the hash function is not an easy task, and any changes made to it by the Triton team will break our implementation. For this reason, we decided to use the `warmup` argument of the JIT function. This argument allows us to compile the kernel without running it. Nevertheless, this comes with one major drawback, and that is that although the kernel does not need the GPU to run, it requires it to be present to infer the target architecture. There are no arguments that can be passed to the JIT function to specify the target architecture, so the multi-node approach is not possible (unless we have multiple nodes with the same GPU). Regardless, compiling the kernel on today's computers means we can compile on at least 4-6 cores at a minimum. This is a significant speedup compared to the single-core compilation. Fortunately for us, the nodes we will be using in Chapter 5 have 64 cores, allowing us to precompile large search spaces in a reasonable time.

After evaluating both compilation methods, we adopted the `warmup` argument approach of the JIT function as our final solution. While the undocumented kernel compiler module initially seemed promising due to its independence from specific GPU hardware, the hash incompatibility issue made it impractical for our use case. The `warmup` approach, despite requiring GPU presence for target architecture inference, provides several key advantages:

- **Cache Compatibility:** Compiled kernels are stored with the correct hash, ensuring subsequent JIT function calls can find and reuse them
- **Documented API:** The `warmup` parameter is part of Triton's official interface, reducing the risk of breaking changes
- **Sufficient Parallelization:** Single-node multi-core compilation provides sufficient speedup for our evaluation scenarios

This approach allows us to precompile entire search spaces across multiple CPU cores while maintaining full compatibility with Triton's caching mechanism. The requirement for GPU presence is acceptable in our evaluation context, where benchmarking naturally occurs on GPU-equipped systems.

Now that we have established our compilation approach, the actual implementation is straightforward. We will add another function to Kernel Tuner's interface that will accept a kernel function, its name, arguments, and tunable parameters. We will reuse the searchspace module to build the search space, or in other words, all possible parameter combinations.

Since Python can only run one thread at a time because of its global interpreter lock, we will use the `multiprocessing` module to parallelize compilation. This module allows us to create new processes (instead of threads) and run them in parallel as workers. Each worker will be assigned a different parameter combination and will compile the kernel.

We could just keep a set of workers that is equal to the number of cores and assign each worker a new parameter combination every time one finishes. However, in rare cases, some LLVM processes can crash during the compilation. These crashes occur during the compilation pipeline of the LLVM compiler. Unfortunately, Python's `try-except` blocks cannot catch these failures. This causes every process within the pool to terminate, leaving us with no workers and losing all progress. There are several approaches we could use to address this issue. We could spawn a

separate process for each kernel compilation. This would isolate the failure of one process from the rest, but it would introduce a significant overhead of process creation and termination after each compilation. Another approach would be to use alternative libraries like `Dask` or `Ray` to handle the workers. While this would allow for more robust error handling, it would introduce additional complexity and dependencies. Since Python 3.13 introduces experimental support for disabling the GIL, we could use threads to compile the kernels. This would allow parallel compilation and most likely solve the issue, but it would require the latest Python version which does not meet Kernel Tuner's requirements. Finally, the approach we selected is sticking with the `multiprocessing` module and compiling the kernels in batches. This provides a balance between robustness and simplicity. The batches ensure that, in case of a crash, we only lose one batch of kernels, and we can continue with the rest of the search space. The downside is that we lose speed, as the slowest kernel in the batch will determine the time it takes to compile the entire batch.

While this introduces some inefficiency (batches wait for their slowest member), the robustness benefits outweigh this cost in our evaluation scenarios where reliability is paramount. The exact implementation and details are available in the public GitHub repository here.

4.6 Benchmark Framework

While Triton has a built-in benchmarking function that accepts a kernel function, its arguments, and the configurations to benchmark, it does not provide a way to store configurations for different inputs and different GPUs in a structured manner. Furthermore, it has no command-line interface, which makes it more difficult to use on clusters where we define SLURM jobs. For this reason, we developed a simple benchmark framework that allows us to define inputs and parameters in a YAML file and have a command-line interface to run the benchmarks.

The YAML files support two main types: input size-based benchmarks, such as matrix multiplication where we define different matrix sizes, and model parameter-based benchmarks, such as attention mechanisms where we define different model parameters like sequence length or head dimension.

The framework was designed to be used through a command-line script, where the user would specify the benchmark type (like `matmul` or `attention`) and afterwards select which configuration they want to run. For example, `run_benchmark.py matmul --config_type kernel_tuner --keys 4096,8192` would run the matmul benchmark with the Triton configuration for matrix sizes 4096x4096 and 8192x8192. Results are stored in YAML files within a `results/` directory, capturing the benchmark timings (median, percentiles), the configuration used, benchmark parameters, system information, and details about the GPU used during the run. This structured approach simplifies the process of comparing performance across different implementations and hardware setups. Further details about the framework's structure and usage can be found in the project's repository.

Chapter 5

Evaluation

5.1 Experiments

5.1.1 Hardware Configuration

Experiments were conducted on two different GPU architectures to provide a comprehensive evaluation:

NVIDIA RTX 5000 Ada Generation This GPU is based on the Ada Lovelace architecture and provides high computational power for deep learning and numerical optimization tasks. The key hardware specifications are:

- **GPU Model:** NVIDIA RTX 5000 Ada Generation
- **Memory:** 32 GB GDDR6
- **CUDA Cores:** 7680
- **Power Consumption:** Up to 250W

The system was configured to run in performance mode (`P0`), ensuring maximum computational efficiency.

AMD Radeon Pro W7800 To provide a comparison across different GPU architectures, we also conducted experiments on an AMD Radeon Pro W7800 GPU. This workstation-class GPU is based on the RDNA 3 architecture and offers the following specifications:

- **GPU Model:** AMD Radeon Pro W7800
- **Memory:** 32 GB GDDR6
- **Stream Processors:** 7168
- **Compute Units:** 56
- **Power Consumption:** Up to 211W

The AMD GPU was configured to run with optimized power settings for maximum performance.

5.1.2 Software Stack

The tuning experiments were conducted using the following software stacks:

NVIDIA Environment

- **NVIDIA Driver:** 545.23.06
- **CUDA Version:** 12.3
- **Kernel Tuning Framework:** Kernel Tuner with Triton (3.1.0)
- **Operating System:** Rocky Linux 8.5 (Green Obsidian)

AMD Environment

- **AMD Driver:** AMD ROCm 5.7.1
- **HIP Version:** 5.7.31511
- **Kernel Tuning Framework:** Kernel Tuner with Triton (3.1.0)
- **Operating System:** Rocky Linux 8.5 (Green Obsidian)

5.1.3 GPU tunable parameters

In the following sections, we will be tuning parameters that are specific to certain algorithm implementations. However, in each case, we will also be tuning two special parameters that are provided to us by the Triton compiler. These are `num_warps` and `num_stages`. The former tells us how many warps (groups of 32 threads) per block will be used to execute the kernel. For example, if we specify `num_warps=8`, we will use 8 warps per block, or 256 threads per block. The latter tells us how many stages of memory coalescing will be used to load the data from memory. The higher the number, the more data we will try to fit into the L2 cache. In some cases this will result in a performance increase, while in others, some parameters like block sizes will need to be reduced to fit the data in the L2 cache, resulting in a performance decrease.

5.1.4 Data types

In the following sections, we will be working mainly with floats for our experiments. The default data type for floats in our experiments is float16 unless specified otherwise. This is because most of Triton's own examples use float16, and we want to keep the comparison fair. Integers will always be of type `int32`.

5.1.5 Obtaining the baseline configuration

In order to obtain the baseline configuration, we will be using the auto-tuner provided by the Triton compiler. Unfortunately, the auto-tuner does not provide us with an interface to obtain the configuration it found, but it does provide us with the ability to set an environment variable that will print the best configuration to the standard output. The environment variable is `TRITON_PRINT_AUTOTUNING` and it should be set to the value of 1. With this set, we can run the kernel with a given input, and we will note down the configuration it used.

5.1.6 Strategies

In this section, we will describe the possible strategies we can use to explore the search space. Many of these can come in handy when the search space is large and we need a more efficient way to explore it. Or, moreover, when we want to tune a kernel but we have limited resources and are time-constrained. All of the strategies we will mention in this section are already implemented in the `kernel_tuner` library.

Brute force

The default strategy used by `kernel_tuner` is the brute force strategy. This strategy will try all the possible combinations of the parameters. For example, if we have 3 parameters, and each parameter has 10 possible values, the brute force strategy will try all $10^3 = 1000$ combinations. While this strategy is simple and guarantees us finding the best configuration, it is not the most efficient, as all the combinations are tried, even if some of them are clearly worse than others.

Basin hopping

Basin hopping is a global optimization algorithm that is particularly useful for exploring complex, non-convex search spaces, such as those encountered when autotuning GPU kernels. The method is designed to escape local minima and find the global minimum by iteratively performing local optimization steps and accepting or rejecting new solutions based on a probabilistic criterion.

Bayesian Optimization

Bayesian optimization is a probabilistic method that uses a Gaussian process to model the objective function. It is particularly useful for optimizing complex, non-convex functions. It works by building a probabilistic model called the surrogate function (which models the objective function), which it uses to guide the search. The new configurations are evaluated and used to update the surrogate function [35].

Differential Evolution

Differential evolution is a population-based optimization algorithm that is particularly useful for optimizing complex, non-convex search spaces. Furthermore, unlike some strategies, it does not use gradients to minimize the objective function, which means that the function does not need to be differentiable. Each candidate solution is a vector of parameters whose quality is evaluated by the fitness function, which needs to be pre-defined. In the case of kernel tuning, the fitness function is the performance of the kernel (time in milliseconds). Similar to other population-based strategies, new solutions/population members are generated by mutating the current population. The mutation is done by taking the difference between two randomly sampled population members. The result is then multiplied by a constant factor (F) and added to a third random individual. The result is then used to generate a new candidate solution [36].

Genetic Algorithm

Similar to the differential evolution algorithm (Section 5.1.6), the genetic algorithm is a population-based strategy. It retains all of the benefits of the differential evolution algorithm but uses a different selection, mutation, and crossover strategy. The fitness function and the

initial population are the same as in the differential evolution algorithm. In the selection step, a portion of the population is selected (usually) based on their fitness in order to create a new population. The new population is created by crossing over the selected individuals. Crossover can be done in a number of ways; one of the simplest is the single-point crossover. In the single-point crossover, a random point is chosen, and the values of the parents are swapped at that point. In kernel tuning, that means that we will be switching the values of randomly picked parameters between the parents [37]. Applying this to the population, we will be creating a new population of children, which will replace the old population [38]. Finally, to have some randomness and escape local minima, there will be a random portion of the population that will be mutated and not crossed over [38].

Greedy ILS

Greedy algorithms are a type of optimization algorithm that picks the locally optimal solution at each step. Iterative local search (ILS) algorithms make initial solutions and then iteratively try to improve them. The combination of the two is a greedy iterative local search algorithm. The algorithm starts with a greedy solution and then applies local search to try to improve it. This can help escape local minima, which is a common problem with greedy algorithms.

Greedy MLS

Multi-Start Local Search (MLS) works in two phases. Firstly, they generate a solution, in this case using a greedy algorithm. Secondly, they apply local search to the solution to try to improve it. Since it is a multistart algorithm, it will generate different initial solutions, apply local search to each, and then pick the best solution.

Minimize

This method allows the user to specify a minimizer method, which will then be used to optimize the objective function. Essentially this becomes a local search algorithm where the user can specify the local search method. Some of the methods available are Nelder-Mead, L-BFGS-B, and Powell. By default, Kernel Tuner uses L-BFGS-B as the local search method.

Random Search

Random search is a simple optimization algorithm that works by randomly sampling the search space. It samples random configurations and evaluates them.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another population-based optimization algorithm. It works by having a population of particles, which are randomly initialized and then iteratively updated. Each particle has a position and a velocity. The position is the current configuration, and the velocity is the difference between the current and the best configuration found by the particle. The particles are updated by adding the velocity to the position and then projecting the new position back into the search space.

5.1.7 Benchmarking

Comparing the performance of different tuning configurations (benchmarking) will be done using Triton's built-in benchmarking tool. Triton provides a handy `triton.testing.do_bench` function where you can pass a function that will be timed, the quantiles that you want to calculate, the number of warmup runs, and the number of runs. Warmup runs are runs designed to warm up the GPU and make sure that the kernel is compiled and the memory is allocated. The number of runs is the number of times the function will be timed.

With this, the way we compare the performance of two different configurations is the following. Firstly, for each configuration, we clean all the data from the GPU and the memory. Then we push the data to the GPU and call the `triton.testing.do_bench` function. For the warmup runs we use 10 runs, while for the actual runs we use 100 runs. We store the results obtained from the benchmarking tool in a YAML file.

5.1.8 Matrix Multiplication

Matrix multiplication is one of the most used operations in the field of deep learning. It is notoriously difficult to optimize, and the algorithm implementations are often provided by vendor libraries such as cuBLAS. However, these lack the customizability that modern deep learning applications often require [39].

OpenAI's Triton provides an example implementation of this algorithm on their website. Furthermore, they also use the built-in auto-tuner to find the most performant implementation. In this section, we will take the Triton implementation as a baseline and try to improve it.

Firstly, let us look at the baseline implementation, explain the idea behind the algorithm, and see which parameters can be tuned.

```
# Do in parallel
for m in range(0, M, BLOCK_SIZE_M):
  # Do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

Listing 5.1: Matrix Multiplication pseudo code

The kernel implemented follows the pseudocode in Listing 5.1. In the example, we multiply a matrix of dimensions (M, K) with a matrix of dimensions (K, N) using the block-based matrix multiplication algorithm. Block-based algorithms are chosen because they are more cache-friendly, and they improve the computational intensity of the operation.

Triton's example implements the previously mentioned pseudocode with one key difference. In order to improve the L2 cache hit rate, they launch blocks in an order that promotes data reuse. They arrange "super-group" the blocks in groups of `GROUP_M` rows before switching to the next column.

To better understand this, let us look at the example in Figure 5.1. We can see that in the first example, where we do a regular block-based multiplication, we need to load 90 blocks into

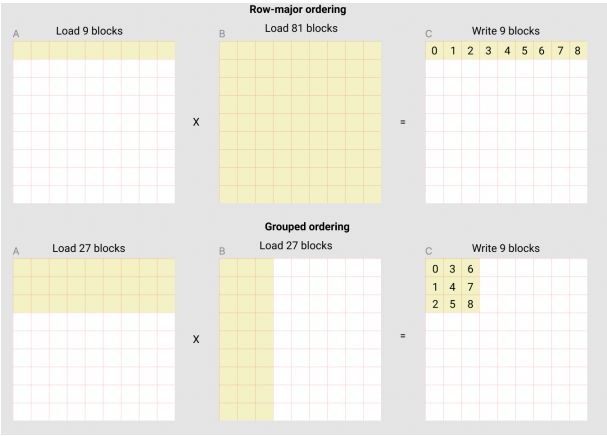

Figure 5.1: Super-grouping blocks

SRAM in order to compute the first 9 blocks of the output. However, in the second example, where we super-group the blocks, we only need to load 54 blocks for the same amount of output blocks.

With that change, the whole kernel code can be found in the Appendix, Listing A.1. From it we can see that we can tune the following parameters:

- `BLOCK_SIZE_M`: The size of the blocks in the M dimension.
- `BLOCK_SIZE_N`: The size of the blocks in the N dimension.
- `BLOCK_SIZE_K`: The size of the blocks in the K dimension.
- `GROUP_SIZE_M`: The number of blocks in the M dimension to super-group.

These are also the parameters that the official implementation tunes. We will be tuning these parameters (and the parameters from Section 5.1.3) for the following sizes of matrices:

- 4096×4096
- 8192×8192
- 16384×16384
- 32768×32768

These GPU sizes are chosen because they are sufficiently large to utilize the GPU's resources and also tend to have a runtime of at least multiple milliseconds, which avoids potential precision issues when benchmarking.

As matrix multiplication is a very well-known and important operation, we want to try to find the best configuration for it. For this reason, we will be using the brute force search strategy (Section 5.1.6) to find the best configuration.

Comparison to vendor libraries

Apart from comparing the performance between the different Triton configurations, we will also compare it to the performance of vendor libraries. For this, we will be using PyTorch's

implementation of the matrix multiplication operation. PyTorch's implementation uses cuBLAS or hipBLAS (depending on the GPU) under the hood, and it is a good baseline to compare to.

We will obtain the results for PyTorch separately by first defining the matrix sizes, pushing the data to the GPU, and then benchmarking the kernel performance using CUDA events. This way we ensure that we time only the kernel execution time and not the data transfer times.

5.1.9 Group GEMM

In the previous section, we have seen an implementation of the General Matrix Multiplication (GEMM) algorithm. But what if we want to multiply a large number of matrices that are not necessarily of the same size? This is where the Group GEMM algorithm is used to multiply multiple matrices at once.

The key idea is to partition the matrices into groups and launch a single kernel to process multiple groups of matrices at once. This is in contrast to the regular GEMM algorithm, where we launch a kernel for each matrix. The groups are usually formed in such a way that matrix multiplications in a single group have the same dimensions. Afterwards, the matrices within the same group are multiplied using the regular GEMM algorithm (see Section 5.1.8) without using the super-blocks technique. In this way, we launch a single kernel, load the data for multiple matrices onto the main memory of the GPU at once, and with that, improve performance. The full kernel implementation can be found in the Appendix, Listing A.2.

Because the algorithm uses regular GEMM in the inner loop, we can use the same tuning parameters as in the previous section. In contrast to the previous section, we will not be tuning the parameters described in Section 5.1.3, because the official example does not use them. However, there is one more parameter that we can tune, which is the number of streaming multiprocessors (SMs). The algorithm is designed in such a way that each tile index will be processed by a different SM. Since different GPUs have different numbers of SMs, and since we do not know how many SMs will be available, we can pick multiples of the GPUs' SMs. For this reason, we will be tuning the following parameters:

- `BLOCK_SIZE_M`: The size of the blocks in the M dimension.
- `BLOCK_SIZE_N`: The size of the blocks in the N dimension.
- `BLOCK_SIZE_K`: The size of the blocks in the K dimension.
- `NUM_SM`: The number of SMs to use.

Given the large search space, it is not feasible to use the brute force search strategy (Section 5.1.6) to find the best configuration. For this reason, we will be using the genetic algorithm strategy (Section 5.1.6) to find a more performant configuration. This strategy does perform well, but it does not guarantee us to find the best configuration, but it is a good compromise given that we are resource and time constrained. The full kernel implementation can be found in the Appendix, Listing A.2.

We will be tuning the previously mentioned parameters for the following matrix sizes, with the group size being set to 4:

- 4096×4096
- 8192×8192

- 16384×16384

5.1.10 Attention

With recent advances in the field of deep learning and natural language processing, giving rise to large language models, the attention mechanism has become a crucial component in the field. The attention kernel is a fundamental building block that helps the models to focus on the most important parts of the input. For this reason, achieving higher performance in the attention kernel can dramatically reduce the cost and latency and improve the overall efficiency of the model.

The attention kernel we have decided to tune is the one provided by the FlagGems library. This library provides general operators (kernel functions) that can be used to accelerate LLM training and inference. Their kernel implementations have been optimized and also use Triton's auto-tuner to find the best configuration [40].

To get a better understanding, let us look at what the attention kernel achieves on a high level.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \tag{5.1}$$

In Equation 5.1, we can see that the attention kernel takes as input three matrices: Q, K, and V. These matrices are the query, key, and value matrices, respectively. The d in the denominator is the dimension of the query and key matrices (a scaling factor). The kernel then computes the attention weights by taking the dot product of the query and key matrices and applying a softmax function. The result is then multiplied with the value matrix to get the final attention output.

While the equation above is for a single head, the kernel computes the multiheaded attention in parallel. The number of heads is usually the same as the dimension of the query and key matrices. Furthermore, the kernel encompasses some key features that make the operation more efficient. Firstly, the computations are done in blocks to improve cache efficiency. Secondly, because of the exponential nature of the softmax function, it is prone to underflowing and overflowing, especially in low-precision arithmetic. To avoid this, the kernel implements the softmax with the log-sum-exp trick to make it more stable [41]. Thirdly, it allows for the usage of FP8 value tensors in order to reduce memory usage and improve performance. Nevertheless, for this experiment, we will be using the default data type of float16. Finally, the kernel also supports pre-loading the value matrix if the `PRE_LOAD_V` flag is set to true.

The FlagGems implementation of the kernel offers a couple of options to choose from. You can choose to run either casual or non-casual attention, as well as use masked or unmasked attention. In order to reduce the number of parameters and keep things simple, we will be using the non-causal and unmasked attention for tuning the kernel. The kernel implementation can be found in the Appendix, Listing A.3.

To keep the comparison fair, we will be tuning the exact same parameters as the ones used in the FlagGems implementation. FlagGems uses a YAML file to store the possible configurations for each parameter and uses Triton's auto-tuner to find the best configuration. This means they are effectively using the brute force search strategy to find the best configuration out of the ones provided.

We will be tuning the following parameters:

- `BLOCK_M`: The block size for tiled computation.
- `BLOCK_N`: The block size for tiled computation.
- `PRE_LOAD_V`: Whether to pre-load the value matrix.

Furthermore, we will be tuning the parameters from Section 5.1.3 (`num_warps` and `num_stages`) as well.

The input data sizes we will be tuning for are the following:

- Batch size: 20
- Sequence length: 1024
- Number of heads: 32
- Head dimension: 64

With this, the query, key, and value tensors will be of the shape $(20, 1024, 32, 64)$. In this case the search space will be, like most of the other kernels, large, and the compilation time will be longer due to the complexity of the kernel and the size of the input data. For this reason we will be using the genetic algorithm strategy (Section 5.1.6) to find a more performant configuration.

5.1.11 Conv2d

Another widely used operation in the field of deep learning is the 2D convolution. It is especially prevalent in the field of computer vision, where it used to process images by applying different filters to them. These filters can be used to extract features from the image, such as edges, corners, and other visual patterns. Convolutional neural networks (CNNs) use this operation to extract features from the input data; only here there are no predefined filters, but rather they are learned during the training process. Meaning that the conv2d operation is applied a large number of times to the input data, making optimizations to this kernel very valuable.

To better understand the conv2d operation, let us look at the mathematical equation

$$O[x, y, m] = \sum_{c=1}^{C} \sum_{i=1}^{k_H} \sum_{j=1}^{k_W} I[sx + i - p, sy + j - p, c] \cdot F[i, j, c, m] \tag{5.2}$$

In Equation 5.2, we can see that the conv2d operation takes as input a feature map I and a filter F. The input feature map is of dimensions (H, W, C), where H and W are the height and width of the feature map, and C is the number of channels. The filter is of dimensions (k_H, k_W, C, M), where k_H and k_W are the height and width of the filter, C is the number of channels, and M is the number of filters. The symbol p denotes the padding and s is the stride of the operation. This $O[x, y, m]$ denotes that for position (x, y) in the output feature map and for filter m, the operation is a sum of the element-wise product between the filter and the feature map patch.

Once again, we will use the implementation provided by the FlagGems library as a baseline (see Section 5.1.10 for more details). Conv2d is a very well-known operation, and a lot of research has been done to optimize it. In this case, the implementation provided by FlagGems uses a block-based approach, where the feature map is divided into blocks for the purpose of better parallelization and memory coalescing. The kernel implementation can be found in Listing A.4.

Given that FlagGems uses Triton's auto-tuner, we will be tuning the same parameters as the ones used in the FlagGems implementation. These are the following:

- `BLOCK_NI_HO_WO`: Determines how many output elements are computed in parallel.
- `BLOCK_CI`: Determines how many channels are processed in parallel.
- `BLOCK_CO`: Determines how many output channels are computed in parallel.

The names of these parameters are not self-explanatory and are chosen because of the way the algorithm is implemented. For this reason, we will briefly explain the names of the parameters. NI is the batch dimension, HO is the output height, WO is the output width, CI is the input channel dimension, and CO is the output channel dimension.

Apart from these, we will also be tuning the parameters from Section 5.1.3. The input data sizes we will be tuning for are the following:

- Batch size: 16
- Input height: 112
- Input width: 112
- Input channels: 128
- Output channels: 256
- Kernel height: 3
- Kernel width: 3
- Stride: 1
- Padding: 1

Once again, we will be using the genetic algorithm strategy (Section 5.1.6) to find a more performant configuration.

5.1.12 Strategy comparison

In this section, we will describe how we will be comparing the performance of different available strategies. For the full list of strategies, please refer to Section 5.1.6. In order to compare them, we need to have a common ground, a kernel to tune, and a fixed set of input parameters. For this experiment, we have chosen to compare different tuning strategies on the GROUP GEMM kernel (see Section 5.1.9). We have chosen this kernel due to the small search space and the fact that it is a good representative of the performance gains that can be achieved. This way, we can compare every strategy to the results obtained from the brute force search strategy.

We will be tuning the exact same parameters as described in Section 5.1.9. Furthermore, we will also have the same set of input parameters, except for the matrix size, which will be fixed

to 4096 $\times$ 4096. Many of the strategies offer setting custom parameters, such as population size in the case of the genetic algorithm. However, in order to keep the comparison fair and manageable for this report, we will be using the default parameters for every strategy. This is because we want to give an idea of how the strategies can perform and improve tuning and not go in depth into the details of each strategy.

One more thing to note is that, as mentioned in Section 4.1.2, Triton compiles the kernel when it's first invoked. This can be problematic when comparing the strategies, as the strategy that invokes a certain kernel configuration first will have a disadvantage due to the additional compilation time. For this reason, we will be using an environment variable called `TRITON_ALWAYS_COMPILE` that will force Triton to compile the kernel for every invocation. This way, we can ensure that each strategy is treated equally, and the only difference is how the strategies explore the search space.

The search space we will be exploring is the cartesian product of the following parameters:

- `BLOCK_SIZE_M`: [16, 32, 64, 128, 256, 512]
- `BLOCK_SIZE_N`: [16, 32, 64, 128, 256, 512]
- `BLOCK_SIZE_K`: [16, 32, 64, 128, 256, 512]
- `NUM_SMs`: [60, 72, 82, 90, 105]

This gives us a total of 1080 configurations to explore.

As with the previous experiments, we will be using NVIDIA's software and hardware configuration described in Section 5.1.2 and Section 5.1.1. We will compare the strategies based on the total time taken to tune the kernel and the tuned kernel performance.

5.2 Results

5.2.1 Matrix Multiplication

In this section, we present and discuss the results for the matrix multiplication experiments described in Section 5.1.8.

Results

Starting with the NVIDIA GPU (Figure 5.2), we can see that the Kernel Tuner tuned kernel outperforms the baseline implementation in all cases. The speedup ranges from 1.02x to 1.11x for the 16384 $\times$ 16384 matrix. Comparing the relative performance to cuBLAS (Figure 5.4), we observe that in all cases apart from the 16384 $\times$ 16384 matrix, both the Kernel Tuner tuned kernel and the baseline implementation outperform cuBLAS by a few percent. The Kernel Tuner tuned kernel outperforms cuBLAS in all cases, with the largest speedup being 1.23x for the 8192 $\times$ 8192 matrix.

On the AMD side, the relative performance graph (Figure 5.3) shows that the Kernel Tuner tuned kernel outperforms the baseline implementation in all cases. The speedup ranges from 2.53x to 3.14x for the 4096 $\times$ 4096 matrix. Comparing the relative performance to hipBLAS (Figure 5.5), we can see that AMD Triton kernel is able to outperform hipBLAS for the smallest and the largest matrix sizes with speedups of 1.13x and 1.32x respectively. For the 16384 $\times$

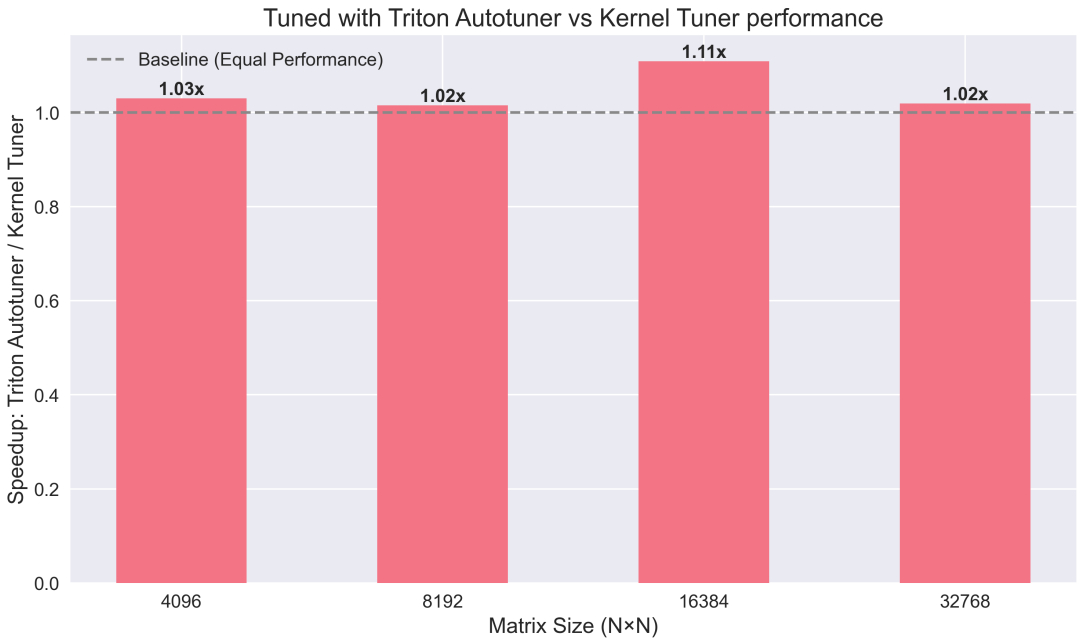

Figure 5.2: Relative performance comparison of matrix multiplication implementations on NVIDIA RTX 5000 Ada GPU, showing the speedup achieved by the Kernel Tuner tuned kernel over the baseline Triton implementation across different matrix sizes (4096x4096 to 32768x32768)

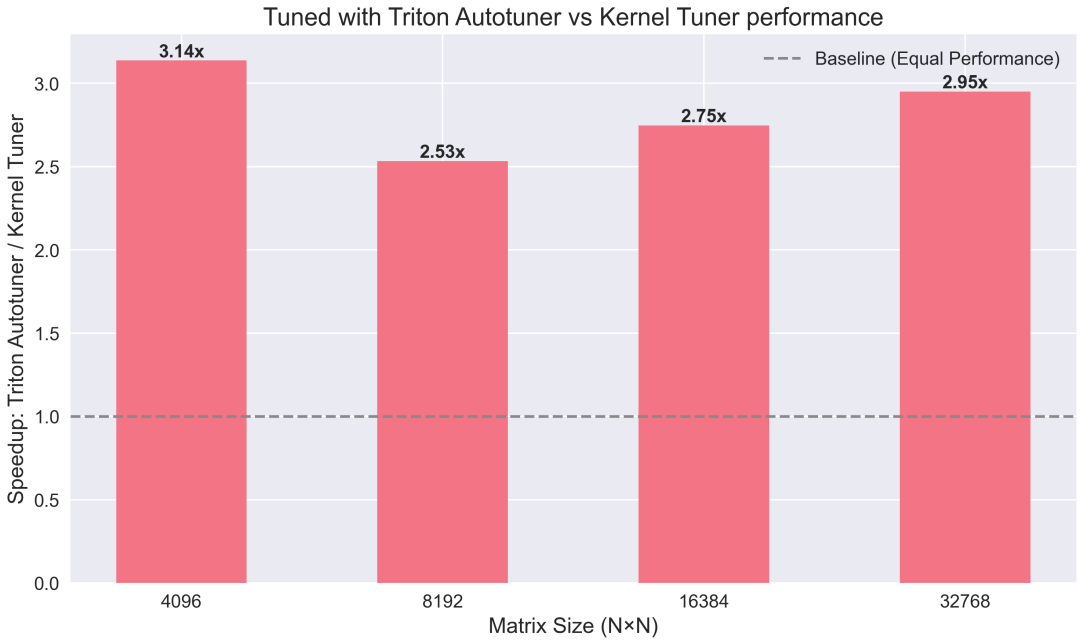

Figure 5.3: Relative performance comparison of matrix multiplication implementations on AMD Radeon Pro W7800 GPU, showing the speedup achieved by the Kernel Tuner tuned kernel over the baseline Triton implementation across different matrix sizes (4096x4096 to 32768x32768)

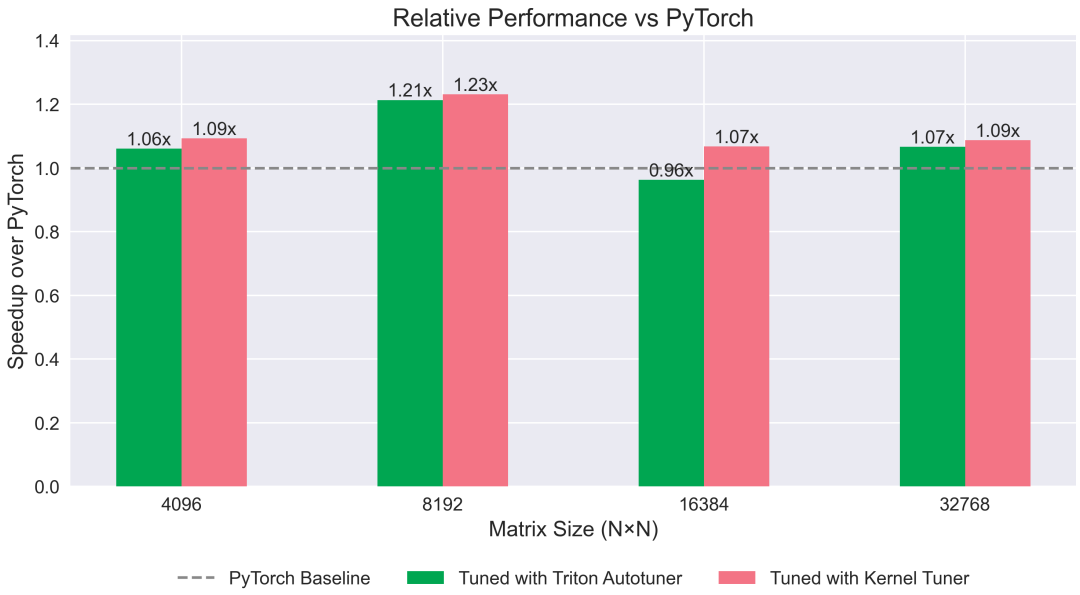

Figure 5.4: Performance comparison of matrix multiplication implementations on NVIDIA RTX 5000 Ada GPU, showing the relative speedup of both Triton implementations (baseline and Kernel Tuner tuned) compared to cuBLAS across different matrix sizes (4096x4096 to 32768x32768)

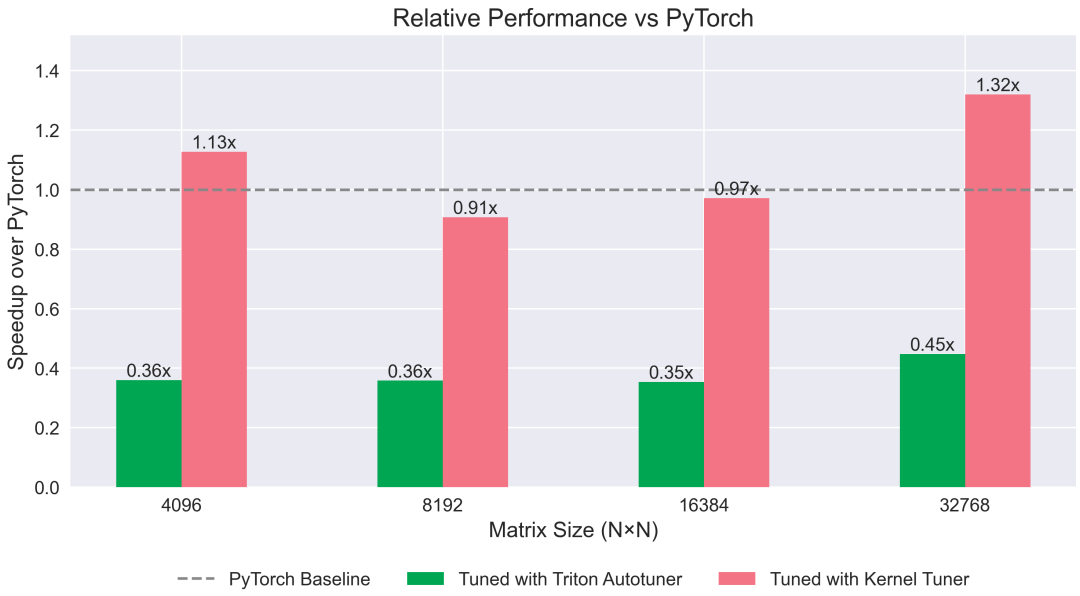

Figure 5.5: Performance comparison of matrix multiplication implementations on AMD Radeon Pro W7800 GPU, showing the relative speedup of both Triton implementations (baseline and Kernel Tuner tuned) compared to hipBLAS across different matrix sizes (4096x4096 to 32768x32768)

16384 matrix, and the 8192 $\times$ 8192 matrix, the tuned kernel comes close to hipBLAS, 0.91x for the 8192 $\times$ 8192 matrix and 0.95x for the 16384 $\times$ 16384 matrix.

Discussion

Regarding the NVIDIA GPU results, we observe that our tuned kernel was able to outperform the baseline implementation in all cases. That said, the margins were not as large as hoped for, with the speedups ranging from 1.02x to 1.11x. Furthermore, we observed that even the baseline implementation was able to outperform cuBLAS in all but one case. Initially, this indicated that finding a better configuration would be challenging, as outperforming cuBLAS is already difficult. Moreover, this suggests that the Triton team likely spent more time optimizing configurations for NVIDIA GPUs to showcase their auto-tuner's performance. It is for this reason that we decided to use the brute force search strategy to find the best configuration for the NVIDIA GPU, ensuring no potential configuration was missed. Because of this comprehensive search, the relatively small performance gains achieved are not surprising.

Interestingly, as mentioned before, the Kernel Tuner tuned kernel was able to outperform cuBLAS in all cases, with significantly larger margins ranging from 1.04x to 1.23x (Figure 5.4). We believe this can be attributed to several factors. Firstly, our kernel invocation did not need to find the best configuration before launch; it did not need to compute the optimal grid and block sizes dynamically. In comparison, when invoked, cuBLAS needs to select the appropriate kernel and determine the best grid/block sizes. Secondly, cuBLAS might implement additional checks or handle edge cases that add overhead but improve robustness. Finally, Triton's just-in-time compilation can generate a specifically optimized kernel for the exact problem size and hardware architecture at runtime, while cuBLAS must maintain compatibility across a wide range of use cases. The important takeaway is that Triton, if the kernel is written and tuned properly, can reach performance similar to vendor libraries, proving to be a viable alternative to its CUDA counterparts.

Given the impressive results on the NVIDIA GPU, especially for the 8192 $\times$ 8192 matrix, we decided to use NVIDIA's NSight Compute to profile both kernels for the 8192 $\times$ 8192 matrix. All of the reports generated by NSight Compute can be found in our GitHub repository. The key difference between the two kernels was the difference in memory throughput. Our Triton kernel had an average DRAM throughput of 0.63%, while cuBLAS had a much higher 21.5%. This suggests that the Triton kernel achieves much better data locality, heavily re-using the data from the faster shared memory. It is for this performance difference and the fact that cuBLAS has to select the best kernel beforehand that we believe that the performance gains we achieved were able to outperform cuBLAS by such a large margin.

The matrix multiplication results on AMD GPUs showed interesting outcomes. The AMD GPU managed to gain up to 3.14x and as low as 2.53x performance over the baseline implementation (Figure 5.3), highlighting the importance of tuning for the specific hardware architecture. Especially for AMD GPUs, which are not as widely used as their NVIDIA counterparts, testing different configurations is crucial to find the best-performing one, as many default configurations target NVIDIA hardware.

In comparison with hipBLAS, the highly optimized vendor library for matrix multiplication on AMD GPUs, our tuned kernel achieved 132% of its performance in the 32768 $\times$ 32768 case, and 113% in the 4096 $\times$ 4096 case (Figure 5.5). For the other two matrix sizes, it comes

close to hipBLAS but does not manage to outperform it. Compared to NVIDIA, AMD does not manage to outperform hipBLAS in all cases. We believe this is due to the fact that the AMD implementation of Triton is relatively new, and not as mature and well optimized as the NVIDIA implementation. In cases where AMD was able to outperform hipBLAS, same arguments can be made as for cuBLAS. Like cuBLAS, hipBLAS needs to select appropriate kernel, determine the grid and block sizes, and perform additional checks and optimizations.

Our experiments demonstrate that, even though hipBLAS is highly optimized, it is possible to achieve a similar or even better performance with a properly tuned Triton kernel. Furthermore, it shows that Triton-generated kernels can achieve comparable performance to vendor libraries while retaining flexibility and tunability for specific use cases. Moreover, we believe that hipBLAS performance could potentially be exceeded in the future for all matrix sizes as the AMD implementation of Triton matures.

5.2.2 Group GEMM

In this section, we present and discuss the results for the Group GEMM experiments described in Section 5.1.9.

Results

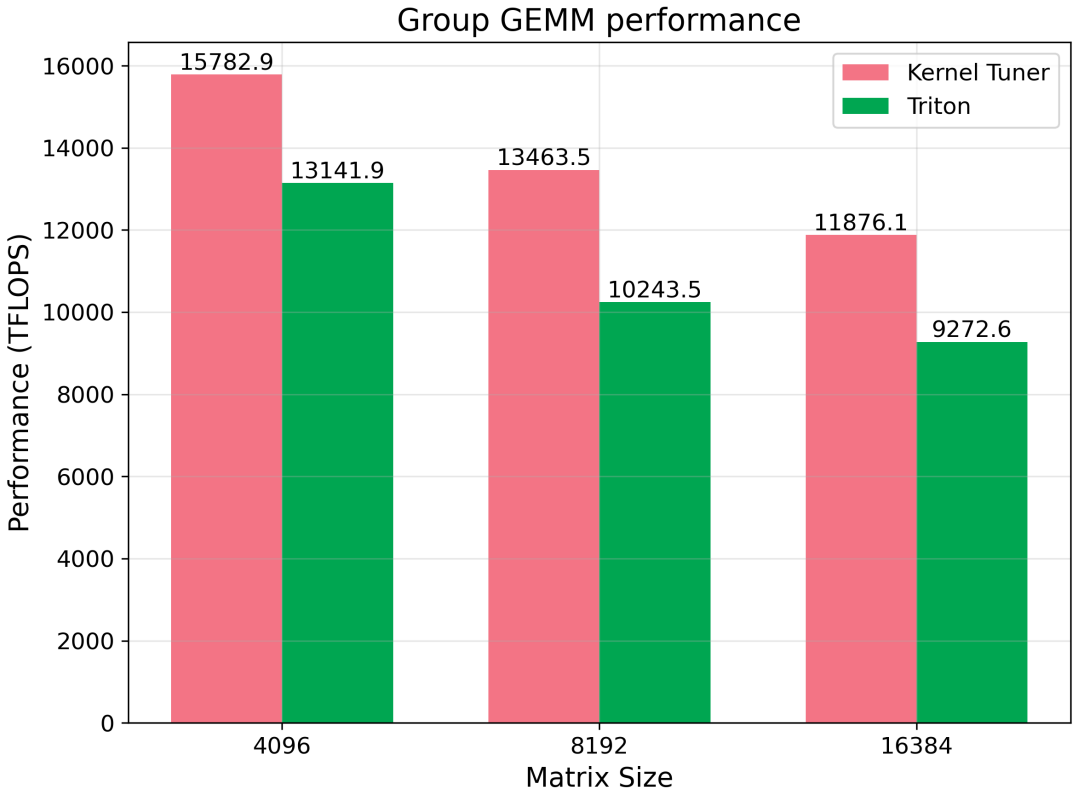

Figure 5.6: Performance comparison of Group GEMM implementations on NVIDIA RTX 5000 Ada GPU, showing the raw performance in TFLOPS for both Triton baseline and Kernel Tuner tuned implementations across different matrix sizes (4096x4096 to 16384x16384)

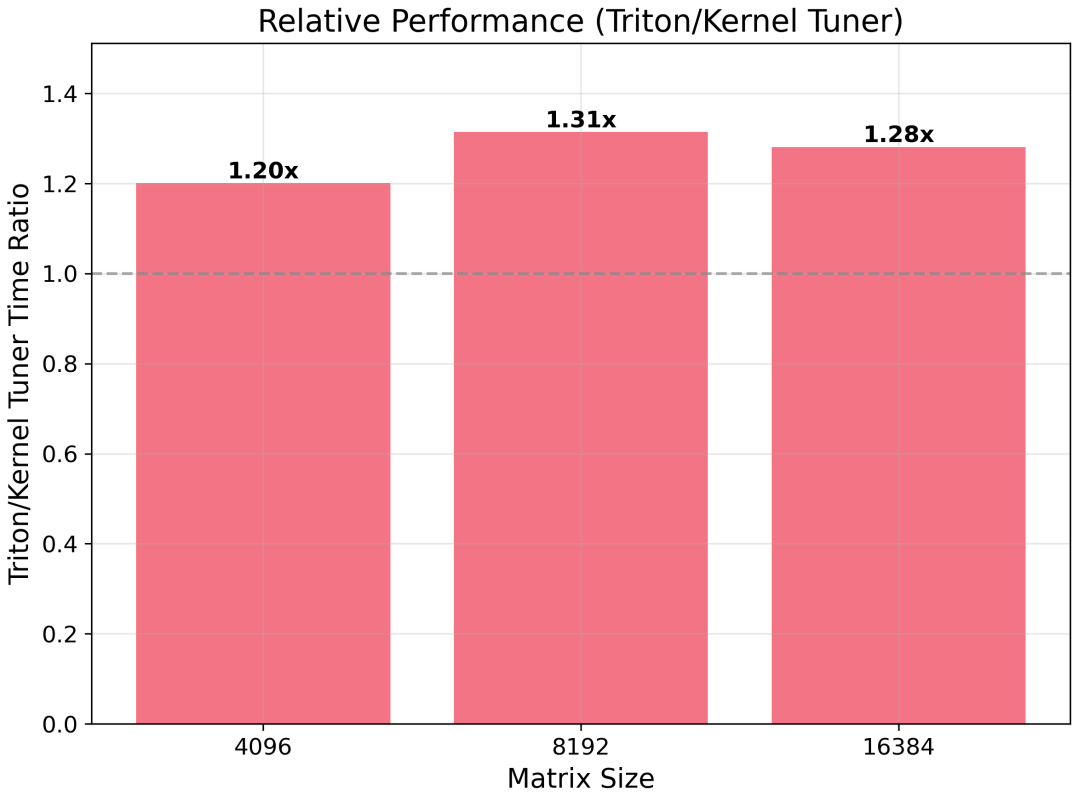

Figure 5.7: Relative performance comparison of Group GEMM implementations on NVIDIA RTX 5000 Ada GPU, showing the speedup achieved by the Kernel Tuner tuned kernel over the baseline Triton implementation across different matrix sizes (4096x4096 to 16384x16384)

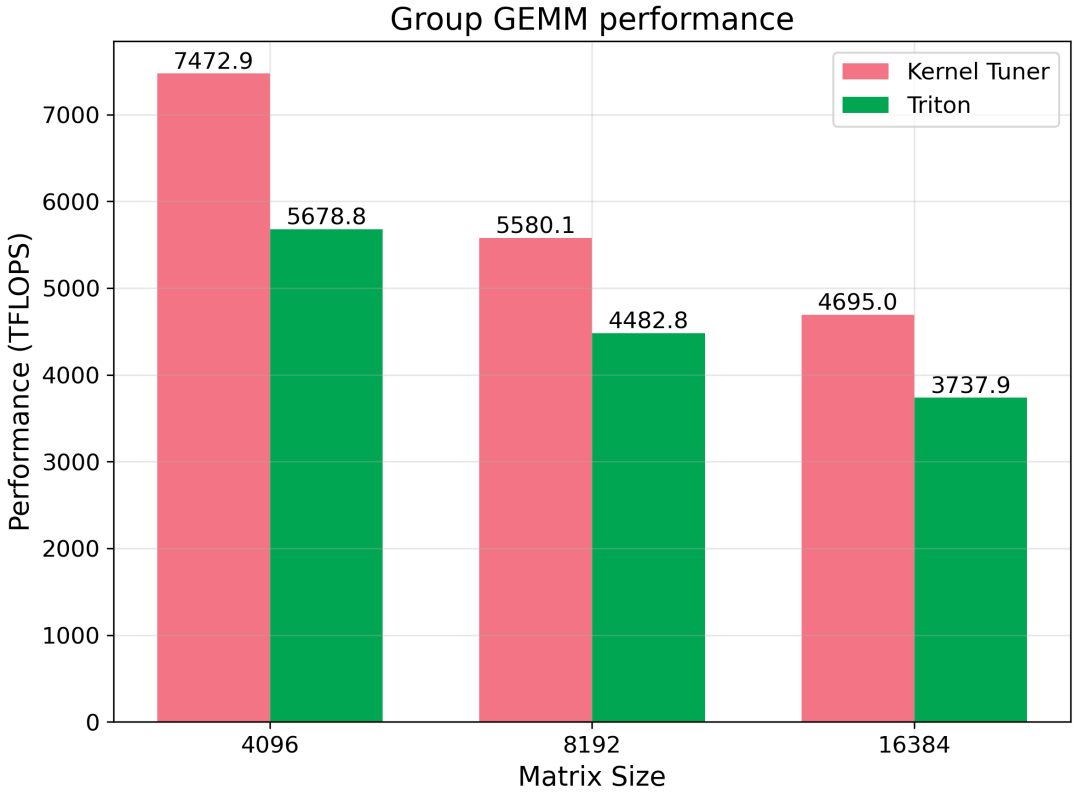

Figure 5.8: Performance comparison of Group GEMM implementations on AMD Radeon Pro W7800 GPU, showing the raw performance in TFLOPS for both Triton baseline and Kernel Tuner tuned implementations across different matrix sizes (4096x4096 to 16384x16384)

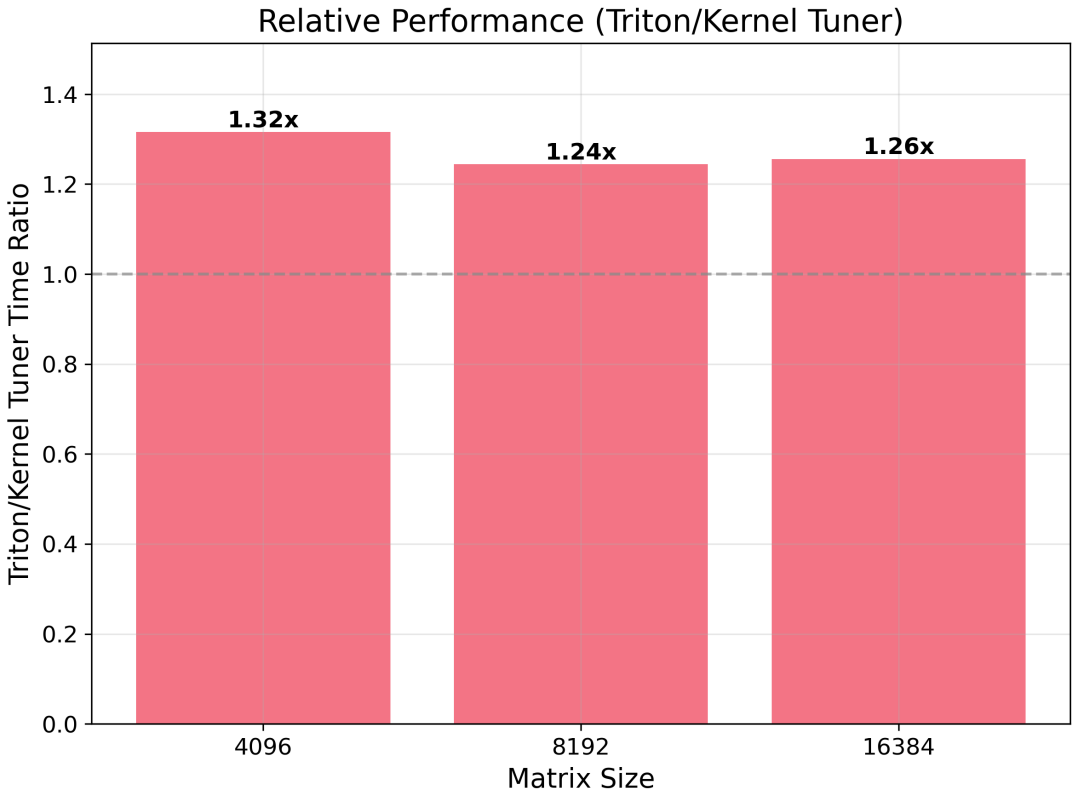

Figure 5.9: Relative performance comparison of Group GEMM implementations on AMD Radeon Pro W7800 GPU, showing the speedup achieved by the Kernel Tuner tuned kernel over the baseline Triton implementation across different matrix sizes (4096x4096 to 16384x16384)

The benchmarking results compare the performance of Group GEMM implementations using Triton versus Kernel Tuner across different matrix sizes (4096, 8192, and 16384).

The performance graph for NVIDIA (Figure 5.6) shows that the GPU performance drops as the matrix size increases. Nevertheless, it also shows that the Kernel Tuner tuned kernel outperforms the baseline implementation in all cases. The same can be said for the AMD GPU (Figure 5.8).

The relative performance graph for NVIDIA (Figure 5.7) shows the speedup achieved by the Kernel Tuner tuned kernel over the baseline configuration. From it, we can see that the Kernel Tuner tuned kernel consistently outperforms the baseline configuration, with the largest speedup being 1.31x for the 8192×8192 matrix. Relative performance for AMD is shown in Figure 5.9.

Discussion

From the Group GEMM results (Figure 5.7, Figure 5.9), we can observe that the tuned kernels managed to outperform the baseline configuration on both NVIDIA and AMD GPUs. While different block sizes can influence performance, we found that the main contributing factor in this case is the larger number of SMs specified in the tuned configuration via the `NUM_SM` parameter. As mentioned previously (Section 5.2.2), work is distributed across SMs. GPUs with more SMs benefit from higher performance if `NUM_SM` is sufficiently large and work is evenly distributed. For this reason, the best choice is usually a multiple of the actual number of SMs in the GPU. The official Triton example for Group GEMM suggests setting `NUM_SM` to match the number of SMs in the GPU. However, in our experiments, setting it to a multiple (e.g., 3 or 4 times) of the actual SM count proved more effective. This approach likely ensures better work distribution and higher utilization of each SM. This showcases a shortcoming of the Triton auto-tuner, where they could not include a wider range of SM counts in the search space, as it would cause the brute force search to take too long to complete. On the other hand, using one of the advanced strategies within Kernel Tuner, we can quickly find a good-performing configuration, given that minimizing the objective is directly related to the number of SMs.

5.2.3 Attention

In this section, we present the results for the attention experiments described in Section 5.1.10.

Results

Implementation	NVIDIA RTX 5000 Ada (ms)	AMD Radeon Pro W7800 (ms)
Kernel Tuner	1.301	64.69
FlagGems Baseline	1.302	65.38

Table 5.1: Performance comparison of attention kernel implementations. NVIDIA configuration: batch size 2, sequence length 128, 4 attention heads, and head dimension 64. AMD configuration: batch size 20, sequence length 1024, 32 attention heads, and head dimension 64. Values shown are average execution times in milliseconds.

Discussion

Looking at the results displayed in Table 5.1, we can see that we did not manage to find configurations that outperform the baseline FlagGems implementation by a large margin. The performance differences were minimal (1.01x for both GPUs) and could also be attributed to a measurement error. We believe the reason for this is the relatively small search space, or in other words limited amount of parameters that successfully compiled. While the search space seems to be large in the beginning, the actual number of kernels that can be compiled is very limited. Attention kernel is a complex kernel that takes in many different input arguments, and operating with large block sizes and grid sizes is simply not feasible. For this reason, most of the compiled kernels could not run due to a lack of shared memory that Triton requires for the kernel to run. This left us with a limited search space that was also mostly explored by the FlagGems baseline.

5.2.4 Conv2d

In this section, we present and discuss the results for the Conv2D experiments described in Section 5.1.11.

Results

Implementation	NVIDIA RTX 5000 Ada (ms)	AMD Radeon Pro W7800 (ms)
Kernel Tuner	7.15	21.87
FlagGems Baseline	10.26	22.12

Table 5.2: Performance comparison of Conv2D kernel implementations on different GPUs with input size 112x112, 128 input channels, 256 output channels, 3x3 kernel size, stride 1, padding 1, and batch size 16.

From the absolute performance table for NVIDIA (Table 5.2), we can see that the performance of the Kernel Tuner tuned kernel significantly outperforms the FlagGems-tuned baseline implementation. The performance increase in this case is 1.44x for the given input parameters. On the other hand, the AMD GPU did not achieve a large performance gain, barely outperforming the baseline implementation by 1.01x (Table 5.2).

Discussion

The Conv2D tuning experiments revealed interesting contrasts between NVIDIA and AMD hardware optimization, particularly highlighting the challenges in tuning kernels for AMD architectures. As shown in Table 5.2, the results demonstrate significantly different outcomes. On the NVIDIA side, Kernel Tuner found a configuration 1.44x faster than the baseline. However, on the AMD side, the performance gain was minimal at 1.01x. This slight improvement underscores a challenge often encountered when tuning kernels for AMD architectures: performance landscapes can be sparse "peakier", meaning there might be only one or two configurations performing significantly better than the baseline, making them harder to find [33]. In contrast, NVIDIA GPUs sometimes exhibit more gradual performance improvements across a wider range of configurations, potentially making tuning less sensitive.

5.2.5 Strategy Comparison

In this section, we present and discuss the results comparing different tuning strategies.

Results

Table 5.3: Strategy Performance Comparison Sorted by Tuning Time (Ascending)

Strategy	Tuning Time (s)	Performance (ms)
minimize	8.38	27.53
firefly_algorithm	65.81	13.69
random_sample	85.85	14.53
diff_evo	118.99	11.73
greedy_ils	133.23	11.72
pso	136.37	11.72
greedy_mls	137.46	11.72
ordered_greedy_mls	141.73	11.72
mls	144.15	11.74
bayesian_optimization	153.63	11.74
genetic_algorithm	180.05	11.73
basinhopping	235.50	11.72
brute_force	715.39	11.74

From Figure 5.10 and Table 5.3, we can see that most of the algorithms managed to find the same, best-performing configuration (around 11.72-11.74 ms). Furthermore, the majority managed to do so within a similar time frame of 118 to 150 seconds. Notably, differential evolution found the best-performing configuration the fastest among advanced strategies, taking only 118.99 seconds. On the other hand, the firefly algorithm offered a good trade-off between tuning time and performance, finding a configuration only around 17% slower (13.69 ms vs. 11.73 ms) than the best-performing one in just 65.81 seconds. Looking at it from a broader perspective, using alternative search strategies can be very beneficial, as they can find a good-performing (in this case, the best) configuration in a fraction of the time required by brute-force search. In our case, the best configuration was found by differential evolution in just 16.6% of the time taken by the brute-force search.

Discussion

The strategy comparison also yielded some interesting results showcasing the performance versus tuning time trade-off. It is at this exact trade-off where we see the most striking observation. We observed a substantial difference in tuning time between strategies that achieved similar performance. The brute-force approach, serving as a baseline and guaranteeing the best performance, took 715 seconds to tune the kernel. This tuning time ended up being 6 times longer than the differential evolution algorithm, which found an equally well-performing configuration in just 118 seconds. This demonstrates that search strategies can be very beneficial, especially in environments where tuning time is limited or search space is too large to be explored with brute force.

Looking at the specific group of strategies, we can observe that all of the evolutionary algorithms

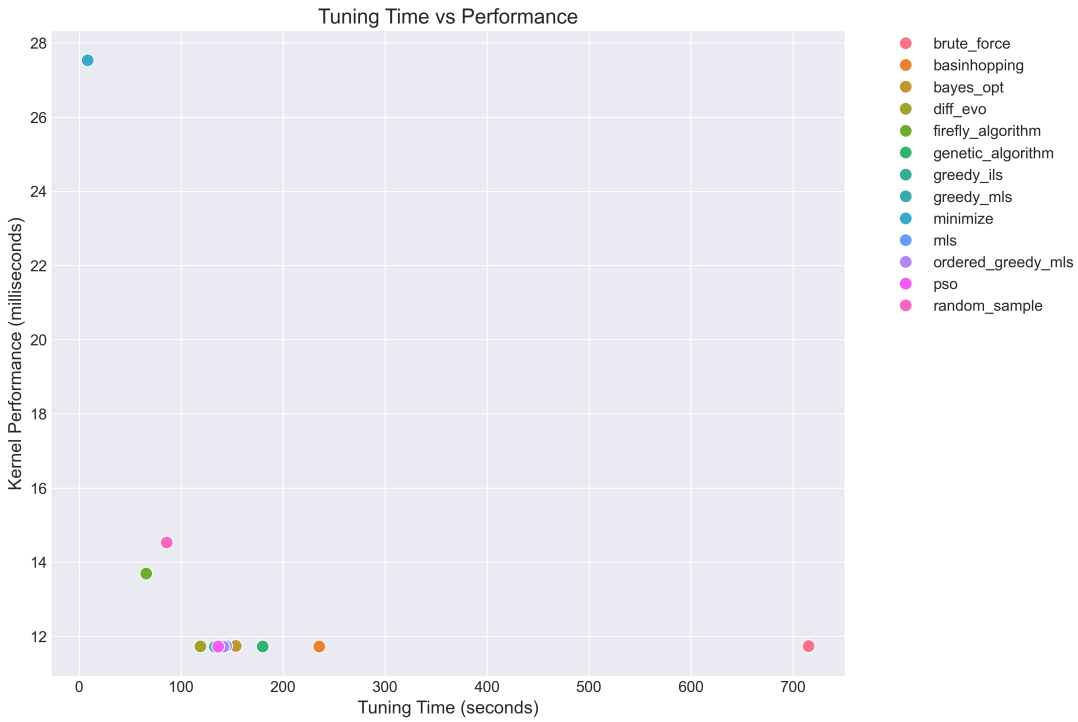

Figure 5.10: Comparison of different kernel tuning strategies for Group GEMM on NVIDIA RTX 5000 Ada GPU, showing the trade-off between tuning time (in seconds) and achieved performance (in milliseconds) for each strategy

performed exceptionally well. Differential evolution in particular stood out by finding the optimal solution faster than any other advanced strategy. Local search methods also performed well, finding the optimal solution in approximately 135-142 seconds. This usually indicates that the performance landscape has a relatively clear gradient toward optimal configurations. Which, in this case, is true, as we mentioned before; the group GEMM kernel is heavily influenced by the `NUM_SM` parameter. Interestingly, even simpler strategies like random sampling managed to find a reasonably good configuration within a limited time frame. However, random sampling is still luck-based, and there was still a noticeable gap in performance between it and the more sophisticated strategies.

Following the same line of thought, we want to mention the firefly algorithm as the best tuning time-to-performance trade-off. It managed to find a configuration that is only 16% slower than the best-performing one in just 65.81 seconds. This means that it required roughly half the tuning time of the other evolutionary algorithms and less than 10% of the time needed for brute-force search. For scenarios where tuning time is constrained, but some performance can be sacrificed, the firefly algorithm offers a compelling compromise.

In general, these results showcase important implications for practical kernel tuning scenarios. When finding the best-performing kernel is the most important aspect, then the brute-force search is the way to go. However, in most real-world scenarios where tuning time is limited, for example, in CI/CD pipelines, development environments, or when tuning for multiple hardware targets, evolutionary algorithms offer a better balance between performance and tuning time. Finally, the fact that most of the strategies converged to a similar performance figure suggests that the Group GEMM kernel may have a relatively well-defined global optimum that can be

discovered through various search approaches. For kernels with more complex performance landscapes featuring multiple local optima, the differences between strategies might be more pronounced.

Chapter 6

Conclusion

Throughout this work, we address the growing need for efficient and accessible GPU programming, caused by the increased demand in high-performance computing and, especially, AI-related workloads. While languages such as CUDA offer a great deal of control and performance, their complexity and the steep learning curve present a significant barrier for many developers.

OpenAI's Triton programming language was developed to address these issues while still maintaining cutting-edge performance. They have achieved this by embedding Triton into Python code, which provides a Python-like syntax, and they have closely integrated it with PyTorch, a popular framework used by many data scientists and AI developers. Furthermore, they have abstracted away the intricacies of the underlying hardware by completely removing the need for on-device memory management by automatically handling which data goes to which memory space.

Nevertheless, achieving high performance is not always straightforward and requires careful tuning of the configuration parameters, a task for which Triton's built-in autotuner offers limited capabilities.

The primary contribution of this work was the integration of Triton into the Kernel Tuner, a mature and feature-rich framework for autotuning GPU kernels. This integration allows Triton kernel developers to use a powerful and flexible toolset for optimizing their kernel. Moreover, it gives them access to a plethora of different optimization strategies that often speed up the tuning process by a large margin. Key steps in the implementation included:

- **Developing a dedicated Triton backend**: This involved adapting the existing Kernel Tuner architecture to support Triton's unique Just-In-Time (JIT) compilation model and leveraging PyTorch for memory management between the host and device.
- **Applying parameters to Triton kernels**: We used a novel approach using Abstract Syntax Trees (ASTs) to dynamically apply tuning parameters to Python function-based Triton kernels. This allowed us to overcome some of the limitations posed by string templating in combination with Triton's JIT.
- **Managing Triton dependencies**: We developed a system for automatically including the dependencies used by Triton kernels, such as other Triton kernels and Triton library imports.

Following the implementation, we conducted an extensive evaluation across various kernels, inputs, and GPU architectures. To be specific, we evaluated the performance of tuned Matrix Multiplication, Group GEMM, Attention, and Conv2D kernels on both NVIDIA (RTX 5000 Ada) and AMD (Radeon Pro W7800) GPUs. The results consistently demonstrated the value of our integration:

- **Improved performance**: Kernels tuned with Kernel Tuner were able to, in most cases, significantly outperform those tuned with the native autotuner. Speedups were observed across most of the tested kernels, with some of the notable ones being a 3.14x improvement for the GEMM kernel on the AMD and a 1.44x speedup for the Conv2D kernel on the NVIDIA GPU.
- **Competitiveness with Vendor Libraries**: Remarkably, the tuned Triton kernels were able to, in most cases, outperform the vendor libraries provided by NVIDIA and AMD (cuBLAS and hipBLAS, respectively). This shows that Triton is a viable and more flexible alternative to the vendor libraries, if the kernel is tuned beforehand.
- **Hardware-Specific Tuning**: Our results also underscore the critical importance of tuning for the specific hardware architecture. The performance gains were specifically noticeable on AMD's GPU, where the default or baseline configurations were often more suitable for the NVIDIA GPU (because of the wider adoption). Furthermore, AMD's GPUs tend to have fewer configurations that compare exceptionally well compared to NVIDIA, where there are a handful of configurations that are significantly faster. In this case, Kernel Tuner's ability to efficiently explore wider search spaces proved to be crucial for unlocking the full performance potential of AMD's hardware.
- **Efficiency of Search Strategies**: The comparison between the different search strategies available in Kernel Tuner showed the importance of using a non-brute force approach. Advanced space exploration strategies like Differential Evolution and other evolutionary algorithms consistently found the optimal or near-optimal configuration in a fraction of the time (up to 6x times faster) that it takes the default brute force approach. This demonstrates the practical benefits for time-constrained environments or problems with prohibitively large search spaces. Moreover, strategies like Firefly presented a great balance between the tuning time and the final performance of the kernel, achieving 84% of the performance of the best configuration within 10% of the total tuning time required by the brute force approach.

In conclusion, this work successfully added support for tuning Triton with Kernel Tuner, bridging the gap between Triton's ease of use and the need for a more sophisticated tuning framework. By providing access to advanced search strategies and a robust mechanism for parameter application, this integration empowers developers to more effectively harness the power of GPUs using Triton, achieving performance levels competitive with established, low-level approaches and vendor-specific libraries. The findings validate the potential of combining high-level GPU programming abstractions like Triton with powerful, generic autotuning frameworks like Kernel Tuner, ultimately making high-performance GPU computing more accessible and efficient for a broader range of applications and developers. Future work could focus on refining the AMD tuning process and exploring a wider range of kernels and applications. Furthermore, new search strategies could be added to Kernel Tuner, for example, an LLM guiding the search process. The codebase for this work is available at the Kernel Tuner fork for the Triton Kernel Tuner support and at the Master's Thesis repository for the rest of the work.

Appendix A

Appendix

A.1 Matrix Multiplication Implementation

```
def matmul_kernel(
    # Pointers to matrices
    a_ptr, b_ptr, c_ptr,
    # Matrix dimensions
    M, N, K,
    # The stride variables represent how much to increase the ptr by when moving by 1
    # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
    # by to get the element one row down (A has M rows).
    stride_am, stride_ak,  #
    stride_bk, stride_bn,  #
    stride_cm, stride_cn,
    # Meta-parameters
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,  #
    GROUP_SIZE_M: tl.constexpr,  #
    ACTIVATION: tl.constexpr  #
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    # -----------------------------------------------------------
    # Map program ids `pid` to the block of C it should compute.
    # This is done in a grouped ordering to promote L2 data reuse.
    # See above `L2 Cache Optimizations` section for details.
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m

    # -----------------------------------------------------------
    # Create pointers for the first blocks of A and B.
```

```
    # We will advance this pointer as we move in the K direction
    # and accumulate
    # `a_ptrs` is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K] pointers
    # `b_ptrs` is a block of [BLOCK_SIZE_K, BLOCK_SIZE_N] pointers
    # See above `Pointer Arithmetic` section for details
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] *
   stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] *
   stride_bn)

    # -----------------------------------------------------------
    # Iterate to compute a block of the C matrix.
    # We accumulate into a `[BLOCK_SIZE_M, BLOCK_SIZE_N]` block
    # of fp32 values for higher accuracy.
    # `accumulator` will be converted back to fp16 after the loop.
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.
   float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Load the next block of A and B, generate a mask by checking
   the K dimension.
        # If it is out of bounds, set it to 0.
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K,
    other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K,
    other=0.0)
        # We accumulate along the K dimension.
        accumulator = tl.dot(a, b, accumulator)
        # Advance the ptrs to the next K block.
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    # You can fuse arbitrary activation functions here
    # while the accumulator is still in FP32!
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)
    c = accumulator.to(tl.float16)

    # -----------------------------------------------------------
    # Write back the block of the output matrix C with masks.
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[
   None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)


# We can fuse `leaky_relu` by providing it as an `ACTIVATION` meta-
   parameter in `matmul_kernel`.
@triton.jit
def leaky_relu(x):
    return tl.where(x >= 0, x, 0.01 * x)
```

Listing A.1: Triton MatMul Kernel implementation

A.2 Group GEMM implementation

```
def grouped_matmul_kernel(
    # device tensor of matrices pointers
    group_a_ptrs,
    group_b_ptrs,
    group_c_ptrs,
    # device tensor of gemm sizes. its shape is [group_size, 3]
    # dim 0 is group_size, dim 1 is the values of <M, N, K> of each gemm
    group_gemm_sizes,
    # device tensor of leading dimension sizes. its shape is [group_size, 3]
    # dim 0 is group_size, dim 1 is the values of <lda, ldb, ldc> of each gemm
    g_lds,
    # number of gemms
    group_size,
    # number of virtual SM
    NUM_SM: tl.constexpr,
    # tile sizes
    BLOCK_SIZE_M: tl.constexpr,
    BLOCK_SIZE_N: tl.constexpr,
    BLOCK_SIZE_K: tl.constexpr,
):
    tile_idx = tl.program_id(0)
    last_problem_end = 0
    for g in range(group_size):
        # get the gemm size of the current problem
        gm = tl.load(group_gemm_sizes + g * 3)
        gn = tl.load(group_gemm_sizes + g * 3 + 1)
        gk = tl.load(group_gemm_sizes + g * 3 + 2)
        num_m_tiles = tl.cdiv(gm, BLOCK_SIZE_M)
        num_n_tiles = tl.cdiv(gn, BLOCK_SIZE_N)
        num_tiles = num_m_tiles * num_n_tiles
        # iterate through the tiles in the current gemm problem
        while (tile_idx >= last_problem_end and tile_idx < last_problem_end + num_tiles):
            # pick up a tile from the current gemm problem
            k = gk
            lda = tl.load(g_lds + g * 3)
            ldb = tl.load(g_lds + g * 3 + 1)
            ldc = tl.load(g_lds + g * 3 + 2)
            a_ptr = tl.load(group_a_ptrs + g).to(tl.pointer_type(tl.float16))
            b_ptr = tl.load(group_b_ptrs + g).to(tl.pointer_type(tl.float16))
            c_ptr = tl.load(group_c_ptrs + g).to(tl.pointer_type(tl.float16))
            # figure out tile coordinates
            tile_idx_in_gemm = tile_idx - last_problem_end
            tile_m_idx = tile_idx_in_gemm // num_n_tiles
            tile_n_idx = tile_idx_in_gemm % num_n_tiles

            # do regular gemm here
            offs_am = tile_m_idx * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
            offs_bn = tile_n_idx * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
```

```
            offs_k = tl.arange(0, BLOCK_SIZE_K)
            a_ptrs = a_ptr + offs_am[:, None] * lda + offs_k[None, :]
            b_ptrs = b_ptr + offs_k[:, None] * ldb + offs_bn[None, :]
            accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=
    tl.float32)
            for kk in range(0, tl.cdiv(k, BLOCK_SIZE_K)):
                # hint to Triton compiler to do proper loop pipelining
                tl.multiple_of(a_ptrs, [16, 16])
                tl.multiple_of(b_ptrs, [16, 16])
                # assume full tile for now
                a = tl.load(a_ptrs)
                b = tl.load(b_ptrs)
                accumulator += tl.dot(a, b)
                a_ptrs += BLOCK_SIZE_K
                b_ptrs += BLOCK_SIZE_K * ldb
            c = accumulator.to(tl.float16)

            offs_cm = tile_m_idx * BLOCK_SIZE_M + tl.arange(0,
    BLOCK_SIZE_M)
            offs_cn = tile_n_idx * BLOCK_SIZE_N + tl.arange(0,
    BLOCK_SIZE_N)
            c_ptrs = c_ptr + ldc * offs_cm[:, None] + offs_cn[None, :]

            # assumes full tile for now
            tl.store(c_ptrs, c)

            # go to the next tile by advancing NUM_SM
            tile_idx += NUM_SM

        # get ready to go to the next gemm problem
        last_problem_end = last_problem_end + num_tiles
```

Listing A.2: Triton Group GEMM Kernel implementation

A.3 Attention implementation

```
@triton.jit
def _attn_fwd_inner(
    acc,
    l_i,
    m_i,
    q,  #
    K_block_ptr,
    V_block_ptr,  #
    mask_block_ptr,  #
    stride_k_seqlen,
    stride_v_seqlen,
    stride_attn_mask_kv_seqlen,  #
    start_m,
    qk_scale,  #
    q_load_mask,
    BLOCK_M: tl.constexpr,
    HEAD_DIM: tl.constexpr,
    BLOCK_N: tl.constexpr,  #
    STAGE: tl.constexpr,
    offs_m: tl.constexpr,
    offs_n: tl.constexpr,  #
```

```
    KV_CTX: tl.constexpr,
    fp8_v: tl.constexpr,
    HAS_ATTN_MASK: tl.constexpr,
    PRE_LOAD_V: tl.constexpr,
):
    # range of values handled by this stage
    if STAGE == 1:
        lo, hi = 0, start_m * BLOCK_M
    elif STAGE == 2:
        lo, hi = start_m * BLOCK_M, (start_m + 1) * BLOCK_M
    # causal = False
    else:
        lo, hi = 0, KV_CTX

    K_block_ptr += lo * stride_k_seqlen
    V_block_ptr += lo * stride_v_seqlen
    kv_load_mask = lo + offs_n < KV_CTX
    if HAS_ATTN_MASK:
        mask_block_ptr += lo * stride_attn_mask_kv_seqlen

    LOG2E: tl.constexpr = 1.44269504

    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        # start_n = tl.multiple_of(start_n, BLOCK_N)
        # -- compute qk ----
        k = tl.load(K_block_ptr, mask=kv_load_mask[None, :], other=0.0)
        if PRE_LOAD_V:
            v = tl.load(V_block_ptr, mask=kv_load_mask[:, None], other
    =0.0)

        qk = tl.dot(q, k, allow_tf32=False)
        # qk = qk.to(tl.float32)

        if HAS_ATTN_MASK:
            attn_mask = tl.load(
                mask_block_ptr,
                mask=q_load_mask[:, None] & kv_load_mask[None, :],
                other=0.0,
            )

        if STAGE == 2:
            mask = offs_m[:, None] >= (start_n + offs_n[None, :])

            if HAS_ATTN_MASK:
                qk = qk * qk_scale + attn_mask
                qk *= LOG2E
                qk = qk + tl.where(mask, 0, -1.0e6)
            else:
                qk = qk * qk_scale * LOG2E + tl.where(mask, 0, -1.0e6)

            m_ij = tl.maximum(m_i, tl.max(qk, 1))
            qk -= m_ij[:, None]
        else:
            m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
            if HAS_ATTN_MASK:
                qk = qk * qk_scale + attn_mask
```

```
                qk *= LOG2E
                qk = qk - m_ij[:, None]
            else:
                qk = qk * qk_scale * LOG2E - m_ij[:, None]
        p = tl.math.exp2(qk)
        l_ij = tl.sum(p, 1)
        # -- update m_i and l_i
        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        # -- update output accumulator --
        acc = acc * alpha[:, None]
        # update acc
        if not PRE_LOAD_V:
            v = tl.load(V_block_ptr, mask=kv_load_mask[:, None], other
   =0.0)
        if fp8_v:
            p = p.to(tl.float8e5)
        else:
            p = p.to(q.dtype)
        p = p.to(v.dtype)
        acc = tl.dot(p, v, acc, allow_tf32=False)
        # update m_i and l_i
        m_i = m_ij

        K_block_ptr += BLOCK_N * stride_k_seqlen
        V_block_ptr += BLOCK_N * stride_v_seqlen

        if HAS_ATTN_MASK:
            mask_block_ptr += BLOCK_N * stride_attn_mask_kv_seqlen

    return acc, l_i, m_i

@triton.jit
def _attn_fwd(
    Q,
    K,
    V,
    attn_mask,
    sm_scale,
    Out,  #
    stride_q_batch,
    stride_q_head,
    stride_q_seqlen,
    stride_q_headsize,
    stride_k_batch,
    stride_k_head,
    stride_k_seqlen,
    stride_k_headsize,
    stride_v_batch,
    stride_v_head,
    stride_v_seqlen,
    stride_v_headsize,
    stride_attn_mask_batch,
    stride_attn_mask_head,
    stride_attn_mask_q_seqlen,
    stride_attn_mask_kv_seqlen,
    stride_o_batch,
```

```
    stride_o_head,
    stride_o_seqlen,
    stride_o_headsize,
    Z,
    q_numhead,
    kv_numhead,
    Q_CTX,
    KV_CTX,
    HEAD_DIM: tl.constexpr,
    BLOCK_M: tl.constexpr,
    BLOCK_N: tl.constexpr,
    STAGE: tl.constexpr,
    HAS_ATTN_MASK: tl.constexpr,
    PRE_LOAD_V: tl.constexpr,
):
    tl.static_assert(BLOCK_N <= HEAD_DIM)
    start_m = tl.program_id(0)
    off_hz = tl.program_id(1)
    batch_id = off_hz // q_numhead
    head_id = off_hz % q_numhead
    kv_head_id = off_hz % kv_numhead

    q_offset = (
        batch_id.to(tl.int64) * stride_q_batch + head_id.to(tl.int64) *
   stride_q_head
    )
    kv_offset = (
        batch_id.to(tl.int64) * stride_k_batch + kv_head_id.to(tl.int64)
    * stride_k_head
    )

    offs_headsize = tl.arange(0, HEAD_DIM)

    # initialize offsets
    offs_m = start_m * BLOCK_M + tl.arange(0, BLOCK_M)
    q_load_mask = offs_m < Q_CTX
    offs_n = tl.arange(0, BLOCK_N)

    Q_block_ptr = (
        Q
        + q_offset
        + offs_m[:, None] * stride_q_seqlen
        + offs_headsize[None, :] * stride_q_headsize
    )
    K_block_ptr = (
        K
        + kv_offset
        + offs_n[None, :] * stride_k_seqlen
        + offs_headsize[:, None] * stride_k_headsize
    )
    V_block_ptr = (
        V
        + kv_offset
        + offs_n[:, None] * stride_v_seqlen
        + offs_headsize[None, :] * stride_v_headsize
    )

```

```
    if HAS_ATTN_MASK:
        attn_mask_offset = (
            batch_id.to(tl.int64) * stride_attn_mask_batch
            + head_id.to(tl.int64) * stride_attn_mask_head
        )
        mask_block_ptr = (
            attn_mask
            + attn_mask_offset
            + offs_m[:, None] * stride_attn_mask_q_seqlen
            + offs_n[None, :] * stride_attn_mask_kv_seqlen
        )
    else:
        mask_block_ptr = None

    O_block_ptr = (
        Out
        + q_offset
        + offs_m[:, None] * stride_o_seqlen
        + offs_headsize[None, :] * stride_o_headsize
    )

    # initialize pointer to m and l
    m_i = tl.zeros([BLOCK_M], dtype=tl.float32) - float("inf")
    l_i = tl.zeros([BLOCK_M], dtype=tl.float32) + 1.0
    acc = tl.zeros([BLOCK_M, HEAD_DIM], dtype=tl.float32)
    # load scales
    qk_scale = sm_scale
    # qk_scale *= 1.44269504  # 1/log(2)
    # load q: it will stay in SRAM throughout
    q = tl.load(Q_block_ptr, mask=q_load_mask[:, None], other=0.0)
    # stage 1: off-band
    # For causal = True, STAGE = 3 and _attn_fwd_inner gets 1 as its
STAGE
    # For causal = False, STAGE = 1, and _attn_fwd_inner gets 3 as its
STAGE
    if STAGE & 1:
        acc, l_i, m_i = _attn_fwd_inner(
            acc,
            l_i,
            m_i,
            q,
            K_block_ptr,
            V_block_ptr,
            mask_block_ptr,
            stride_k_seqlen,
            stride_v_seqlen,
            stride_attn_mask_kv_seqlen,
            start_m,
            qk_scale,
            q_load_mask,
            BLOCK_M,
            HEAD_DIM,
            BLOCK_N,
            4 - STAGE,
            offs_m,
            offs_n,
            KV_CTX,
```

```
            V.dtype.element_ty == tl.float8e5,
            HAS_ATTN_MASK,
            PRE_LOAD_V,
        )
    # stage 2: on-band
    if STAGE & 2:
        # barrier makes it easier for compielr to schedule the
        # two loops independently
        acc, l_i, m_i = _attn_fwd_inner(
            acc,
            l_i,
            m_i,
            q,
            K_block_ptr,
            V_block_ptr,
            mask_block_ptr,
            stride_k_seqlen,
            stride_v_seqlen,
            stride_attn_mask_kv_seqlen,
            start_m,
            qk_scale,
            q_load_mask,
            BLOCK_M,
            HEAD_DIM,
            BLOCK_N,
            2,
            offs_m,
            offs_n,
            KV_CTX,
            V.dtype.element_ty == tl.float8e5,
            HAS_ATTN_MASK,
            PRE_LOAD_V,
        )
    # epilogue
    acc = acc / l_i[:, None]
    tl.store(O_block_ptr, acc.to(Out.type.element_ty), mask=q_load_mask
   [:, None])
```

Listing A.3: Triton Attention Kernel implementation

A.4 Conv2d implementation

```
@triton.jit
def conv2d_forward_kernel(
    input_pointer,
    weight_pointer,
    output_pointer,
    in_n,
    input_height,
    input_width,
    out_c,
    out_height,
    out_width,
    input_n_stride,
    input_c_stride,
    input_height_stride,
    input_width_stride,
```

```
    weight_n_stride,
    weight_c_stride,
    weight_height_stride,
    weight_width_stride,
    output_n_stride,
    output_c_stride,
    output_height_stride,
    output_width_stride,
    weight_c: tl.constexpr,
    weight_height: tl.constexpr,
    weight_width: tl.constexpr,
    stride_height: tl.constexpr,
    stride_width: tl.constexpr,
    padding_height: tl.constexpr,
    padding_width: tl.constexpr,
    dilation_height: tl.constexpr,
    dilation_width: tl.constexpr,
    groups: tl.constexpr,
    BLOCK_NI_HO_WO: tl.constexpr,
    BLOCK_CI: tl.constexpr,
    BLOCK_CO: tl.constexpr,
):
    pid_ni_ho_wo = tl.program_id(0)
    pid_co = tl.program_id(1)
    pid_group = tl.program_id(2)

    # caculate in_n out_height out_weight value in kernel
    ni_ho_wo_offset = pid_ni_ho_wo * BLOCK_NI_HO_WO + tl.arange(0, BLOCK_NI_HO_WO)
    ni_ho_offset = ni_ho_wo_offset // out_width
    in_n_point_value = ni_ho_offset // out_height
    output_height_point_value = ni_ho_offset % out_height
    output_width_point_value = ni_ho_wo_offset % out_width

    # Load the input and weight pointers. input and weight are of shape
    # [in_n, groups, in_c, input_height, input_width] and [groups, out_c, in_c, weight_height, weight_width]
    out_per_group_c = out_c // groups
    output_c_offset = pid_co * BLOCK_CO + tl.arange(0, BLOCK_CO)
    input_pointer += (
        input_n_stride * in_n_point_value + input_c_stride * pid_group * weight_c
    )[:, None]
    weight_pointer += (
        weight_n_stride * output_c_offset
        + weight_n_stride * pid_group * out_per_group_c
    )[None, :]

    accum = tl.zeros((BLOCK_NI_HO_WO, BLOCK_CO), dtype=tl.float32)
    BLOCK_CI_COUNT = (weight_c + BLOCK_CI - 1) // BLOCK_CI
    for hwc in range(weight_height * weight_width * BLOCK_CI_COUNT):
        c = (hwc % BLOCK_CI_COUNT) * BLOCK_CI
        hw = hwc // BLOCK_CI_COUNT
        h = hw // weight_width
        w = hw % weight_width

        input_c_offset = c + tl.arange(0, BLOCK_CI)
```

```
        input_height_offset = (
            h * dilation_height
            - padding_height
            + stride_height * output_height_point_value
        )
        input_width_offset = (
            w * dilation_width - padding_width + stride_width *
    output_width_point_value
        )

        curr_input_pointer = (
            input_pointer
            + (input_c_stride * input_c_offset)[None, :]
            + (input_height_stride * input_height_offset)[:, None]
            + (input_width_stride * input_width_offset)[:, None]
        )
        curr_weight_pointer = (
            weight_pointer
            + (weight_c_stride * input_c_offset)[:, None]
            + (weight_height_stride * h)
            + (weight_width_stride * w)
        )

        input_mask = (
            (in_n_point_value < in_n)[:, None]
            & (input_c_offset < weight_c)[None, :]
            & (0 <= input_height_offset)[:, None]
            & (input_height_offset < input_height)[:, None]
            & (0 <= input_width_offset)[:, None]
            & (input_width_offset < input_width)[:, None]
        )
        weight_mask = (input_c_offset < weight_c)[:, None] & (
            output_c_offset < out_per_group_c
        )[None, :]

        input_block = tl.load(curr_input_pointer, mask=input_mask)
        weight_block = tl.load(curr_weight_pointer, mask=weight_mask)

        accum += tl.dot(input_block, weight_block, allow_tf32=False)

    output_pointer += (
        (output_n_stride * in_n_point_value)[:, None]
        + (output_c_stride * (pid_group * out_per_group_c +
    output_c_offset))[None, :]
        + (output_height_stride * output_height_point_value)[:, None]
        + (output_width_stride * output_width_point_value)[:, None]
    )
    output_mask = (
        (in_n_point_value < in_n)[:, None]
        & (output_c_offset < out_per_group_c)[None, :]
        & (output_height_point_value < out_height)[:, None]
        & (output_width_point_value < out_width)[:, None]
    )

tl.store(output_pointer, accum, mask=output_mask)
```

Listing A.4: Triton Conv2d Kernel implementation

A.5 Additional Experimental Results

References

[1] Y. Wang et al. “Gunrock: a high-performance graph processing library on the gpu”. In: (2015), pp. 265–266. DOI: 10.1145/2688500.2688538.

[2] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. “Triton: an intermediate language and compiler for tiled neural network computations”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019, pp. 10–19.

[3] OpenAI. *Triton: An open-source GPU programming language*. Accessed: 2025-04-14. 2021. URL: https://openai.com/index/triton/.

[4] Ben van Werkhoven. “Kernel Tuner: A search-optimizing GPU code auto-tuner”. In: *Future Generation Computer Systems* 90 (2019), pp. 347–358. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18313359.

[5] NVIDIA. *cuBLAS: Basic Linear Algebra on NVIDIA GPUs*. Accessed: 2025-02-18. 2025. URL: https://developer.nvidia.com/cublas.

[6] NVIDIA. *cuFFT API Reference: CUDA Fast Fourier Transform library*. Accessed: 2025-02-18. 2025. URL: https://docs.nvidia.com/cuda/cufft/.

[7] NVIDIA. *NVIDIA cuDNN: CUDA Deep Neural Network library*. Accessed: 2025-02-18. 2025. URL: https://developer.nvidia.com/cudnn.

[8] NVIDIA. *cuSPARSE: CUDA Sparse Matrix library*. Accessed: 2025-02-18. 2025. URL: https://docs.nvidia.com/cuda/cusparse/.

[9] James Stanier and Des Watson. “Intermediate representations in imperative compilers: A survey”. In: *ACM Computing Surveys (CSUR)* 45.3 (2013), pp. 1–27.

[10] Clint Greene. *Developing Triton kernels on AMD GPUs*. https://rocm.blogs.amd.com/artificial-intelligence/triton/README.html. [Accessed 24-09-2024]. 2024.

[11] ENCCS. *Introduction to GPU programming models*. Accessed: 2025-02-18. 2024. URL: https://enccs.github.io/gpu-programming/5-intro-to-gpu-prog-models/.

[12] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016.

[13] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[14] Aaftab Munshi. “The opencl specification”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314.

[15] Paul Bauman et al. “Introduction to amd gpu programming with hip”. In: *Presentation at Oak Ridge National Laboratory. Online at: https://www. olcf. ornl. gov/calendar/intro-to-amd-gpu-programming-with-hip* (2019).

[16] Hercules Cardoso Da Silva, Flavia Pisani, and Edson Borin. “A comparative study of SYCL, OpenCL, and OpenMP”. In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE. 2016, pp. 61–66.

[17] H Carter Edwards and Christian R Trott. “Kokkos: Enabling performance portability across manycore architectures”. In: *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE. 2013, pp. 18–24.

[18] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A llvm-based python jit compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[19] ROYUD Nishino and Shohei Hido Crissman Loomis. “Cupy: A numpy-compatible library for nvidia gpu calculations”. In: *31st confernce on neural information processing systems* 151.7 (2017).

[20] NVIDIA. *CUDA Python: Python wrappers for CUDA driver and runtime APIs*. Accessed: 2025-02-18. 2025. URL: `https://developer.nvidia.com/cuda-python`.

[21] Andreas Kloeckner. *PyCUDA: Python wrapper for NVIDIA CUDA*. Accessed: 2025-02-18. 2025. URL: `https://pypi.org/project/pycuda/`.

[22] AMD. *HIP Python: Python bindings for HIP runtime and libraries*. Accessed: 2025-02-18. 2025. URL: `https://rocm.docs.amd.com/projects/hip-python/en/latest/`.

[23] Stijn Heldens and Ben van Werkhoven. “Kernel Launcher: C++ Library for Optimal-Performance Portable CUDA Applications”. In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2023, pp. 744–753.

[24] Amir H Ashouri et al. “A survey on compiler autotuning using machine learning”. In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–42.

[25] Ananta Tiwari et al. “Auto-tuning full applications: A case study”. In: *The International Journal of High Performance Computing Applications* 25.3 (2011), pp. 286–294.

[26] Jason Ansel et al. “Opentuner: An extensible framework for program autotuning”. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.

[27] Cedric Nugteren and Valeriu Codreanu. “CLTune: A generic auto-tuner for OpenCL kernels”. In: *2015 IEEE 9th International Symposium on Embedded Multicore/Manycore Systems-on-Chip*. IEEE. 2015, pp. 195–202.

[28] Ari Rasch and Sergei Gorlatch. “ATF: A generic auto-tuning framework”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pp. 3–4.

[29] Filip Petrovič and Jiří Filipovič. “Kernel tuning toolkit”. In: *SoftwareX* 22 (2023), p. 101385.

[30] Ignacio Laguna et al. “Gpumixer: Performance-driven floating-point tuning for gpu scientific applications”. In: *High Performance Computing: 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20, 2019, Proceedings 34*. Springer. 2019, pp. 227–246.

[31] *Kernel Float*. https://kerneltuner.github.io/kernel_float/. [Accessed 24-09-2024].

[32] Konstantinos Parasyris et al. “Scalable Tuning of (OpenMP) GPU Applications via Kernel Record and Replay”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023, pp. 1–14.

[33] Milo Lurati et al. *Artifact of the paper: Bringing auto-tuning to HIP: Analysis of tuning impact and difficulty on AMD and Nvidia GPUs*. June 2024. DOI: 10.5281/zenodo.11617999. URL: https://doi.org/10.5281/zenodo.11617999.

[34] URL: https://triton-lang.org/main/python-api/generated/triton.Config.html.

[35] F.J. Willemsen. *F.J. Willemsen's Blog*. Accessed: Feb 6, 2025. 2025. URL: https://fjwillemsen.com/.

[36] R. Storn and K. Price. “Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces”. In: *Journal of Global Optimization* 11 (1997), pp. 341–359. DOI: 10.1023/A:1008202821328. URL: https://doi.org/10.1023/A:1008202821328.

[37] Wikipedia. *Crossover (evolutionary algorithm)*. Accessed: 2025-02-18. 2025. URL: https://en.wikipedia.org/wiki/Crossover_(evolutionary_algorithm).

[38] Wikipedia. *Genetic Algorithm*. https://en.wikipedia.org/wiki/Genetic_algorithm. Accessed: 2025-02-14.

[39] Triton Team. *Matrix Multiplication — Triton Tutorials*. Accessed: 2025-02-06. 2025. URL: https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html.

[40] FlagOpen. *FlagGems*. https://github.com/FlagOpen/FlagGems. Accessed: 2025-02-13. 2025.

[41] Pierre Blanchard, Desmond J Higham, and Nicholas J Higham. “Accurately computing the log-sum-exp and softmax functions”. In: *IMA Journal of Numerical Analysis* 41.4 (Aug. 2020), pp. 2311–2330. ISSN: 0272-4979. DOI: 10.1093/imanum/draa038. eprint: https://academic.oup.com/imajna/article-pdf/41/4/2311/40758053/draa038.pdf. URL: https://doi.org/10.1093/imanum/draa038.