Abstract

In this thesis, we present KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a freely available tool to dynamically determine the values of kernel launch parameters of a CUDA kernel. We describe a technique for building a helper program, at the compile-time of a CUDA program, that is used at run-time to determine near-optimal kernel launch parameters for the kernels of that CUDA program. This technique leverages the MWP-CWP performance prediction model, runtime data parameters, and runtime hardware parameters to dynamically determine the launch parameters for each kernel invocation. This technique is implemented within the KLARAPTOR tool, utilizing the LLVM Pass Framework and NVIDIA Nsight Compute CLI profiler. We demonstrate the effectiveness of our approach through experimentation on the PolyBench benchmark suite of CUDA kernels.

Keywords: Performance estimation, Performance portability, CUDA, Program Parameters, Kernel Launch Parameters, LLVM Pass Framework, GPGPU

Summary for Lay Audience

KLARAPTOR is a tool designed to optimize the performance of GPU programs by dynamically determining the best kernel launch parameters for each kernel invocation. A kernel is a small piece of code that runs on a GPU and performs calculations in parallel, enabling faster processing times. The kernel launch parameters greatly impact the running time of a GPU program, and their optimal choice depends on various factors such as input data, hardware resources, and program parameters. To address this issue, KLARAPTOR leverages a two-step approach: (1) at compile-time, it determines formulas describing low-level performance metrics for each kernel and inserts them into the host code of a CUDA program; (2) at runtime, a helper program evaluates these formulas using the actual data and hardware parameters to determine the thread block configuration that minimizes the kernel's execution time. The effectiveness of KLARAPTOR is demonstrated through experimentation on a set of benchmarks consisting of CUDA kernels, showing that it can accurately predict near-optimal thread block configurations.

Co-Authorship Statement

KLARAPTOR is a joint project with Alexander Brandt, Marc Moreno Maza, Davood Mohajerani and Linxiao Wang. A preprint of this work is accessible as [10]. The contributions of the thesis author are highlighted in Section 1.1.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

In a Graphics Processing Unit (GPU) program, also called a kernel, the *kernel launch parameters* specify how threads are mapped to, and executed by, the hardware resources of the GPU. The choice of kernel launch parameters has a profound impact on the running time of a GPU program. Moreover, as we will show, the optimal choice of kernel launch parameters is not fixed for a given GPU kernel. Rather, it depends on the values of other *data* and *hardware parameters*. In this thesis we present a generic technique, and an implementation of that technique specialized to CUDA, for automatically and dynamically determining kernel launch parameters which minimize the running time of each kernel invocation in a CUDA program.

Generally, three types of parameters influence the performance of a parallel program: (i) *data parameters*, such as input data and its size; (ii) *hardware parameters*, such as cache capacity and number of available registers; and (iii) *program parameters*, such as granularity of tasks and the quantities that characterize how tasks are mapped to processors. These parameters are independent of the parallel program itself. Data and hardware parameters are independent from program parameters and are determined by the needs of the user and available hardware resources. Program parameters, however, are intimately related to data and hardware parameters. Therefore, it is crucial to determine values of program parameters that yield the best program performance for a given set of hardware and data parameter values.

The program parameters of interest to us in the CUDA programming model are thread block configurations. Their impact on performance is obvious considering, for example, that the memory access pattern of threads in a thread block depend on the thread block's dimension sizes. Similarly, a thread block configuration which minimizes the running time of a kernel invocation may not be optimal if the data sizes or target GPU device changes [39]. This emphasizes not only the impact of data and hardware parameters on program parameters, but also the need for performance portability. Unfortunately, in practice, thread block configurations are typically determined statically for a kernel through simple heuristics or trial and error, so

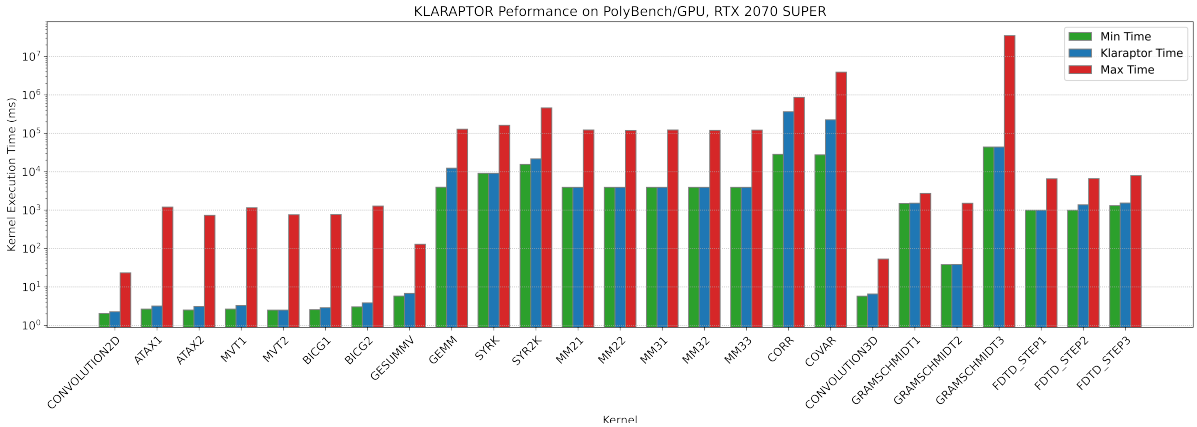

Figure 1.1: Comparing kernel execution time (log-scaled) for the thread block configuration chosen by KLARAPTOR versus the minimum and maximum times as determined by an exhaustive search over all possible configurations. Kernels are part of the `PolyBench/GPU` benchmark suite and executed on a RTX 2070 SUPER with a data size of $N = 8192$ (except convolution3d with $N = 512$)

long as the configurations are constrained to be multiples of 32 [2].

In most cases, the values of the data parameters are only given at runtime, making it difficult to determine optimal values of the program parameters at an earlier stage. On another hand, a bad choice of program parameters can have drastic consequences. Hence, it is crucial to be able to determine the optimal program parameters at runtime without much overhead added to the program execution. This is precisely the intention of the approach proposed here.

KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR) is a tool for automatically and dynamically determining the values of CUDA kernel launch parameters which optimize the kernel's performance, for each kernel invocation independently. That is to say, based on the actual data and target device of a kernel invocation.

KLARAPTOR consists of two main steps: (i) at compile-time, we determine formulas describing low-level performance metrics for each kernel and insert those formulas into the host code of a CUDA program; and (ii) at runtime, a *helper program* corresponding to a particular kernel evaluates those formulas using the actual data and hardware parameters to determine the thread block configuration that minimizes the kernel's execution time.

To minimize compilation and execution overheads, KLARAPTOR performs compile-time analysis and extrapolation of kernel performance on small data sizes at compile-time to predict kernel performance at runtime. Evaluation using the Polybench/GPU benchmark suite demonstrates that KLARAPTOR accurately predicts near-optimal thread block configuration see Figure 1.1.

Our key principle is based on an observation that, in most performance prediction mod-

els, *high-level performance metrics* (e.g. execution time, hardware occupancy) can be seen as decision trees or flowcharts based on *low-level performance metrics* (e.g. memory bandwidth, cache miss rate). These low-level metrics are themselves piece-wise rational functions (PWRFs) of program, data, and hardware parameters. If one could determine these PRFs, then it would be possible to estimate, for example, the running time of a program based on its program, data, and hardware parameters.

Unfortunately, exact formulas for low-level metrics are often not known, instead estimated through empirical measures or assumptions, or collected by profiling. This is a key challenge our technique addresses.

1.1 Contributions

The goal of this work is to determine values of program parameters which optimize a multithreaded program's performance. Towards that goal, the method by which such values are found must be receptive to changing data and changing hardware parameters. Our contributions encapsulate this requirement through the dynamic use of a rational program. Our specific contributions include:

(i) a technique for devising a mathematical expression in the form of a *rational program* to evaluate a performance metric from a set of program and data parameters;

(ii) KLARAPTOR, a tool implementing the rational program technique to dynamically optimize CUDA kernels by choosing optimal launch parameters; and

(iii) an empirical and comprehensive evaluation of our tool on kernels from the `Polybench/GPU` benchmark suite.

As the KLARAPTOR project is a joint research initiative, my contributions to the project are as follows:

1. Profiler Overhaul: I upgraded the profiler to support the latest GPU architectures, transitioning from compute capability 6.0 to 7.5. This update enables KLARAPTOR to target a broader range of NVIDIA GPUs and makes the tool more relevant to contemporary GPU computing environments.
2. Outlier Removal Process: I introduced a preprocessing step that incorporates an outlier removal algorithm based on quartile fencing. This method helps mitigate the impact of potential noise in the empirical data collected from ncu, stemming from factors such as DVFS and other variances in GPU performance.
3. Enhanced Integration of Rational Programs and MWP-CWP Model: I made improvements to the overall integration of the theory of rational programs and the MWP-CWP model.

1.2 Structure of the Thesis

The remainder of this thesis is organized as follows. In the Chapter 2 we provide a comprehensive introduction to the CUDA programming model, covering its core concepts and memory model. Additionally, we discuss the theorectical performance model, MWP-CWP, which is fundamental to our research. We introduce the theory and terminology behind *rational programs* and describe that technique which applies the idea of rational programs to the goal of dynamically selecting optimal program parameters in Chapter 3. Chapter 4 gives on overview of the KLARAPTOR tool which applies our technique to CUDA kernels and the general algorithm underlying our tool, that is, building and using a rational program to predict program performance. Then, Chapter 5 describes our specialization and implementation of this technique to CUDA programs, while our implementation is evaluated in Chapter 6. Lastly, we draw conclusions and explore future work in Chapter 7.

1.3 Related Works

The *Parallel Random Access Machine* (PRAM) model [36, 17], including models tailored to GPU code analysis such as TMM [26] and MCM [19] analyze the performance of parallel programs at an abstract level. More detailed GPU performance models are proposed such as MWP-CWP [20, 34], which estimates the execution time of GPU kernels based on the profiling information of the kernels.

In the context of improving CUDA program performance, other research groups have used techniques such as loop transformation [7], auto-tuning [18, 21, 33, 23], dynamic instrumentation [22], or a combination of the latter two [35]. Auto-tuning techniques have achieved great results in projects such as ATLAS [40], FFTW [15], and SPIRAL [32] in which multiple kernel versions are generated *off-line* and then applied and refined *on-line* once the runtime parameters are known. In contrast, our technique does not optimize the parallel code itself, only the program parameters controlling it.

Although much research has been devoted to compiler optimizations for kernel source code or PTX code, previous works such as [11] and [39] suggest that kernel launch parameters (i.e. thread block configurations) have a large impact on performance and must be considered as a target for optimization. In [25], the authors present an input-adaptive GPU code optimization framework G-ADAPT, which uses statistical learning to find a relation between the input sizes and the thread block sizes. At linking time, the framework predicts the best block size for a given input size using the linear model obtained from compile time. This approach only considers the total size of the thread blocks and not their configuration. Meanwhile, the authors

of [33] use a linear regression model to predict optimal thread block configurations (that is, dimension sizes and not just the total size). However, they assume kernel execution time scales linearly with data size. The authors in [24] have also developed a method determining the best thread block configuration, but similarly, they assume execution time scales linearly with data size. In [16], machine learning techniques are used in combination with auto-tuning to search for optimal configurations of OpenCL kernels, but their examples are limited to stencil computations. On the other hand, ISAAC [38], an auto-tuning framework, utilizes predictive modeling techniques and a regression model on input characteristics to generate optimized hardware and application-specific CUDA kernels from parameterized PTX code templates. However, its functionality is still restricted to matrix multiplication and convolution operations.

Chapter 2

Background

In this chapter, we begin by providing a comprehensive overview of the CUDA programming model and GPU microarchitecture, ensuring that readers have a solid understanding of the key concepts and components that define the CUDA ecosystem. Next, we introduce the MWP-CWP analytical model, which sheds light on the parallelism exhibited by Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP) in GPU architectures. This model plays a critical role in analyzing and optimizing the performance of CUDA kernels. Through a clear explanation of these fundamental concepts, this chapter sets the stage for a deeper exploration of KLARAPTOR and its application in enhancing the performance of CUDA programs.

2.1 CUDA

In this section, a concise introduction to Compute Unified Device Architecture (CUDA) is presented. CUDA, designed by NVIDIA, is a heterogeneous serial-parallel programming model[27] that capitalizes on the capabilities of GPUs for general-purpose computation. Primarily targeting C/C++ programmers, CUDA has revolutionized the field of high-performance computing by facilitating substantial performance enhancements across various domains, including scientific simulations, artificial intelligence, and more. This background information aims to provide the foundations for comprehending the fundamental concepts, techniques, and challenges associated with GPU-accelerated computation.[2]

2.1.1 Graphics Processing Unit (GPU)

A GPU is a specialized computational device designed for quick manipulation and alteration of memory, primarily to generate images for display on a screen. Initially, GPUs were mainly used for rendering 3D graphics in video games and other visually intensive applications. However,

their highly parallel structure and ability to process large amounts of data simultaneously have made them increasingly valuable for general-purpose computation tasks.

Within the realm of CUDA programming, a GPU serves as a powerful computational device capable of executing thousands of threads in parallel. This allows it to perform complex calculations and data processing at a much faster rate than traditional Central Processing Units (CPUs), which focus on executing a smaller number of threads rapidly. By understanding the architecture and capabilities of GPUs, developers can utilize the CUDA programming model effectively to harness their computational power for various applications beyond graphics rendering.

The relevance of GPUs to the CUDA programming model lies in their ability to significantly enhance the performance of parallel computation. CUDA exploits the inherent advantages of GPUs to facilitate general-purpose computation on these devices, effectively transforming them into potent computational tools beyond their conventional graphics rendering roles[2].

GPUs offer substantial advantages over CPUs in the context of instruction throughput and memory bandwidth, while maintaining comparable costs and power consumption. This is primarily due to the distinct design goals of GPUs and CPUs. Mentioned above, CPUs focus on executing a few tens of threads rapidly, while GPUs are engineered to excel at executing thousands of threads in parallel, compensating for slower single-thread performance to achieve greater overall throughput. Figure 2.1 shows the distribution of chip resources for a CPU versus a GPU.

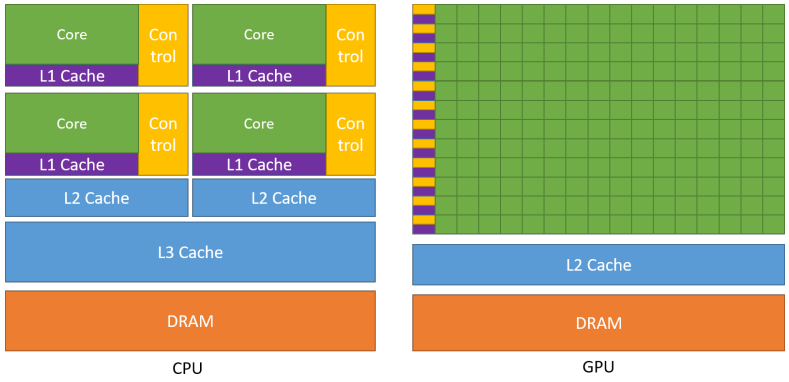

Figure 2.1: The GPU Devotes More Transistors to Data Processing

2.1.2 CUDA Programming Model

The CUDA parallel programming model addresses the challenges of scalable parallelism[28] while retaining accessibility for programmers familiar with languages such as C/C++. At

the core of this model lie three fundamental abstractions—hierarchy of thread groups, shared memories, and barrier synchronization—introduced as minimal language extensions. These abstractions enable a combination of fine-grained and coarse-grained parallelism, directing programmers to partition problems into independently solvable sub-problems and fostering cooperative parallel problem-solving within thread blocks. This methodology not only maintains language expressivity but also permits automatic scalability, as each thread block can be scheduled on any accessible GPU multiprocessor, with the runtime system overseeing the physical multiprocessor count[2]. Figure 2.2 shows an example of a CPU program vs a CUDA program.

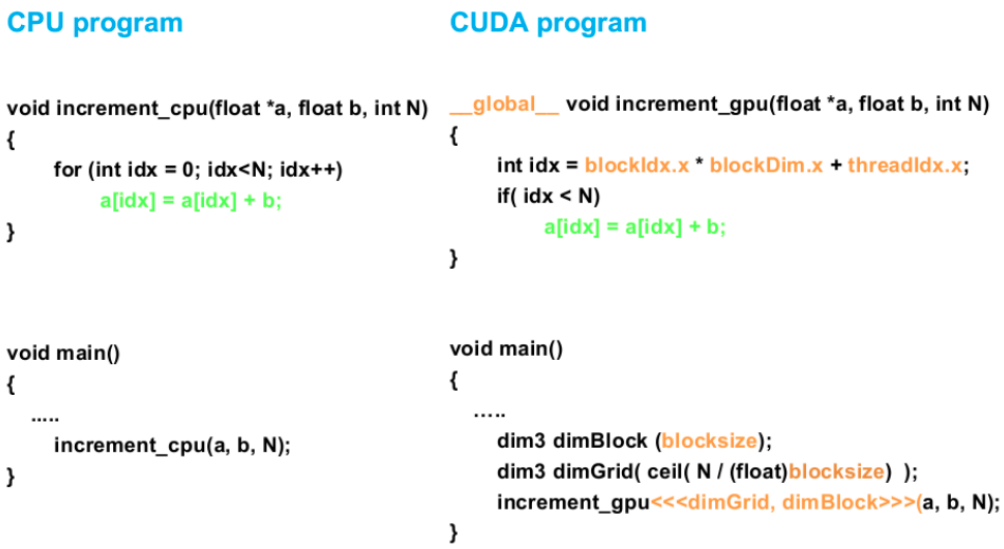

Figure 2.2: Example of a CPU program vs a CUDA program

Kernels

CUDA kernels are the fundamental building blocks of a CUDA program, representing the functions that execute on the GPU device (also referred to as the device) in parallel. In the context of CUDA, the CPU is referred to as the host, responsible for managing and orchestrating the execution of kernels on the device[28]. The device is typically composed of multiple streaming multiprocessors (SMs) that provide the parallel processing capability. A kernel is defined in the CUDA programming model using the `__global__` declaration specifier, which indicates that the function is callable from the host and executes on the device. The execution configuration of a kernel is specified using `<<<...>>>`, which determines the grid and block dimensions, thereby dictating the number of parallel threads and their organization during kernel execution[2]. This configuration allows programmers to efficiently utilize the available resources on the GPU and optimize the performance of their CUDA applications.

Threads

In the CUDA programming model, threads constitute the basic units of parallel execution and are organized within a hierarchical structure encompassing threads, thread blocks, and grids. Each thread is uniquely identified by its `threadIdx`, which is a multi-dimensional variable with up to three dimensions (x, y, and z)[2]. Thread limits are dictated by the GPU architecture, with the maximum number of threads per block typically capped at 1024. Threads are assembled into blocks, which are then organized into a grid. The `blockIdx` and `blockDim` variables identify the position of a specific block within the grid and the dimensions of each block, respectively. Thread and block dimensions can be one-, two-, or three-dimensional, offering flexibility in addressing the problem space in a manner that best accommodates the underlying data structure and computation. This hierarchical organization of threads and blocks enables efficient mapping of intricate, multi-dimensional problems onto the GPU's parallel processing resources, promoting optimal performance and resource utilization.

Thread blocks hold a pivotal role in the CUDA programming model[28], functioning as a means to organize threads that collaboratively work on a specific sub-problem. Within a thread block, threads can communicate and synchronize their operations via shared memory, facilitating efficient data exchange and cooperative problem-solving. It is important to note that thread blocks can be executed in any order, both concurrently and sequentially, across the available SMs on a GPU. This flexibility permits automatic scalability, allowing the compiled CUDA program to adapt to varying GPU architectures and multiprocessor counts[2]. Consequently, thread blocks enable efficient utilization of GPU resources while ensuring that applications can scale effectively on diverse hardware configurations.

Kernel Launch Parameters

When launching a CUDA kernel, the execution configuration specified plays a crucial role in determining the organization of threads and blocks for the kernel's execution on the GPU. The launch parameters within this syntax define the grid and block dimensions, which impact the performance and resource utilization of the application.

The launch parameters consist of two main components: the number of blocks per grid (`gridDim`) and the number of threads per block (`blockDim`). Both of these parameters can be specified as one-, two-, or three-dimensional structures, represented by dim3 variables in CUDA. The choice of dimensions depends on the problem space and the structure of the underlying data, with the aim of efficiently mapping the computation onto the GPU's resources.

For example, if a kernel is launched with the configuration `<<<gridDim, blockDim>>>`, it will create $gridDim.x * gridDim.y * gridDim.z$ blocks in the grid, with each block containing

blockDim.x∗*blockDim.y*∗*blockDim.z* threads. This results in a total of (*gridDim.x*∗*gridDim.y*∗ *gridDim.z*)∗(*blockDim.x*∗*blockDim.y*∗*blockDim.z*) threads executing the kernel concurrently.

Selecting optimal launch parameters is essential for achieving maximum performance and resource utilization on the GPU[3]. The choice of grid and block dimensions should consider factors such as the size of the problem, the hardware limitations of the GPU (e.g., maximum threads per block), and the level of parallelism required. Additionally, it is important to account for shared memory constraints, as larger block sizes may lead to increased shared memory requirements, potentially causing resource allocation issues or reduced parallelism[3].

By carefully choosing the launch parameters for a CUDA kernel, developers can create efficient, high-performance parallel applications that effectively exploit the capabilities of the underlying GPU hardware.[3]

2.1.3 CUDA by Example

In the following section, we shall explore a fundamental example of CUDA programming by implementing a vector addition program. This hands-on approach will enable readers to gain a deeper understanding of the various components and concepts involved in writing and executing CUDA code. This example follows a GitHub repository, If you would like to explore this tutorial further and experiment with the code, it is available at the following [31].

```cuda
#include <stdio.h>

// Size of array
#define N 1048576

// Kernel
__global__ void add_vectors(double *a, double *b, double *c)
{
        int id = blockDim.x * blockIdx.x + threadIdx.x;
        if(id < N) c[id] = a[id] + b[id];
}

// Main program
int main()
{
        // Number of bytes to allocate for N doubles
        size_t bytes = N*sizeof(double);

        // Allocate memory for arrays A, B, and C on host
        double *A = (double*)malloc(bytes);
        double *B = (double*)malloc(bytes);
        double *C = (double*)malloc(bytes);

        // Allocate memory for arrays d_A, d_B, and d_C on device
        double *d_A, *d_B, *d_C;
        cudaMalloc(&d_A, bytes);
        cudaMalloc(&d_B, bytes);
        cudaMalloc(&d_C, bytes);

        // Fill host arrays A and B
        for(int i=0; i<N; i++)
        {
                A[i] = 1.0;
                B[i] = 2.0;
        }

        // Copy data from host arrays A and B to device arrays d_A and d_B
        cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

        // Set execution configuration parameters
        //                thr_per_blk: number of CUDA threads per grid block
        //                blk_in_grid: number of blocks in grid
        int thr_per_blk = 256;
        int blk_in_grid = ceil( float(N) / thr_per_blk );

        // Launch kernel
        add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

        // Copy data from device array d_C to host array C
        cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

        // Free CPU memory
        free(A);
        free(B);
        free(C);

        // Free GPU memory
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);

        return 0;
}
```

We initiate the process at line 17, where the memory requirement for an array comprising N double-precision elements is determined. Subsequently, memory allocation for vectors A, B, and C occurs on the host in lines 20-22. Continuing forward, lines 25-28 allocate memory on the device for the same vectors. It is worth noting the prevalent naming convention for device variables is "d_" prefix to indicate device allocation.

The host input vectors A and B are copied to their device counterparts, d_A and d_B with `cudaMemcpy()` as seen in lines 38-39. To prepare for kernel launch, we set the kernel launch parameters in lines 44-45, defining the number of threads per block and the number of blocks per grid. The kernel is then launched in line 48, where the actual computation is executed.

Focusing on the `add_vectors` kernel function at line 7, a unique thread ID is identified in line 9 for each thread within the grid. The **`if`** statement on line 10 serves to prevent memory access beyond the array's bounds, which may occur when the number of grid threads is not a multiple of the number of threads per block. For instance, if N has an extra element, `blk_in_grid` would equal 4097 due to the `ceil` function in line 45, resulting in a total of $4097 * 256 = 1048832$ threads. Without the **`if`** statement, the final thread would attempt to access memory beyond the array's boundaries.

In conclusion, the device output vector d_C is copied to the host output vector C in line 51. Host memory is then freed in lines 54-56, followed by the release of device memory in lines 59-61.

2.1.4 CUDA Memory Model

The CUDA memory model is designed to accommodate the unique requirements of parallel programming on GPUs and consists of several distinct memory spaces. Host memory refers to the system memory allocated and managed by the CPU. In contrast, the GPU has its own memory spaces, including global, shared, and local memory[2].

Global memory, accessible by all threads within a kernel as well as the host, serves as the primary means for data storage and communication between the host and the device. The lifetime of data in global memory spans from the point of allocation to deallocation, which is explicitly managed by the programmer. Although global memory offers a large storage capacity, it has higher access latency compared to other memory spaces on the GPU[2].

Shared memory, as the name suggests, is a fast, on-chip memory space that can be shared among threads within the same thread block. This memory space enables efficient inter-thread communication and data exchange, making it particularly useful for problems that require threads to cooperate and share information while solving sub-problems. However, shared memory is limited in capacity, and its contents are only available for the duration of a thread

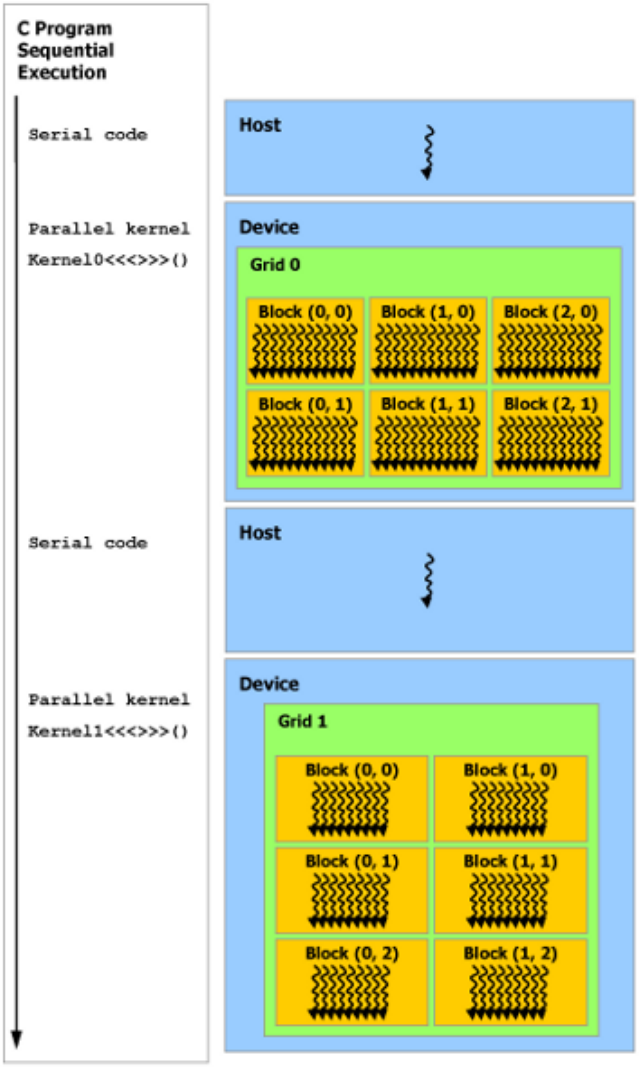

Figure 2.3: Heterogeneous Programming Model for CUDA

block's execution[2].

Local memory is private to each individual thread and is used for storing thread-specific data, such as function call frames and automatic variables. While local memory is accessible only by the thread that owns it, it allows threads to store temporary data without affecting other threads. Similar to global memory, local memory resides off-chip, and therefore, its access latency is higher than that of shared memory.

Refer to Figure 2.4 for a visual representation of the CUDA memory hierarchy.

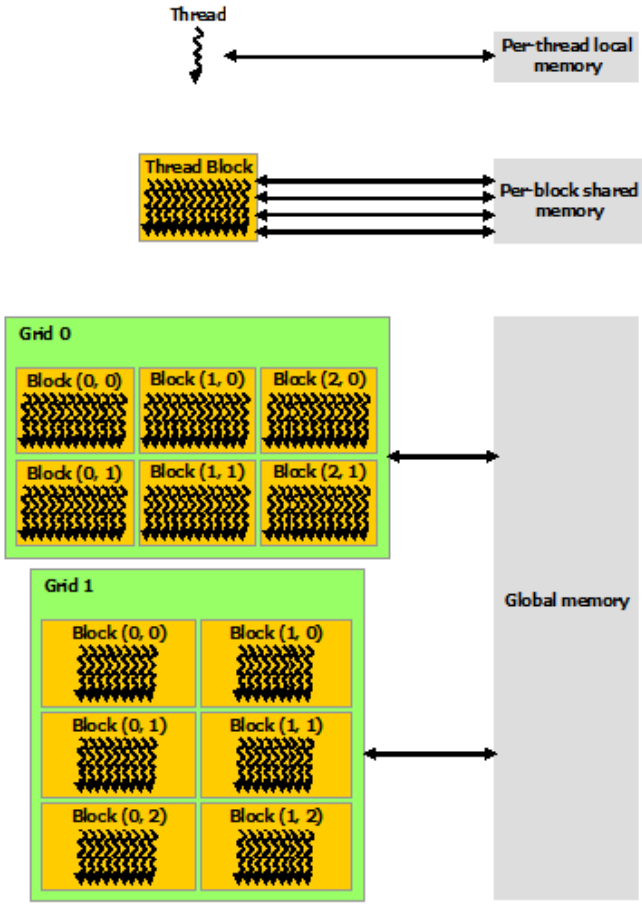

Figure 2.4: CUDA Memory Hierarchy

2.1.5 GPU Microarchitecture

We introduce the fundamental concepts of GPU microarchitecture, highlighting their relation to the CUDA programming model[2]. Understanding these core principles allows for a deeper comprehension of the efficient execution of parallel workloads on GPUs and informs the development of effective CUDA applications.

SIMT and SIMD

SIMT (Single Instruction, Multiple Thread) and SIMD (Single Instruction, Multiple Data) constitute vital parallel execution models that impact the performance of GPU architectures and their integration with the CUDA programming model. SIMT operates by executing identical instructions on distinct threads, while SIMD conducts the same operation on multiple data elements. NVIDIA GPUs utilize the SIMT paradigm, permitting threads to be organized into warps that execute instructions concurrently, thus optimizing resource usage and reducing thread management overhead[2]. The CUDA programming model corresponds directly to the

SIMT architecture, empowering developers to harness the inherent parallelism of GPUs while efficiently handling threads, memory, and synchronization to achieve exceptional parallel computing performance.

Streaming Multiprocessors (SMs)

Streaming Multiprocessors (SMs) are the primary building blocks of NVIDIA GPUs, acting as the core computational units that enable efficient parallel processing. Each SM houses multiple execution units, registers, and shared memory, facilitating the concurrent execution of a large number of threads. In CUDA, developers organize threads into blocks, which are then scheduled onto SMs[2]. This arrangement allows for effective management of thread parallelism, memory, and synchronization, ensuring optimal GPU resource utilization and high-performance parallel computing.

Warp and Warp Scheduling

Warps and warp scheduling are essential components of GPU architectures, particularly in the context of CUDA programming. In CUDA, a warp is a group of threads, typically 32, that execute simultaneously on a single streaming multiprocessor (SM). Threads within a warp share a common program counter and follow a simultaneous execution pattern, optimizing resource utilization and reducing thread management overhead. Warp scheduling, on the other hand, is instrumental in managing how warps execute on SMs. Different GPU architectures use various scheduling policies, determining the selection and instruction issuance for ready warps[2]. Efficient warp scheduling is crucial for maximizing GPU resource utilization and overall performance.

2.2 MWP-CWP

The MWP-CWP analytical model offers a novel approach to understanding the performance of GPU architectures by examining the parallelism exhibited by MWP and CWP. As multi-threaded architectures, GPUs allow multiple warps to be executed concurrently on a streaming multiprocessor (SM), effectively hiding the execution costs of these warps. The MWP-CWP model focuses on determining the maximum number of warps that can access memory simultaneously and the number of warps that an SM processor can execute during a memory warp waiting period. By carefully analyzing the relationship between MWP and CWP, this model provides valuable insights into the factors that govern execution time, revealing whether it is dominated by computation or memory access costs. Through a series of illustrative cases, we

will demonstrate the importance of sufficient warps and the intricate interplay between MWP and CWP in optimizing GPU performance.[20, 27]

(a) MWP (b) CWP

Figure 2.5: MWP-CWP model

2.2.1 $MWP \leq CWP$

In the case where CWP is greater than MWP, the application's performance exhibits a distinct characteristic: computation cycles are effectively hidden by memory waiting periods. This implies that the computational resources are kept busy while the memory accesses are being processed, leading to a more efficient use of the available resources. However, the overall performance of the application is dominated by the memory cycles, as the computational work is executed in parallel with memory access operations. In other words, the system exhibits a higher degree of parallelism in computation than in-memory operations. This implies that while the application effectively utilizes the available computational resources, it may face limitations in leveraging memory-level parallelism. This imbalance between CWP and MWP can result in the underutilization of memory bandwidth and potentially lead to performance bottlenecks.[20, 27]

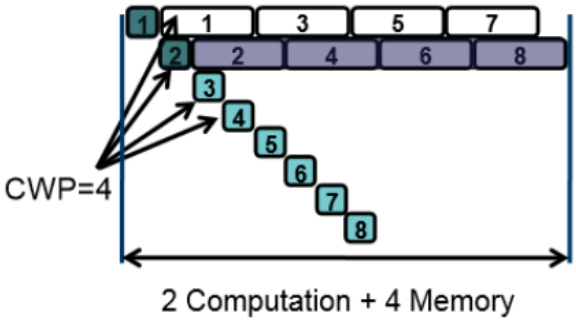

Figure 2.6: Computation cycles are concealed by memory waiting periods, resulting in the overall performance being predominantly dictated by memory cycles.

2.2.2 $MWP > CWP$

In the scenario where MWP is greater than CWP, the application's performance demonstrates a contrasting behavior: memory accesses are predominantly hidden due to the high MWP. This means that the memory subsystem is capable of handling multiple memory requests concurrently while the computation resources are being utilized, effectively masking memory latencies. As a result, the overall performance of the application is dominated by the computation cycles, since the memory accesses are efficiently processed in parallel with the computational work. To put it differently, the application's performance may be hindered due to an imbalance between memory and computational resources. This disparity indicates that the application has a higher degree of parallelism in memory access than in computation, which can potentially result in the underutilization of computational resources.[20, 27]

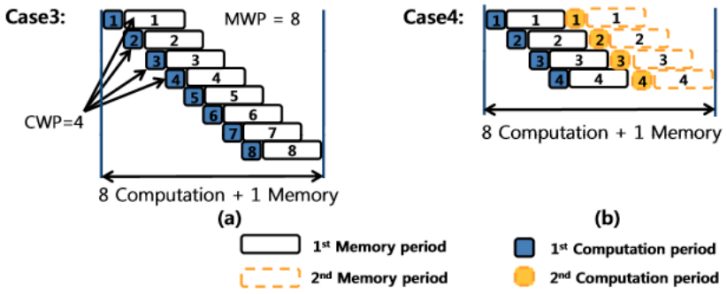

Figure 2.7: Memory accesses are largely concealed by high MWP, leading to the overall performance being primarily governed by computation cycles.

Chapter 3

Theoretical Foundations

Let $\mathcal{P}$ be a multithreaded program to be executed on a specific multiprocessor. Parameters influencing the performance of $\mathcal{P}$ include: (i) *data parameters*, specifying the size and possibly structural characteristics of the data, (ii) *hardware parameters*, specifying characteristics of hardware resources, and (iii) *program parameters*, specifying how work (e.g. threads) is mapped to hardware resources. By fixing the target architecture, the hardware parameters, say, $\boldsymbol{H} = (H_1, \ldots, H_h)$ become fixed and we can assume that the performance of $\mathcal{P}$ depends only on data parameters $\boldsymbol{D} = (D_1, \ldots, D_d)$ and program parameters $\boldsymbol{P} = \left(P_1, \ldots, P_p\right)$. For example, in programs targeting GPUs the parameters $\boldsymbol{D}$ are typically dimension sizes of data structures, like arrays, while $\boldsymbol{P}$ typically specifies the format of the grid and the format of the thread blocks.

Let $\mathcal{E}$ be a high-level performance metric (running time, memory consumption) for $\mathcal{P}$ that we want to optimize. More precisely, given the values of the data parameters $\boldsymbol{D}$, our goal is to find values of the program parameters $\boldsymbol{P}$ such that the execution of $\mathcal{P}$ optimizes $\mathcal{E}$. Performance prediction models attempt to estimate $\mathcal{E}$ from a combination of $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$, and some model- or platform-specific low-level metrics $\boldsymbol{L} = (L_1, \ldots, L_\ell)$ (memory bandwidth, cache miss rate, etc.). It is natural to assume that these low-level performance metrics are themselves functions of $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$. This is an obvious observation from models based on PRAM such as TMM [26] and MCM [19].

Therefore, we look to obtain values for these low- and high-level metrics given values for program, and data parameters. To address our optimization goal, we use the following strategy. At the compile-time of program $\mathcal{P}$, for each metric, we determine a mathematical formula expressing that metric as a function of the data and program parameters. This mathematical formula takes the form of what we call a *rational program*. At the runtime of $\mathcal{P}$, given specific values of $\boldsymbol{D}$ and a choice of $\boldsymbol{P}$, we can evaluate the rational program to obtain a value for each metric and thus for $\mathcal{E}$. Repeating this for all possible choices of $\boldsymbol{P}$ (assumed to be finite

in number) yields values of $\boldsymbol{P}$ optimizing $\mathcal{E}$. This strategy is detailed in Section 4.2, while Section 3.1 is dedicated to the notion of a rational program.

One could view a rational program as a computer program that, for input values $x_1, \ldots, x_n$, computes and returns a value $y = f(x_1, \ldots, x_n)$, where f is a function in the sense of a programming language, say C/C++. However, a rational program is more than that, due to the process we use to determine f.

3.1 Rational Programs

Let $X_1, \ldots, X_n, Y$ be pairwise different variables[1]. Let $\mathcal{S}$ be a sequence of three-address code (TAC [4]) instructions such that the variables occurring in $\mathcal{S}$ that are never assigned a value by an instruction of $\mathcal{S}$ are exactly $X_1, \ldots, X_n$.

Definition 1 *We say that the sequence $\mathcal{S}$ is a* rational program *in $X_1, \ldots, X_n$ evaluating Y if the following two conditions hold:*

1. *every arithmetic operation used in $\mathcal{S}$ is either an addition, a subtraction, a multiplication, a division, or a comparison (for equality or the natural order $\leq$) of two rational numbers numbers, in either fixed or arbitrary precision.*
2. *after specializing in $\mathcal{S}$ the variables $X_1, \ldots, X_n$ to rational numbers $x_1, \ldots, x_n$, the execution of the specialized sequence always terminates and the last executed instruction assigns a rational number to Y.*

It is worth noting that the above definition can easily be extended to include Euclidean division, the integer part operations floor and ceiling, and arithmetic over rational numbers. For Euclidean division one can write a rational program evaluating the quotient q of integer a by b, leaving the remainder r to be simply calculated as $a - qb$. Then, floor and ceiling can be computed via Euclidean division. Rational numbers and their associated arithmetic are easily implemented using only integer arithmetic. Therefore, by adding these operations to Definition 1, the class of rational programs does not change. We regard rational programs as such henceforth.

3.2 Rational Programs as Flowcharts

For any sequence $\mathcal{S}$ of computer program instructions, one can associate $\mathcal{S}$ with a *control flow graph* (CFG). In the CFG of $\mathcal{S}$, the nodes are the *basic blocks* of $\mathcal{S}$. Recall that a *flowchart* is

[1] Variables refer to both its mathematical meaning and programming language concept.

another graphic representation of a sequence of computer program instructions. In fact, CFGs can be seen as particular flowcharts.

If, in a given flowchart C, every arithmetic operation occurring in every (process or decision) node is either an addition, subtraction, multiplication, or comparison of integers in either fixed or arbitrary precision then C is the flowchart of a rational sequence of computer program instructions. Therefore, it is meaningful to depict rational programs using flowcharts, and vice versa, flowcharts as rational programs. For example, one could consider the metric of theoretical *hardware occupancy* as defined by NVIDIA. The following example details its definition, its depiction as a flowchart, and its dependency on program, data, and hardware parameters.

Example 1 The hardware *occupancy* is a measure of a program's effectiveness in using the Streaming Multiprocessors (SMs) of a GPU. It is calculated from a number of hardware parameters, namely:

- the maximum number $R_{\max}$ of registers per thread block,
- the maximum number $Z_{\max}$ of shared memory words per thread block,
- the maximum number $T_{\max}$ of threads per thread block,
- the maximum number $B_{\max}$ of thread blocks per SM and
- the maximum number $W_{\max}$ of warps per SM,

as well as low-level kernel-dependent performance metrics, namely:

- the number R of registers used per thread and
- the number Z of shared memory words used per thread block,

and a program parameter, namely the number T of threads per thread block. The occupancy of a CUDA kernel is defined as the ratio between the number of active warps per SM and the maximum number of warps per SM, namely:

$$W_{\text{active}}/W_{\max}, \quad \text{where} \quad W_{\text{active}} \leq \min\left(\lfloor B_{\text{active}} T/32 \rfloor, W_{\max}\right) \tag{3.1}$$

and B_{active} is given by the flowchart in Figure 3.1. This flowchart shows how one can derive a rational program computing B_{active} from $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. It follows from Formula (3.1) that W_{active} can also be computed by a rational program from $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. Finally, the same is true for the occupancy of a CUDA kernel using W_{active} and $W_{\max}$.

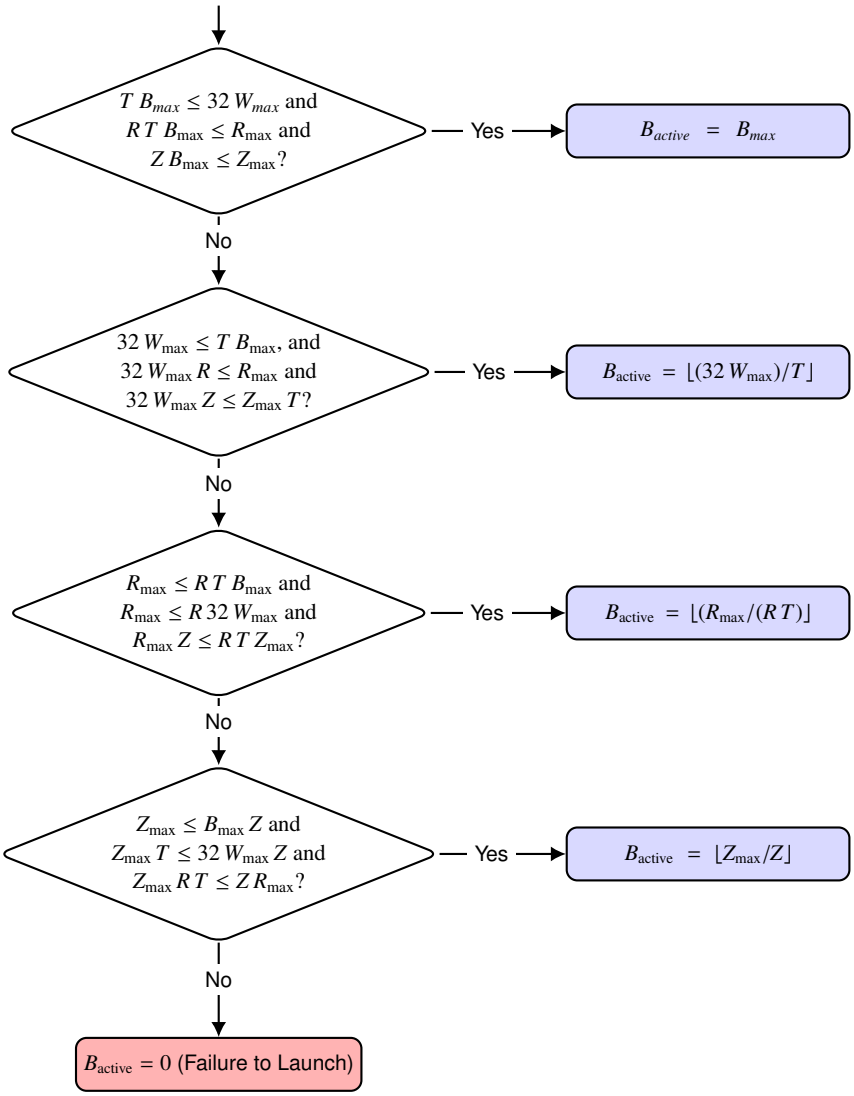

Figure 3.1: Rational program (presented as a flow chart) for the calculation of the number of active blocks per streaming processor in a CUDA kernel.

3.3 Piece-Wise Rational Functions in Rational Programs

We begin with an observation describing the fact that a rational program can be viewed as a piece-wise rational function [2] .

Observation 1 Let $\mathcal{S}$ be a rational program in $X_1, \ldots, X_n$ evaluating Y. Let s be any instruction of $\mathcal{S}$ other than a branch or an integer part instruction. Hence, this instruction can be of the form $C = A + B$, $C = A - B$, $C = A \times B$, where A and B can be any rational number. Let $V_1, \ldots, V_v$ be the variables that are defined at the entry point of the basic block of the instruction s. An elementary proof by induction yields the following fact. There exists a rational function in $V_1, \ldots, V_v$ denoted $f_s(V_1, \ldots, V_v)$ such that $C = f_s(V_1, \ldots, V_v)$ for all possible values of $V_1, \ldots, V_v$. From there, one derives the following observation. There exists a partition $\mathcal{T} = \{T_1, T_2, \ldots\}$ of $\mathbb{Q}^n$ (where $\mathbb{Q}$ denotes the field of rational numbers) and rational functions $f_1(X_1, \ldots, X_n)$, $f_2(X_1, \ldots, X_n)$, ... such that, if $X_1, \ldots, X_n$ receive respectively the values $x_1, \ldots, x_n$, then the value of Y returned by $\mathcal{S}$ is one of $f_i(x_1, \ldots, x_n)$ such that $(x_1, \ldots, x_n) \in T_i$ holds. In other words, $\mathcal{S}$ computes Y as a *piece-wise rational function* (PRF). Notice if $\mathcal{S}$ contains only one basic block then $\mathcal{S}$ can be trivially given by a single rational function.

Example 1 shows that the hardware occupancy of a CUDA kernel is given as a piece-wise rational function in the variables $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. Hence, in this example, we have $n = 8$, and, as shown by Figure 3.1, its partition of $\mathbb{Q}^n$ contains 5 parts as there are 5 terminating nodes in the flowchart.

Suppose that a flowchart C representing the rational program $\mathcal{R}$ is partially known; to be precise, suppose that the decision nodes are known (that is, the mathematical expressions defining them are known) while the process nodes are not. Then, from Observation 1, each process node can be given by a one or more rational functions. Trivially, a single formula can also be seen as a flowchart with a single process node. Determining each of those rational functions can be achieved by solving an *interpolation* or *curve fitting* problem. More generally, if the sequence of instructions in a process node involves non-rational functions (e.g. log) we can apply Stone-Weierstrass Theorem [37] to approximate each of those by a PRF.

It then follows that any performance metric, which can be depicted as a flow chart or a formula, can also be represented as a piece-wise rational function, and thus a rational program. For high-level performance metrics, which relies on low-level metrics, one could work recursively, first determining rational programs for the low-level metrics which depend on $\boldsymbol{P}$, $\boldsymbol{D}$, and $\boldsymbol{H}$, and then constructing a rational program for the high-level metric from those rational programs. Hence, by this recursive construction, we can fully determine a rational program for

[2] Here, rational function is in the sense of algebra, see Section 5.5.

a high-level metric depending only on $\boldsymbol{P}$, $\boldsymbol{D}$, and $\boldsymbol{H}$. Of course, hardware parameters could be fixed given a target architecture to yield a rational program which depends only on $\boldsymbol{P}$ and $\boldsymbol{D}$. Again, notice that even where formulas for low-level metrics are not known, it is still possible to estimate them as PRFs, and thus rational programs, via a curve fitting.

As an example, consider occupancy (Example 1). One could first determine PRFs for the number of registers user per thread and the amount of shared memory used per thread block. Then, a PRF is determined for the number of active blocks (Figure 3.1) from these two low-level metrics, and a few more hardware and program parameters. Thus, by recursive construction, we have a PRF depending only on program and hardware parameters.

Lastly, we make one final remark. We assumed that the decision nodes in the flowchart of the rational program were known, however, we could relax this assumption. Indeed, each decision node is given by a series of rational functions. Hence, those could also be determined by solving curve fitting problems. However, we do not discuss this further since it is not needed in our proposed technique or implementation presented in the remainder of this thesis.

Chapter 4

An Overview of KLARAPTOR

In this chapter, we present an overview of KLARAPTOR, a compile-time tool designed to optimize the performance of CUDA kernels by dynamically choosing the most suitable thread block configuration. We discuss the underlying theory of rational programs and the MWP-CWP performance model, which form the basis of KLARAPTOR's functionality. Furthermore, we explain the process of building and utilizing rational programs to determine optimal kernel launch parameters, detailing both the compile-time and runtime aspects of the tool. This chapter aims to provide an in-depth understanding of KLARAPTOR's methodology and how it contributes to enhancing kernel performance in CUDA applications.

4.1 Dynamic Optimization of CUDA Kernel Launch Parameters

The theory of rational programs is put into practice for the CUDA programming model by our tool KLARAPTOR. KLARAPTOR is a compile-time tool implemented using the LLVM Pass Framework and the MWP-CWP performance model to dynamically choose a CUDA kernel's launch parameters (thread block configuration) which optimize its performance. Most high-performance computing applications require computations be as fast as possible and so kernel performance is simply measured as its execution time.

As mentioned in Chapter 1, thread block configurations drastically affect the running time of a kernel. Determining optimal thread block configurations typically follows some heuristics, for example, constraining block size to be a multiple of 32 [2]. However, it is known that the dimension sizes of a thread block, not only its total size, affect performance [39, 11]. Moreover, since thread block configurations are intimately tied to the size of data being operated on, it is very unlikely that a static thread block configuration optimizes the performance of all data

sizes. Our tool effectively uses rational programs to dynamically determine the thread block configuration which minimizes the execution time of a particular kernel invocation, considering the invocation's particular data size and target architecture. This is achieved in two main steps.

1. At the compile-time of a CUDA program, its kernels are analyzed in order to build rational programs estimating some performance metrics for each individual kernel. Each rational program, written as code in the C language, is inserted into the code of the CUDA program so that it is called before the execution of the corresponding kernel.
2. At runtime, immediately preceding the launch of a kernel, where data parameters have specific values, the rational program is evaluated to determine the thread block configuration which optimizes the performance of the kernel. The kernel is then launched using this thread block configuration.

Not only are we concerned with kernel performance, but also programmer performance. By that, we mean the efficiency of a programmer to produce optimal code. When a programmer is attempting to optimize a kernel, choosing optimal launch parameters can either be completely ignored, performed heuristically, determined by trial and error, or determined by an exhaustive search. The latter two options quickly become infeasible as data sizes grow large. Regardless, any choice of optimal thread block configuration is likely to optimize only a single data size.

For KLARAPTOR to be practical, not only does the choice of optimal kernel launch parameters need to be correct, but it must also be more efficient than trial and error or exhaustive search. Namely, the compile-time analysis cannot add too much overhead to the the compilation time and the runtime decision of the kernel launch parameters cannot overwhelm the program execution time. For the former, our analysis is performed quickly by analyzing kernel performance on only small data sizes, and then results are extrapolated. For the later, the rational program evaluation is quick and simple, being only the evaluation of a few rational functions. Moreover, we maintain a runtime invocation history to instantly provide results for future kernel launches. Our implementation is detailed in Chapter 5.

We have made use of the `Polybench/GPU` benchmark suite as an empirical evaluation of the correctness of our tool on a range of CUDA programs. In Figure 1.1 we have already seen that KLARAPTOR accurately predicts the optimal or near-optimal thread block configuration. Before presenting more detailed results and experimentation in Chapter 6, we describe the steps followed by our tool to build and use rational programs for determining a thread block configuration which optimizes performance.

4.2 An Algorithm to Select Program Parameters

In this section the notations and hypotheses are the same as in Chapter 3. Namely, $\mathcal{E}$ is a high-level performance metric for the multithreaded program $\mathcal{P}$, $\boldsymbol{L}$ is a set of low-level metrics of size ℓ, and $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$ are sets of program, data, and hardware parameters, respectively. Recall $\boldsymbol{P}$ has size p. Let us assume that the values of $\boldsymbol{H}$ are known at the compile-time of $\mathcal{P}$ while the values of $\boldsymbol{D}$ are known at runtime. Further, let us assume that $\boldsymbol{P}$ and $\boldsymbol{D}$ take integer values. Hence the values of $\boldsymbol{P}$ belong to a finite set $F \subset \mathbb{Z}^p$. That is to say, the possible values of $\boldsymbol{P}$ are tuples of the form $(\pi_1, \ldots, \pi_p) \in F$, with each π_i being an integer. Let us call such a tuple a *configuration* of the program parameters. Due to the nature of program parameters, those are not necessarily all independent variables For example, in CUDA, the product of the dimension sizes of a thread block is usually a multiple of the warp size (32).

Given a performance-prediction model for $\mathcal{E}$, one could work recursively to determine a single helper program $\mathcal{R}$, depending on only $\boldsymbol{D}$ and $\boldsymbol{P}$, evaluating $\mathcal{E}$, from a combination of rational programs constructed for each low-level metric in $\boldsymbol{L}$ and values of $\boldsymbol{D}$ and $\boldsymbol{P}$. Following Section 3.3, each of these helper programs are constructed by computing rational functions. Without loss of generality, let us assume each low-level metric is given by a single formula and thus a single rational function. Hence, we look to determine $g_1(\boldsymbol{D}, \boldsymbol{P})$, ..., $g_\ell(\boldsymbol{D}, \boldsymbol{P})$ for the ℓ low-level metrics. Finally, at runtime, given particular values of $\boldsymbol{D}$, the helper program for $\mathcal{E}$ can be evaluated for various values of $\boldsymbol{P}$ to determine the optimal configuration. In the remainder of this section we describe the general process to build and use helper programs to determine optimal configurations. The entire process is decomposed into five steps: the first three occur at compile-time and the next three at runtime.

1. **Data collection**: To perform a curve fitting of the rational functions $g_1(\boldsymbol{D}, \boldsymbol{P})$, ..., $g_\ell(\boldsymbol{D}, \boldsymbol{P})$ we require data points to fit. These are collected by (i) selecting a subset of K points from the space of possible values of ($\boldsymbol{D}$, $\boldsymbol{P}$); and (ii) executing the program $\mathcal{P}$, recording the values of the low-level performance metrics $\boldsymbol{L}$ as $\boldsymbol{V} = (V_1, \ldots, V_\ell)$, at each point in K. The data used for executing the programs is generated randomly, but could follow some scheme provided by the user.
2. **Rational function approximation**: For each low-level metric L_i we use the set of points K and the corresponding value V_i measured at each point in order to approximate the rational function $g_i(\boldsymbol{D}, \boldsymbol{P})$. In practice, these empirical values are likely to be noisy from profiling, and/or numerical approximations. Consequently, we actually determine a rational function $\widehat{g_i}(\boldsymbol{D}, \boldsymbol{P})$ which approximates $g_i(\boldsymbol{D}, \boldsymbol{P})$.
3. **Code generation**: In order to generate the helper program $\mathcal{R}$, we proceed as follows:
   (i) we convert the helper program representing $\mathcal{E}$ into code,

(ii) we convert each $\widehat{g_i}(\boldsymbol{D}, \boldsymbol{P})$ into a sub-routine estimating L_i, and

(iii) we include those sub-routines into the code computing $\mathcal{E}$, which yields the desired helper program $\mathcal{R}$ depending only on $\boldsymbol{D}$ and $\boldsymbol{P}$.

4. **Helper program evaluation**: At the runtime of $\mathcal{P}$, the data parameters $\boldsymbol{D}$ are given particular values. For those specified values of $\boldsymbol{D}$ and for all practically meaningful values of $\boldsymbol{P}$ from the set F,[1] we compute an estimate of $\mathcal{E}$ using $\mathcal{R}$. The evaluation of $\mathcal{E}$ over so many different possible program parameters is feasible for three reasons:
   (i) the number of program parameters is small, typically $p \leq 3$, see Chapter 5;
   (ii) the set of meaningful values for $\boldsymbol{P}$ is small (consider that in CUDA the product of thread block dimension sizes should be a multiple of 32 less than 1024), and
   (iii) the program $\mathcal{R}$ simply evaluates a few polynomial formulae and thus runs almost instantaneously.
5. **Program execution**: Once an optimal configuration is selected, the program $\mathcal{P}$ is executed using this configuration along with the values of $\boldsymbol{D}$.

[1] The values for $\boldsymbol{P}$ are likely to be constrained by the values $\boldsymbol{D}$. For example, if P_1, P_2 are the two dimension sizes of a two-dimensional thread block of a CUDA kernel operating on a square matrix of order D_1, then $P_1 P_2 \leq D_1^2$ is meaningful.

Chapter 5

The implementation of KLARAPTOR

This section is an overview of the implementation of our previously presented technique (Section 4.2) specialized to CUDA in the KLARAPTOR tool. Our tool is built in the C language, making use of the LLVM Pass Framework (see Section 5.2) and the NVIDIA Nsight Compute CLI (NCU) (see Section 5.3). KLARAPTOR is freely available in source at https://github.com/orcca-uwo/KLARAPTOR.

In the case of a CUDA kernel, the data parameters specify the input data size. In many examples this is a single parameter, say N, describing the size of an array (or the order of a multi-dimensional array), the values of which are usually powers of 2. Program parameters describe the kernel launch parameters, i.e. grid and thread block dimension sizes, and are also typically powers of 2. For example, a possible thread block configuration may be $1024 \times 1 \times 1$ (a one-dimensional thread block), or $16 \times 16 \times 2$ (a three-dimensional thread block). Lastly, the hardware parameters are values specific to the target GPU, for example, memory bandwidth, the number of SMs available, and their clock frequency.

We organize this section as follows. Sections 5.1 and 5.2 are specific to our implementation and do not correspond to any step of Section 4.2. The compile time steps 1 (data collection) and 2 (rational function estimation) are reflected in Sections 5.3 and 5.5, respectively, while step 3 requires no explanation. The runtime steps 4 (rational program evaluation) and 5 (program execution) are trivial to perform. Throughout this section, the term *rational program* refers to the mathematical concept defined in Section 3.1 whereas the term *helper program* refers to the generated code which implements rational programs in order to select kernel configurations.

5.1 Annotating and preprocessing source code

Beginning with a CUDA program written in C/C++, we minimally annotate the host code to make it compatible with our *pre-processor*. We take into account the following points:

(i) the code targets at least CUDA Compute Capability (CC) 7.5;

(ii) there should be no CUDA runtime API calls as such calls will interfere with later CUDA driver API calls used by our tool, for example, `cudaSetDevice`;

(iii) the block dimensions and grid dimensions must be declared as the typical CUDA `dim3` structs.

For each kernel in the CUDA code, we add two pragmas, one specifying the dimension of the kernel (1, 2, or 3), and one defining the index of the kernel input argument corresponding to the data size N. For instance, consider the code segment below of a CUDA kernel and added pragmas. This kernel operates of a two-dimensional array of order N, making it a two-dimensional kernel.

```
#pragma kernel_info_size_param_idx_Sample = 1;
#pragma kernel_info_dim_sample_kernel = 2;
__global__ void Sample (int *A, int N) {
    int tid_x = threadIdx.x + blockIdx.x*blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y*blockDim.y;
    ...
}
```

Lastly, for each kernel, the user must fill two formatted configuration files which follow `Python` syntax. One specifies the constraints on the thread block configuration while the other specifies the grid dimensions. For example, for the 2D kernel `Sample` above, one could specify that its thread block configuration (bx, by, bz) must satisfy $bx < by^2$, $bx < N$ and $by < N$. Since the kernel dimension is given as 2, we assume $bz = 1$. Similarly, the grid dimensions (gx, gy, gz), could be specified as $gx = \lceil \frac{N}{bx} \rceil$, $gy = \lceil \frac{N}{by} \rceil$, $gz = 1$.

Now, a preprocessor processes the annotated source code, replacing CUDA runtime API calls with driver API kernel launches. This step includes source code analysis in order to extract a list of kernels, a list of kernel calls in the host code, and finally, the body of each kernel to be used for further analysis. A so-called "PTX lookup table" is built to store kernel information and static parameters. This table will be inserted into the "instrumented binary", the compiled CUDA program augmented by the helper programs.

5.2 Input/Output builder

The Input/Output builder Pass, or IO-builder, is a compiler pass written in the LLVM Pass Framework to build the previously mentioned "instrumented binary". This pass embeds an IO mechanism (i.e. a function call) to communicate with the helper program of a kernel for each of its invocations. Thus, at the runtime of the CUDA program being analyzed (step 5 of Section 4.2), an IO function is called before each kernel invocation to return six integers,

(gx, gy, gz, bx, by, bz), the optimal kernel launch parameters.

The IO-builder pass goes through the following steps:

(i) obtain the LLVM intermediate representation of the instrumented source code and find all CUDA driver API kernel calls;

(ii) relying on the annotated information for each kernel, determine which variables in the IR contain the value of N for a corresponding kernel call; and

(iii) insert a call to an IO function immediately before each kernel call in order to pass the runtime value of N to the corresponding helper program and retrieve the optimal kernel launch parameters.

5.3 Building a helper program: data collection

In order to perform the eventual rational function approximation, we must collect data and statistics regarding certain performance counters and runtime metrics (see [20] and [1]). These metrics can be partitioned into three categories.

Firstly, *architecture-specific performance counters* of a kernel, characteristics influenced by the CC of the device. These can be obtained at compile-time, since the target CC is specified at this time. These characteristics include the number of registers used per thread, the amount of static shared memory per thread block, and the number of (arithmetic and memory) instructions per thread.

Secondly, *runtime-specific performance counters* that depend on the behavior of the kernel at runtime. This includes values impacted by memory access patterns, namely, the number of memory accesses per warp, the number of memory instructions of each thread, and the total number of warps that are being executed. We have developed a customized profiler using NVIDIA's Nsight Compute CLI to collect the required runtime performance counters.

Thirdly, *device-specific parameters*, which describe an actual GPU card, allow us to compute a more precise performance estimate. A subset of such parameters can be determined by microbenchmarking the device (see [29] and [41]), this includes the memory bandwidth, and the departure delay for memory accesses. The remaining parameters can easily be obtained by consulting the vendor's guide [3], or by querying the device itself via the CUDA driver API. Such parameters include the number of SMs on the card, the clock frequency of SM cores, and the instruction delay.

5.4 Building a helper program: outlier removal

As mentioned in Section 1.1, we now describe the outlier removal step, which is performed by quartile fencing algorithm. The rationale behind integrating this algorithm lies in addressing potential noise in the empirical data gathered from NVIDIA's Nsight Compute (ncu), which could stem from factors such as dynamic voltage and frequency scaling (DVFS) and other variations in GPU performance. By pinpointing and eliminating upper outliers from the dataset, our objective is to enhance the accuracy of the subsequent parameter estimation process.

The quartile fencing algorithm is a robust statistical method for detecting and removing outliers from a dataset. It is based on the concept of interquartile range (IQR), which is the difference between the first quartile (Q1) and the third quartile (Q3) of the data. The algorithm defines outlier boundaries, called the lower and upper inner fences, by extending the IQR beyond Q1 and Q3[14]:

(i) $IQR = Q3 - Q1$.
(ii) $UIF = Q3 + 1.5 \times IQR$.

Data points falling outside the upper inner fence are considered as outliers and are removed from the dataset. To implement the outlier removal process, we first profile our program $\mathcal{P}$ for small input sizes of N and obtain the MWP-CWP estimation for clock-cycles for various thread block configurations. Next, we calculate the first and third quartiles (Q1 and Q3) along with the interquartile range (IQR) for the estimated clock-cycles. Using the quartile fencing algorithm, we determine the upper inner fence and identify the thread block configurations exceeding this threshold as outliers.

Once the outliers are identified, we remove them from the dataset before proceeding with the parameter estimation process. This preprocessing step helps minimize the impact of upper outliers on the data and enhances the accuracy of the resulting parameter estimation.

5.5 Building a helper program: rational function approximation

Using the runtime data collected in the previous step, we look to determine the rational functions $\widehat{g_i}(\boldsymbol{D}, \boldsymbol{P})$ (see Section 4.2) which estimate the low-level metrics or other intermediate values in the rational program $\mathcal{R}$. For ease of discussion, we replace the parameters $\boldsymbol{D}$ and $\boldsymbol{P}$ with the generic variables $X_1, \ldots, X_n$.

A rational function is simply a fraction of two polynomials:

$$f(X_1, \ldots, X_n) = \frac{\alpha_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \alpha_i \cdot (X_1^{u_1} \cdots X_n^{u_n})}{\beta_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \beta_j \cdot (X_1^{v_1} \cdots X_n^{v_n})} \tag{5.1}$$

With a *degree bound* (an upper limit on the exponent) on each variable X_k in the numerator and the denominator, u_k and v_k, respectively, these polynomials can be defined up to some *parameters* (using the language of parameter estimation), namely the coefficients of the polynomials, $\alpha_1, \ldots, \alpha_i$ and $\beta_1, \ldots, \beta_j$. Through algebraic analysis of performance models like the MWP-CWP model, and empirical evidence, these degree bounds are relatively small.

We perform a parameter estimation (for each rational function) on the coefficients $\alpha_1, \ldots, \alpha_i, \beta_1, \ldots, \beta_j$ to determine the rational function precisely. This is a simple linear regression which can be solved by an over-determined system of linear equations, say by the method of linear least squares. This system of linear equations is often described in matrix format $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ (the *sample matrix* or *design matrix*) and $\mathbf{b}$ (the right-hand side vector) encode the collected data, while the solution vector $\mathbf{x}$ encodes the model-fitting parameters. $\mathbf{A}$ being derived from a rational function implies that it is essentially the sample matrix for the denominator polynomial appended to the sample matrix for the numerator polynomial[1].

Many different methods exist for solving this so-called linear least squares problem, such as the *normal-equations*, or *QR-factorization*, however, these methods are either numerically unstable (normal-equations), or will fail if the sample matrix is rank-deficient (both normal-equations and QR) [13]. We rely then on the *singular value decomposition* (SVD) of $\mathbf{A}$ to solve this problem. This decomposition is very computationally intensive, much more than that of normal-equations or QR, but is also much more numerically stable, as wel as being capable of producing solutions with a rank-deficient sample matrix.

We are highly concerned with the robustness of our method due to three problems present in our particular situation:

(1) the sample matrix is very ill-conditioned;
(2) the sample matrix will often be (numerically) rank-deficient;
(3) we are interested in using our fitted model for extrapolation, meaning any numerical error in the model fitting will grow very quickly [13].

While (3) is an issue inherent to our model fitting problem, (1) and (2) result from our choice of model, and how the sample points $(X_1, \ldots, X_n)$ are chosen, respectively. Using a rational function (or polynomial) as the model for which we wish to estimate parameters presents numerical problems. The resulting sample matrix is essentially a Vandermonde matrix. These

[1] Keen observers will notice that, for rational functions, we must actually solve a system of homogeneous equations. Such details are omitted here, but we refer the reader to [9, Chapter 5].

matrices, while theoretically of full rank, are extremely ill-conditioned [13, 8].

Refering to (2), we discuss the difficulties in obtaining poised sample points for modeling functions that involve CUDA thread block dimensions as variables. The geometric constraints, combined with the requirement that the product of dimensions be a multiple of 32, makes it challenging to achieve a full-rank sample matrix, which is crucial for accurate and stable model fitting. As a result, there is a higher likelihood of encountering a rank-deficient sample matrix [12, 30], which complicates the entire model estimation process.

Despite all of these challenges our parameter estimation techniques are well-implemented in optimized C code. We use optimized algorithms from LAPACK (Linear Algebra PACKage) [5] for singular value decomposition and linear least squares solving while rational function and polynomial implementations are similarly highly optimized thanks to the Basic Polynomial Algebra Subprograms (BPAS) library [6, 9]. With parameter estimation complete the rational functions required for the rational program are fully specified and we can finally construct it.

5.6 Helper programs

In practice, the use of helper programs is split into two parts: the generation of the rational program at the compile-time of the multithreaded program $\mathcal{P}$, and the use of the helper program during the runtime of $\mathcal{P}$.

5.6.1 Compile-time code generation

We are now at Step 3 of Section 4.2. We look to define a helper program which evaluates the high-level metric $\mathcal{E}$ of the program $\mathcal{P}$ using the MWP-CWP model. In implementation, this is achieved by using a previously defined *helper program template* which contains the formulas and case discussion of the MWP-CWP model, independent of the particular program being investigated. Using simple regular expression matching and text manipulation we combine the helper program template with the rational functions determined in the previous step to obtain a helper program specialized to the multithreaded program $\mathcal{P}$. The generation of this helper program is performed completely during compile-time, before the execution of the program itself.

5.6.2 Runtime optimization

At runtime, the input data sizes (data parameters) are well known. In combination with the known hardware parameters, since the program is actually being executed on a specific device,

we are able to specialize every parameter in the helper program and obtain an estimate for the high-level metric $\mathcal{E}$. This helper program is then easily and quickly evaluated during (or immediately prior to) the execution of $\mathcal{P}$. Evaluating the helper program for each possible thread block configuration, as restricted by our data parameters and the CUDA programming model itself, we determine a thread block configuration which optimizes $\mathcal{E}$. The program $\mathcal{P}$ is finally executed using this optimal thread block configuration. Therefore, we have completed Steps 4 and 5 of Section 4.2.

Chapter 6

Experimentation

In this section we examine the performance of KLARAPTOR by applying it to the CUDA programs of the `Polybench/GPU` benchmark suite [18]. We note here that many of the kernels in this suite perform relatively low amounts of work; they are best suited to being executed many times from a loop in the host code. Data in this section was collected using a RTX 2070 SUPER.

Table 6.1 provides experimental data for the main kernels in the benchmark suite `Polybench/GPU`. Namely, this table compares the execution times of the thread block configuration chosen by KLARAPTOR against the optimal thread block configuration found though exhaustive search. The table shows a couple of data sizes in order to highlight that the best configuration can change for different input sizes. While it may appear for some examples that there are large variations between timings of the KLARAPTOR-chosen configuration and the optimal, these should be considered within the full range of possible configurations. Recall from Figure 1.1 that compared to the worst possible timings, the KLARAPTOR-chosen configuration and the optimal result in very similar in timings.

In Table 6.2 we report the time it takes for KLARAPTOR to perform its compile-time analysis and build the rational programs for each example in the `PolyBench/GPU` suite. This table also compares the times taken by KLARAPTOR and the exhaustive search. Exhaustive search times are given as a sum over all possible configurations and all powers of 2 up to N=8192, meanwhile the data collection for KLARAPTOR is done for $128 \leq N \leq 2048$, again all powers of 2 within this range. Note that KLARAPTOR can determine near-optimal thread block configurations for any $N \geq 128$. As seen in Table 6.2, ∞ represents the upper bound of KLARAPTOR's search space. The best and worst execution times for the main kernel in each example (for $N = 8192$) is also given to highlight the fact that our optimization step is sometimes faster than even a single execution of a kernel with a poor choice of thread block configuration. We note that for some kernels, with very quick running times, exhaustive search

Table 6.1: KLARAPTOR vs. exhaustive search for thread block configuration choice for kernels in `Polybench/GPU`.

Kernel	N	KLARAPTOR Time (ms)	Chosen Config.	Optimal Time (ms)	Optimal Config.
atax K1	2048	0.404	8, 4	0.386	16, 2
	4096	0.856	16, 2	0.856	16, 2
	8192	3.175	16, 2	2.646	32, 1
atax K2	2048	0.586	8, 4	0.583	16, 2
	4096	1.198	32, 1	1.196	64, 1
	8192	3.096	32, 1	2.499	256, 1
bicg K1	2048	0.598	8, 4	0.598	8, 4
	4096	1.217	32, 1	1.212	64, 1
	8192	2.886	32, 1	2.568	32, 2
bicg K2	2048	0.388	8, 4	0.376	16, 2
	4096	0.838	16, 2	0.838	16, 2
	8192	3.855	16, 2	3.042	32, 1
convolution2d	2048	0.183	16, 2	0.139	32, 4
	4096	0.517	32, 2	0.517	32, 2
	8192	2.265	16, 32	2.033	32, 4
corr	2048	1173.972	2, 16	1173.972	2, 16
	4096	10286.199	2, 16	5799.606	16, 2
	8192	366938.750	8, 64	28347.246	32, 1
covar	2048	1243.915	8, 4	1179.058	2, 16
	4096	12050.661	64, 4	5831.021	16, 2
	8192	225534.281	128, 4	27667.753	32, 1
fdtd_step1	2048	63.408	32, 2	63.310	256, 1
	4096	250.627	32, 2	250.302	128, 1
	8192	997.219	32, 2	996.090	16, 8
fdtd_step2	2048	91.518	32, 1	63.998	128, 1
	4096	251.701	32, 4	250.677	128, 1
	8192	1385.221	16, 2	996.072	16, 8
fdtd_step3	2048	96.936	32, 1	84.321	32, 2
	4096	330.271	32, 2	330.259	128, 1
	8192	1526.632	16, 2	1311.767	16, 8
gemm	2048	143.621	16, 2	62.054	16, 64
	4096	1455.516	4, 16	491.389	16, 64
	8192	12334.238	4, 16	3940.204	16, 64
gesummv	2048	1.321	4, 8	0.773	16, 2
	4096	1.818	16, 2	1.789	128, 1
	8192	6.778	16, 2	5.765	32, 1
gramschmidt K1	2048	82.855	1024, 1	81.918	4, 16
	4096	370.100	1024, 1	366.439	32, 2
	8192	1511.951	1024, 1	1494.540	32, 2
gramschmidt K2	2048	8.581	8, 4	8.245	64, 1
	4096	17.738	16, 2	17.516	32, 1
	8192	38.508	32, 1	38.364	64, 1
gramschmidt K3	2048	2227.957	32, 1	2204.946	4, 8
	4096	10020.573	32, 1	10020.573	32, 1
	8192	43879.894	32, 1	43879.894	32, 1
mm2 K1	2048	67.812	16, 8	61.852	16, 64
	4096	489.912	16, 64	489.912	16, 64
	8192	3925.331	16, 64	3925.331	16, 64
mm2 K2	2048	67.827	16, 8	61.847	16, 64
	4096	488.863	16, 64	488.863	16, 64
	8192	3924.813	16, 64	3924.813	16, 64
mm3 K1	2048	67.833	16, 8	61.840	16, 64
	4096	489.801	16, 64	489.801	16, 64
	8192	3926.057	16, 64	3926.057	16, 64
mm3 K2	2048	67.905	16, 8	61.802	16, 64
	4096	489.185	16, 64	489.185	16, 64
	8192	3925.962	16, 64	3925.962	16, 64
mm3 K3	2048	67.611	16, 8	61.555	16, 64
	4096	489.250	16, 64	489.250	16, 64
	8192	3926.108	16, 64	3926.108	16, 64
mvt K1	2048	0.404	8, 4	0.382	16, 2
	4096	0.856	16, 2	0.856	16, 2
	8192	3.298	16, 2	2.647	32, 1
mvt K2	2048	0.585	8, 4	0.584	16, 2
	4096	1.198	32, 1	1.194	64, 1
	8192	2.487	32, 1	2.487	32, 1
syr2k	2048	244.153	8, 64	242.817	8, 4
	4096	1942.944	8, 128	1941.899	8, 64
	8192	21648.312	8, 4	15545.357	8, 128
syrk	2048	143.478	8, 8	143.442	8, 64
	4096	1137.063	8, 128	1136.299	8, 64
	8192	9085.384	8, 128	9085.384	8, 128

is not a bad option. However, some examples such as `GRAMSCHMIDT`, take an exorbitant amount of time for exhaustive search. This table also shows that the one-time compile-time cost can often be amortized by only a few executions of the kernel.

Table 6.2: KLARAPTOR Optimization Times on `Polybench/GPU`, RTX 2070 SUPER Comparing times for (1) compile-time optimization steps of KLARAPTOR, (2) exhaustive search over all thread block configurations, the execution time for a kernel given (3) the best thread block configuration, and (4) the worst thread block configuration. Exhaustive search is given as a sum for values up to $N = 8192$ (except convolution3d with $N = 512$).

Kernel	KLARAPTOR Time (s) $128 \leq N < \infty$	Ex. Search Time (s) $128 \leq N \leq 8192$	Min Time (s) $N = 8192$	Max Time (s) $N = 8192$
2DCONV	210.29	82.78	0.002	0.023
ATAX	507.59	59.60	0.006	1.940
MVT	508.03	60.03	0.005	1.978
BICG	510.91	60.16	0.006	2.050
GESUMMV	398.54	142.78	0.006	0.129
GEMM	456.50	987.77	3.941	126.052
SYRK	579.84	2772.64	9.069	160.944
SYR2K	1173.68	9553.64	15.534	459.169
2MM	700.49	1889.62	7.851	240.828
3MM	944.54	2798.12	11.779	361.310
CORR	1032.92	10924.12	28.365	861.289
COVAR	1141.45	23251.12	27.670	3900.855
3DCONV	132.88	52.06	0.006	0.053
GRAMSCHM	2113.27	94206.06	45.418	35146.314
FDTD_2D	489.21	495.79	3.304	21.107

Chapter 7

Conclusions and future work

The performance of a single CUDA program can vary wildly depending on the target GPU device, the input data size, and the kernel launch parameters. Moreover, a thread block configuration yielding optimal performance for a particular data size or a particular target device will not necessarily be optimal for a different data size or different target device. In this thesis we have presented the KLARAPTOR tool for determining optimal CUDA thread block configurations for a target architecture, in a way which is adaptive to each kernel invocation and input data, allowing for dynamic data-dependent performance and portable performance. This tool is based upon our technique of encoding a performance prediction model as a rational program. The process of constructing such a rational program is a fast and automatic compile-time process which occurs simultaneously to compiling the CUDA program by use of the LLVM Pass framework. Our tool was tested using the kernels of the `Polybench/GPU` benchmark suite with great results.

One of the main challenges we face in our research is understanding GPU hardware saturation, as executing a CUDA kernel with an optimal number of threads to efficiently utilize available hardware resources is a complex task. As a result, this limitation may be impeding our ability to attain precise occupancy values. As newer GPU models, such as NVIDIA's Ampere architecture, have become available, it may be necessary to explore improved performance prediction models to better account for factors such as concurrency and arithmetic intensity. We anticipate that such models may enable us to achieve more accurate results in our research.

Our use of linear least squares for extrapolation beyond the "training" range has proven to be less than ideal. In future work, it would be beneficial to test the model within the training range, but with multiples of 32 instead of powers of 2, in order to more accurately capture the behavior of CUDA thread block dimensions. The combination of the MWP-CWP model as a theoretical performance model with linear regression may not be the most effective approach.

Considering the tree-like structure of rational programs, a random forest regression could potentially provide a better fit for our problem. Future work should investigate the feasibility and performance of random forest regression as an alternative to linear regression within the context of KLARAPTOR.

Lastly, beyond the scope of rational programs, the exploration of other machine learning models, such as neural networks, could prove valuable in determining the optimal thread block configuration for CUDA kernels. These alternative models may provide more robust and accurate predictions, improving the performance and efficiency of the kernels.

In summary, the future work for this thesis should focus on refining the methods used for model fitting, exploring alternative regression techniques that better suit the structure of rational programs, and investigating the potential of machine learning models for determining the optimal CUDA kernel configurations.

Bibliography

[1] CUDA runtime API: v10.0. NVIDIA Corporation, September 2018. http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[2] CUDA C Best Practices Guide, v12.0, February 2023. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[3] CUDA C Programming Guide, v12.0, February 2023. https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[6] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. Moir, M. Moreno Maza, L. Wang, N. Xie, and Y. Xie. Basic Polynomial Algebra Subprograms (BPAS), 2019. http://www.bpaslib.org.

[7] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pages 225–234, 2008.

[8] Bernhard Beckermann. The condition number of real vandermonde, krylov and positive definite hankel matrices. *Numerische Mathematik*, 85(4):553–577, 2000.

[9] Alexander Brandt. High performance sparse multivariate polynomials: Fundamental data structures and algorithms. Master's thesis, University of Western Ontario, London, ON, Canada, 2018.

[10] Alexander Brandt, Davood Mohajerani, Marc Moreno Maza, Jeeva Paudel, and Lin-Xiao Wang. KLARAPTOR: A tool for dynamically finding optimal kernel launch parameters targeting CUDA programs. *CoRR*, abs/1911.02373, 2019.

[11] C. Chen, X. Chen, A.-K. Keita, M. Moreno Maza, and N. Xie. MetaFork: A compilation framework for concurrency models targeting hardware accelerators and its application to the generation of parametric CUDA kernels. In *CASCON 2015*.

[12] Kwok Chiu Chung and Te Hai Yao. On lattices admitting unique lagrange interpolations. *SIAM Journal on Numerical Analysis*, 14(4):735–743, 1977.

[13] R. Corless and N. Fillion. *A graduate introduction to numerical methods*. Springer, 2013.

[14] B.S. Everitt and A. Skrondal. *The Cambridge Dictionary of Statistics*. Cambridge University Press, 2010.

[15] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *IEEE,ICASSP '98*, pages 1381–1384, 1998.

[16] Joseph D. Garvey and Tarek S. Abdelrahman. Automatic performance tuning of stencil computations on gpus. In *ICPP 2015*, pages 300–309, 2015.

[17] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. ACM, 1989.

[18] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Proc. InPar*, pages 1–10, 2012.

[19] S.A. Haque, M. Moreno Maza, and N. Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. In *Parallel Computing: On the Road to Exascale, Proc. of ParCo*, volume 27, pages 35–44. IOS Press, 2015.

[20] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA 2009*, pages 152–163, 2009.

[21] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9(4), 2013.

[22] T. Kistler and M/ Franz. Continuous program optimization: A case study. *(TOPLAS)*, 25(4):500–548, 2003.

[23] J. Kurzak, Y. Tsai, M. Gates, A. Abdelfattah, and J. J. Dongarra. Massively parallel automated software tuning. In *ICPP 2019*, pages 92:1–92:10.

[24] Robert V. L., Boyana N., and Allen D. M. Autotuning GPU kernels via static and predictive analysis. In *ICPP 2017*, pages 523–532, 2017.

[25] Yixun Liu, Eddy Z Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[26] L. Ma, K. Agrawal, and R. D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Comp. Syst.*, 30, 2014.

[27] Marc Moreno Maza. Models of computation for graphics processing units, November 2017. `https://www.csd.uwo.ca/~mmorenom/Publications/CASCON-2017.pdf`.

[28] Marc Moreno Maza. Many-core computing with cuda, March 2022. `https://www.csd.uwo.ca/~mmorenom/HPC-Slides/Many_core_computing_with_CUDA.pdf`.

[29] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):72–86, 2017.

[30] Peter J Olver. On multivariate interpolation. *Studies in Applied Mathematics*, 116(2):201–240, 2006.

[31] Tom Papatheodore. Vector addition (cuda. `https://github.com/olcf-tutorials/vector_addition_cuda`, 2022.

[32] M. Püschel, J.M.F. Moura, B. Singer, J. Xiong, J.R. Johnson, D.A. Padua, M.M. Veloso, and R.W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[33] K. Sato, H. Takizawa, K. Komatsu, and H. Kobayashi. Automatic tuning of CUDA execution parameters for stencil processing. In *Software Automatic Tuning, From Concepts to State-of-the-Art Results*. 2010.

[34] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, 2012.

[35] C. Song, L.-P. Wang, and T.J. Mart´inez. Automated code engine for graphical processing units: Application to the effective core potential integrals and gradients. *J. Chem. Theory Comput.*, 12(1):92–106, 2015.

[36] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[37] M. H. Stone. The generalized weierstrass approximation theorem. *Mathematics Magazine*, 21(5):237–254, 1948.

[38] Philippe Tillet and David Cox. Input-aware auto-tuning of compute-bound hpc kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Y. Torres, A. González-Escribano, and D. R. Llanos. ubench: exposing the impact of CUDA block geometry in terms of performance. *The Journal of Supercomputing*, 65(3):1150–1163, 2013.

[40] R. Clinton Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 IEEE/ACM Conference on Supercomputing*, 1998.

[41] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE ISPASS*, pages 235–246, 2010.

Appendix A

Examples of Annotated Code

A.1 PolyBench Code

Below is an example of the PolyBench code for the 2DConvolution benchmark. As mentioned in Section 5.1 the pragmas are the manual annotations required for our *pre-processor* to work.

```
1  ///**
2  // * 2DConvolution.cu: This file is part of the PolyBench/GPU 1.0 test
   ↪  suite.
3  // *
4  // *
5  // * Contact: Scott Grauer-Gray <sgrauerg@gmail.com>
6  // * Louis-Noel Pouchet <pouchet@cse.ohio-state.edu>
7  // * Web address:
   ↪  http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU
8  // */
9
10 #include "2dconv_utils.h"
11
12
13 #pragma kernel_info_size_param_idx_Convolution2D_kernel = 2;
14 #pragma kernel_info_dim_Convolution2D_kernel = 2;
15
16 void
17 conv2D (DATA_TYPE* A, DATA_TYPE* B)
18 {
```

```
19    int i, j;
20    DATA_TYPE c11, c12, c13, c21, c22, c23, c31, c32, c33;
21
22    c11 = +0.2;
23    c21 = +0.5;
24    c31 = -0.8;
25    c12 = -0.3;
26    c22 = +0.6;
27    c32 = -0.9;
28    c13 = +0.4;
29    c23 = +0.7;
30    c33 = +0.10;
31
32    for (i = 1; i < NI - 1; ++i) // 0
33      {
34        for (j = 1; j < NJ - 1; ++j) // 1
35          {
36            B[i * NJ + j] = c11 * A[(i - 1) * NJ + (j - 1)]
37                + c12 * A[(i + 0) * NJ + (j - 1)]
38                + c13 * A[(i + 1) * NJ + (j - 1)]
39                + c21 * A[(i - 1) * NJ + (j + 0)]
40                + c22 * A[(i + 0) * NJ + (j + 0)]
41                + c23 * A[(i + 1) * NJ + (j + 0)]
42                + c31 * A[(i - 1) * NJ + (j + 1)]
43                + c32 * A[(i + 0) * NJ + (j + 1)]
44                + c33 * A[(i + 1) * NJ + (j + 1)];
45          }
46      }
47  }
48
49  void
50  init (DATA_TYPE* A)
51  {
52    int i, j;
53
54    for (i = 0; i < NI; ++i)
```

```
55      {
56        for (j = 0; j < NJ; ++j)
57          {
58            A[i * NJ + j] = (float) rand () / RAND_MAX;
59          }
60      }
61  }
62
63  int
64  compareResults (DATA_TYPE* B, DATA_TYPE* B_outputFromGpu)
65  {
66    int i, j, fail;
67    fail = 0;
68
69    // Compare a and b
70    for (i = 1; i < (NI - 1); i++)
71      {
72        for (j = 1; j < (NJ - 1); j++)
73          {
74            if (percentDiff (
75                B[i * NJ + j],
76                B_outputFromGpu[i * NJ + j]) >
                   ↪  PERCENT_DIFF_ERROR_THRESHOLD)
77              {
78                fail++;
79                return (EXIT_FAILURE);
80              }
81          }
82      }
83
84    // Print results
85  //  printf (
86  //      "Non-Matching CPU-GPU Outputs Beyond Error Threshold of %4.2f
     ↪  Percent: %d\n",
87  //      PERCENT_DIFF_ERROR_THRESHOLD,
88  //      fail);
```

```
89    return (EXIT_SUCCESS);
90
91  }
92
93  void
94  GPU_argv_init ()
95  {
96    cudaDeviceProp deviceProp;
97    cudaGetDeviceProperties (&deviceProp, GPU_DEVICE);
98  //  printf ("setting device %d with name %s\n", GPU_DEVICE,
     ↪  deviceProp.name);
99    printf ("[running on device %d: %s]\n", GPU_DEVICE, deviceProp.name);
100   cudaSetDevice ( GPU_DEVICE);
101 }
102
103 __global__ void
104 Convolution2D_kernel (DATA_TYPE *A, DATA_TYPE *B, int NI, int NJ)
105 {
106   int j = blockIdx.x * blockDim.x + threadIdx.x;
107   int i = blockIdx.y * blockDim.y + threadIdx.y;
108
109   DATA_TYPE c11, c12, c13, c21, c22, c23, c31, c32, c33;
110
111   c11 = +0.2;
112   c21 = +0.5;
113   c31 = -0.8;
114   c12 = -0.3;
115   c22 = +0.6;
116   c32 = -0.9;
117   c13 = +0.4;
118   c23 = +0.7;
119   c33 = +0.10;
120
121   if ((i < NI - 1) && (j < NJ - 1) && (i > 0) && (j > 0))
122     {
123       B[i * NJ + j] = c11 * A[(i - 1) * NJ + (j - 1)]
```

```
124             + c21 * A[(i - 1) * NJ + (j + 0)] + c31 * A[(i - 1) * NJ + (j
                ↪  + 1)]
125             + c12 * A[(i + 0) * NJ + (j - 1)] + c22 * A[(i + 0) * NJ + (j
                ↪  + 0)]
126             + c32 * A[(i + 0) * NJ + (j + 1)] + c13 * A[(i + 1) * NJ + (j
                ↪  - 1)]
127             + c23 * A[(i + 1) * NJ + (j + 0)] + c33 * A[(i + 1) * NJ + (j
                ↪  + 1)];
128     }
129 }
130
131 void
132 convolution2DCuda (DATA_TYPE* A, DATA_TYPE* B, DATA_TYPE*
    ↪  B_outputFromGpu)
133 {
134
135   cuda_timer t_conv;
136   cuda_timer_init (t_conv);
137
138   DATA_TYPE *A_gpu;
139   DATA_TYPE *B_gpu;
140
141   cudaMalloc ((void **) &A_gpu, sizeof(DATA_TYPE) * NI * NJ);
142   cudaMalloc ((void **) &B_gpu, sizeof(DATA_TYPE) * NI * NJ);
143   cudaMemcpy (A_gpu, A, sizeof(DATA_TYPE) * NI * NJ,
      ↪  cudaMemcpyHostToDevice);
144
145   dim3 block (DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
146   dim3 grid ((size_t) ceil (((float) NI) / ((float) block.x)),
147              (size_t) ceil (((float) NJ) / ((float) block.y)));
148
149   cuda_timer_record_start (t_conv);
150   Convolution2D_kernel <<<grid, block>>> (A_gpu, B_gpu, NI, NJ);
151   cudaCheckKernel()
152   ;
153   cuda_timer_record_stop (t_conv);
```

```
154   cudaThreadSynchronize ();
155   cudaMemcpy (B_outputFromGpu, B_gpu, sizeof(DATA_TYPE) * NI * NJ,
156               cudaMemcpyDeviceToHost);
157   cudaFree (A_gpu);
158   cudaFree (B_gpu);
159
160   cuda_timer_record_get_elapsed_time (t_conv);
161
162   printf (
163       "[trace: n=%d, bx=%d, by=%d, elapsed_Convolution2D_kernel=%0.4f
        ↪  (ms)] ... ",
164       NI, DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y, t_conv.elapsed_time);
165 }
166
167 int
168 main (int argc, char **argv)
169 {
170
171   int n = 4096, bx = 32, by = 8;
172   if (argc > 1)
173     n = atoi (argv[1]);
174   if (argc > 2)
175     bx = atoi (argv[2]);
176   if (argc > 3)
177     by = atoi (argv[3]);
178
179   NI = NJ = n;
180   DIM_THREAD_BLOCK_X = bx;
181   DIM_THREAD_BLOCK_Y = by;
182
183   double t_start, t_end;
184
185   DATA_TYPE* A;
186   DATA_TYPE* B;
187   DATA_TYPE* B_outputFromGpu;
188
```

```
189   A = (DATA_TYPE*) malloc (NI * NJ * sizeof(DATA_TYPE));
190   B = (DATA_TYPE*) malloc (NI * NJ * sizeof(DATA_TYPE));
191   B_outputFromGpu = (DATA_TYPE*) malloc (NI * NJ * sizeof(DATA_TYPE));
192
193   //initialize the arrays
194   init (A);
195
196   GPU_argv_init ();
197
198 #pragma START_TRACING
199   convolution2DCuda (A, B, B_outputFromGpu);
200 #pragma STOP_TRACING
201 //  t_start = rtclock ();
202 //  conv2D (A, B);
203 //  t_end = rtclock ();
204 //  fprintf (stdout, "CPU Runtime: %0.6lfs\n", t_end - t_start); //);
205
206 //  int s = compareResults (B, B_outputFromGpu);
207   int s = EXIT_SUCCESS;
208   if (s == EXIT_SUCCESS)
209     printf ("PASS\n");
210   else
211     printf ("FAIL\n");
212
213   free (A);
214   free (B);
215   free (B_outputFromGpu);
216
217   return 0;
218 }
```

A.2 Annotated PolyBench Code

Below is an example of how KLARAPTOR automatically annotates code for the 2DConvolution benchmark.

```c
#include "2dconv_utils.h"

//////////////////////////////////////
/////// AUTOMATICALLY ANNOTATED ///////
//////////////////////////////////////
#include "kernel_invoker.h"
//////////////////////////////////////
//////////////////////////////////////
const int kernel_info_size_param_idx_Convolution2D_kernel
 ↪  __attribute__((used))  = 2;
const int kernel_info_dim_Convolution2D_kernel __attribute__((used))  =
 ↪  2;
void
conv2D (DATA_TYPE* A, DATA_TYPE* B)
{
  int i, j;
  DATA_TYPE c11, c12, c13, c21, c22, c23, c31, c32, c33;
  c11 = +0.2;
  c21 = +0.5;
  c31 = -0.8;
  c12 = -0.3;
  c22 = +0.6;
  c32 = -0.9;
  c13 = +0.4;
  c23 = +0.7;
  c33 = +0.10;
  for (i = 1; i < NI - 1; ++i)
    {
      for (j = 1; j < NJ - 1; ++j)
 {
   B[i * NJ + j] = c11 * A[(i - 1) * NJ + (j - 1)]
       + c12 * A[(i + 0) * NJ + (j - 1)]
       + c13 * A[(i + 1) * NJ + (j - 1)]
       + c21 * A[(i - 1) * NJ + (j + 0)]
       + c22 * A[(i + 0) * NJ + (j + 0)]
       + c23 * A[(i + 1) * NJ + (j + 0)]
```

```
35          + c31 * A[(i - 1) * NJ + (j + 1)]
36          + c32 * A[(i + 0) * NJ + (j + 1)]
37          + c33 * A[(i + 1) * NJ + (j + 1)];
38   }
39      }
40  }
41  void
42  init (DATA_TYPE* A)
43  {
44    int i, j;
45    for (i = 0; i < NI; ++i)
46      {
47        for (j = 0; j < NJ; ++j)
48   {
49     A[i * NJ + j] = (float) rand () / RAND_MAX;
50   }
51      }
52  }
53  int
54  compareResults (DATA_TYPE* B, DATA_TYPE* B_outputFromGpu)
55  {
56    int i, j, fail;
57    fail = 0;
58
59    for (i = 1; i < (NI - 1); i++)
60      {
61        for (j = 1; j < (NJ - 1); j++)
62   {
63     if (percentDiff (
64          B[i * NJ + j],       B_outputFromGpu[i * NJ + j]) >
           ↪  PERCENT_DIFF_ERROR_THRESHOLD)
65       {
66         fail++;
67         return (EXIT_FAILURE);
68       }
69   }
```

```
70      }
71
72    return (EXIT_SUCCESS);
73  }
74  void
75  GPU_argv_init ()
76  {
77    cudaDeviceProp deviceProp;
78    cudaGetDeviceProperties (&deviceProp, GPU_DEVICE);
79    printf ("[running on device %d: %s]\n", GPU_DEVICE, deviceProp.name);
80    cudaSetDevice ( GPU_DEVICE);
81  }
82  //__global__ void
83  //Convolution2D_kernel (DATA_TYPE *A, DATA_TYPE *B, int NI, int NJ)
84  //{
85  //  int j = blockIdx.x * blockDim.x + threadIdx.x;
86  //  int i = blockIdx.y * blockDim.y + threadIdx.y;
87  //  DATA_TYPE c11, c12, c13, c21, c22, c23, c31, c32, c33;
88  //  c11 = +0.2;
89  //  c21 = +0.5;
90  //  c31 = -0.8;
91  //  c12 = -0.3;
92  //  c22 = +0.6;
93  //  c32 = -0.9;
94  //  c13 = +0.4;
95  //  c23 = +0.7;
96  //  c33 = +0.10;
97  //  if ((i < NI - 1) && (j < NJ - 1) && (i > 0) && (j > 0))
98  //    {
99  //      B[i * NJ + j] = c11 * A[(i - 1) * NJ + (j - 1)]
100 //   + c21 * A[(i - 1) * NJ + (j + 0)] + c31 * A[(i - 1) * NJ + (j +
    ↪  1)]
101 //   + c12 * A[(i + 0) * NJ + (j - 1)] + c22 * A[(i + 0) * NJ + (j +
    ↪  0)]
102 //   + c32 * A[(i + 0) * NJ + (j + 1)] + c13 * A[(i + 1) * NJ + (j -
    ↪  1)]
```

```
103 //   + c23 * A[(i + 1) * NJ + (j + 0)] + c33 * A[(i + 1) * NJ + (j +
    ↪  1)];
104 //    }
105 //}
106 void
107 convolution2DCuda (DATA_TYPE* A, DATA_TYPE* B, DATA_TYPE*
    ↪  B_outputFromGpu)
108 {
109   cuda_timer t_conv;
110   cuda_timer_init (t_conv);
111   DATA_TYPE *A_gpu;
112   DATA_TYPE *B_gpu;
113   cudaMalloc ((void **) &A_gpu, sizeof(DATA_TYPE) * NI * NJ);
114   cudaMalloc ((void **) &B_gpu, sizeof(DATA_TYPE) * NI * NJ);
115   cudaMemcpy (A_gpu, A, sizeof(DATA_TYPE) * NI * NJ,
    ↪  cudaMemcpyHostToDevice);
116   dim3 block (DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
117   dim3 grid ((size_t) ceil (((float) NI) / ((float) block.x)),
    ↪  (size_t) ceil (((float) NJ) / ((float) block.y)));
118   cuda_timer_record_start (t_conv);
119
120
    ↪  ////////////////////////////////////////////////////////////////////
    ////
121   ////////// WARNING: AUTOMATICALLY ANNOTATED REGION BEGINS HERE
    ↪  ////////
122
    ↪  ////////////////////////////////////////////////////////////////////
    ////
123
124
125
126  char kernel_Convolution2D_kernel_0_name[] =
    ↪  "kernel_Convolution2D_kernel_sm_75";
127
128  //launch_params: 3 for grid_dim, 3 for block_dim, 1 for
    ↪  dynamic_shared_mem_bytes;
129  int kernel_Convolution2D_kernel_sm_75_0_launch_params[6];
```

```
130
    ↪ set_kernel_launch_params(kernel_Convolution2D_kernel_sm_75_0_launch_params,
    ↪ grid, block);
131
132 void * kernel_Convolution2D_kernel_sm_75_0_kernel_params[]={&A_gpu ,
    ↪ &B_gpu , &NI , &NJ};
133
134 kernel_invoker(kernel_Convolution2D_kernel_0_name,
    ↪ kernel_Convolution2D_kernel_sm_75_0_launch_params,
    ↪ kernel_Convolution2D_kernel_sm_75_0_kernel_params);
135
136
    ↪ ////////////////////////////////////////////////////////////////////////
137 ////////// WARNING: AUTOMATICALLY ANNOTATED REGION ENDS HERE
    ↪ //////////
138
    ↪ ////////////////////////////////////////////////////////////////////////
139
140  cudaCheckKernel()
141  ;
142  cuda_timer_record_stop (t_conv);
143  cudaThreadSynchronize ();
144  cudaMemcpy (B_outputFromGpu, B_gpu, sizeof(DATA_TYPE) * NI * NJ,
     ↪ cudaMemcpyDeviceToHost);
145  cudaFree (A_gpu);
146  cudaFree (B_gpu);
147  cuda_timer_record_get_elapsed_time (t_conv);
148  printf (
149      "[trace: n=%d, bx=%d, by=%d, elapsed_Convolution2D_kernel=%0.4f
       ↪ (ms)] ... ",      NI, DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y,
       ↪ t_conv.elapsed_time);
150 }
151 int
152 main (int argc, char **argv)
153 {
154   int n = 4096, bx = 32, by = 8;
```

```
155   if (argc > 1)
156     n = atoi (argv[1]);
157   if (argc > 2)
158     bx = atoi (argv[2]);
159   if (argc > 3)
160     by = atoi (argv[3]);
161   NI = NJ = n;
162   DIM_THREAD_BLOCK_X = bx;
163   DIM_THREAD_BLOCK_Y = by;
164   double t_start, t_end;
165   DATA_TYPE* A;
166   DATA_TYPE* B;
167   DATA_TYPE* B_outputFromGpu;
168   A = (DATA_TYPE*) malloc (NI * NJ * sizeof(DATA_TYPE));
169   B = (DATA_TYPE*) malloc (NI * NJ * sizeof(DATA_TYPE));
170   B_outputFromGpu = (DATA_TYPE*) malloc (NI * NJ * sizeof(DATA_TYPE));
171
172   init (A);
173   GPU_argv_init ();
174 #pragma START_TRACING
175   convolution2DCuda (A, B, B_outputFromGpu);
176 #pragma STOP_TRACING
177   int s = EXIT_SUCCESS;
178   if (s == EXIT_SUCCESS)
179     printf ("PASS\n");
180   else
181     printf ("FAIL\n");
182   free (A);
183   free (B);
184   free (B_outputFromGpu);
185   return 0;
186 }
```

Curriculum Vitae

Name:	Taabish Jeshani
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2021 - 2023 M.Sc. Computer Science Ontario Tech University Oshawa, ON 2016 - 2021 B.Sc. Computer Science
Related Work Experience:	Teaching Assistant University of Western Ontario 2021 - 2023 Teaching Assistant Ontario Tech University 2020 - 2021 Analytics Intern PwC Jan. 2020 - Apr. 2020 Analytics Intern Bruce Power Summer 2019

Software

- KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs. A. Brandt, T. Jeshani, D. Mohajerani, M. Moreno Maza, L. Wang. 2023. http://github.com/orcca-uwo/KLARAPTOR.