DISSERTATION

submitted to the

Combined Faculty of Mathematics, Engineering and Natural Sciences

of the

Ruprecht–Karls University

Heidelberg, Germany

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Lorenz Braun

born in

Achern, Baden-Württemberg

Heidelberg, 2023

Automated Partitioning of CUDA Kernels for Multi-GPU Systems

Advisor: Professor Dr. Holger Fröning

Oral Examination: .....

I dedicate this dissertation to my family and friends.

A special feeling of gratitude goes to my loving parents,
Doris and Thomas Braun,
who always stood behind me and my choices.

Abstract

Supercomputers and powerful workstations with multiple GPUs have become the state of the art. GPUs are favored for their immense computational power, high memory bandwidth and energy efficiency for highly parallel workloads. The translation of mathematical problems to multi-GPU compute kernels has already been solved in research by using domain-specific languages and libraries or clever analysis and transformation during compilation. The process of optimizing the partitioning of kernels has received very little attention in research. This work explores the viability of automated partitioning of GPU kernels. The problem is approached by modeling the compute graph of selected applications. The Execution of the applications is simulated on a wide range of systems with different interconnects and GPUs. To cut down on simulation time, a simulator was developed for this specific use case. With the simulation results, simple but per-case individual models were created, with which we show that the application behavior can be well predicted. Results show that the automated partitioning is only 10.17% slower than the optimal partitioning.

Analyzing and improving this problem for real applications depends on good information about the compute kernel. Thus, this work additionally considers the problem of obtaining such information. We use profiling of CUDA kernels to obtain information on instruction counts. A LLVM-based compiler extension providing instruction counts per kernel on PTX level is proposed and evaluated. This approach is the advantage that profiling is much faster compared to NVIDIAs profiler nvprof. The average overhead could be improved by a factor of 10 to 13.2 times the normal execution time.

The metrics of the new profiling approach are used to develop a methodology for kernel performance prediction. Because the metrics of the profiler are GPU-independent, they only need to be measured once, which is a great advantage. 168 kernels from the benchmark suites such as Parboil, Rodinia, Polybench-GPU and

SHOC are evaluated on five GPUs. The models are based on random forests and are built for execution time and power consumption prediction. The evaluation of the model prediction performance is using cross-validation and the results show that the median average percentage error ranges from 8.86 to 52% for time and from 1.84 to 2.94% for power prediction.

Zusammenfassung

Supercomputer und leistungsstarke Workstations mit mehreren Grafikkarten sind mittlerweile Stand der Technik. GPUs werden wegen ihrer immensen Rechenleistung, hohen Speicherbandbreite und Energieeffizienz für hochgradig parallele Berechnungen bevorzugt. Die Übersetzung mathematischer Probleme in Multi-GPU Kernel wurde in der Forschung bereits durch die Verwendung domänenspezifischer Sprachen und Bibliotheken oder durch geschickte Analyse und Transformation während der Kompilierung gelöst. Dem Prozess der Optimierung der Partitionierung von Kernels wurde in der Forschung bisher nur wenig Aufmerksamkeit geschenkt. In dieser Arbeit wird die Machbarkeit einer automatischen Partitionierung von GPU-Kerneln untersucht. Das Problem wird durch Modellierung des Berechnungsgraphen ausgewählter Anwendungen untersucht. Die Ausführung der Anwendungen auf einer breiten Auswahl von Systemen mit unterschiedlichen Interconnects und GPUs wird simuliert. Um die Simulationszeit zu reduzieren, wurde für diesen spezifischen Use-Case ein Simulator entwickelt. Mit den Ergebnissen der Simulation, wurden einfache, aber individuelle Modelle erzeugt, mit denen wir zeigen können, dass das Anwendungsverhalten gut vorhergesagt werden kann. Die Ergebnisse zeigen, dass die automatische Partitionierung nur 10,17% langsamer ist als die optimale Partitionierung.

Die Analyse und Verbesserung dieses Problems für reale Anwendungen hängt von guten Informationen über den Kernel ab. Daher befasst sich diese Arbeit zusätzlich mit dem Problem der Beschaffung solcher Informationen. Wir verwenden Profiling von CUDA-Kernel, um Informationen über die Anzahl der ausgeführten Instruktionen zu erhalten. Es wird eine LLVM-basierte Compiler-Erweiterung präsentiert und evaluiert, die die Anzahl der Instruktionen pro Kernel auf PTX-Ebene ermittelt. Dieser Ansatz hat den Vorteil, dass das Profiling im Vergleich zu NVIDIAs Profiler nvprof viel schneller ist. Der durchschnittliche Overhead konnte um den Faktor 10 bis 13,2 gegenüber der normalen Ausführungszeit verbessert

werden.

Die Metriken des neuen Profiling-Ansatzes werden verwendet, um eine Methodik für die Vorhersage der Kernel-Performance zu entwickeln. Da die Metriken des Profilers GPU-unabhängig sind, müssen sie nur einmal gemessen werden, was ein großer Vorteil ist. 168 Kernel aus den Benchmark-Suiten wie Parboil, Rodinia, Polybench-GPU und SHOC werden auf fünf GPUs ausgewertet. Die Modelle basieren auf Random Forests und wurden für die Vorhersage der Ausführungszeit und des Stromverbrauchs entwickelt. Die Ergebnisse zeigen, dass der durchschnittliche prozentuale Fehler bei der Zeitvorhersage zwischen 8,86 und 52% und bei der Stromverbrauchsvorhersage zwischen 1,84 und 2,94% liegt.

Contents

CHAPTER 1

Taming Multi-GPU Partitioning

1.1 Motivation

Many of today's supercomputers feature multiple GPUs per node. Four of the five best systems in the Top500 list from November 2022 feature four or more GPUs per Node [1–4]. Not only supercomputers make use of multiple GPUs. Cloud providers such as Amazon, Google and Microsoft (MS) Azure offer compute instances with multiple GPUs per Node [5–7]. MS Azure offers up to 8 GPUs, while Amazon and Google offer up to 16 GPUs.

Since multiple GPUs per node in supercomputers is the new normal, it is no surprise that the problem sizes of parallel workloads have immensely grown. Just to give an example: largest problem size classes from NAS Parallel Benchmarks [8] have memory requirements ranging from 12 gigabytes up to 5 terabytes! These are requirements for benchmarks and real-world applications may be different, but a fraction of these problem sizes would still be sufficient to fully utilize most recent multi-GPU compute nodes. When processing multi-GPU workloads the number of participating GPUs is selected from experience. But even for experienced developers, this is a hard task to get right and often requires test runs. This clearly shows a need for a more structured approach for partitioning workloads on multiple GPUs.

1.2 Challenges of Multi-GPU Programming

From formulating a computational problem to processing it efficiently on multiple GPUs several stages have to be completed:

- Translation of the mathematical problem into one or multiple compute kernels.
- Adapting data distribution and kernel placement for the usage of multiple compute devices.
- Partitioning of the workload for efficient execution.

Translating a mathematical problem into a kernel for a single GPU is in itself a difficult task. Managing the data of multiple kernels and multiple devices is very time-consuming and error-prone. For this very reason, few GPU applications are developed with support for multiple GPUs.

This emphasizes the need for good multi-GPU programming frameworks. To the best of our knowledge, there is no framework, which has been widely established or pushed by GPU manufacturers. There are some works, which solve these problems (e.g. Ben-Nun et al [9] and Matz et al. [10]), but these are research prototypes and have not been developed further.

These multi-GPU programming frameworks, do not solve the task of efficient partitioning but merely provide multi-GPU applications which can be partitioned. Now that, there are tools available, which make generating multi-GPU applications easier. Tools that can predict an optimal number of GPUs depending on the workload problem size are desirable. Additionally, how can multi-GPU programming frameworks be made partitioning agnostic?

Just being able to divide a GPU workload into any number of partitions, does not enable partitioning decisions. Metrics with which the performance of kernels can be estimated are required. Device-independent metrics would be preferable. Ideally, there is one framework that handles the multi-GPU code generation, performance metrics, performance modeling and optimizing the partitioning of the multi-GPU application. This way, the power of multi-GPU systems would be made accessible and hopefully used efficiently.

1.3 Methodology

This work will first deal with the big picture of automating partitioning decisions. Our approach is to try to generate models which can predict the execution time of an application on a multi-GPU system with a given problem size and a set number of GPUs. For this task, we develop Pick-Sim a simple simulator to estimate the execution time of multi-GPU compute graphs on varying hardware and use it to develop execution time prediction models. This enables the rapid collection of data in many settings. We consider this as a minimum viable prototype, which can be improved iteratively. Next, the approach is enhanced by diving into sub-problems of the automated partitioning problem. Predicting kernel performance is very important for good partitioning decisions. Thus, we work on providing device-independent metrics for kernels. To achieve this the LLVM compiler framework was leveraged to build CUDA Flux, a CUDA kernel profiler based on instrumentation. The profiler is compared with NVIDIA's nvprof profiler. With these metrics, we develop the GPU mangrove performance prediction methodology. The models created using the metrics as input features are used to predict the performance of CUDA kernel. We collect metrics on a wide range of kernels and with different GPU models. Additionally, the prediction performance is evaluated and set into context with similar recent work.

1.4 Limitations

Because the topic has a lot of ground to cover, the scope of this work has to be limited. This work does not deal with the generation of multi-GPU code or single to multi-GPU code transformation. This problem has already been solved with multiple approaches. Even though, the CUDA compute grid will be sliced in multiple dimensions, we do not probe every possible configuration with a set number of GPUs. The partitioning will only deal with GPUs as whole processing units, instead of breaking them further down into smaller allocatable processing units (such as streaming multiprocessors). Additionally, applications will have few kernels and will not change the number of GPUs participating in the computation. These limitations make the optimization of the number of partitions manageable. While we would like to lift these limitations, we

think that the basics need to be matured to be able to make more sophisticated optimizations.

Dynamic frequency settings are not considered. Lowering the frequency of the core or the memory when either one is not used to full capacity, is a great way to save energy. Neither in the Pick-Sim simulator nor in the GPU Mangrove approach this is considered. The reasons for this are mainly that for this, much more data collection would be required which takes more effort and time. Additionally, strategies for lowering clocks for energy efficiency have already been covered in recent research.

1.5 Structure of this Work

The remaining dissertation consists of five more chapters.

- The background chapter provides the necessary background information on the GPU programming model, related work, and GPGPU workloads.
- Chapter three deals with the big picture problem for automated partitioning of CUDA kernels. It explores the approach of compute graph-based workload partitioning and describes its implementation. It closes with the evaluation and conclusion of the results.
- Chapter four is a deep dive into better profiling of CUDA kernels. It introduces CUDA Flux a LLVM-based profiler and reviews the performance of the profiler and the quality of the metrics provided.
- In chapter five, CUDA Flux is used to collect metrics on a wide range of kernels. These metrics are used to build GPU Mangrove a collection of performance prediction models. The models are evaluated on five different GPUs.
- Finally, chapter six, summarizes the key findings of the dissertations and offers some concluding remarks on the topic of automated partitioning of CUDA kernels on multi-GPU systems.

CHAPTER 2

Background

2.1 GPU Programming Model

GPUs are traditionally used for rendering computer graphics. Because the hardware of GPUs enables substantial parallel computation they are used for computational tasks as well. The GPU programming model leverages the SIMD/SIMT paradigm. A GPU has typically multiple processing units which work in parallel and virtually independently. These properties allow GPUs to process many parallel workloads at very high speed and energy efficiency.

The programming model requires defining compute kernels which are the processing routines running on the GPU. These kernels are implemented using languages such as CUDA or OpenCL, which are specifically designed to leverage parallel accelerator hardware.

Implementing a compute kernel involves breaking the computational workload down into smaller independent tasks, which can be executed in parallel. As the processing is distributed across multiple cores which work in a SIMD/SIMT manner, the programmer has to consider the GPU's architecture and optimize the implementation accordingly.

Since this work is based on NVIDIA GPUs, the following explanations describe the CUDA programming model in more detail.

The CUDA programming language is an extension of C/C++. CUDA source files include the compute kernel definition and may also include host code that

runs on the CPU. The kernel functions are called from the host CPU and are executed by the processing units in a SIMT manner.

Execution of work is divided by thread blocks, also called collaborative thread arrays (CTAs), which are divided further into threads. Threads within the same thread block are executed on the same processing unit and share computational resources.

The processing units of NVIDIA GPUs are called streaming multiprocessors (SMs). Each SM executes at least one thread block and can execute more if there are enough computational resources available.

The thread blocks are organized into a grid of blocks, which defines the total number of blocks a kernel launch will process. Each thread block and thread receive a unique ID which is used to let the threads and blocks process different data.

Figure 2.1 from the CUDA Programming Guide [11] illustrates how the blocks of a grid are executed. The grid defined in the green box can be distributed differently, depending on how many SMs are available for computation. In this example, there are two GPUs with two and four SMs. Each SM can only execute one thread block in this case. Even though, the CUDA programming model is inherently parallel, not all blocks are necessarily processed at the same time. In the case of two SMs, the first two blocks are processed in parallel and after those are finished, the next two blocks are processed until all blocks are done. Likewise, with four SMs per GPU, four blocks are being processed in parallel.

Similar to the architecture of CPUs, NVIDIA GPUs also have a memory hierarchy. In contrast to CPUs, the usage of different types of memory is much more explicit.

There are several types of memory:

- Global memory: the largest and slowest type of memory and equally accessible to all threads of the grid.
- Shared memory: small and fast type of memory located on the SM and only accessible to the threads on the same thread block and SM.
- Constant memory: fast read-only memory which is accessible for all threads.
- Texture memory: specialized memory featuring optimized 2D access and interpolation of data.

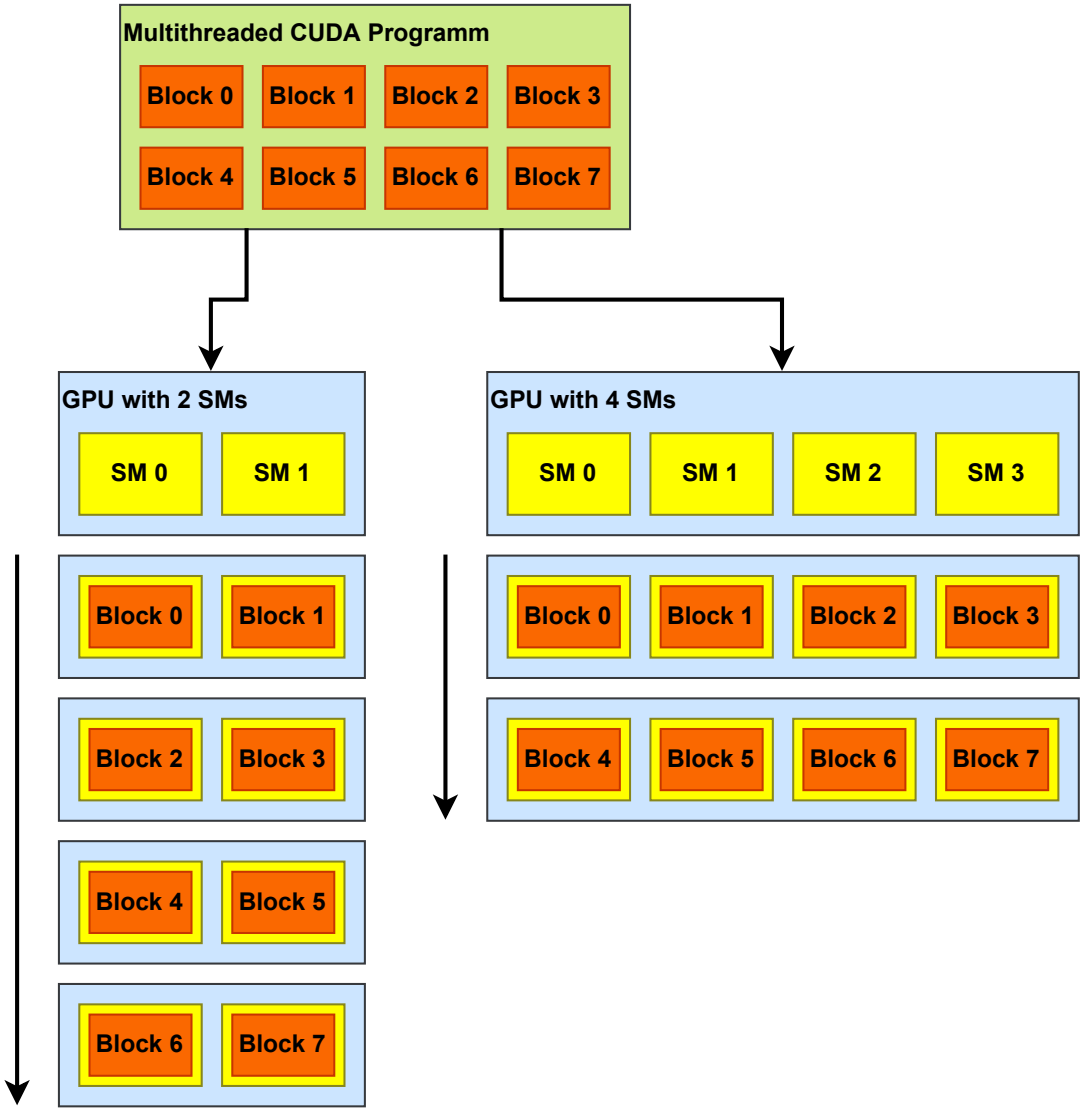

Figure 2.1 Distribution of thread blocks on GPUs with different SM counts [11]

Each of these types of memory has to be used explicitly. This requires the programmer to be aware of the properties of the memory type.

The execution model of a simple CUDA program has typically three stages:

- Copy data from host memory to device (GPU) memory
- Execution of the compute kernel on the GPU
- Copy results back from the device memory to the host memory

More complicated CUDA programs may use multiple compute kernels (e.g. as a synchronization barrier) and use multiple memory copies. Memory copy operations and compute kernel can run at the same time, which allows for optimizing the time needed to process the whole application. Also, multiple GPUs can be used and the GPUs can exchange data with each other without the need of using CPU memory as an intermediate stage.

2.2 Related Work

In this section, important related work is briefly reviewed. Some chapters will review related work more extensively.

The related work for this dissertation can be categorized into three different topics:

- Multi-GPU programming
- GPU kernel profiling
- GPU kernel performance prediction

Multi-GPU programming

There are multiple approaches to automated multi-GPU programming, which we define as generating code that processes an algorithm on multiple GPUs from some sort of description that does not reason about multiple processing units. The code can be generated by either a high-level description, such as a compute graph or an algorithm described by a domain specific language (DSL) or from existing algorithms for single processing units. Distribution and exchanging data for the kernel running on the GPUs is the main problem here. Initially moving data to multiple GPUs and exchanging updates between the devices

requires knowledge of the memory access patterns. Memory access patterns and the resulting requirements for synchronization have to be generated from the DSL implementation of the algorithms or extracted from existing code via analysis or profiling. Optimal memory access patterns do not only depend on the algorithms used but also on the mapping of the grid on the GPUs and the memory layout. When implementing multi-GPU applications with plain CUDA on a single compute node or with message passing on multiple compute nodes, the programmer has to do the tedious work of determining the access patterns and issuing resulting memory transfers.

To alleviate the programmer of this error-prone task several solutions to this problem have been proposed:

- Use of DSLs or libraries which make the memory access patterns known implicitly [9, 12–14].
- Analysis of memory access patterns during compile time [10].
- Sampling of the memory access patterns during runtime [15, 16].

Ben-Nun et al. [9] proposes MAPS-Multi a multi-GPU partitioning framework. His framework introduces smart containers to use when accessing data in a kernel. The developer has to define the properties of the container which let the memory analyzer determine the memory access patterns. Most features of CUDA should be compatible with this approach and a high level of optimization in the kernel is still possible. The memory access patterns supported are derived from the classification of Berkeley's „parallel dwarfs" [17], and thus should give great coverage of known parallel algorithms.

SkePU, which was initially released by Enmyren et al. [18] in 2010 and later redesigned by Ernestsson [14] in 2018, is a high-level skeleton programming framework for heterogeneous parallel systems, which also supports multi-GPU computing. The computation has to be expressed by the developer with predefined skeletons. These skeletons imply memory access patterns similar to MAPS-Multi containers. The high-level approach of SkePU simplifies the programming but will not be able to cover programming patterns not covered by the skeletons. Steuwer et al. [19] introduce SkelCL, a high-level GPU programming framework, which is also making use of skeleton programming. The skeletons perform very similar functions and the memory access patterns are also inferred by the selected skeleton.

Compiler-assisted analysis and container/skeleton-based deduction of memory access patterns work well when memory locations that are accessed are deterministic and do not depend on runtime data.

The analysis of memory access patterns during compilation is a powerful tool with the advantage, that no changes to the source code have to be made. Matz et al. [10] uses polyhedral compilation to model memory access patterns of single-GPU applications to later transform them into multi-GPU applications. The analysis is limited to regular memory access patterns, meaning that memory accesses must not depend on runtime data.

Matz et al. [16] also showed that memory access patterns of various applications can be captured by using an open-source memory tracing plugin for clang/LLVM. While the analysis this tool provides is very detailed, the time and memory overhead for such tracing is of significant magnitude.

A more optimized approach to trace memory accesses at runtime is proposed by Kim et al. [15]. Instead of profiling on the GPU their runtime performs a sampling run on the CPU. For efficiency, they sample only a representative subset of items in the thread grid. This requires some conditions to be met, which means that not all memory access patterns can be analyzed. Also, data-dependent memory accesses can not be computed.

Closing the loop of obtaining memory access patterns and using them to automatically generate multi-GPU code is achieved in [9, 10, 14]. Data-dependent memory access patterns can usually not be covered. Notably, Ernestsson and Matz both use similar components to track and manage data placement dynamically. We think that dynamic management of data dependencies and placement is very important when implementing automated multi-GPU programming because the performance of GPU workloads is most sensitive to orchestrating data (except when data exchange is minimal and/or trivial). This is underlined by results, which were produced using unified memory (UM) for data orchestration. UM allows GPUs with support for paging (CUDA compute capability 6.0 and higher) to fetch data on demand to the GPU memory similar to paging on CPUs. A single memory allocation is sufficient to be used on all devices. Appendix A shows a small study on the subject of unified memory performance regarding multiple GPUs. Comprehensive studies on whether using this particular feature for multi-GPU computing is beneficial are yet to be done. When used with single GPUs, performance is usually lower than using conventional memory

allocation (see figure A.1, Appendix A). Reasons for performance degradation are presumably: latency overhead when fetching pages; and page trashing; when CPU and GPU are modifying memory on the same page. UM gives a functional solution to the problem, but is usually not worth using it as performance is rarely improved.

GPU Kernel Profiling

Related work on the topic GPU kernel profiling includes work of Villa et al. [20], Diamos et al. [21] and Farooqui et al. [22] and several works with GPU simulators like [23–27]. While simulators could be used to profile CUDA kernels profiling of a wide range of kernels is prohibitively expensive and thus, simulators are not explored further.

Villa et al. introduce NVBit, a dynamic binary instrumentation framework for CUDA applications. Their tool allows users to selectively instrument binaries and execute them on GPUs. They show that NVBit can be used to easily implement profilers, performance evaluation, error checking, and bug detection tools. NVBit operates at the SASS assembly level, which is GPU-specific, and uses dynamic recompilation. NVBit enables the following modifications to the code basic-block instrumentation, multi-function injection to the same location, inspection of ISA visible state, dynamic selection of instrumented or uninstrumented code, permanent modification of register state, correlation with source code, and instruction removal. Their work can be viewed as a continuation of the work of Stephenson et al. [28] who proposed a software profiling tool for GPU architectures that can be used to identify performance bottlenecks and optimize GPU applications. They share a common goal of improving GPU performance through profiling and analysis. Stephenson et al. introduce SASSI an instrumentation tool for the assembly SASS. They show several use cases of SASSI, which also includes profiling of conditional branches which is a very effective way to infer the executed instructions of a CUDA kernel. GPUOcelot by Diamos et al. [21] and Lynx by Farooqui et al. [22] offer the possibility to dynamically instrument PTX assembly of CUDA kernels. Instead of instrumenting at compile time they intercept the kernel before it is executed and apply the instrumentation at runtime. This approach is very similar to our work, but their tools are not maintained anymore.

GPU Kernel Performance Prediction

On the topic of GPU kernel performance prediction, Braun et al. [29] provide an extensive list of related work on that topic. In the scope of this work, it is an advantage to be able to make predictions based on static information available ahead of execution time. For this reason, the works of Guerreiro et al. [30] and Fan et al. [31] are the most relevant. Besides using similar features, Guerreiro and Fan use a comparable number of kernels for their study.

Guerreiro et al. [30] use GPU assembly as input for modeling execution time, power and energy consumption of GPU kernel, with respect to dynamic voltage and frequency scaling. They change the core and memory frequencies of the GPUs and predict how this changes the GPU's behavior. They use a deep learning model and use it to find Pareto-optimal frequency settings. Fan et al. [31] follow a very similar approach. They also examine the effect of frequency scaling on the performance of a GPU kernel. Their machine learning models predict speedup and normalized energy and they also try to find Pareto-optimal frequency settings.

2.3 GPGPU Workloads and Benchmark Suites

Parts of this work are based on GPU benchmark suites. This section introduces important benchmark suites for this work.

The Rodinia benchmark suite by Che et al. [32] is a very important collection for GPU computing. It includes a wide range of benchmarks for evaluating the performance of heterogeneous computing systems, including GPUs. The included applications are inspired by the Berkeley Dwarves taxonomy.

The SHOC (Scalable Heterogeneous Computing) benchmark suite by Danalis et al. [33] is also a heterogeneous benchmark suite. The CUDA implementation makes this benchmark collection well-suited for GPU benchmarking. The suite contains low-level benchmarks and implementations of typical basic parallel algorithms.

The parboil benchmark by Stratton et al. [34] is another benchmark suite that is used in this work. The benchmark suite covers a wide range of applications for heterogeneous computing systems and makes supporting additional platforms or compilers easier than most other benchmark suites.

Lastly, we want to introduce the Polybench-GPU Benchmark by Grauer-Gray et al. [35]. The benchmark suite was originally developed for compiler research and is therefore not heavily optimized. The benchmark suite covers important workloads and is not heavily optimized. For this reason, we think that it is an interesting counterpoint to the more optimized implementations and reflects properties of naive implementations, which are also important because many developers may not know how to program GPUs efficiently.

CHAPTER 3

Compute Graph-Based Workload Partitioning for Multi-GPU Systems

This chapter explores a possible solution to the problem of multi-GPU kernel partitioning. The problem is approached by modeling the compute graph, GPU hardware and the interconnect processing of the application. We use simple building blocks to create models for partitioning GPU workloads on specific multi-GPU systems. To do so, we provide a "bare-bones" modeling and simulation of partitioned compute graphs on different multi-GPU systems. The simulation focuses on the key aspects of multi-GPU applications which are kernel execution, memory transfers and synchronization. We use real and simulated benchmark data to build and compare partitioning models. We evaluate our models on various systems with multiple GPUs by leveraging simulation.

3.1 Partitioning Approach

The main objective is a decision process on how many GPUs should be used to compute a sequence of kernels in an application. This task can be broken down to finding the appropriate number of GPUs for a given problem size N.

We emphasize that the execution time of an application is defined as kernel execution time plus communication time in the context of this work. For this reason, we focus on estimating kernel execution time, communication time and the interaction of kernel and communication task of a GPU application. Running a complete application realistically includes time spent on host code, which usually handles I/O and the data to be processed by the GPUs. The host code execution time could change with different numbers of GPUs, but these changes are most likely very small and orders of magnitude lower than time changes in kernel execution or communication. For that reason, this work does not consider host code execution time.

3.1.1 Data Collection

For the best results when building a model, it is usually preferable to work with realistic data which is as close to the real world as possible. Measurements on real hardware, while being desirable, require a lot of effort. Modeling a single application for a specific system requires a lot of measurements because there are many combinations of the factors (e.g. problem size, GPU/interconnect specifications).

Since this work is concerned with the general approach of partitioning and thus should be able to cover many combinations of applications, interconnects, GPUs and problem sizes there must be a solution to collect data faster using than real execution of the workloads.

Because there are only a few systems with different interconnect and GPUs available to us, simulating the applications and systems is a good solution to that problem. Using existing GPU simulators such as GPGPU Sim by Bakhoda *et al.* [23], Barra by Collange *et al.* [25] or Multi2Sim by Gong *et al.* [27] would require even more time than the execution of the application on real hardware.

Selecting the best number of GPUs for a given problem size is the actual goal. Most simulators are therefore too detailed and too slow for us to use. Simulating

only the essentials needed for our approach seems to be a viable option, as it would be fast and allow us to collect more data in less time.

Kernel execution time and communication are very important and must be calculated by the simulation. Including every aspect that affects kernel execution and communication would make the simulation prohibitively complex. For this reason, the calculation of kernel execution time and communication throughput is allowed to be approximate and only needs to scale similarly to real-world behavior. This idealized notion of the multi-GPU system also excludes overheads, which may lead to deviations seen in real application executions. Since the simulation is based on compute graphs, not specific implementations, and the observed overheads may be avoided or hidden by optimized implementations of such compute graphs, this is an acceptable limitation.

The goal is to simulate the application with respect to the behavior of the big O notation. With this in mind, several simplifications can be made:

- a kernel has always 100% throughput for all hardware resources
- no overhead for 2D/3D mem. copies
- no overhead for synchronization

Ideally, the simulation should work on a GPU compute graph that can be easily exported from real applications such that modeling of such applications on arbitrary hardware is possible.

3.1.2 Behavior Modeling and Partition Size Optimization

Our approach uses separate machine learning models for each application, GPU and interconnect combinations. This allows using small models, which brings a few advantages such as:

- Improved accuracy and less over-fitting.
- Fast training times which also allows fast re-fitting of selected models, when there is more data available.
- Better comprehensibility of the models because they have fewer parameters and are less complex.

In order to evaluate the approach properly, multiple applications, GPUs and interconnects need to be examined. Ideally as many as possible.

The prerequisites for the collection of a fair amount of samples for modeling are resource intensive:

- There are not many multi-GPU CUDA applications that allow arbitrary selection of problem size N and the number of GPUs.
- Access to many GPUs is needed. For each model for which an examination is sought a minimum of 4 devices is needed; 8 or 16 devices would be even better.
- Different interconnects of the host systems may have a significant influence on the partitioning performance. Therefore, benchmarking with different interconnect is required.

With limited access to GPUs, host systems and computational time a simulation approach is favorable. For this reason, we decided to simulate the execution of compute graph on virtual hardware, which can be easily exchanged. The compute graphs of each application are calculated and execution is simulated on virtual hardware. To reduce the number of simulations to be done there will be only one partitioning for each number of GPUs in a host system.

For the modeling of the applications, we chose a polynomial regression to fit the model parameters. Training data consists of execution time samples with only two input features problem size n and number of GPUs g. Multiple additive terms $n^x g^y$ are fitted to minimize the error between the model and samples.

The optimal partitioning is found by finding the minimum of the resulting function $t(n, g)$. The optimum is sought concerning n, which simplifies the optimization even more.

Although beyond the scope of this work, we would like to mention that partitioning can be done on partial compute graphs, which allows optimizing the graph execution with changing numbers of GPUs. Finding the optimum while allowing to change the number of GPUs is a much harder problem, but could be solved based on the approach presented here.

3.1.3 Application Selection

The examined applications are roughly at the same complexity level as the SHOC benchmark level 1 [33]. This allows us to reason about workloads with multiple kernels and data movements in between. Applications with larger compute graphs

may not be suited to be executed with a fixed number of GPUs. We choose to study only three applications with very different communication behaviors in order to keep the effort of implementing the generation of compute graphs for the simulation to a minimum. The selected applications are:

- Matrix Multiplication
- 2D 5-point stencil
- N-Body

3.2 Pick-Sim Implementation

Because existing GPU simulators are providing much more detailed information than needed for this work, we developed the **P**lain **I**nter**C**onnect and **K**ernel simulator (Pick-Sim) specifically for our research task. It is composed of the following components:

- Kernel execution time estimation
- Flow-based communication processing
- Application compute graph generator
- Hardware specifications

Together, these components allow for fast approximate simulation of multi-GPU applications. The following sections describe each component in more depth.

3.2.1 Kernel Execution Time Estimation

Kernel execution time can be modeled to a high degree of accuracy. The approximate simulation does not need a very high degree of accuracy and thus the computed kernel execution time can be simplified.

The implementation is required to simulate realistic scaling behavior to changes in workload size (similar to big-o notation) and to calculate the results for the hardware resources available on the GPU.

The implementation is using the idea of a simplified resource consumption model. A compute kernel can not perform better than the theoretical computational resources of the GPU allow. The same idea is also used in the roofline model, which is used to analyze bottlenecks of computational kernels by viewing the attainable GFlops/s with respect to the operational intensity [36]. The speed at which a kernel is executed is not only limited by floating-point operations but could also involve other operations like integer or bit operations. Only looking at one type of resource like the operations per second a processor can perform, will omit other important limiting factors like memory bandwidth.

The list of the considered resources includes:

- Global GPU Memory Bandwidth

- Shared Memory Bandwidth (per SM)
- 64-bit floating-point operations
- 32-bit floating-point operations
- Integer operations
- Logic and bit operations

64- and 32-bit floating-point operations are separated because depending on the GPU type the number of 64-bit ALUs can differ significantly.

The time is a function of hardware resources the GPU provides and kernel properties and is calculated as follows:

$$t(\overrightarrow{kernel\_prop}, \overrightarrow{HW\_res}) = max(\frac{\overrightarrow{kernel\_prop}}{\overrightarrow{HW\_res}}) \tag{3.1}$$

Using the maximum operation and not considering the interaction of the instructions simplifies the time estimation a lot while still respecting the very fundamental limitations for the kernel execution time. The main advantage of this method is the speed of this computation. There are certainly better models for execution time estimation, but these depend on the implementation details on the kernel and in-depth information on the GPU being used. Theoretically, this component can be easily exchanged if more details on kernel and GPU are provided.

3.2.2 Flow-based Communication Processing

Time spent on communication (CUDA memcpy operations) is simulated using a directed graph representing the interconnect of the GPU. The graph comprises different kinds of nodes such as GPU, CPU, Memory and PCIe switches. The connections between these nodes are represented by directed edges in the graph which are annotated with their corresponding bandwidth. Section 3.2.4 describes in detail how these graphs are composed.

The goal is to compute the estimated duration of each of the concurrent transactions, taking into account the reduced bandwidth at share edges. The final duration of the entire transaction is evaluated in the encompassing multi-GPU simulation. Each transaction is allocated a bandwidth which is used by the

simulation to calculate the progress of the copy operation. This problem is very similar to the multicommodity flow problems.

In real GPU interconnects, there are many variables determining how the actual transactions are being executed, such as driver behavior, OS and interconnect implementations. These influences introduce randomness to the exact timings and are ignored as they can not be captured well. As a result, all source-destination connection pairs are treated equally regarding bandwidth allocation. Multiple transfers for the same source-destination connection have to share the allocated bandwidth of the connection.

Listing 3.1 shows how the bandwidth is allocated, which is a key function. The procedure works on the edge properties of the interconnect graph.

Listing 3.1 Allocate bandwidth pseudo code.

```
1  def allocate_bandwidth(graph, unused_bw, connections):
2    link_uses = graph.new_edge_property()
3    for c in connections:
4    link_uses[c.path] += 1
5
6    # available bw for each connection
7    available_bw = unused_bw / link_uses
8
9    for c in connections:
10   min_bw = min(available_bw[c.path])
11   c.bw += min_bw
12   unused_bw[c.path] -= min_bc
13
14   return unused_bw
```

`unused_bw` is initialized with the bandwidths of the graph describing the interconnect. Line 7 shows that the bandwidth is divided by the number of uses for each link. For each connection from a source to a destination, the allocated bandwidth is the minimum of the available bandwidth (see line 10). Depending on the layout of the interconnect there might be still unused bandwidth. For this reason, the procedure can be called until the bandwidth does not change anymore. The bandwidth for each connection will be increased accordingly.

3.2.3 Application Compute Graph Generator

Applications in the simulation are modeled by a directed acyclic graph (DAG) – the compute graph. Each of the generated compute graphs has a start and an end node. The task nodes between the start and end are either kernel or memcpy nodes and are annotated as such. The latter can be further distinguished by processor type of source and destination: host-to-device (h2d), device-to-host(d2h) and device-to-device(d2d).

Figure 3.1 shows an example of such a compute graph for a matrix multiplication. The application is partitioned into two partitions (0,0) and (1,0). The partition ID is a tuple to enable partitioning in two or possibly more dimensions. The graph is partitioned into a number of partitions which is equal to the number of GPUs, which shall process the application. At this stage, the GPUs are still virtual and could be mapped arbitrarily.

Nodes in the compute graph are annotated with the following information:

- name – task name for visualization and debugging purposes
- pid – partition ID
- size – the size of the partition for kernel nodes or size of the memory transactions for copy nodes
- kernel properties – info on the resource usage of the kernel
- src/dest – source and destinations for copy nodes

The compute graphs are synthetically generated for a specific problem size and number of GPUs. The generation is user-defined and has to be implemented manually. In theory, the graphs can also be captured from real applications via instrumentation and profiling. Because there are only a few configurable multi-GPU applications (concerning problem size and GPUs used) implementing instrumentation for this use case remains part of future work. The graph generation was defined for the following selected applications: matrix multiplication 2D 5-point stencil and n-body.

3.2.4 Hardware Specifications

The hardware specification module contains several predefined interconnects and GPU specifications. Figure 3.2 shows such a predefined interconnect with two

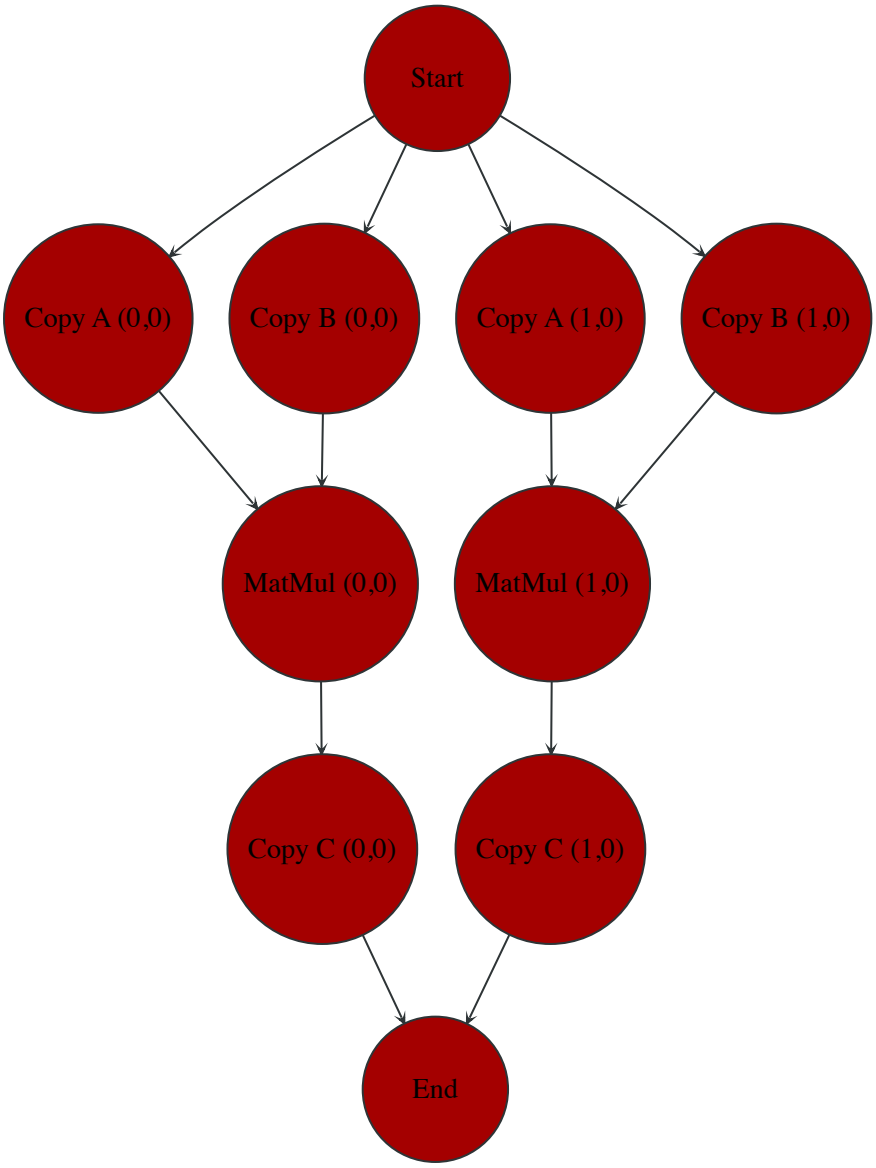

Figure 3.1 Compute graph of the matrix multiplication with two partitions.

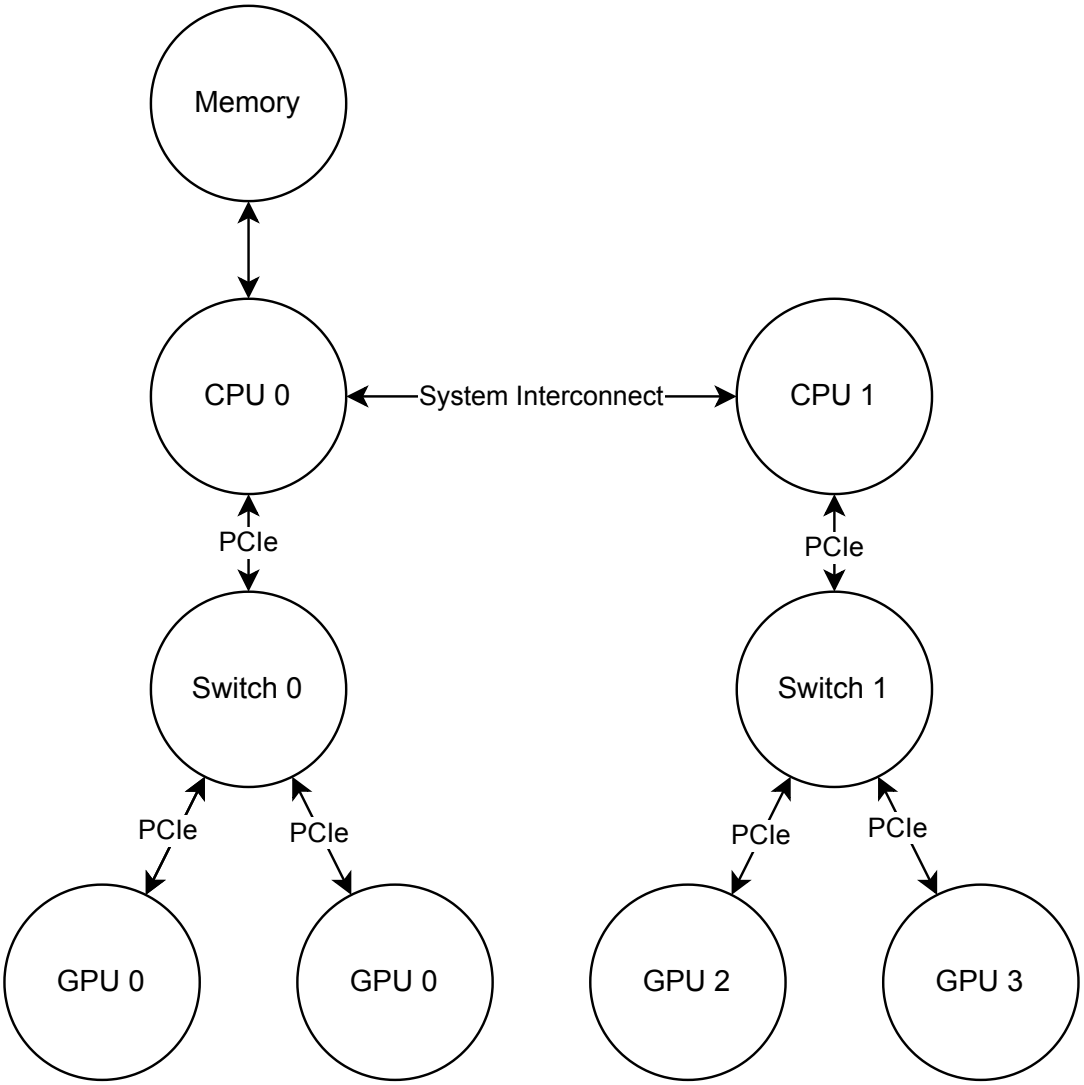

Figure 3.2 Interconnect graph modeling the *octane* dual socket system with four GPUs.

CPU sockets and four GPU slots. The interconnect is modeled as a directed graph and includes a connection between CPU, memory and GPU compute devices. Note that all connections represent two edges (one in each direction). Each connection can be configured with the corresponding bandwidth. This is important because the system interconnect (e.g. Intel QPI or AMD Hypertransport) differs with the type of CPU being used. The edges of the CPU and Memory are sharing their read and write bandwidth.

The interconnect does not define the specific GPU. This is done in the simulation. The connections are given a maximum bandwidth, which can be put through the link. Between CPU and GPU there may be switches or similar

intermediate hops. These are necessary to account for bottlenecks between devices.

3.2.5 Multi-GPU Execution Simulation

The simulation is processing the application compute graph. GPUs are mapped to the logical partitions of the compute graph and the specifics of the selected GPU are applied. The compute graph is processed step by step until the end node is reached.

Listing 3.2 shows the gist of the simulation loop. The compute graph is processed from the start node. Nodes are explored in a BFS search manner as successors of a task node will be added once it is done. There are active, waiting and finished tasks nodes with their corresponding sets in which they are managed. Active task nodes will make progress in each simulation step. Each step ends when the first task is finished. This ensures waiting task nodes can become active as soon as possible and each task node can be processed with the corresponding available resources in the current simulation step. This is important for copy task nodes for which the bandwidth allocation depends on all copies running concurrently (as discussed in flow-based communication). Kernel task nodes are not sharing GPUs and are therefore completely independent and do not interfere with each other. After each task was updated in the simulation step one or multiple tasks should be done (line 13 and following). Active, finished and waiting tasks are managed accordingly. In an intermediate step successors of task nodes are added to the waiting tasks if they are not already active or finished (lines 19–21). In the end, waiting tasks are added to the active task if the predecessors are finished (lines 23–25).

Listing 3.2 Compute graph simulation pseudo code.

```
active_tasks = set(start.successors())
waiting_tasks = set()
finished_task = set()

while not active_tasks.empty():
  bandwidth_allocation(active_tasks)
  # compute time to finish each task
  for task in active_tasks:
    task.compute_task_eta()
  step_time = min([task.eta for task in active_tasks])
  # progress each task with the shortest task time
  for task in active_tasks:
    task.update(step_time)
    if task.done():
      active_tasks.remove(task)
      finished_tasks.add(task)
      # successors of the finished task become waiting,
      # if they are not already active or finished
      successors = task.successors() - active_tasks
      successors = successors - finished_tasks
      waiting_tasks.update(successors)
  for task in waiting_tasks:
    if task.predecessors().issubset(finished_tasks):
      active_tasks.add(task)
      waiting_tasks.remove(task)
```

3.3 Modeling System-Application Interaction with Polynomial Regression

Each combination of application and system is modeled separately. This has several advantages:

- Model Size – only a few model parameters are needed
- Model Simplicity – model parameters can be interpreted better

- Low use of computational resources – model fitting and usage of the model are fast

For the above reasons, a linear regression model was chosen. To allow modeling more complex behavior the input parameters N and g are augmented with polynomials. The exponent for g ranges from -1 to 3 and for N from 0 to 3. This should allow for covering most applications' behavior.

The implementation of the system-application model is kept simple by restriction to regression models. Feature engineering is employed to model non-linear behavior by augmenting the features. This approach uses polynomial features, but other (preferably derivable) features are also possible. The features are augmented by computing the cartesian product of n and g vectors with different exponents

When applying the regression model it is favorable to get a simple model with fewer parameters. For this reason, the LassoLars regression is used [37]. The algorithm provides a good fit and regularization so that only important features remain.

Predictions for the optimal number of GPUs g(n) are quickly calculated. The usually low number of GPUs allows the enumeration of all possible settings and picking the lowest one. For large numbers of GPUs t(g, n) can be derived to compute the minimum analytically.

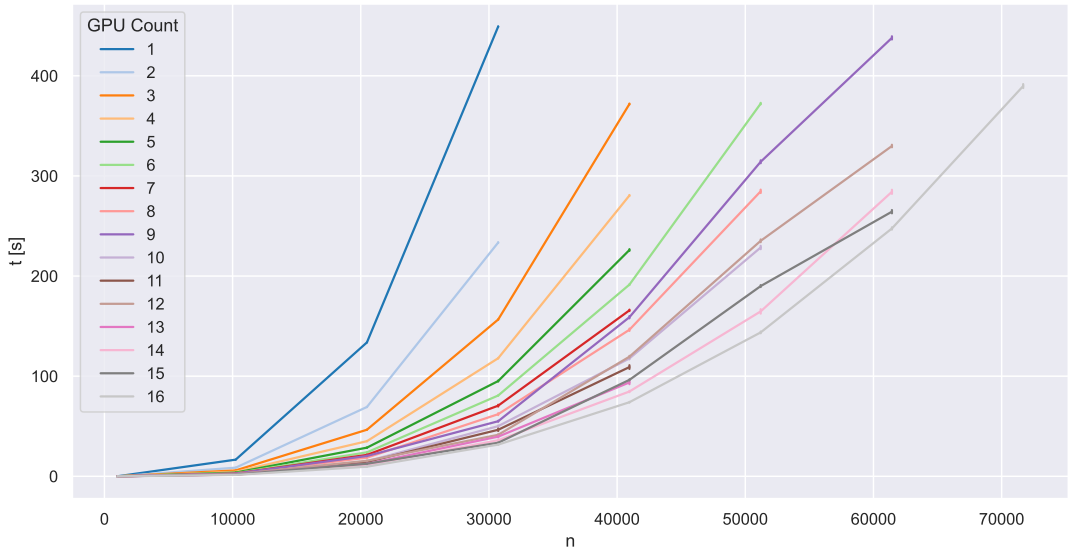

Figure 3.3 Performance measurements of the matrix multiplication application with varying problem sizes and up to 16 GPUs. Measurements were executed on the victoria system and repeated five times for statistical stability.

3.4 Evaluation

The evaluation involves several stages of the approach presented in this chapter. First, benchmark data from real hardware is examined. This is followed by a comparison of the simulation data with the real benchmark data. Next, linear regression is applied to generate time prediction models, and finally, the optimization of the number of GPU devices is evaluated.

3.4.1 Benchmark Results on Real Hardware

Figure 3.3, 3.4 and 3.5 show the measured execution times over the problem size with 1 to 16 GPUs used for the applications matrix multiplication, n-body and the 2D 5-point stencil. Each data point was measured five times and the median of the results is used. The error bars show the standard deviation. The error of the execution time measurement is very small, so the error bars are very hard to see (if even visible at all). Some lines for lower numbers of GPUs end early, because there is not enough device memory for larger problem sizes.

The results of the matrix multiplication (figure 3.3) are mostly as expected. Some lines do cross each other at certain thresholds. This suggests that the approach of simply increasing the number of GPUs with the problem size can be

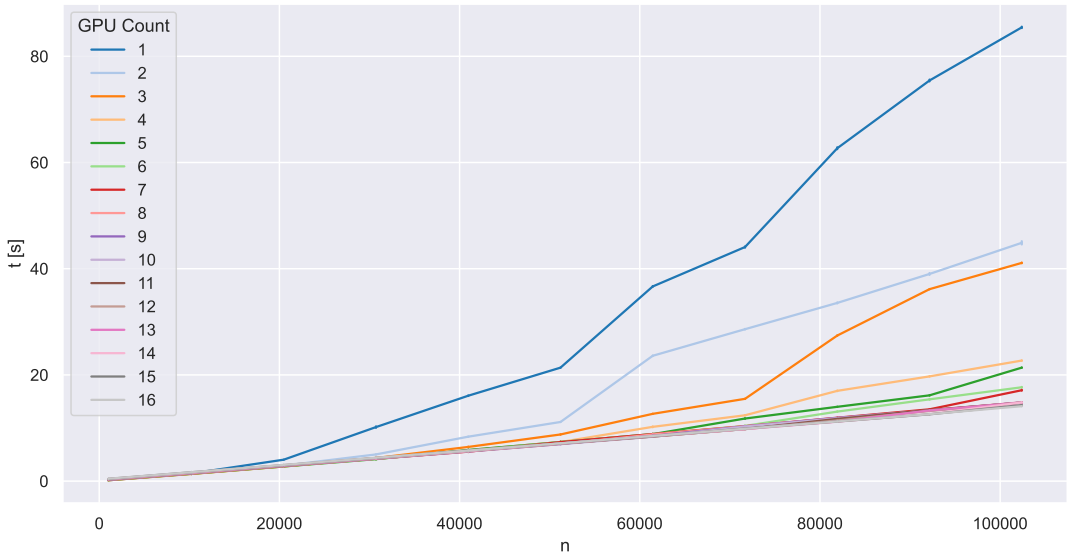

Figure 3.4 Performance measurements of the n-body application with varying problem sizes and up to 16 GPUs. Measurements were executed on the victoria system and repeated five times for statistical stability.

optimized. Here are some examples: At n = 40960, eight GPUs outperform nine, and 14 GPUs outperform 15. Only at n = 61440 are 15 GPUs faster than 14 GPUs again.

The n-body results in figure 3.4 show that the problem can easily benefit from almost any number of GPUs because it requires a lot of computational resources and communicates very little. The graph shows that the interesting problem sizes for partitioning tasks are the smaller ones, where the overhead is more important. Optimizing the number of GPUs may not seem profitable, but small differences can add up at high iteration counts. The measured data contains one hundred iterations.

The results of the stencil application in figure 3.5 are not as expected. The time required for computation and communication increases approximately linearly. Since the computation increases quadratically and the communication increases linearly, it can be concluded that communication time is the dominant factor for the stencil application. Using the one GPU measurement (blue line in the graph) as a baseline, the other measurements can be divided into two groups. All lines above the one GPU measurement are measurements that are slower despite having more processing power available. The lines below the blue line are measurements that run faster. The reason for this is not directly the number of GPUs. Since the stencil grid is two-dimensional, the computational grid can

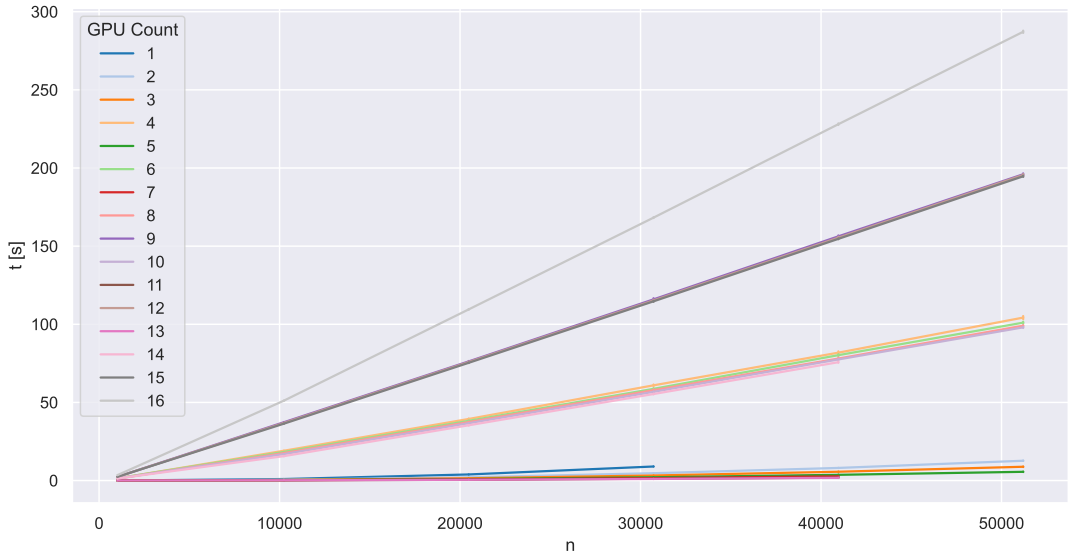

Figure 3.5 Performance measurements of the stencil application with varying problem sizes and up to 16 GPUs. Measurements were executed on the victoria system and repeated five times for statistical stability.

also be divided into two dimensions if the number of partitions can be factorized. Partition counts that are prime numbers run faster because the stencil grid is only sliced in such a way that the halos to be exchanged between the GPUs are continuous in memory. The other partition counts have halos that must be accessed with a stride. Although the cudaMemcpy2D functions from the CUDA runtime were used, this resulted in severe performance penalties and thus long runtimes. To test the ability of the models to cover such difficult behavior, it was decided to use this benchmark data anyway.

The benchmark data was used to fit models for predicting execution time. To get a good visual impression of the fit 3D plots are provided in figures 3.6, 3.7 and 3.8. The models seem to capture the behavior quite well. The surface is generally smoother, which is to be expected because the models have very few parameters.

The plots in figure 3.6 are visually almost indistinguishable, indicating a good fit. A few large problem sizes show more changes in execution time in the benchmark data compared to the simulation data.

Figure 3.7 shows a similar picture, where benchmark data and the model correlate very well. Again, the model is smother than the benchmark data.

Figure 3.8 has very different plots for the benchmark data and the model. Clearly, the model is unable to reproduce the behavior. Given the severity of

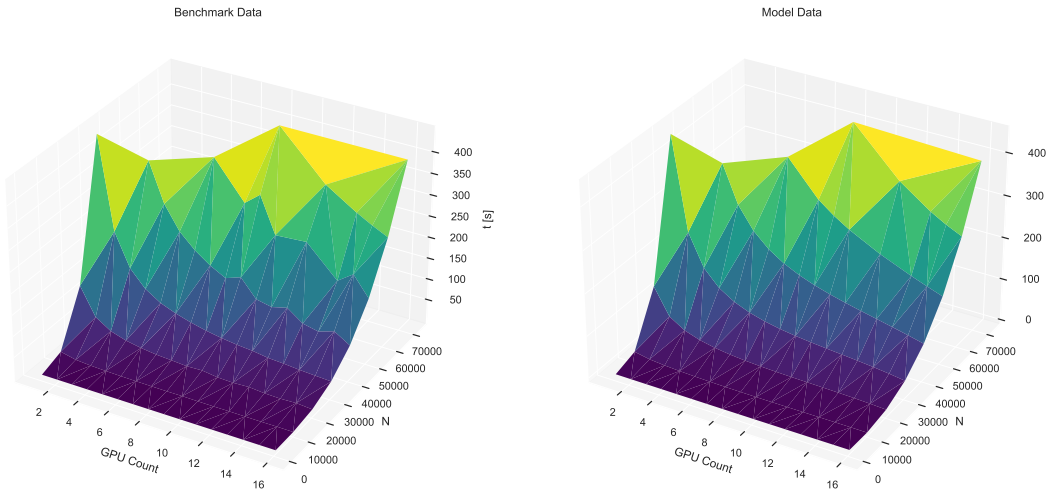

Figure 3.6 Two 3D surface plots showing execution time (z-axis) over the GPU count (x-axis) and problem size N (y-axis) for the **matrix multiplication**. The left plot shows the median of the benchmark data. The right plot shows the data from the model.

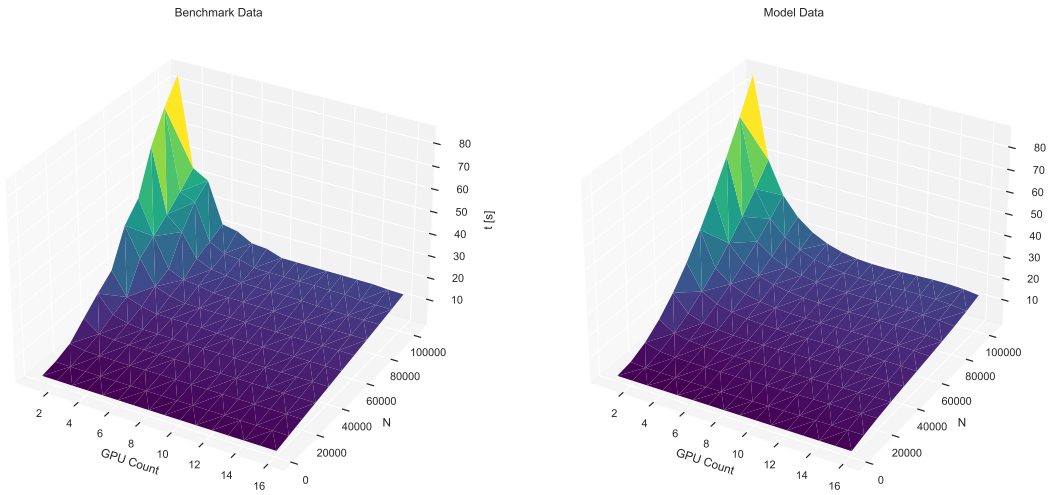

Figure 3.7 Two 3D surface plots showing execution time (z-axis) over the GPU count (x-axis) and problem size N (y-axis) for **n-body**. The left plot shows the median of the benchmark data. The right plot shows the data from the model.

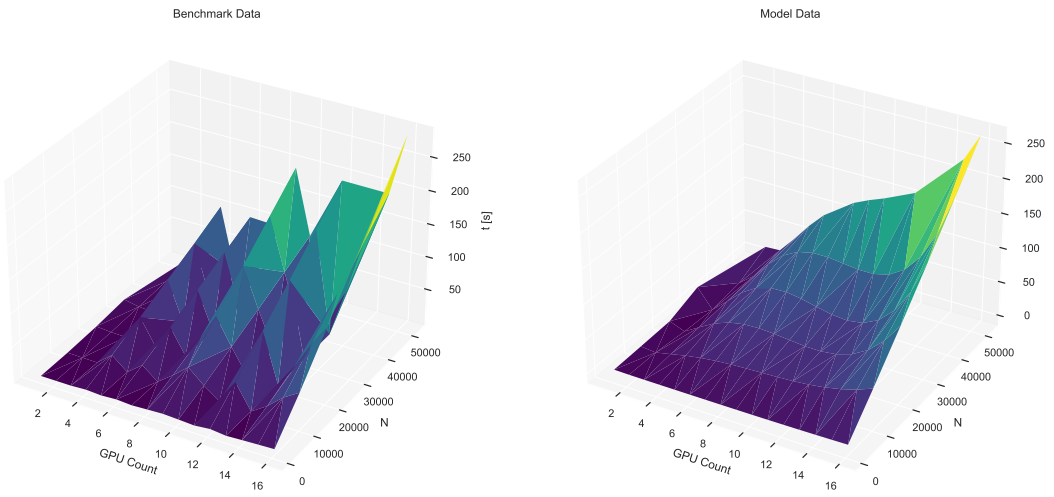

Figure 3.8 Two 3D surface plots showing execution time (z-axis) over the GPU count (x-axis) and problem size N (y-axis) for the **stencil**. The left plot shows the median of the benchmark data. The right plot shows the data from the model.

the performance penalty, this is acceptable, as optimized implementations would certainly avoid this *performance bug*.

In addition to the visual evaluation, a quantitative result is needed to judge the fitness of the model. The error is calculated in percent and displayed in the box plot of figure 3.9. In most cases, the model slightly underestimates the execution time by a few percent. The error for the stencil code is generally higher. For matrix multiplication and especially for stencil code, the range of error is quite high. This should be taken with a grain of salt, as the error is computed in percent, and low absolute errors for small problem sizes can lead to very high percent errors. The mean error percentage is -95.82% while the mean absolute error is 6.74 seconds. Without the stencil application, the mean error percentage is -1.12% and the absolute error is 1.04 seconds.

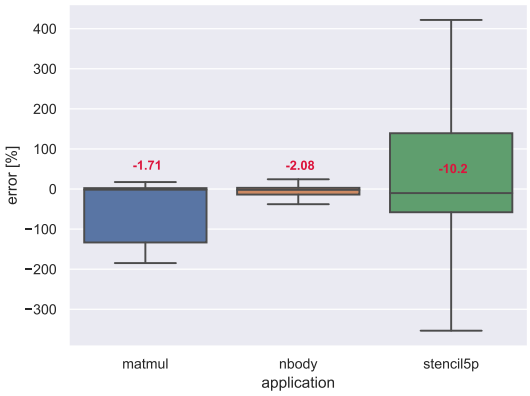

Figure 3.9 Boxplot of the model error in percent over the applications matrix multiply, n-body and stencil on the victoria system. The median is annotated in red for better readability. Whiskers are limited to 1.5 of the interquartile range, outliers are not shown.

3.4.2 Comparing Measured Data with Simulation Data

To compare the scaling behavior of the execution times of benchmark and simulation data, the time is scaled by the problem size. Figure 3.10, 3.11 and 3.12 show the scaled execution time over the number of GPUs. The number of GPUs on the x-axis is preferred because we want to scale by n and show how the number of GPUs influences time. The top plot shows the scaled time of the benchmarks and the bottom plot the time of the simulation. Each line represents a different problem size n.

The plot for the matrix multiplication used the log scale on the y-axis as this makes slight differences in execution time better visible. We can make the following observations:

- Larger problem sizes take longer because matrix multiplication has a complexity of $O(n^2)$.
- Using more GPUs improves performance, if there is enough work.
- Not all aspects of the application can be fully parallelized, which results in diminishing performance improvements when increasing the number of GPUs.
- Besides the effect of Amdahl Law, increased overhead can be observed which is very pronounced in the simulation results of the problem size 1024.
- Comparing the performance delta of an even count of GPUs with the next

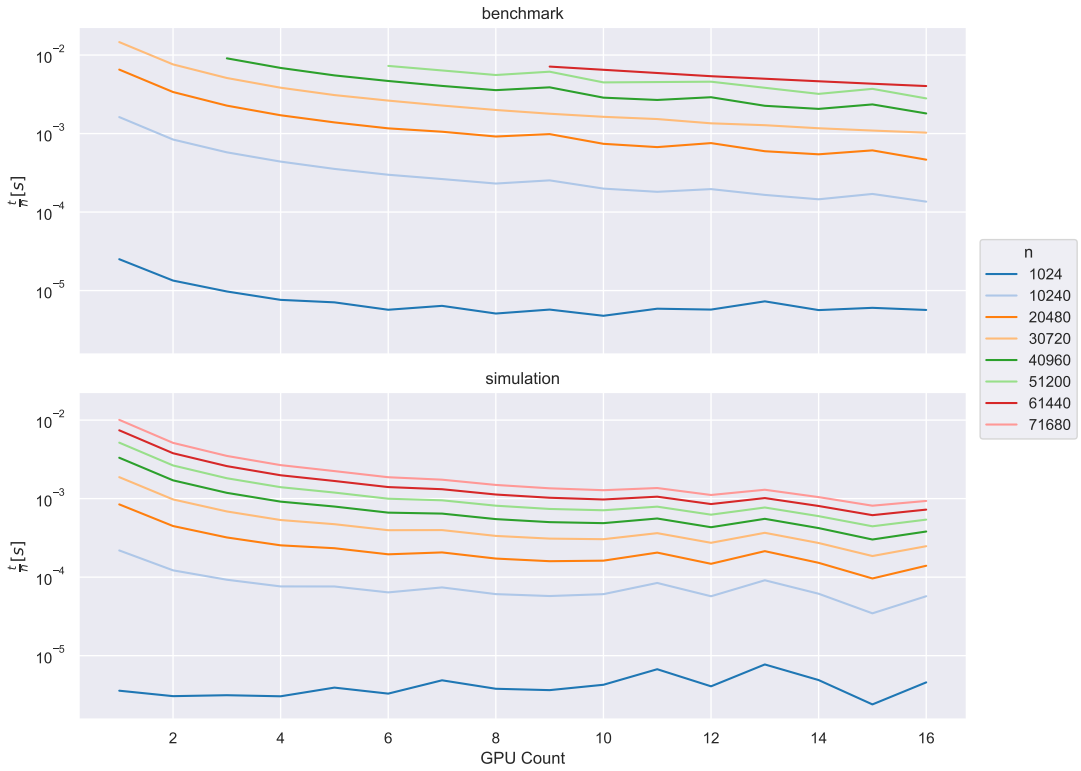

Figure 3.10 Two plots showing the execution time t normalized by the problem size n of matrix multiplication over the number of GPUs used for computation. The top shows the simulation results and the bottom the benchmark results. Log scale is applied on the y-axis.

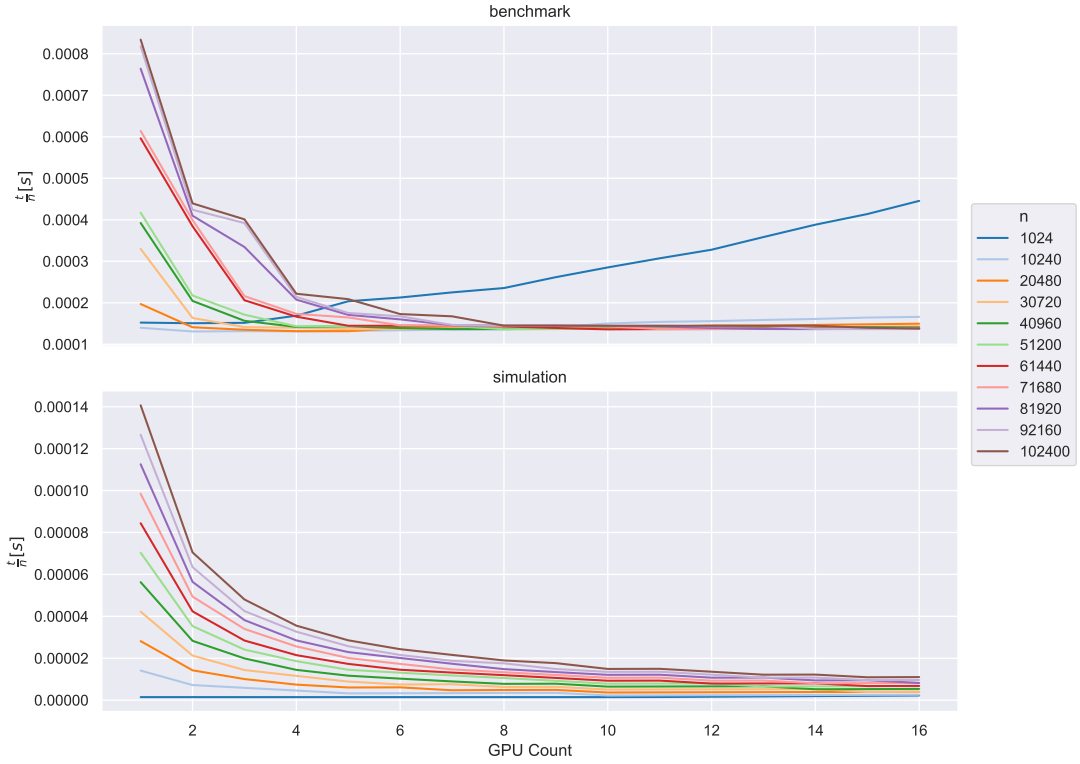

Figure 3.11 Two plots showing the execution time t normalized by the problem size n of n-body over the number of GPUs used for computation. The top shows the simulation results and the bottom the benchmark results.

odd count of GPUs reveals a very low gain or even loss in performance. This is explained by the higher communication overhead of usually not being able to partition the C matrix in two dimensions among the GPUs.

- This effect is stronger in the simulation. The benchmarked results show this effect to a lesser extent because there are additional overheads that are not reproduced by the simulation (such as copying data slices that are not stored continuously in memory).
- The simulation and benchmark results show similar behavior and often similar results regarding the optimal number of GPUs. The overhead of real execution is most likely not affecting computation and simulation to the same extent. This may slightly shift the optimal number of GPUs.

For the n-body application, the observations are as follows:

- N-body is a compute-intensive application with little communication. Thus, it can benefit multiple GPUs even with low problem sizes.

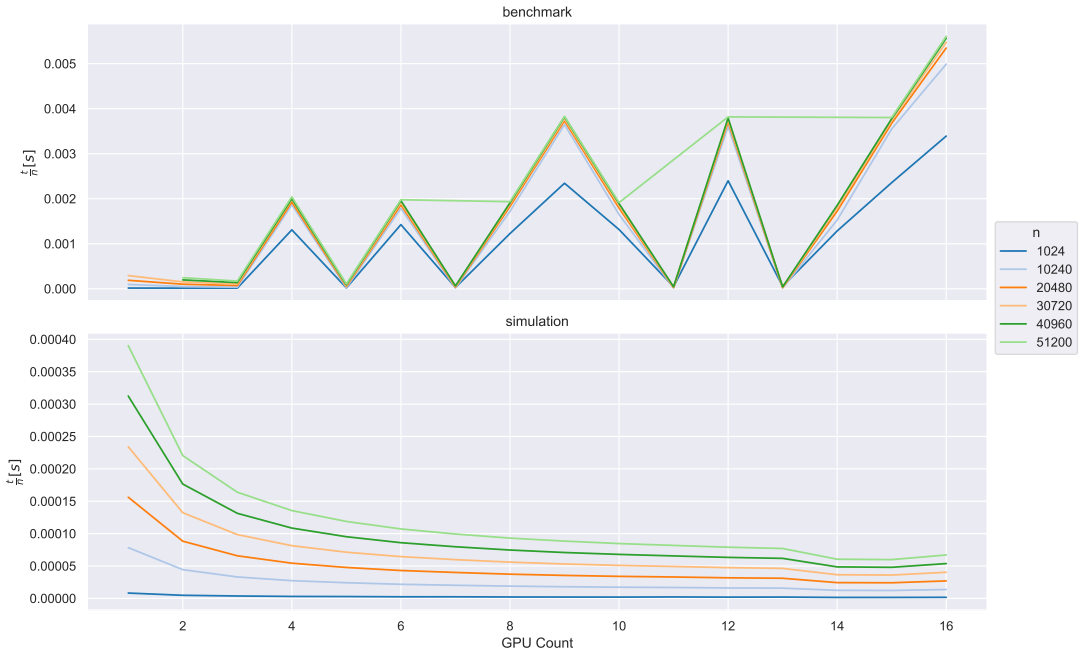

Figure 3.12 Two plots showing the execution time t normalized by the problem size n of the stencil computation over the number of GPUs used for computation. The top shows the simulation results and the bottom the benchmark results.

- The simulation scales very well, due to the lack of overhead.
- The benchmark results show that communication overhead can outweigh performance gains when the problem size is low.
- The effect of using too many GPUs is not very strong and does not affect the performance a lot but decreases the efficiency significantly.
- The maximum problem size is suitable for processing with about eight GPUs. Beyond that, the gains are quite low. Larger problem sizes would scale better.
- Using 3 GPUs is performing worse than expected. We suspect that the rather short communication time of this workload is sensitive to the processing order of the memory transfers and overheads in general.

Stencil observations:

- Our stencil implementation behaves very differently in simulation and real execution.
- As explained earlier this behavior comes from very high overhead from exchanging unaligned halos.

- The simulation results show, that the application could scale quite well if it was optimized better.
- All GPU counts, which cannot be factorized performed well, because the grid is only split along the y-axis and the slices to copy are continuous in memory.

3.4.3 Evaluating Model Fit on Simulation Data

With the simulation able to reproduce the behavior of real benchmarks, the simulation results are used to evaluate the model fit on a large combination of applications, GPUs and interconnects. The applications used include matrix multiplication, n-body simulation and a 5-point 2D stencil, as used in this work before. As GPUs, we used the NVIDIA GPUs Tesla K20, Tesla K80, Tesla P100, Tesla V100 and Tesla A100. As interconnects, we used three different systems modeled after real hardware:

- octane: 2 CPU sockets, 4 GPUs
- victoria: 2 CPU sockets, 16 GPUs
- brook: 1 CPU socket, 8 GPUs

The results are used to assess model error for prediction application runtime.

Figure 3.13 shows a box plot of the error in time prediction for the fitted model of the simulation results. The error in the average case is very close to zero. The distribution of the error is slightly more spread out for the matrix multiplication.

The error distribution on the different interconnects is shown in the box plots of figure 3.14. Again, the error of the average case is very close to zero. The victoria interconnect has the widest error spreads, followed by brook and then octane which has almost no spread at all. This observation has an easy explanation: The victoria interconnect can host up to 16 GPUs, brook only eight and octane only four GPUs. Fewer GPUs seem to allow better fitting of the model parameters.

Finally, table 3.1 shows the main quartiles of the error of the model fit over all simulated samples. These numbers serve as a summary of the model fit. There are three columns with the error in percent, absolute error and squared error. The median (second quartile) is almost zero for any error metric, which is to

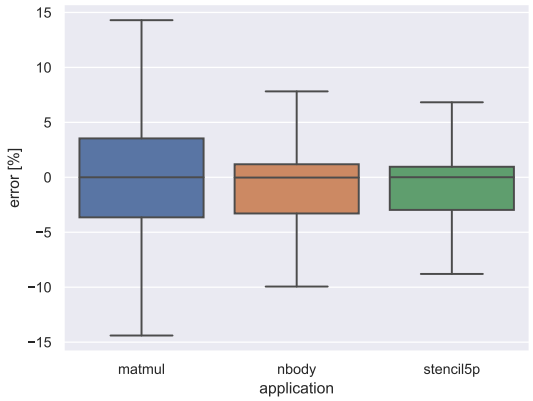

Figure 3.13 Boxplot of the model error in percent over the applications matrix multiply, n-body and stencil on all simulated systems. Whiskers are limited to 1.5 of the interquartile range, outliers are not shown.

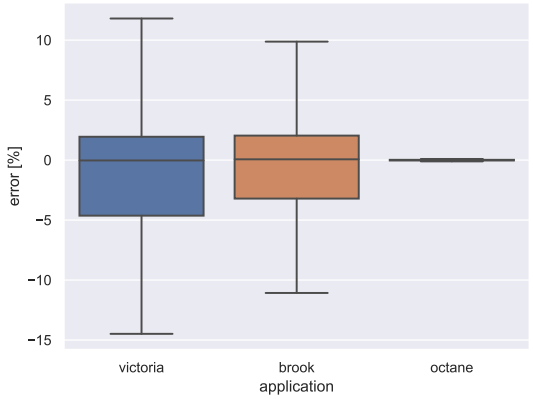

Figure 3.14 Boxplot of the performance loss of the optimal partitioning according to the model compared to the actual optimal partitioning over the simulated interconnects. Whiskers are limited to 1.5 of the interquartile range, outliers are not shown.

	error [%]	absolute error	squared error
Q1	-3.17	0.00	0.00
Q2	0.00	0.02	0.00
Q2	1.48	0.28	0.08

Table 3.1 First, second and third quartile of the model fit over all simulation samples.

be expected from the results shown in figure 3.13 and 3.14. The first and third quartiles also show that the majority of data points of the models are within -3.17 and 1.48 percent. The mean absolute error is 0.74 seconds and the mean error percentage is -0.73%.

3.4.4 Compute Device Optimization Performance

Similar to the previous section figure 3.13 and 3.14 show box plots of the relative performance as multiple of the actual best time over the applications and interconnects. The majority of the predicted optimal number of GPUs is within the range of about 1.2 and 1.6 of the actual optimum. The average case is about 1.1 times slower for matrix multiplication and n-body. The average relative performance of stencil code is only slightly higher than the optimum.

Figure 3.14 shows that the interconnects octane and brook are much better optimized than the victoria interconnect, in which the average case is about 1.3 times slower than the optimum. With only four GPUs the octane interconnect seems trivial to optimize. The quite substantial number of eight GPUs hosted by the brook interconnect is performing well. Most predictions are within 1.0 and 1.2 of the optimum. The victoria interconnect on the contrary is much harder to predict correctly. Most of the predicted optimums differ substantially from the actual optimal time.

Finally, table 3.2 shows the main quartiles of the differences regarding the GPU device count optimization of the model fit over data points. These numbers serve as a summary of the optimization results. There are three columns with the differences in GPU count, absolute time difference and time difference in percent. The median of the GPU difference is one, which is not as good as one can hope. But the third quartile is also one which means that in the majority of cases, the number of GPUs is the same or off by one. However, this may be okay if the time difference in these cases is not too high. For this reason, the time

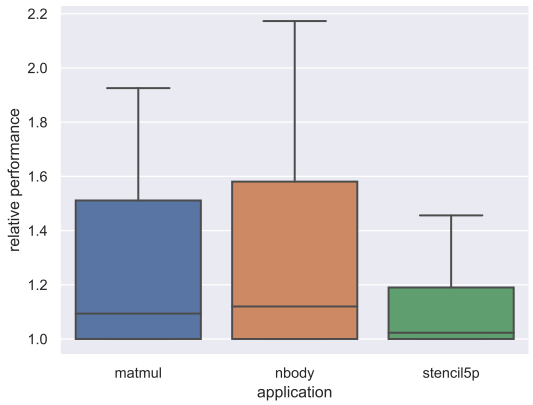

Figure 3.15 Boxplot of the performance loss of the optimal partitioning according to the model compared to the actual optimal partitioning over the simulated applications. Whiskers are limited to 1.5 of the interquartile range, outliers are not shown.

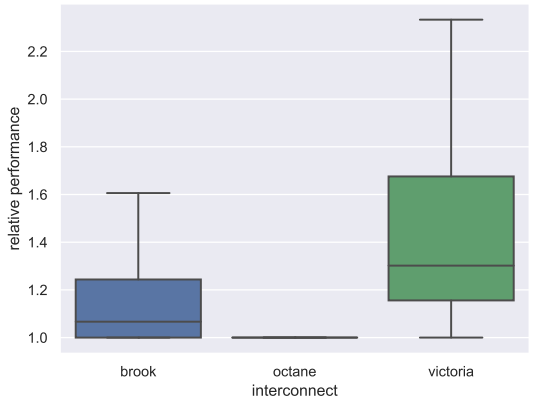

Figure 3.16 Boxplot of the performance loss of the optimal partitioning according to the model compared to the actual optimal partitioning over the simulated interconnects. Whiskers are limited to 1.5 of the interquartile range, outliers are not shown.

	GPU count difference	time difference [s]	time difference [%]
Q1	0	0.00	0.00
Q2	1	0.02	3.99
Q3	1	0.93	30.15

Table 3.2 First, second and third quartile of optimization differences over all data points.

difference in seconds and percent are also important metrics. Considering the other time metrics, it can be seen, that the absolute time difference is very low. The percentage difference is ranging from 0.00% for the first quartile to 30.15% for the third quartile. The mean time difference is 10.17%.

3.5 Conclusion

Graph-based workload partitioning can be a powerful tool for optimizing the execution of multi-GPU workloads. Optimizing multiple multi-GPU applications for various compute systems with different interconnects and GPU models can be prohibitively expensive. Thus, conventional simulation and kernel profiling do not solve this problem.

We have shown that a simple approach with a building block-like implementation is feasible. Benchmark data from real hardware showed that even applications with lower computational requirements can benefit from optimized partitioning. We have shown that a simple regression model can reproduce the scaling behavior of the application quite well. For real benchmark data, the mean average error percentage is -95.82% and the mean absolute error is 6.74 seconds. We attribute this high percentage error to the odd behavior of the memory transfers of the stencil application. The simulated benchmarks could be modeled much better with a mean average error percentage of -0.73% and an absolute error of 0.74 seconds. The implementation and use of Pick-Sim allowed the collection of a large amount of data with different interconnects and GPUs. Even with the basic implementation of a GPU simulator, the behavior shown is quite similar. We want to emphasize that parts of the simulation, such as kernel execution time estimation or interconnect throughput calculation, can be easily replaced with more accurate building blocks that are closer to the real application behavior. The models built on the simulation data allow for partitioning decisions that could predict the optimal number of GPUs in many cases, with little performance penalty when not predicting the optimum. The average performance penalty over all simulated GPU, interconnect and application combination is 10.17%.

There is certainly room for improvement and more ideas to explore:

For some use cases with very short execution time per run, simulation may not be needed and the model could work with execution statistics alone. For use cases with long execution times, a hybrid approach that uses simulation and execution statistics to model application behavior would be advantageous. With a model that can adapt as more real data is captured, auto-tuning of partitioning becomes feasible, which may be worthwhile for larger compute graphs where sub-graphs are to be optimized.

For these hybrid approaches, the execution times should match the actual

application, not just scale similarly. This requires accurate kernel time estimation and interconnect simulation. While this is computationally expensive, long-running applications may still benefit. In general, API behavior should also be considered, as we have seen in the stencil application.

We believe that the ideas in this work should be further explored and tested on real-world applications with large compute graphs. We know that this is a difficult task and that there are many sub-problems to solve in order to improve partitioning for multi-GPU applications. The pursuit of compute graph-based partitioning will hopefully not only make applications run faster and more efficiently on different hardware but also shape the way multi-GPU applications are developed. In this process, we would like to move away from low-level placement of compute tasks and data onto compute devices to high-level task descriptions that can be scheduled similarly to processes on CPUs.

Chapter 4

Benefits of Instrumentation for Profiling of CUDA Kernels

This chapter is based on work of Braun et al. - „CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications" [38].

Profilers are important development tools for programmers in the CPU programming domain. This is also true for other processors like for example GPUs In 2013 nvprof was added to the CUDA SDK Version 5.0 [39]. Since then it is a great tool for collecting additional data on executed CUDA binaries, which also included the possibility to use hardware performance counters to analyze the performance of CUDA kernels.

With a relatively large overhead for nvprof instruction counter profiling, collecting large amounts of instruction counter samples was impractical. For CPUs, it is not uncommon to use instrumentation for detailed profiling of binaries under execution. Thus, the same principle can be applied to GPU binaries / CUDA kernels.

This leads to the question, of how profiling on GPUs implemented by instrumentation of the kernel code compares to conventional profiling with nvprof. Additionally, since CUDA is inherent parallel, this also raises the question if sampling only a subset of the executed threads would be a viable trade-off for faster execution but less accurate results. Finally, not only the performance is important, but also the quality of the results. How accurate are the results in practice and might there be even more detailed information available without

large performance penalties?

There is little public work using instrumentation for CUDA kernels. Lynx[22] which is based on GPU Ocelot[21] and was released in 2013[40] and is not actively developed anymore and also currently not available for download[1]. There is also NVBit[20], which can be downloaded at GitHub[2].

Because of the lack of available tools for instrumentation, CUDA Flux was developed. This chapter introduces the tool CUDA Flux and the evaluation of its performance and the quality of its results.

[1] URL: https://code.google.com/archive/p/gpulynx/ - 09.05.2022
[2] URL: https://github.com/NVlabs/NVBit

4.1 The ecosystem of CUDA kernel profiling

This section will give background information on the LLVM compilation framework which is used to compile and instrument CUDA kernels. The level of abstraction when profiling is also discussed. Finally, this section ends with an overview of similar tools which can characterize CUDA kernels.

4.1.1 LLVM and gpucc

With the introduction of gpucc [41] it became possible to compile CUDA applications with the LLVM compiler framework [42]. With LLVM version 3.9 gpucc was integrated such that clang is natively able to compile CUDA applications [43].

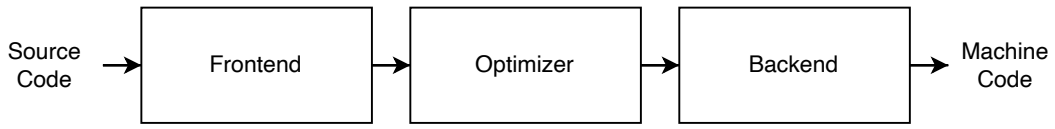

Figure 4.1 The LLVM compilation pipeline is composed of frontend, optimizer and backend.

The three most important modules of the framework are the frontend (e.g. clang for C/C++/CUDA), optimizer or middle"end" (code-analysis, -transformation and -optimization) and backend (emitting machine code for target architecture), see figure 4.1. The framework supports many programming languages and target architectures as it is designed to be independent of language or target features. This is greatly helped by the well-defined intermediate representation (IR) of the code, which is target-independent and serves as an interface between different modules. As a result, the compilation pipeline can be easily extended and modified. Also, most so-called passes which perform code analysis, transformation and optimization can be reused. This leads to very similar compilation pipelines for CUDA and C/C++ code.

CUDA code is compiled in two separate compilation pipelines – one for host (CPU) code and one for device (GPU) code. Figure 4.2 shows the compilation of CUDA code. Device code generation has to run first and yields a fat binary which is included in the binary which contains the host code.

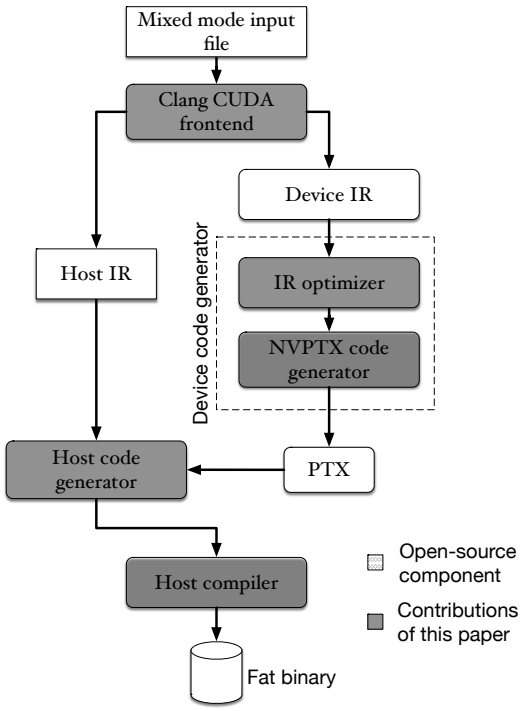

Figure 4.2 Compilation of CUDA code with LLVM in two separate compilation pipelines by Wu et al. [41].

4.1.2 Choices of Instruction Set Level for CUDA Kernel Characterization

The code running in CUDA kernels is represented in different levels of abstraction. For characterization, of such kernels, one must choose a fitting abstraction level. C/C++ operators on different datatypes are too high level as there is a too detailed insight into what instructions are executed on the GPU. LLVM IR is also too unspecific to the actual hardware, even though it is much more detailed.

Two possible levels of abstraction remain. These are the parallel thread execution (PTX) ISA [44] and the SASS assembly [28]. PTX is an intermediate representation of the kernel assembly. There are different versions of the PTX ISA, but it is supposed to span multiple GPU generations. SASS is the result of translating PTX to a specific GPU and is the hardware instruction set.

Both PTX and SASS could be good candidates for characterization. We choose PTX for the following reasons:

- PTX is well documented and stable which is not the case for SASS
- PTX can be used across different GPU generations

- Obtaining PTX code is easy as the LLVM framework can produce it
- The instruction set is already well optimized for performance and includes specific instructions for function units like fmul or sinus/cosine functions

4.1.3 Limitations of Currently Available Tools for CUDA Kernel Characterization

4.1.3.1 Performance Counter Based

As an example for profiling using performance counters on NVIDIA GPUs the work of Burtscher et al. [45] is considered. They use the Nvidia profiler to measure control flow and memory-access irregularities. At the time of their work, they were using a Fermi-based GPU which required the profiling process to run multiple times to collect the metrics.

Similar to NVIDIAs profiler there is also a profiler for AMD GPUs. Since it is possible to transform CUDA kernels for execution on AMD GPUs [46], profiling on AMD hardware becomes also possible. AMDs Rocm Profiler [47] supports similar features like cache hits/misses, read/write to memory, instruction counter etc. Unfortunately, there is no public work available that investigates the performance. Compared to the NVIDIA profiling tools, the AMD tools do provide less information on the application. For example, the number of metrics is significantly less. Combined with the possible inaccuracies when translating CUDA this approach would be only used in very few (if any) cases when the starting points are CUDA kernels.

4.1.3.2 GPU Simulators

Bakhoda et al. [23] extend GPGPU Sim by Fung et al. [24], which is a micro-architecture performance simulator, to support CUDA. GPGPU Sim runs the PTX assembly defined by CUDA. They exchange the libcuda library with a custom one that runs the kernels on the simulator. They use the GPGPU Sim to study how different micro-architecture choices impact the performance of a few selected CUDA applications.

Another GPU simulator is introduced by Collange et al. [25]. The simulator called Barra does not work on the PTX intermediate instruction set but on the NVIDIA Tesla instruction set architecture (ISA). They achieve similar

performance than CUDA emulation but with more detailed information about internal processes. Like Bakhoda et al. [23] they use a custom libcuda library to run kernels on the simulator.

Gong et al. [27] extended the Multi2Sim framework [26] with support for simulating NVIDIA Kepler GPUs. Multi2Sim Kepler is a micro-architecture simulator running on shader assembly (SASS) which was chosen over PTX for accuracy reasons. Like Barra [25] and GPGPU Sim [23] they also integrate the simulator by exchanging the libcuda library.

4.1.3.3 Instrumentation

Farooqui et al. provide with Ocelot [48] and their subsequent work Lynx [22] instrumentation frameworks for PTX code. They exchange the CUDA runtime with their own library which dynamically instruments code during the runtime of the CUDA application. In the case of Ocelot, the instrumentation is defined programmatically whereas Lynx allows using C-on-Demand (COD) which is a subset of the C programming language.

Stephenson et al. introduced SASSI [28] in 2015, which is a tool instrumenting SASS code for GPUs. In their paper, they show that SASSI can be used to characterize GPU applications. SASSI exposes several options to insert user-specific code instrumentation code. The instrumentation takes place during compile time as a final compiler pass in `ptxas`, the ptx assembler.

In 2019, Villa et al. made NVBit [20] public. Like SASSI, NVBit allows instrumentation of GPU code on SASS level. The improvement over SASSI is, that NVBit performs the instrumentation dynamically during runtime when the code is loaded onto the GPU. Their approach does not require access to the source code of the application and thus also works with precompiled binaries and libraries.

Shen et al. use the LLVM framework to implement CUDA Advisor [49]. The framework provides memory analysis capabilities, including reuse distance, caching, and memory divergence (e.g. non coalesced memory access). In addition, the framework can analyze branch divergence. The framework requires source code.

4.2 CUDA Flux Tool Design

CUDA Flux is integrated into the LLVM compilation toolchain. A simple wrapper ensures that the instrumentation passes of CUDA Flux will be executed during compile time. Besides instrumentation, an analysis of the PTX code of the kernel will be performed and the basic blocks and the instructions which they execute are stored. When the instrumented binary is executed the basic block frequencies of the kernel are collected and stored as well. After execution processing of the PTX analysis and the basic block frequencies yield the instruction counts of the corresponding kernel. This extra step keeps the execution overhead low.

The following sections will discuss the major components of CUDA Flux which are the device instrumentation pass, the PTX parser, the host instrumentation pass and the post-processing step. Figure 4.3 shows how CUDA Flux is integrated into the CUDA compilation of clang.

The device pass is executed first. Before any modifications are done, the kernel is compiled into PTX. The PTX kernel is then processed by the PTX parser. The device pass then performs the instrumentation. After the device code has been compiled the host pass will run. The instruction summary produced by the PTX parser and fat binary which includes the compiled kernel is included in the host IR. The host pass performs additional instrumentation, which completed the modifications needed for functioning profiling.

4.2.1 Device Instrumentation Pass

The device instrumentation pass is executed after all the optimizations have been performed on the device code. The compilation module is dumped to disk and `llc` is used to compile the LLVM IR to PTX code.

Next, the PTX parser is invoked. After the lexical analysis dividing the PTX code into tokens, the actual parsing takes place. The parser extracts all the kernel functions containing their basic blocs including the instruction sequences each basic block executes.

Afterward, the original kernel code has been parsed, the device runtime is compiled and linked to the current compilation module. This has the advantage that the device runtime can be compiled for the same CUDA architecture as the compilation module.

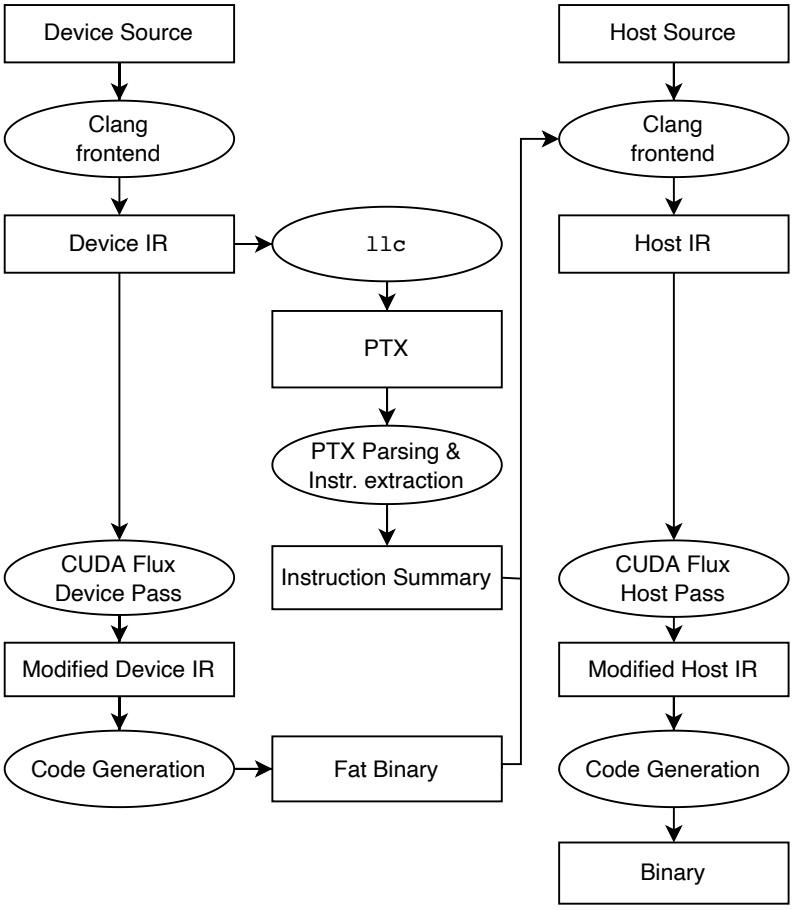

Figure 4.3 Integration of CUDA Flux into the CUDA compilation of clang. Braun et al. [38]

In the next step, the kernel functions are cloned and modified for profiling. Additional arguments are added, which include a pointer to the counters for the basic bloc frequencies, the number of banks used for the counter and the profiling mode.

The kernel code is instrumented as in the pseudocode in listing 4.1.

Listing 4.1 Pseudo-code showing the process of instrumentation of each basic block.

```
1 irBuilder = IRBuilder(moduleContext)
2 For each basicblock in kernel:
3   irBuilder.SetInsertPoint(basicblock.getFirstInsertPt())
4   irBuilder.CreateCall(
5     incBlockCounter,
6     {bbCounter, IDMap[basicblock],
7     n_banks, profiling_mode}
8   )
```

The rather naive approach instruments every basic block. With a bit more sophisticated analysis of the control flow graph, the number of accesses to the basic block counter can be reduced a lot. As the performance is already more than sufficient with this approach further optimizations were not attempted. Note that an IDMap for the basic blocks is used. This allows stable mapping of the basic blocks and their IDs to ensure correctness.

The function `incBlockCounter` is shown in listing 4.2. The profiling mode lets the user choose which threads are performing profiling. In an early development stage, the usage of fewer profiling threads for better performance was considered. The performance gain compared to profiling all threads is quite low. The option remains, but in practice, all threads are profiled.

As can be seen in line 15 only one thread per warp writes to the basic block counters. This keeps the overhead of CUDA flux already quite low. Since each thread can find which other threads are active with the `__activemask()` function this optimization is quite simple. Another optimization is trading memory usage for speed. This is achieved by using multiple banks of memory for different CTAs. The atomicAdd in line 19 is a possible bottleneck as all concurrent accesses to the same memory location need to be serialized. By using multiple banks this problem is alleviated.

Listing 4.2 Function increasing the basic block counters at the beginning of each basic block.

```
extern "C" __device__ void incBlockCounter(
  uint64_t *bbCounter,  uint32_t block_id,
  uint32_t n_banks, ProfilingMode profilingMode) {
  /* Choose Profiling Threads */
  if (profilingMode == ..) { ... }
  // get Number of Active Threads
  uint32_t lane_id;
  asm volatile("mov.u32 %0, %%laneid;" : "=r"(lane_id));

  uint32_t active_bitvec = __activemask();
  uint32_t active_threads = __popc(active_bitvec);
  uint32_t first_active_thread = __ffs(active_bitvec) - 1;

  // first thread of warp
  if (lane_id == first_active_thread) {
    uint32_t ctaID = blockIdx.x + blockIdx.y * gridDim.x +
    blockIdx.z * gridDim.x * gridDim.y;
    uint32_t bank = ctaID % n_banks;
    atomicAdd(bbCounter+(block_id * n_banks + bank),
      active_threads);
  }
}
```

4.2.2 PTX Parser

The PTX Parser performs the lexical analysis and the parsing of the tokens into function lists containing information on the basic blocks and their instructions. The parser does not support the full PTX specification and only looks into functions, basic blocks and instructions but not the operators of the instructions. The lexical analysis is implemented using RE2C which was originally developed by Bumbulis et al. [50]. RE2C compiles regular expressions to a deterministic finite automaton. With RE2C the PTX code is broken down into token vectors.

The token vector is analyzed and returns a structure containing basic blocks and functions are returned (see 4.3).

Listing 4.3 Data structure returned by the PTX Parser.

```
struct PTXFunction {
  std::string name;
  struct Block {
    std::string name;
    std::vector<std::string> inst;
  };
std::vector<Block> bb;
};
```

4.2.3 Host Instrumentation Pass

The host instrumentation pass runs before all the optimizations in the host code compilation pipeline. Here, it is not necessary to have access to the optimized code. The un-optimized code is preferable because the instrumentation is simpler on unoptimized code.

The host runtime is linked to the compilation module. It is compiled during the compilation of CUDA Flux but could also be compiled during the compilation of the instrumentation target. Besides the host runtime, the instruction summary is also included in the host compilation module. When the binary is executed the summary is written to disk along with the counters. This is technically not necessary, but it proved to be very useful for the further processing of the results.

The host instrumentation pass proceeds with adding kernel call wrappers for the cloned kernels. In addition, the cloned kernel need also to be registered so that it can be executed on the compute device. Only then each kernel launch site can be modified. Listing 4.4 shows a simplified pseudocode of this process.

Listing 4.4 Kernel launch replacement

```
for each launchSite in getLaunchSites(module):
  basicBlockCounter = createBBCounterMemoryAllocation()
  replaceKernelLaunch(launchSite, basicBlockCounter)
  insertCounterSerialization(launchSite)
```

Creating code for the allocation of memory for basic block counters is rather simple. The kernel launch is replaced by the launch of the instrumented kernel clone, which will write the basic block execution frequencies to the ba-

sic block counter. After kernel execution, the counters need to be processed. This functionality is implemented in the host runtime. A call to the function `serializeCounters` is inserted after the kernel launch. These functions take care of copying the results to the host code, adding up the counter banks, writing the counters to disk and freeing the memory.

4.2.4 Post-Processing of Profiling Data

With the execution of the binary, the instruction summary and the block execution frequencies are obtained. Besides the execution frequencies, the grid and thread block size and shared memory are provided. Getting the full instruction count for each kernel is quite simple. Instructions can be counted for each basic block and multiplied by the execution of the corresponding block. Depending on the usage of the counters, the instructions could already be summarized at basic block level.

4.3 Evaluation Methodology

CUDA Flux is evaluated qualitatively and performance-wise. The qualitative comparison shall give insight into the accuracy of the results when directly compared to performance-counter-based profiling methods like `nvprof`. The performance evaluation will compare the execution time of CUDA Flux, with a baseline and an alternative profiling approach, which is nvprof.

For the qualitative comparison to conventional profiling with nvprof a case example is used (SHOC, FFT). The performance evaluation makes use of the benchmark suites Rodinia [32], Parboil [34], SHOC [33], and Polybench-GPU [35].

Table 4.1 reports all applications of the benchmark suites and their number of unique kernels.

The build process needs to be adapted to clang because nvcc and clang support use slightly different command line arguments. Any build process that works with unmodified clang/LLVM will also work with the CUDA FLUX. Because clang does not support texture memory within the LLVM tool-chain, the following applications had to be excluded:

- Rodinia: hybridsort, mummergpu, leukocyte, kmeans
- Parboil: bfs, sad
- Shoc: DeviceMemory, MD, spmv

Additionally, some applications could not be used because of compilation issues:

- Rodinia: cfd, huffman, pathfinder, hotspot
- Parboil: cutcp, mri-gridding
- Shoc: qtclustering

The GEMM benchmark from the shoc benchmark suite was not used, because instead of using a kernel a CUBlAS library routine is called. Since there is no source code for this routine it could not be instrumented.

Qualitative Comparison to conventional profiling is performed by executing the FFT application from the SHOC benchmark suite with nvprof and with CUDA Flux. The instruction counts from CUDA Flux are summarized into similar groups like the instruction counter from nvprof and these are then

Benchmark	Application	Unique Kernels
parboil-2.5	**cutcp**	1
	histo	4
	lbm	1
	mri-gridding	8
	mri-q	2
	sgemm	1
	spmv	1
	stencil	1
	tpacf	1
polybench-gpu-1.0	**2DConvolution.exe**	1
	2mm.exe	2
	3DConvolution.exe	1
	3mm.exe	3
	atax.exe	2
	bicg.exe	2
	correlation.exe	4
	covariance.exe	3
	fdtd2d.exe	3
	gemm.exe	1
	gesummv.exe	1
	gramschmidt.exe	3
	mvt.exe	2
	syr2k.exe	1
	syrk.exe	1
rodinia-3.1	**3D**	1
	b+tree.out	2
	backprop	2
	dwt2d	4
	euler3d	4
	gaussian	2
	heartwall	1
	lavaMD	1
	lud_cuda	3
	myocyte.out	1
	needle	2
	particlefilter_float	4
	particlefilter_naive	1
	sc_gpu	1
	srad_v2	2
shoc	**DeviceMemory**	6
	FFT	6
	MaxFlops	36
	Reduction	2
	Scan	6
	Sort	5
	Stencil2D	2

Table 4.1 Applications of all used benchmark suites and the corresponding number of unique kernels.

compared.

Regarding performance evaluation, the procedure is the following:
Three different sets of measurements are done for comparison:

- nvcc without profiling (baseline)
- nvcc with nvprof profiling: overhead of nvprof-based profiling
- LLVM with CUDA Flux instrumentation: overhead of profiling based on CUDA Flux

Each measurement is performed five times and the average execution time is evaluated.

4.4 Qualitative Evaluation

This section begins with a theoretical comparison of the CUDA Flux approach and nvprof.

Modern GPU architectures have many performance counters. Before compute capability 5.0, only the following metrics represented the executed instructions:

- flop_count_dp
- flop_count_dp_add
- flop_count_dp_fma
- flop_count_dp_mul
- flop_count_sp
- flop_count_sp_add
- flop_count_sp_fma
- flop_count_sp_mul
- flop_count_sp_special
- inst_bit_convert
- inst_compute_ld_st
- inst_control
- inst_executed
- inst_fp_32
- inst_fp_64
- inst_integer
- inst_inter_thread_communication
- inst_issued
- inst_misc

The profiling approach using performance counters is thus limited when it comes to fine-grained information on the exact types of the instructions. Modern GPUs offer more different types of performance counters, but increasing the number of metrics to profile increases the overhead further. Another downside is, that the SASS instructions are not documented well. Mapping nvprof metrics or SASS counters from SASS-based instrumentation to PTX instructions is a serious hurdle. SASS instrumentation has at least the advantage of being even more verbose when it comes to the exact instruction type.

Listing 4.5 FFT512 kernel code of the shoc benchmark suite

```
template<class T2, class T> __global__
void FFT512_device( T2 *work )
{
    int tid = threadIdx.x;
    int hi = tid>>3;
    int lo = tid&7;

    work += (blockIdx.y * gridDim.x + blockIdx.x) * 512 +
       tid;

    T2 a[8];
    __shared__ T smem[8*8*9];

    load<8, T2>( a, work, 64 );

    FFT8<T2,T>( a );

    twiddle<8,T2,T>( a, tid, 512 );
    transpose<8, T2, T>( a, &smem[hi*8+lo], 66, &smem[lo
       *66+hi], 8 );

    FFT8<T2,T>( a );

    twiddle<8,T2,T>( a, hi, 64);
    transpose<8, T2, T>( a, &smem[hi*8+lo], 8*9, &smem[hi
       *8*9+lo], 8, 0xE );

    FFT8<T2,T>( a );

    store<8, T2>( a, work, 64 );
}
```

For the purpose of a qualitative comparison, we use the FFT application from the SHOC benchmark 4.5. The FFT kernel is well suited for this comparison because it uses special function units and also vectorized loads and stores for complex numbers. The instructions emitted by the compiler and executed by the

kernel can leverage the potential of fine-grained instruction types. The kernel also uses shared memory which allows viewing information on instructions working with this kind of memory, as well (lines 18 and 23). The special function units are being used in the twiddle function for sine and cosine in lines 17 and 22.

In table 4.2 are the instruction counts from CUDA Flux for the double-precision version of the FFT kernel. It shows that the sine and cosine instructions are used. The load and store instructions distinguish between global and shared memory and the v2 extension shows that vectorized loads and store are executed by the kernel.

Metric Name	nvprof	CUDA Flux	Ratio
flop_count_dp	645,922,816	547,356,672	1.18
flop_count_dp_add	301,989,888	144,703,488	2.09
flop_count_dp_fma	121,634,816	109,051,904	1.12
flop_count_dp_mul	100,663,296	113,246,208	0.89
flop_count_sp	0	58,720,256	0.00
flop_count_sp_add	0	0	-
flop_count_sp_fma	0	0	-
flop_count_sp_mul	0	0	-
flop_count_sp_special	58,720,256	58,720,256	1.00
inst_bit_convert	92,274,688	94,371,840	0.98
inst_compute_ld_st	167,772,160	167,772,160	1.00
inst_control	2,097,152	2,097,152	1.00
inst_executed	30,212,096	29,163,520	1.04
inst_executed_global_atomics	0	0	-
inst_executed_global_loads	524,288	524,288	1.00
inst_executed_global_reductions	0	0	-
inst_executed_global_stores	524,288	524,288	1.00
inst_executed_local_loads	0	0	-
inst_executed_local_stores	0	0	-
inst_executed_shared_atomics	0	0	-
inst_executed_shared_loads	2,097,152	2,097,152	1.00
inst_executed_shared_stores	2,097,152	2,097,152	1.00
inst_executed_surface_atomics	0	0	-
inst_executed_surface_loads	0	0	-
inst_executed_surface_reductions	0	0	-
inst_executed_surface_stores	0	0	-
inst_executed_tex_ops	0	0	-
inst_fp_32	88,080,384	58,720,256	1.50
inst_fp_64	524,288,000	547,356,672	0.96
inst_integer	35,651,584	20,971,520	1.70
inst_inter_thread_communication	0	0	-
inst_issued	30,216,709	0	inf
inst_misc	56,623,104	44,040,192	1.29

Table 4.2 CUDA Flux profiling results for the double precision FFT512 kernel.

Table 4.3 shows the profiling results of nvprof in comparison with CUDA Flux. The more detailed instructions of CUDA Flux are grouped to match the metrics which nvprof supports. Some deviation may come from a faulty grouping of the instructions. Other errors may occur because not all PTX instructions may have a direct SASS counterpart. Considering these sources of errors the

Metric Name	nvprof	CUDA Flux	Ratio
flop_count_dp	645,922,816	547,356,672	1.18
flop_count_dp_add	301,989,888	144,703,488	2.09
flop_count_dp_fma	121,634,816	109,051,904	1.12
flop_count_dp_mul	100,663,296	113,246,208	0.89
flop_count_sp	0	58,720,256	0.00
flop_count_sp_add	0	0	-
flop_count_sp_fma	0	0	-
flop_count_sp_mul	0	0	-
flop_count_sp_special	58,720,256	58,720,256	1.00
inst_bit_convert	92,274,688	94,371,840	0.98
inst_compute_ld_st	167,772,160	167,772,160	1.00
inst_control	2,097,152	2,097,152	1.00
inst_executed	30,212,096	29,163,520	1.04
inst_executed_global_atomics	0	0	-
inst_executed_global_loads	524,288	524,288	1.00
inst_executed_global_reductions	0	0	-
inst_executed_global_stores	524,288	524,288	1.00
inst_executed_local_loads	0	0	-
inst_executed_local_stores	0	0	-
inst_executed_shared_atomics	0	0	-
inst_executed_shared_loads	2,097,152	2,097,152	1.00
inst_executed_shared_stores	2,097,152	2,097,152	1.00
inst_executed_surface_atomics	0	0	-
inst_executed_surface_loads	0	0	-
inst_executed_surface_reductions	0	0	-
inst_executed_surface_stores	0	0	-
inst_executed_tex_ops	0	0	-
inst_fp_32	88,080,384	58,720,256	1.50
inst_fp_64	524,288,000	547,356,672	0.96
inst_integer	35,651,584	20,971,520	1.70
inst_inter_thread_communication	0	0	-
inst_issued	30,216,709	0	inf
inst_misc	56,623,104	44,040,192	1.29

Table 4.3 Profling results of nvprof and CUDA Flux for the FFT512 kernel in comparison.

instructions count are very close to each other.

4.5 Performance Evaluation

The performance of CUDA Flux profiling a single warp is evaluated by comparison with an un-instrumented baseline and execution time of the same application with nvprof. The metrics to profile with nvprof are chosen to match CUDA Flux functionality best.

In the bar plot in figure 4.4 the application execution time on a Tesla K20 GPU is displayed. The y-axis is using a logarithmic scale. With a few exceptions, namely spmv from the parboil benchmark suite, CUDA Flux outperforms nvprof consistently.

In some cases, the CUDA Flux binary executes faster than the baseline. Because the application is measured as a whole it is hard to tell if the difference comes from faster kernel execution. Of the seven applications that are faster, six belong to the polybench benchmark suite. These benchmarks will check the results on the host side. It could be the case that LLVM optimized these benchmarks better than nvcc, since the benchmark suite polybench originates from the LLVM developer community. Overall, clang produces binaries, which perform similarly to nvcc, according to Wu et al. [41].

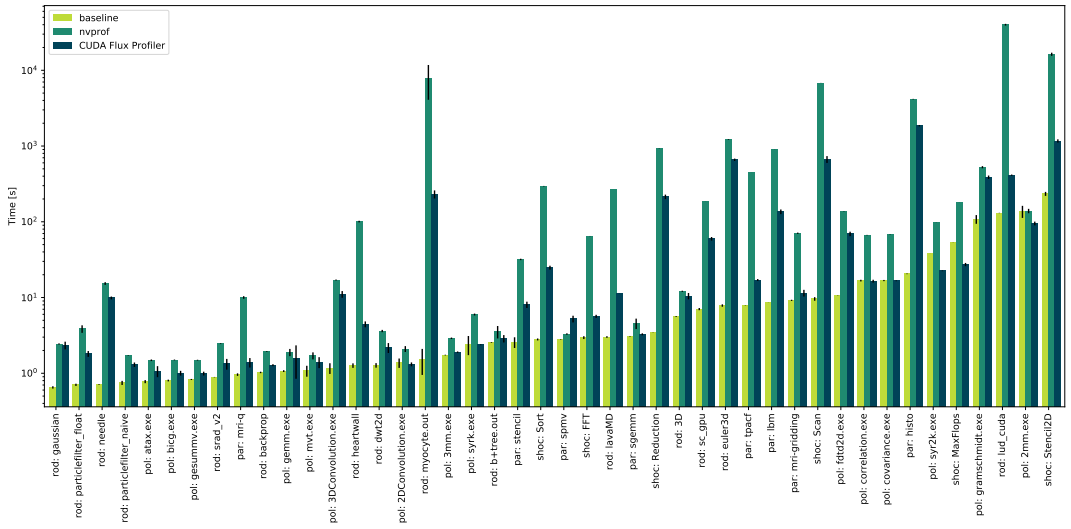

Figure 4.4 Plot of benchmark execution times using no profiling, nvprof and CUDA Flux profiling a single warp with a Tesla K20 GPU. Braun et al. [38]

.

Detailed reports of the results are shown in table 4.4. The average overhead when using CUDA FLUX is 13.20. Compared to the average overhead of 171.01

when using nvprof this is a big improvement.

	Application	baseline [s]	nvprof [norm.]	CUDA Flux [norm.]	# k. launches
parboil-2.5	**histo**	20.54	202.66	90.19	40,000
	lbm	8.63	104.23	15.73	3,000
	mri-gridding	9.18	7.69	1.25	50
	mri-q	0.96	10.43	1.45	3
	sgemm	3.08	1.48	1.06	1
	spmv	2.81	1.17	1.86	50
	stencil	2.57	12.38	3.13	100
	tpacf	7.89	57.39	2.16	1
polybench-gpu-1.0	**2DConvolution**	1.37	1.53	0.96	1
	2mm	137.39	1.01	0.69	2
	3DConvolution	1.17	14.57	9.46	254
	3mm	1.74	1.67	1.09	3
	atax	0.78	1.91	1.37	2
	bicg	0.80	1.87	1.25	2
	correlation	16.73	3.99	0.98	4
	covariance	16.76	4.06	0.99	3
	fdtd2d	10.63	12.79	6.52	1,500
	gemm	1.07	1.77	1.49	1
	gesummv	0.83	1.80	1.20	1
	gramschmidt	108.27	4.85	3.58	6,144
	mvt	1.08	1.58	1.30	2
	syr2k	38.40	2.53	0.59	1
	syrk	2.42	2.47	0.98	1
rodinia-3.1	**3D**	5.60	2.16	1.87	100
	b+tree.out	2.56	1.39	1.12	2
	backprop	1.03	1.89	1.25	2
	dwt2d	1.27	2.84	1.71	10
	euler3d	7.84	154.75	83.98	14,003
	gaussian	0.65	3.75	3.60	30
	heartwall	1.26	79.63	3.51	20
	lavaMD	3.01	88.87	3.74	1
	lud_cuda	130.67	306.01	3.16	6,142
	myocyte.out	1.52	5193.69	151.82	6,071
	needle	0.71	21.53	13.90	255
	particlefilter_f	0.70	5.48	2.56	36
	particlefilter_n	0.75	2.30	1.74	9
	sc_gpu	7.04	26.29	8.48	1,611
	srad_v2	0.89	2.76	1.50	4
shoc	**FFT**	2.96	21.88	1.91	60
	MaxFlops	53.70	3.32	0.51	361
	Reduction	3.45	265.75	61.90	5,120
	Scan	9.59	709.20	69.57	15,360
	Sort	2.80	105.58	8.88	480
	Stencil2D	233.89	69.67	5.00	22,000
	Min	0.65	1.01	0.51	1
	Max	233.89	5,193.70	151.82	40,000
	Mean	19.70	171.01	13.20	2790.98

Table 4.4 Table of benchmark execution times using no profiling, nvprof and CUDA Flux Profile. nvprof and CUDA Flux results are normalized to the baseline. The last column refers to the number of kernel launches. Braun et al. [38]

Because CUDA Flux has the ability to selectively profile only a subset of threads additional measurements using only the polybench GPU benchmark

have been made. Only kernel time is measured, and five iterations are used of which the median is used for evaluation. The execution time of a single warp, a full CTA and all threads are compared with nvprof. Figure 4.5 shows the normalized overhead on a Tesla K20 GPU. CUDA Flux consistently outperforms nvprof. As expected profiling only one warp is usually faster. The exceptions are the application atax and gemm. Reasons for this could be that profiling only one warp leads to less efficient memory accesses and therefore performance degradation.

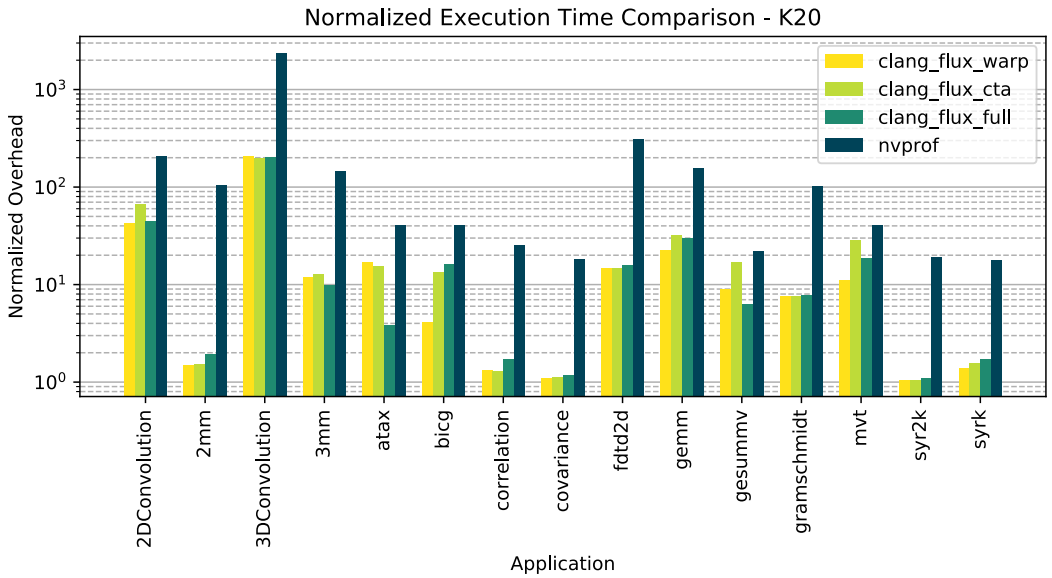

Figure 4.5 Comparison of kernel profiling time on Tesla K20 GPU normalized to baseline measurement.

Additional measurements were made on a GTX Titan Xp GPU (Pascal architecture) as shown in figure 4.6. Again, CUDA Flux consistently outperforms nvprof, but with a lower improvement. Differences in the selection of the profiled threads are very small. The results show that difference in the performance is not very high in general when fewer threads are profiled. This difference depends also on the GPU and depending on the GPU, the trade-off might not be worth it. Mean overhead ranges from 25x (K20) to 60x (Titan Xp). For these selected benchmark suites, the median speed-up to nvprof of about 10x (K20) and 6x (Titan Xp).

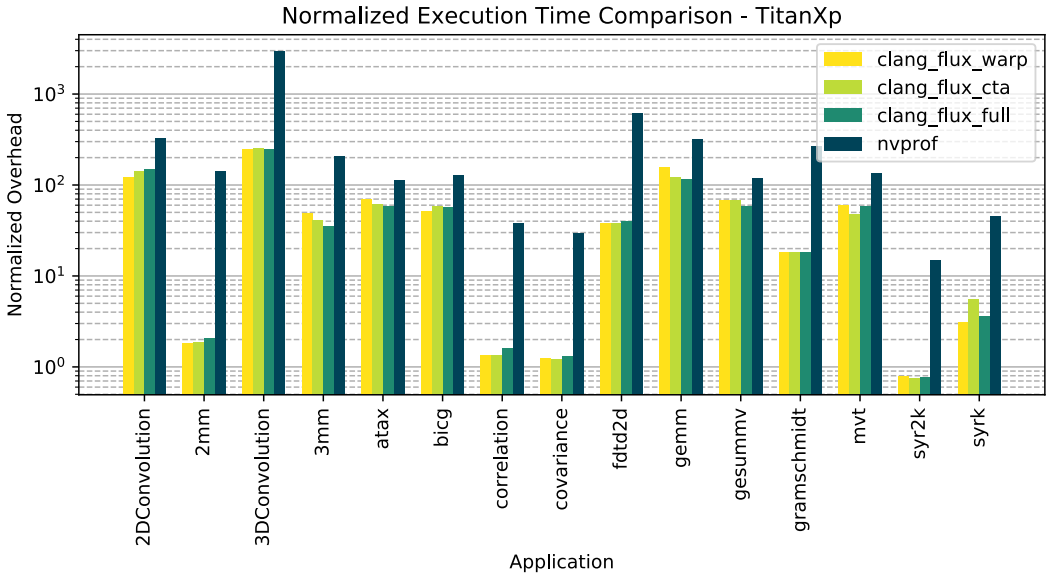

Figure 4.6 Comparison of kernel profiling time on Titan Xp GPU normalized to the baseline measurement.

4.6 Conclusion

The evaluation showed that CUDA Flux is a viable alternative to conventional profiling approaches. The PTX abstraction is a valid option and provides valuable insight into the internals of a kernel, as the qualitative evaluation showed. The performance of instrumentation-based profiling methods is very competitive which could be shown with the example of CUDA Flux. On the application level, the overhead is 13.2 times the normal application execution time on the four benchmark suites. Additional measurements on the polybench-gpu benchmark suite where only kernel time was measured the overhead was around 25x for the Tesla K20 GPU and around 60x for the Titan Xp GPU. It could be shown that profiling fewer threads only gives a minimal advantage over profiling all threads. Regarding accuracy, profiling a subset of the thread is viable when the control flow does not depend on input data or thread/block-ID in any form. For this reason, use cases of profiling subsets of threads are probably rare.

In summary, this work showed that instrumentation-based profiling of GPU kernels is a valuable tool with good performance and accuracy. The information which can be gathered via instrumentation is very detailed and comes with no additional cost because everything can be calculated with basic block frequencies and compile-time information. Trading accuracy for faster execution is not worthwhile. This is even more the case if further optimization of instrumentation is applied.

4.7 Future Work

Instrumentation, which is already used on CPUs for many purposes, has a big potential for GPUs as well. With NVBit NVIDIA published also a framework for kernel instrumentation. We welcome this development but argue that the instrumentation of the PTX code is also very important. The implementation of CUDA Flux leverages the LLVM framework and its ability to compile CUDA code. This makes it easy to instrument host and device code and to exchange information between their compilation pipelines. The approach of GPU Ocelot and Lynx has the advantage, that it can work with already compiled binaries. Future work could return to this mechanism as this makes working with many benchmarks and various build-systems much easier. Further improvements can

be made to the instrumentation itself. Performance can still be improved. Some loop iterations may be calculated beforehand and this could save many memory accesses to the frequency counters. Saving even more profiling overhead can be achieved by inferring frequency counts from other block executions counts. Both of these optimizations need the usage of a more detailed analysis of the kernel code.

Be aware that CUDA Flux is a research prototype. It is nice to improve its implementation further, but this may also be done in future development of similar tools.

CHAPTER 5

Fast and Portable Performance Prediction

This chapter is based on the publication of Braun et al. - „A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels" [29].

5.1 Introduction

GPUs are well known for their ability to process highly parallel workloads very fast. The raw operations per second, memory bandwidth and energy efficiency are hard to match with conventional CPUs. For this reason, many applications in the areas of data science, physical simulations, visual computing etc. are heavily using GPUs. The improved performance of GPUs compared with CPUs is achieved with a restricted programming model, which requires well-structured parallel implementations. This leads to compute kernels that behave more predictably.

The landscape of GPU hardware is highly heterogeneous. Between different generations of GPUs, there may be large changes regarding the architecture of just the sheer number of execution units. For example, the PCIe version of the NVIDIA Tesla P100 GPU can reach up to 9.3 TFlops for single-precision floating-point operations [51]. The next generation GPU the NVIDIA Tesla V100 PCIe can reach 14.0 TFlops, which is an increase of about 50% [52]. Since the introduction of high bandwidth memory (HBM), which only some GPU models

receive, there are big differences with the same GPU generations as well.

Therefore, the scheduling of compute tasks became more complex. Predictive models help in this regard to select the fastest or most energy-efficient processor for a given task. In the area of multi-GPU computing these predictive models help to adjust the number of GPUs, such that an optimal count is used and GPUs are not underutilized.

A model, which could make the predictions only with hardware-independent features, allows reasoning about performance in terms of time or energy ahead of execution time. Selection of the appropriate hardware would be greatly aided by removing the need to execute a task to schedule on all possible target processors to get input features for the model. Hardware-independent features include instruction counts and kernel launch configurations but no hardware-dependent features such as cache hit rates. Depending on the detail of the instruction counters additional features like memory volume read or written can be derived.

Performance and power models for GPUs are a well-covered topic in research [30, 53–73]. Braun et al. [29] describe the current research as follows:

> They are usually based on: (1) executing the program under observation, with additional costs depending on the required execution statistics; (2) collecting execution statistics using the processor's performance counters; and (3) inferring executing time and power consumption based on these statistics. As a result, such models rely on a variety of input features, in particular also hardware-related features like cache hit rates. Notably, some models yield good prediction performance without the use of such hardware-related features [59, 61].

They also list some analytical models [57, 59, 63, 68] and machine learning based methods [30, 55, 62]. The currently available models have two disadvantages, with the first one being that it is unknown or documented how these models would perform on different GPU architectures. They lack portability. In the worst case, the approaches presented would only work on the GPUs shown in the work itself. The second disadvantage is the lack of public models which can be used by anyone. Exceptions are the works of Arafa et al. and Guerreiro et al. [30, 74]. Their work is based on static analysis of the kernel assembly, which

limits the applications to those with kernels exhibiting static control flow. This shows that there is also a lack of availability.

GPUs match the BSP programming model [75] very well by leveraging parallel slackness to tolerate latency. Usually, there are much more threads available than execution units. For this reason and commonly followed programming practices GPU code is latency tolerant and optimized for data reuse. This is in accordance with the experiments conducted with the benchmarks used in this work [32–35]. With these assumptions dynamic hardware becomes a less important influence factor for execution time and power. The execution behavior of the kernel mainly depends on the instructions executed, the kernel launch configuration and static properties of the hardware, which makes prediction models solely based on hardware-independent features viable. We are aware that this implicitly dismisses dynamic properties of the interaction of the kernel with the hardware such for example cache hit-rates which is an important effect on execution time and power. We argue that dynamic effects like cache hit-rates are still important, but that in this viewpoint they do not play a major role and have limited influence. This assumes reasonable optimization of the application and will definitely not work with applications that are implemented in an adversarial way.

Using this chain of thought, the method described in this chapter offers a machine learning model for predicting kernel execution time and power consumption. The model is simple because it uses features that can be collected easily. The features do not depend on the executing hardware, therefore this approach is very portable and can quickly be adopted for hardware. With the random forest implementation that drives the model inference is quickly computed. The previous work of CUDA Flux [38] makes collecting input samples fast because of its low profiling overhead.

This chapter will go through related work in this area of research, the methods behind this approach and a detailed evaluation of the model on five different GPUs.

5.2 Related Work

Work of Braun et al. [29] describes the related work for this topic well and gives a good overview of that topic:

> In last years, performance and power modeling of GPUs are attracting considerable interest and several approaches have been proposed. A summary of related works sorted in chronological order including prediction (execution time or power consumption or both), model, accuracy, portability, input source, dataset size and support for different dynamic voltage and frequency scaling (DVFS) settings is given in Table 5.1.
>
> The most common approach is using machine learning methods such as random forest (RF) [66, 67, 72], support vector machines (SVM) [67], artificial neural networks (ANN) [55, 62, 76], long short-term memory networks (LSTM) [30], k-nearest-neighbor (KNN) [77], and so forth. The other common approach is using regression-based models such as statistical regression (SR) [78], regression model (RM) [79], regression trees (RT) [65], linear regression (LR) [63], multiple linear regression (MLR) [62] , ordinary least squares linear regression (OLS), LASSO, polynomial regression (PR) and support vector regression (SVR) [31]. Machine learning and regression methods provide an accurate prediction, albeit, tedious effort is required on feature engineering. However, this fundamental issue can be overcome by automatic methods evaluating the feature impact on the accuracy of the model.
>
> The other major approach is to use analytical models (AM). One example is Aspen as a domain specific language for analytical performance modeling [64], which basically requires to rewrite an application in this language. Another analytical model considers the number of running threads and memory bandwidth for predicting performance [67]. There is also analytical model using the novel collaborating filtering based modeling technique to predict the performance [71]. Alternatively, the interval analysis (IA), which differs from traditional analytical models, uses both trace-driven functional simulators and analytical model to estimate core-level performance

[60]. In this approach, GPUMech, an interval analysis-based performance modeling technique for GPU architectures was used for modeling two popular warp scheduling policies, namely round-robin scheduling (RR) and greedy-then-oldest (GTO), respectively.

Furthermore, there are approaches combining the aforementioned methods, leading to a hybrid model (HM) [70]. For example, combing an analytical model and with an event-based simulation of the code [68], or models as throughput model (TM) [58] and empirical model (EPM) [59, 69, 73] exist as well.

Numerous metrics have been used for measuring the accuracy of models such as Average Absolute Prediction Error Rate (AAPER) [62], Average Error (AE) [55, 60, 69, 71], Average Error Ratio (AER) [63], Average Percentage Error (APE) [65], Absolute and Relative Error (ARE) [64], Average Squared Error (ASE) [63], Geometric Mean of Absolute Error (GMAE) [57], Geometric Mean of the Error (GME) [59], Mean Absolute Error (MAE) [30, 66, 78], Mean Absolute Percentage Error (MAPE) [70, 72, 80], Mean Prediction Accuracy (MPA) [67], Mean Squared Error (MSE) [67], Root Mean Square Error (RMSE) [31] or Symmetric Absolute Percentage Error (SMAPE) [73]. Different performance metrics are used as they serve different purposes [81], but a detailed explanation is beyond the scope of this work.

Besides accuracy, another important characteristic of a model is to enable portability across different GPUs or other accelerators. Several studies [30, 55, 57, 62, 66–68, 70–72, 77, 78] have been conducted on this direction, while many other works focus on one single processor.

Most often, the input features for training the model are performance counters, which are required from tools including CUPTI, AMD CodeXL, nvprof, LLVM, Score-P, nvcc, Barra simulator, cubin generator, GPGPUSim, MacSim, DRAMSIM, GPUOcelot (PTX emulator), Architecture Independent Workload Characterization (AIWC), among others. However, most of the studies do not rely exclusively on those tools but also develop on their own using custom microbenchmarks, further code analysis, kernel compilation information, hardware specifications, analytical equations, program dependence graphs

(PDG) and others.

As previously mentioned, a representative training dataset is important for the model's generalization capability. The size of such a dataset varies highly among the studies, ranging from one single application to up to 169 different kernels. Note that it often remains unclear in the references if an application consists of multiple kernels which are treated independently or not.

More recently, there is the tendency on predicting performance and power consumption for different DVFS settings, namely for different memory and core frequencies. Subsequently, those works aim to determine the best DVFS settings providing the best performance with the minimum power consumption to leverage energy efficiency [30, 31, 72, 78, 80].

Our work distinguishes from most of these related works by using only hardware-independent input features for model training. Only quite few related works are also based on static input features [30, 31, 59, 61], while the vast majority requires a comprehensive application analysis prior to prediction. Last, we are aware of only two other works being publicly available [30, 68], and will discuss them in Section 7.

Source	T	P	Model	Accuracy	Portability	Input source	Dataset	DVFS
[57]	✓		AM	GMAE: 5.4-13.3%	4 NVIDIA GPUs (Fermi)	NVIDIA PTX, custom	20 apps	
[61]	✓		AM	good agreement between predicted and observed	1 NVIDIA GPU (Tesla)	PDG	4 apps	
[76]	✓		RM, ANN	median error: 1.16-6.65%	1 CPU	Xen-specific	4 apps	
[59]	✓	✓	EM, AM	GME: 2.7% (micro-benchs), 8.94% (merge)	1 NVIDIA GPU (Tesla)	GPUOcelot	20 apps	
[63]		✓	LR	ASE: 54.9%, AER: 4.7%	1 NVIDIA GPU (Fermi)	CUDA Profiler	49 kernels	
[58]	✓		TM	error: 5-15%	1 NVIDIA GPU (Fermi)	Barra, cubin, nvcc, HW res	3 apps, micro-benchmarks	
[65]		✓	RF, RT, LR	APE: 7.77%, 11.68%, 11.7%	1 NVIDIA GPU (Tesla)	GPGPUSim	52 kernels	
[64]	✓		AM	ARE plots	Intel CPU & NVIDIA GPU (Fermi)	Aspen	4 kernels	
[62]	✓	✓	MLR, ANN	AAPER: 6.7% (T), 2.1% (P)	2 NVIDIA GPUs (Fermi)	CUPTI, custom	20 kernels	
[69]		✓	EM	AE: 7.7% (micro-bench), 12.8% (merge)	1 NVIDIA GPU (Fermi)	MacSim, DRAMSIM	23 apps	
[60]	✓		AM, IA	AE: 13.2% (RR), 14.0% (GTO)	Fermi-like architecture	GPUOcelot	40 kernels	
[55]	✓	✓	ANN	AE: 15% (T), 10% (P)	6 AMD GPUs (GCN)	AMD CodeXL	108 kernels	✓
[77]	✓		LR, KNN	median divergence: 10%	2 NVIDIA GPUs (Fermi, Kepler)	custom	1 app (SpMV)	
[67]	✓		AM, LR, SVM, RF	MPA (pred/meas): 0.75-1.5% (ML), 0.8-1.2% (AM)	9 NVIDIA GPUs (Kepler, Maxwell)	nvprof, custom	9 apps	
[72]	✓	✓	RF	MAPE: 25% (T), 12% (P)	AMD CPU+APU	AMD CodeXL	73 apps	✓
[73]	✓		EM	SMAPE: 12.97%	CPU-GPU (no info)	Score-P	7 apps	
[56]	✓		AM	predicted/observed: 1.5% (vector ops), 0.76% (matrix ops), 5.49% (reduction)	1 NVIDIA GPU (Kepler)	custom	3 apps	
[80]	✓		AM	MAPE: 3.5%	1 NVIDIA GPU (Maxwell)	Nsight, custom, hard spec	12 kernels	✓
[78]		✓	SR	MAE: 7% (Pascal), 6% (Maxwel), 12% (Kepler)	3 NVIDIA GPUs (Pascal, Maxwell, Kepler)	CUPTI, custom	83 apps	✓
[66]	✓		RF	MAE: 1.2%	3 CPUs, 1 Xeon Phi, 5 NVIDIA GPUs (Kepler, Pascal), 6 AMD GPUs	AIWC	37 kernels	
[68]	✓		HM	w/in 10% to real device performance	2 NVIDIA GPUs (Maxwell, Kepler)	PTX, NVIDIA Visual Profiler	10 kernels	
[70]	✓		HM	MAPE: 17.04%	2 Kepler GPUs, 2 NVIDIA GPUs (Maxwell)	LLVM, custom	20 kernels	
[71]	✓		AM	AE: 9.4%	7 NVIDIA GPUs (Kepler, Maxwell, Pascal, Volta)	no information	30 apps	
[31]	✓	✓	OLS, PR, SVR	RMSE: 6.68-11.13% (speedup), 5.65-15.10% (energy)	1 NVIDIA GPU (Maxwell)	LLVM	118 kernels	✓
[30]	✓	✓	LSTM	MAE: 5.35-7.85% (P), 9.9-19.3% (T)	4 NVIDIA GPUs (Turing, Volta, Pascal, Maxwell)	PTX	169 kernels	✓
Ours	✓	✓	RF	MAPE: 8.86-52.0% (T), 1.84-2.94% (P)	5 NVIDIA GPUs (Kepler, Pascal, Volta, Turing)	CUDA Flux	189 kernels (T), 168 kernels (P)	

Table 5.1 An overview of related work, showing prediction target (time [T], power [P]), used model, accuracy, portability, input feature source, and dataset size. Braun et al.[29].

5.3 Benchmarking the Data Foundation

Design choices of the machine learning model depend a lot on the quantity and quality of data at hand. For this very reason, we like to show the data foundation on which the models are built first. Applications are drawn from four different benchmark suites: Rodinia 3.1 [32], Parboil 2.5 [34], Shoc [33] and Polybench GPU [35].

Three of the four benchmark suites were characterized by Adhinarayanan et al. [82]. They found that all benchmark suites have some unique applications. We decided to use as many applications as possible to have more samples at hand, even though some types of applications may be slightly over-represented.

Since the collection of input features requires CUDA Flux which is based on the LLVM compiler framework, limitations of said framework apply. For this reason, applications using texture memory can not be used. Table 5.2 from the work of Braun et al. [29] shows which benchmarks were used and which benchmarks were not used and the reasons why this was not possible.

Several minor modifications have been implemented on the benchmarks to get more samples. Some applications have been modified such that the kernel launch and the kernel definition are in the same compilation module. This is necessary because of a limitation of CUDA Flux and should have no impact on application performance. Additional problem sizes were introduced when it was possible. For example, the Polybench-GPU benchmark was modified to allow dynamic problem sizes instead of hard-coded ones. In general problem sizes were also increased when possible as more recent GPUs can process much larger problems now. Longer running kernels also help when measuring power as the sampling frequency for power measurements is usually below 100Hz.

5.3.1 Data Acquisition Methodology

Time and power measurements are performed on five different NVIDIA GPUs: Tesla K20 (Kepler), Titan Xp (Pascal), Tesla P100 (Pascal), Tesla V100 (Volta) and the GTX 1650 (Turing). The collection of the input features with CUDA Flux only needs to be measured on one device. Table 5.3 [29] shows the basic properties of these GPUs. The table shows that the power sampling frequency of the GPUs is very low, considering that kernel execution times below 10 ms are

Suite	parboil-2.5	polybench-gpu-1.0	rodinia-3.1	shoc
Included	cutcp histo lbm mri-q sgemm stencil tpacf	2DConvolution 2mm 3DConvolution 3mm atax bicg correlation covariance fdtd2d gemm gesummv gramschmidt mvt syr2k syrk	3D b+tree (irregular) bfs (irregular) backprop dwt2d euler3d gaussian heartwall lud_cuda myocyte needle particlefilter_naive particlefilter_float sc_gpu	BFS (irregular) FFT MD5Hash MaxFlops Reduction S3D Scan Sort Stencil2D Triad
Excluded	bfs[1] mri-gridding[2] sad[1] spmv[6]	correlation[3]	b+tree[3] gaussian[3] hotspot[2] hybridsort[1] kmeans[1] leukocyte[1] mummergpu[1] nn[2] pathfinder[2] srad-v1[6] srad-v2[6]	FFT[3] GEMM[5] MD[1] MaxFlops[3] NeuralNet[4] QTC[1] Sort[3] deviceMemory[1] spmv[1]

Exclusion reasons: [1]texture memory, [2]CUDA Flux compilation error, [3]power measurement methodology could not be applied, [4]hardcoded datasets, [5]no instrumentation possible due to cuBlas use, [6]unstable behavior

Table 5.2 List of included and excluded applications used for sampling. Some kernels are only excluded from the power prediction model. Braun et al. [29].

not unusual. Binaries were compiled with LLVM to measure the same produced code that CUDA Flux is instrumenting. The LLVM version used was 7.0 and the CUDA SDK version was 9.2.

GPU	F32 perf. [TOP/s]	Mem. BW [GB/s]	SMs	CUDA Cores	Core f [MHz]	Mem. f [MHz]	TDP [W]	f_s [Hz]
K20	3.5	208	13	2496	706	2600	225	73.6
Titan Xp	12.0	548	30	3840	1404	5705	250	60.2
P100	9.3	732	56	3584	1189	715	300	61.1
V100	14.0	900	80	5120	1290	877	300	61.2
GTX1650	3.0	128	14	896	300-2250	400-4001	75	10.9

Table 5.3 Overview of used GPUs and their relevant hardware specifications. f_s stands for power sampling frequency. Adapted from Braun et al. [29].

5.3.2 Execution Time Measurement Results

Code for time measurements was inserted into all applications before and after kernel launch. Each kernel launch is measured separately. Measurements of execution time were repeated ten times to get a statistical sample of each execution kernel. Some applications execute their kernels in loops, which yield over 900,000 samples in total. The results are combined with CUDA Flux measurements and identical samples are grouped and the median of each group is used as a prediction target. After the grouping, about 21,000 individual samples remain.

Figure 5.1 shows the execution time distribution on all GPUs. Because the execution time has a very large range from a fraction of milliseconds to a few minutes the bins in the histogram are applied only after the logarithm has been applied to the execution time. There is no shortage of samples with short execution times, but long-running kernels are scarce.

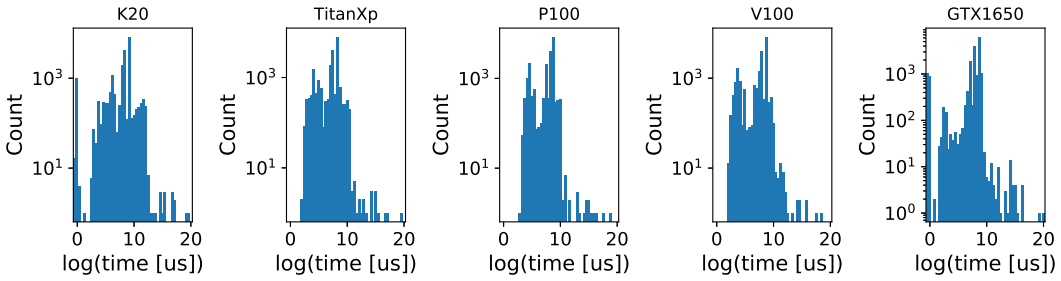

Figure 5.1 Histogram of the kernel execution time in logarithmic timescale. Note that long-running kernels are statistically under-represented. Braun et al. [29].

Since kernel execution time can also be in the low microsecond range. Looking into the variance of the measurements is important because it can have an impact on the prediction accuracy of the model. Figure 5.2 shows the coefficient of variation plotted over the mean execution time for identical kernel executions. In general, the variation is high for low execution time and gets lower when execution time increases. Even for long kernel execution times, there are still many kernel executions with high variance. This could be improved with additional measurements. Real-world systems, which may have more external influencing factors, could have even higher variance. We think that this behavior should be partially covered by the model. In a very advanced model, it may even be possible to quantify the variance.

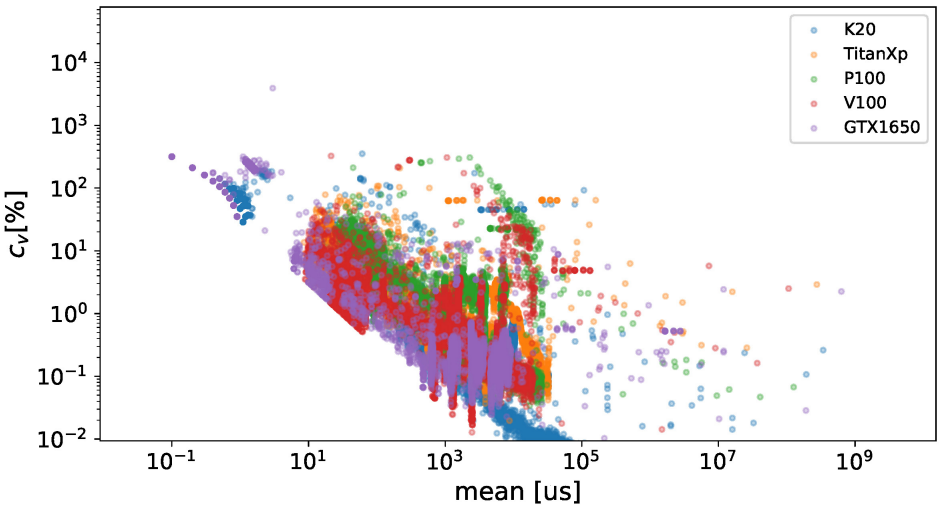

Figure 5.2 Visualization of the variance of execution time: coefficient of variation C_v plotted over the mean of execution time (for identical kernel executions) shows that short-running kernels appear to have a larger variance compared to long-running kernels. Braun et al. [29].

5.3.3 Power Measurement Results

Power consumption is measured using the onboard power sensors of the GPU. Similar to the execution time measurements code is inserted before and after the kernel to measure power. The NVML library [83] is used to read power values from the device. While the power values can be read with a high frequency, the results get only updated with the frequencies provided in table 5.3. If the kernel execution time is in the order or time between measurements of the power values, the measurement may have a high error. To improve the measured power accuracy, the kernel is executed multiple times in a loop until at least one second has passed. This was not possible for some applications therefore these could not be included in the power measurements. Measurements were repeated ten times as well.

To better understand the accuracy of the power measurements figure 5.3 shows the coefficient of variance over the mean power measured for identical kernel executions. The variance is usually well below 5%, which is in line with the findings of Burtscher et al. [84].

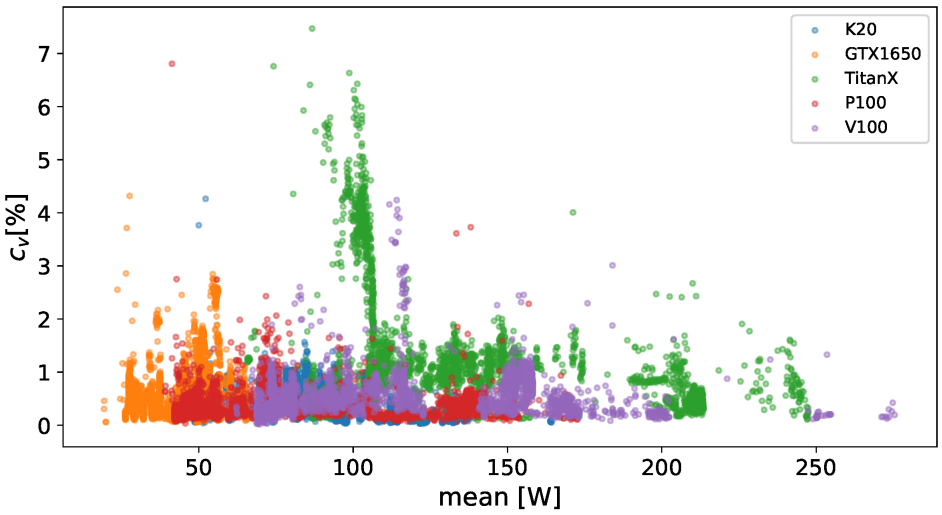

Figure 5.3 Validation of power measurements by comparing the coefficient of variation against mean power consumption. Braun et al. [29].

5.4 The GPU Mangrove Methodology

For our method, we decided to use portable code features together with feature engineering to build our random forest models. This section will also cover how we selected samples as well. The following subsections provide the details on each of these topics.

5.4.1 Portable Code Features

The approach for this method requires features to be hardware independent and portable. For this reason conventional methods of capturing information on the kernels like *nvprof* cannot be used. Instead, the CUDA Flux profiler is used. The PTX instructions provided are portable across multiple GPUs and do not change, unlike the SASS assembly. The output of CUDA Flux is independent of the GPU platform used to generate the results. The data gathered with the profiler is used to train not only one model but multiple. With a given set of benchmarks, which have been sampled, each model only requires sampling the target prediction values for the set of benchmarks.

Features provided by CUDA Flux include instruction counts and kernel launch configurations. These configurations consist of block- and grid-size and static shared memory allocation. The block- and grid-size are especially important because they give insight into the occupancy of the GPU.

5.4.2 Feature Engineering

There are many types of PTX instructions and there are even more subtypes of instructions as some instructions are extended with qualifiers. These qualifiers often define the data types of the operands or describe in more detail how the instructions should behave. Instructions for accessing memory also include qualifiers for the memory location (local, shared, global etc.).

Having separate features for each (sub)-instruction would give too many features to reasonably train most machine learning models accurately, with the number of samples we can collect. The number of features could be reduced by applying a PCA, but we believe that manually grouping the instructions gives more control over the results and interpretable features. Additionally, some analytical features can be introduced.

Preliminary experiments showed that simple grouping of instructions by arithmetic, special, logic and control flow are suitable to reach acceptable prediction accuracy. The grouping approach was inspired by the classification of instructions by Patterson and Hennessy [85]. Instructions accessing memory (load/store) are grouped differently. The operand type qualifiers allow deriving the type bit-width. The bit-width is used to compute the total volume of data which is written or read from memory. Separate features are computed for read and write accesses for all the different memory locations. Because arithmetic intensity is an important metric, which is often used to optimize kernels, it was also added as a feature. It is computed as the ratio of arithmetic instructions and access to global or local memory.

5.4.3 Selection of Samples

Because some applications execute kernels in loops with different launch configurations, these particular kernel executions are over-represented. While it is preferable to have as many samples as possible, the model should not be biased toward a specific type of kernel. This is addressed by reducing the number of samples of a kernel belonging to an application and applying a threshold of allowed samples per kernel in an application. The threshold is set to 100 samples. Therefore, if there are more than 100 samples of a specific kernel in an application, 100 randomly selected samples are used in the final dataset. This number could be tweaked because too few or many very similar samples can impact the quality of the model prediction.

5.4.4 Machine Learning Model Selection and Performance Evaluation

Even though many samples could be collected the remaining samples used are surprisingly low. From the 21,000 individual samples after grouping only around 1,600 samples remain after applying the threshold of 100 allowed samples. With this new approach, it is necessary to perform model selection. At the same time, the performance of the model needs to be evaluated. This is difficult with the low amount of samples at hand. The traditional approach of splitting the dataset into training, testing and validation sets, will most likely make poor usage of the available data. Varma et. al [86] proposed nested cross-validation

solves this problem very well. Their approach uses two nested loops of cross-validation. The inner loop is used to perform tuning of the hyper-parameters. The outer loop computes the estimated error. While this method is certainly computationally expensive it is simple to implement and performs model selection and performance evaluation, which ensures good generalization. For this reason, we apply the method in our approach. A possible alternative could be the method of Tibshirani and Tibshirani [87] which uses only two cross-validations.

For the machine learning method itself, the Extremely Randomized Trees Regression from the scikit-learn library [88] is used. Different learning methods were tested, including small fully connected neural networks, but they couldn't achieve sufficient performance. While this learning method works well for our approach, there are some disadvantages that required some extra work. The random forest can not extrapolate our prediction data. For power prediction, this is not a problem as the power consumption of the GPU has an upper limit that cannot be exceeded. This is not the case for time predictions, for which the model can only predict up to an upper limit, which it has learned from the training data. Therefore, it is necessary to implement a custom grouping for the splits of the cross-validation.

The custom grouping ensured that each of the training sets always contained the five longest-running kernels. Additionally, because the range of the execution time samples is very large, we ensured that each split has a similar amount of samples from short-running (t < 1,000us), medium running (1,000us <= t < 1000,000us) and long-running kernels (t >= 100,000us).

To address the wide range of execution time the prediction target was transformed for time prediction. A simple log transformation was used to achieve better prediction results.

The hyper-parameter search space was reduced to three parameters. These are the number of estimators used, the maximum number of features to determine a split in a tree and the split criterion.

The number of estimators defines the number of trees in a random forest. Usually, the prediction quality improves when more trees are used. At the same time, computational complexity and over-fitting are also increased. Therefore, it is important to find the right balance of this parameter. The other parameters influence when a split is made.

The following values have been used for the hyper-parameter search:

- N estimators: 128, 256, 512, 1024
- Maximum Features: max, log2, sqrt
- Split criterion: MSE, MAE

The error metric with which the models will be evaluated is also important. A good metric should reflect on what is important in the results. For example, if a low average error is wanted, the mean absolute error could be a good error metric. Sometimes the mean square error is preferred, because larger differences between the true outcome of a sample and the predicted value are pronounced more. For this particular approach, a low error relative to the true value is wanted. This has the advantage that larger differences can be pronounced more while low differences also matter if the true value is also low. Therefore, the error metric chosen is the mean average percentage error (MAPE) shown in equation 5.1. This metric turned out to work well with the wide range of execution time measurements that have been made.

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{y_i} \tag{5.1}$$

5.5 Case Study on the Tesla K20 GPU

Most of the GPUs are more or less similar. Looking into the details of the model evaluation gives insight into the success of the proposed methods. We choose the Tesla K20 GPU because with its older GPU architecture and lower core clock, we assume that the spread of execution time will be lower and therefore it could possibly be modeled better than the other GPUs. This section shows how well the predictions for this GPU are. The results for all GPUs are reviewed in the following section.

The first goal of this case study is to find out if there are any samples that can only be predicted poorly and are possible outliers. Nested cross-validation is used to find the best set of hyper-parameters. Next, the model is evaluated using the leave-one-out technique (LOO), which is a special case of the K-fold cross-validation. The number of folds is the number of samples.

5.5.1 Execution Time Prediction

The nested cross-validation is performed with 30 iterations each using a different random seed for splitting the samples. Figure 5.4 shows the mean error of all iterations for execution time (left) and power (right). In the case of time prediction, the error is between 12.11% and 19.37%. Most of the iterations show an error between 12% and 15%. Comparing these numbers to related work show, that the results should be acceptable considering the scope of the input features for the model prediction. The consistent range in which the error stays during the iterations of the nested cross-validations also indicated that the model seems to generalize well.

The best parameters from the nested cross-validation are used to perform another evaluation using LOO. Figure 5.5 shows two plots. In the first plot, the time value of the sample is plotted on the logarithmic x-axis and the predicted value is plotted on the logarithmic y-axis. A dotted line indicates where the points of the scatter plot would be if the prediction was perfect. The majority of the points are very close to the true value. Note that in the top right, the samples are under-estimated. This is because the random forest model fails to extrapolate beyond seen value ranges. Additionally, there are just too few samples with high execution time, so these samples are under-represented and can not be learned

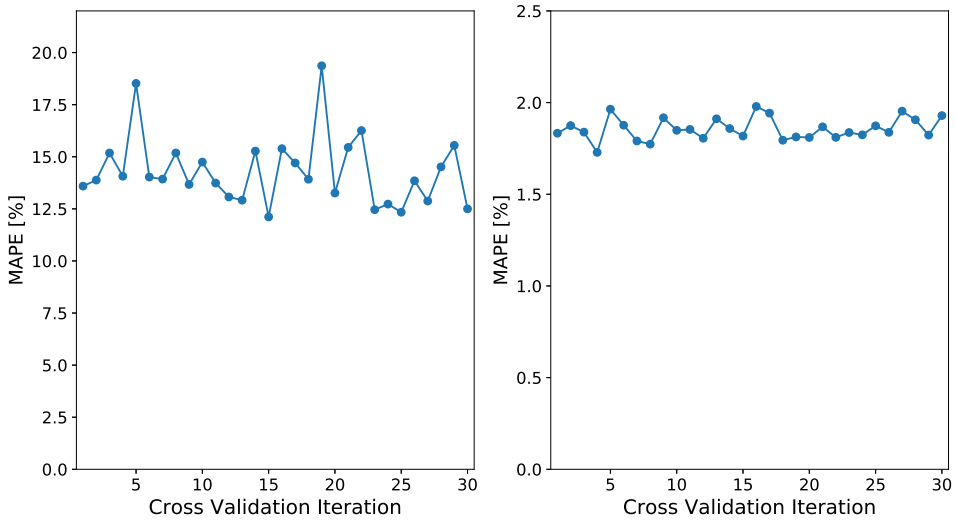

Figure 5.4 Nested cross-validation score for execution time (left) and power (right) prediction on the K20 GPU. Braun et al. [29].

as well as other samples. The right plot shows a histogram counting the number of samples within an error range. Around 82% of the samples are within 10% of the true value or lower. About 8% are between 10% and 25%. The next groups represent ∼4%, ∼4%, ∼% and ∼1%. The amount of samples with an error above 100% is very low and there are only a few outliers. The results for execution time prediction are definitely usable, but there is certainly some caution required when it comes to long execution time.

5.5.2 Power Prediction

The power prediction for the Tesla K20 GPU follows the same approach as for execution time. The right side of figure 5.4 shows the nested cross-validation score for power prediction. The even lower error there which ranges from 1.72% and 1.97% indicate very good generalization. The range of measured power is much smaller, which explains the much better results.

The LOO plot (figure 5.6, left) shows more samples, which are further away from the ideal prediction. This indicates that some samples are harder to predict than others. The overall results are good, nonetheless. The histogram on the right side of figure 5.6 shows that 92% of the samples are below the 5% error margin. Only 4% of the samples have an error of 10% or higher.

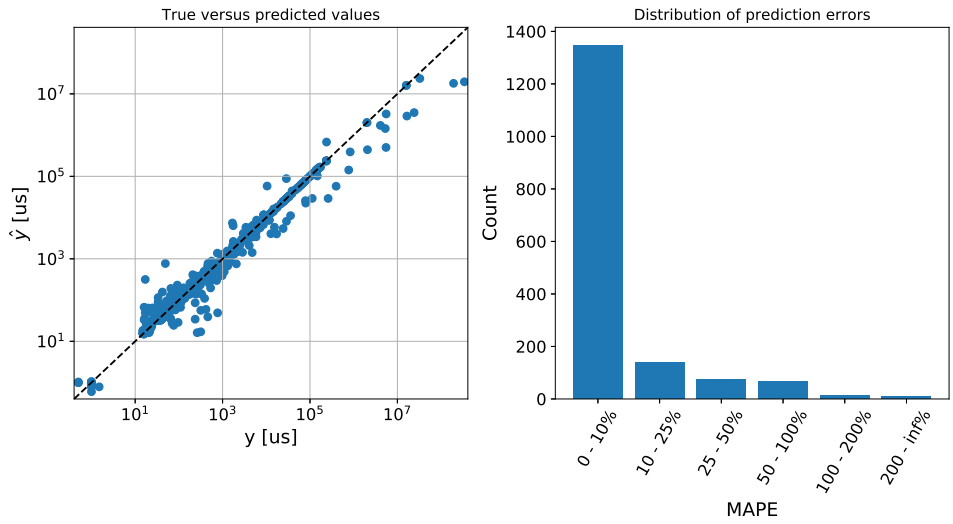

Figure 5.5 Leave-One-Out results for time prediction on the K20 GPU. Left: scatter plot of true values versus predicted values (logarithmic scale). Right: distribution of prediction errors. Braun et al. [29].

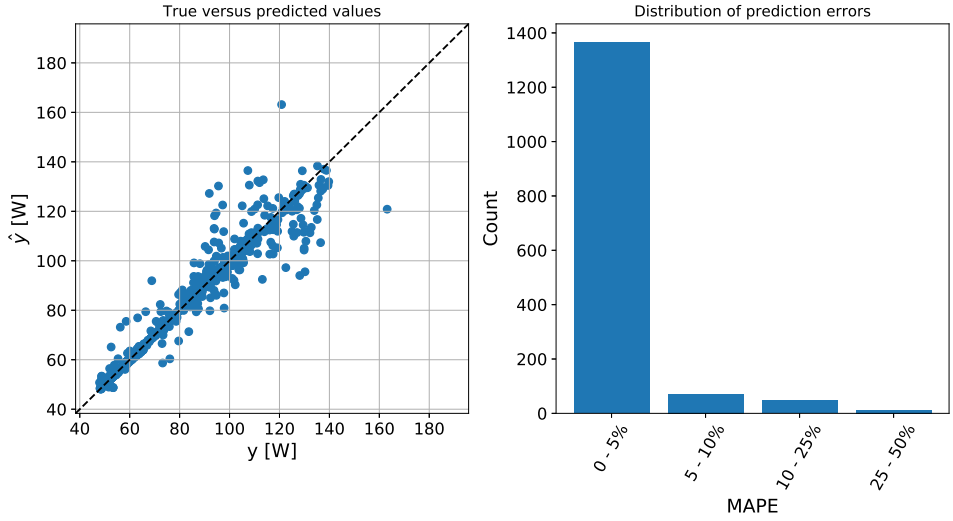

Figure 5.6 Leave-One-Out results for power prediction on the K20 GPU. Left: scatter plot of true values versus predicted values (note the linear scale). Right: distribution of prediction errors. Braun et al. [29].

5.6 Evaluation - Portability

The previous section showed the viability of the proposed approach for the Tesla K20 GPU. This section will examine the portability of the approach which is evaluated on the five GPUs presented earlier. The application statistics (input features) are obtained on one GPU, while the execution time and power usage are measured on each device.

5.6.1 Execution Time Prediction

Similar to the case study with the Tesla K20 GPU, a nested cross-validation with 30 iterations is performed for each GPU. For a more accurate picture of the quality of the prediction models, the scores of all folds of the iterations are considered. Figure 5.7 shows two box plots with scores for execution time (left side) and power (right side) on each GPU. The orange bar in the middle shows the median and the box includes the scores from the first to the third quartile. The whiskers are limited to the interquartile range times 1.5.

The results for execution time prediction are mixed. The median ranges from 8.86% to 52%. The GTX 1650 GPU with a median of 52% is a consumer class GPU and does not have a fixed core or memory clock setting (see table 5.3). The other GPUs are mostly server-class GPUs, which have a median error of at most 13.86% (Tesla K20). The exception is the Titan Xp, which could be viewed as a prosumer (professional consumer) device.

For the server-class GPUs, the variability is low, which shows good prediction results that do not depend on a favorable split of training and test data. This is not so much the case for the GTX 1650 for which the third quartile is at 99.23%. This large interquartile range shows that there is a high variability and poor generalization of the GTX 1650 model. Even with more favorable splits, of training and test data, the error score is still very high (first quartile is around 30%).

Looking at the distribution of the error scores gives more insight (figure 5.8). For the four server class GPUs the error is below 20% for the majority of the cases. This agrees with the good results shown in figure 5.7. For the Tesla K20 and Titan Xp there are a few folds that perform significantly worse than the other. This is shown by the gap between the non-zero count and the previous

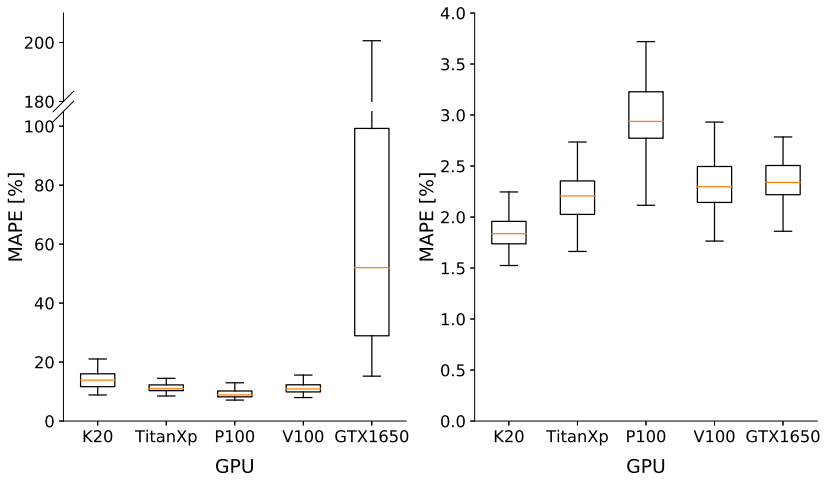

Figure 5.7 Portability of time (left) and power (right) prediction across different GPUs: MAPE scores for all iterations of nested cross-validation with median, first and third quartile. Whiskers are limited to 1.5 times of the interquartile range (Q3-Q1). Outliers are not shown. Braun et al. [29].

one. For these two GPUs, the variability is not as good as the other server-class GPUs. The histogram for the GTX1650 shows, that the majority of error scores are on the low end, but the spread of the error is very high. Some folds of the cross-validation have errors above 500%. This clearly shows that the model cannot predict execution time for this device well. The reason for that could be a lower measurement accuracy compared to the other devices and nondeterministic behavior due to dynamic clock frequencies.

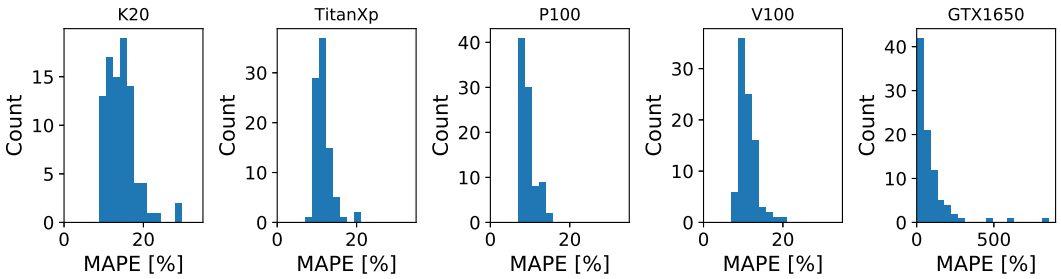

Figure 5.8 Histogram of the MAPE score for each fold of the nested cross-validation. Braun et al. [29].

The error scores using the LOO method are evaluated in figure 5.9. Similar to the Tesla K20 GPU the other devices underestimate long-running kernels. This is no surprise as the model is very similar and the data shows that there is a lack

of long-running kernels for all devices. Note that for the GTX 1650 GPU, there are also more samples with lower execution time, where the prediction is off by a large amount. There is a cluster of very short kernels that are overestimated by a big factor. These samples could be the reason for the bad prediction quality in the nested cross-validation. Because the values are very small a rather low absolute error quickly leads to a high relative error as in the MAPE score.

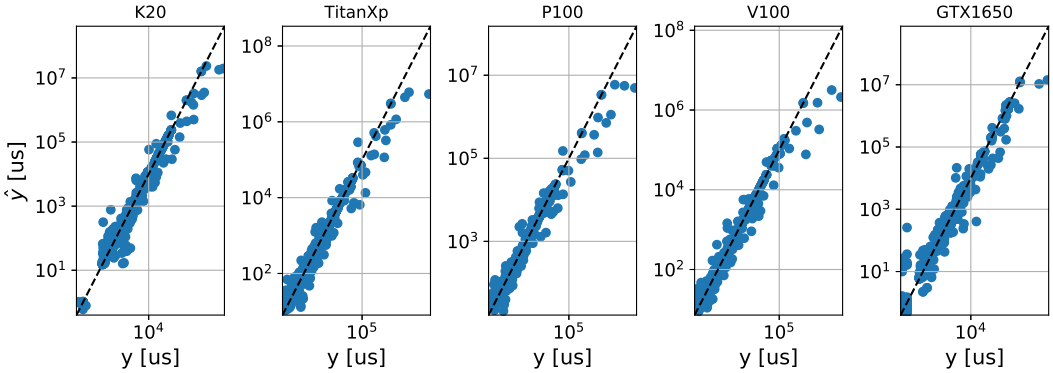

Figure 5.9 Scatter plot of true time values versus predicted values using the leave-one-out method for K20, GTX1650, Titan Xp, P100 and V100 GPUs. Braun et al. [29].

Table 5.4 shows the hyper-parameters, average depth and the prediction latency of the best model for each GPU. The most important hyper-parameter is the number of estimators as this, along with the average depth should influence the prediction latency the most. The prediction latency was measured on an Intel Xeon E5-2667 (introduced in 2012). As the number of estimators is the same for each GPU with the exception of the GTX 1650 and the tree depth is also very similar the prediction latency is very similar for all the GPUs. Even for the GTX 1650 which has only half the number of estimators. The latency of around 108 ms shows that the prediction models should cater more to long-running kernels as it would make little sense (latency-wise) to wait for the prediction for kernels running faster than the prediction latency. With a more recent CPU, the prediction latency could be improved, which may make the model suitable for a larger range of kernels. At the same time, more recent GPUs may improve the execution time of the kernels, too. However, the effects of newer hardware may be limited as the latest trends tend to increase the performance of processors by increasing the capabilities of parallel computing.

GPU	Best hyperparameters	Avg. depth	Prediction latency
K20	MAE, max features, 512 estimators	34.83	108.56 ms
Titan Xp	MAE, max features, 512 estimators	33.30	108.65 ms
P100	MAE, max features, 512 estimators	34.94	108.30 ms
V100	MAE, max features, 512 estimators	34.30	106.91 ms
GTX1650	MSE, max features, 256 estimators	31.06	107.46 ms

Table 5.4 Hyperparameters for the best model for time prediction, together with the corresponding average prediction latency, measured on an Intel Xeon E5-2667 v3 CPU. Braun et al. [29].

5.6.2 Power Prediction

The box plots for the power prediction are shown on the right side in figure 5.7. The median scores for all GPUs are very good and well below 5% for the majority of splits in the nested cross-validation. The low interquartile range indicates good generalization of the models.

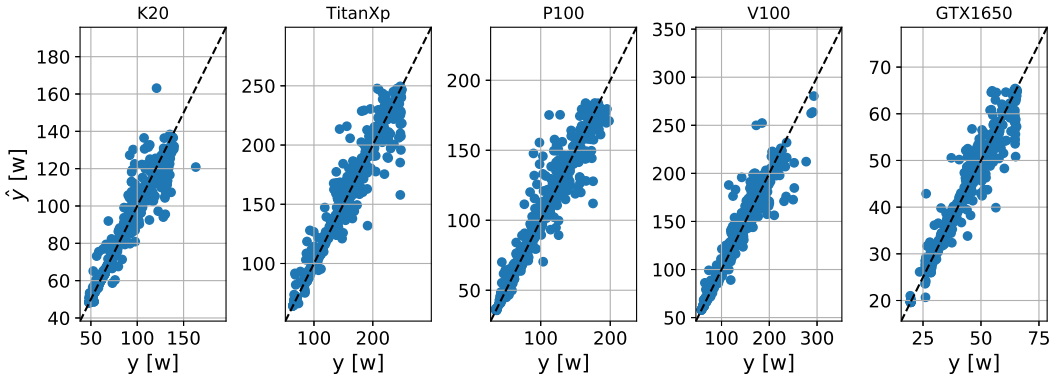

Figure 5.10 Scatter plot of true power values versus predicted values using the leave-one-out method for GTX1650, Titan Xp, P100 and V100 GPUs. Braun et al. [29].

An analysis with the LOO method (figure 5.10) shows that the power prediction also has a problem with underestimating high power usage. There are also some outliers that have identical kernel features but different power consumption. With the available data at hand, the model cannot make better predictions. Therefore, these errors probably come from the limited amount of features or statistical variance.

When looking at the hyper-parameters, tree depth and prediction latency, some differences compared to the execution time prediction can be made out. The number of estimators has a wider range (256 to 1024). While the average

GPU	Best hyperparameters	Avg. depth	Prediction latency
K20	MAE, max features, 256 estimators	32.08	15.36 ms
Titan Xp	MAE, max features, 256 estimators	33.45	15.32 ms
P100	MAE, max features, 512 estimators	32.49	30.14 ms
V100	MSE, max features, 1024 estimators	32.91	60.58 ms
GTX1650	MAE, max features, 1024 estimators	32.19	59.20 ms

Table 5.5 Hyperparameters for power prediction model, together with the corresponding average prediction latency, measured on an Intel Xeon E5-2667 v3 CPU. Braun et al. [29].

tree depth is roughly the same, the prediction latency differs a lot. The latency correlates with the number of estimators and is about 15ms per 256 estimators.

5.6.3 Feature Importance

The random forest machine learning model allows us to derive feature importance. Examining the importance of different features helps to understand better how the models work and which factors influence execution time or power consumption the most. Additionally, it theoretically allows cutting down on prediction time by evaluating only the most important features. However, this optimization would not be for free as this would most certainly decrease prediction accuracy.

In table 5.6 are the feature importance in percent of the best model for each GPU. Regarding time prediction, threads per CTA seem to be the most important feature. At the top 3 of the most important feature is the global memory volume (in place two or three). The arithmetic intensity and synchronization operations are surprisingly unimportant.

To gain more insight into how the feature importance correlates with the properties of the GPU, Pearson correlations coefficient between the features and some hardware properties have been computed (see table 5.7). The following observations can be made:

- The importance which correlates well with the hardware properties are total instruction, synchronization, control flow, CTAs and treads per CTA.
- The total instructions and control flow become less important when there is a higher TDP, memory bandwidth and more CUDA cores.
- While synchronization is not very important in absolute numbers, it correlates well with the memory bandwidth, the number of SMs and power.

	Time					Power				
Feature	K20	Titan Xp	P100	V100	GTX 1650	K20	Titan Xp	P100	V100	GTX 1650
threads per CTA	23.19	27.17	26.62	29.62	23.49	19.74	24.70	17.91	14.77	9.52
CTAs	8.47	10.01	11.74	10.76	5.51	20.64	8.81	16.49	20.26	19.28
total instr.	7.90	7.73	6.40	6.34	9.57	5.58	6.01	5.84	4.36	4.10
special ops	1.16	1.53	1.96	1.46	0.42	2.37	8.00	3.35	1.39	1.07
logic ops	2.15	2.58	2.38	2.30	1.34	3.91	4.32	3.27	3.87	10.94
control ops	4.41	4.50	3.75	3.71	5.51	3.69	6.11	2.68	2.36	2.41
arithm. ops	6.96	8.12	6.75	7.01	11.62	6.75	6.46	6.72	4.84	5.22
sync ops	2.96	3.54	4.89	4.71	2.34	4.97	4.05	8.72	5.08	5.84
global mem vol.	12.46	16.30	16.30	14.13	15.59	8.28	10.61	6.47	5.73	4.99
param mem vol.	20.14	8.15	9.08	8.60	13.45	16.63	11.39	17.72	27.45	30.27
shared mem vol.	4.38	4.22	4.66	4.97	4.72	3.88	3.56	7.49	7.27	4.77
arithm. intensity	5.81	6.14	5.47	6.40	6.43	3.57	5.99	3.34	2.62	1.61

Table 5.6 Feature importance in percent for time and power prediction. Braun et al. [29].

- The number of CTAs becomes more important with more SMs/CUDA Cores.
- Threads per CTA are more important when floating-point performance and memory bandwidth is high.
- Global memory volume is important with few influences from the GPU properties.

The observations fit the intuition of GPU performance well. E.g. threads per CTA must be high for good floating-point performance or to reach the more CTAs which can be executed in parallel the higher the impact of synchronization. It is interesting to note that the top3 and top5 most important features reach 36-44% and 70% respectively of the accumulated importance.

For the power prediction models, the importance of the features looks different. The most important features seem to be threads per CTA, CTAs and parameter memory volume. Importance of the global memory volume is low, which is surprising as memory usually contributes to power usage by a large factor. But this could possibly be explained by the memory consuming a rather static factor of the power. Verifying this would need more experiments, though. The Pearson correlation computed for the power prediction feature importance is shown in table 5.8. The correlation for the power feature importance is much lower than for execution time. This indicates that the GPU properties may be less important for power prediction in general. Logic operations and the shared memory volume

	float perf.	SMs	CUDA Cores	Core Clock	Mem. Clock	Mem. BW	TDP
threads per CTA	0.98	0.91	0.92	0.18	-0.32	0.95	0.73
CTAs	0.78	0.75	0.88	-0.49	-0.50	0.88	0.98
total instr.	-0.75	-0.85	-0.90	0.48	0.68	-0.91	-0.97
special ops	0.68	0.62	0.78	-0.58	-0.42	0.79	0.94
logic ops	0.73	0.48	0.83	-0.36	-0.12	0.69	0.91
control ops	-0.70	-0.81	-0.88	0.53	0.69	-0.86	-0.98
arithm. ops	-0.51	-0.53	-0.76	0.65	0.54	-0.64	-0.94
sync ops	0.78	0.91	0.85	-0.40	-0.68	0.95	0.90
global mem vol.	0.31	0.14	0.01	0.19	0.28	0.25	-0.04
param mem vol.	-0.83	-0.68	-0.60	-0.29	0.04	-0.76	-0.42
shared mem vol.	0.20	0.65	0.20	0.15	-0.70	0.42	0.05
arithm. intensity	0.11	0.01	-0.08	0.96	0.38	-0.11	-0.48

Table 5.7 Pearson correlations of feature importances and GPU properties for execution time prediction.

importance change the most with the GPU properties. When there is more power available to the GPU logic operations become less important. This is probably due to the rather high importance of this feature on the GTX 1650 which has a much lower TDP. It seems that control flow may have a larger power overhead for chips using less power and fewer parallel processing units. Shared memory becomes more important when there are more SMs and memory bandwidth. When there is a lot of high bandwidth memory that can potentially consume a lot of power this correlation makes good sense. The more SMs can request data from memory the higher the power usage. Also, the more memory accesses are going to shared memory instead of global memory, the less power for memory is needed. Surprisingly most of the feature importances do not correlate a lot with the TDP.

Around 50% of the cumulative importance of the power model features is reached by the top 3, which is similar to the execution time model. The top 5 importance is also similar at around 70% (with the GTX 1650 being a bit lower at 63.5%).

In summary, it seems to be important how many CTAs and threads per CTA are executed, both for time and power prediction. This is especially the case for time prediction. This parallel *pressure* helps to keepthe execution units busy. In general, the power feature importance is less conclusive. The importance is distributed more and correlate less with the GPU properties. The total instructions are significantly more important for time prediction than for power.

	float perf.	SMs	CUDA Cores	Core Clock	Mem. Clock	Mem. BW	TDP
threads per CTA	0.38	-0.04	0.46	-0.29	0.32	0.22	0.58
CTAs	-0.43	0.08	-0.22	-0.19	-0.65	-0.17	-0.19
total instr.	0.16	-0.13	0.25	-0.60	0.16	0.13	0.52
special ops	0.41	-0.12	0.29	0.05	0.62	0.15	0.30
logic ops	-0.58	-0.51	-0.80	0.59	0.41	-0.66	-0.95
control ops	0.23	-0.34	0.15	0.15	0.76	-0.08	0.14
arithm. ops	-0.20	-0.38	-0.07	-0.73	0.13	-0.17	0.29
sync ops	-0.11	0.27	-0.08	-0.66	-0.63	0.23	0.18
global mem vol.	0.22	-0.29	0.24	-0.06	0.60	-0.03	0.30
param mem vol.	-0.19	0.19	-0.27	0.44	-0.32	-0.08	-0.47
shared mem vol.	0.45	0.85	0.48	-0.30	-0.87	0.74	0.50
arithm. intensity	0.43	-0.08	0.40	-0.07	0.51	0.19	0.45

Table 5.8 Pearson correlations of feature importance and GPU properties for power prediction.

This makes sense as not all instructions consume the same amount of power. The arithmetic intensity is for both predictions rather unimportant. Could it be that most kernels of the benchmark suites are memory bound? If that were the case, this would explain why the size of the size and number of the CTAs is so important. It helps to hide latency and increase throughput. This stresses the need for parallel slackness for good performance and power efficiency. It is all about keeping the execution units busy.

5.7 Discussion

The execution time and power prediction models work well according to the nested cross-validation results. The exception is the GTX 1650, which does not have a fixed clock setting. It can be assumed that a more dynamic clock setting for core and memory frequency degrades measurement accuracy. We have the hypothesis that the dynamic clock leads to an increase in the non-deterministic behavior of the GPU and that the predictions are much worse for this reason on this device. Of course, further research with experiments would be required to verify this hypothesis. Not taking into account the GTX1650 the median MAPE scores for time prediction is ranging from 8.86% to 13.86%. The results for the power prediction range from 1.84% to 2.93% for all five GPUs.

Despite only using static input features the model produces results, which are comparable to other recent work. The results also proved that the features are indeed portable. Even though the number of samples used for training was rather limited, the models generalized well. This shows that the ensemble learning method random forest can capture the device behavior with the limited information at hand. It was also detected that consumer-grade devices with dynamic clocks cannot be modeled well.

5.7.1 Prediction Latency

The results showed that the prediction latency is in the range of 15-108 milliseconds. Compared to the execution time of the majority of the kernel this seems to be a lot. According to Thinakaran et al. [89], sub-millisecond latency is desirable for load-balancing or work-stealing. The tolerable latency depends on many factors: overhead/workload ratio, the critical path for scheduling decisions, the opportunity for memoization etc.

Since the used benchmarks are a few generations of GPU architectures old, it is not surprising that the execution times are rather short on modern GPUs. Additionally, many benchmarks are tailored to also be used in simulators. These effects lead to a lot of very short-running kernels. With the current workload, the execution times may be a lot higher.

Additionally, we like to point out that the prediction model is not optimized in any way. A custom regression model written in C++ or maybe even CUDA

will surely outperform the current python implementation. If that is not enough for low prediction latency, there is still the option of feature selection and trading accuracy for latency.

Therefore, we certainly see the opportunity of using the prediction models for scheduling reasons. The advantage of not needing to collect input features for every device will surely outweigh some disadvantages and makes this approach feasible for a wide range of heterogeneous devices.

5.7.2 Similar Approaches

Work of Arafa et al. [68] and Guerreiro et al. [30] are recent works that are quite similar to our approach. Guerreiro et al. are also using machine learning on PTX code to predict execution time and power consumption on GPU kernels. Their model is a recurrent neural network and supports DVFS. The kernel code needs to be unrolled and cannot contain dynamic control flow dependencies. This disadvantage allows them to predict kernel performance based on a PTX instruction stream instead of an instruction histogram like in this work. The approach of Arafa et al. is more different and leverages an event-based GPU kernel simulation and an analytical model. Even though simulation is used, they avoid cycle-level simulation, which is usually rather slow. Their tool PPT-GPU is also based on PTX instruction streams. The PTX code is processed for additional information like dependencies. Their model is also able to factor in cache behavior. We tested the public model of PPT-GPU, which currently lacks support for caching. The MAPE error score for the polybench-GPU benchmark suite was 433.8%. Our model could achieve an error score of 218.6%. These numbers have to be considered with care because a fair comparison is difficult and requires a rework of the learning methodology. Still, the simulation is more time-consuming than the machine learning model.

Using a PTX instruction stream gives the model more information, which is certainly an advantage. However, a direct comparison with PPT-GPU showed that just using instruction histograms is competitive. Thanks to the CUDA Flux profiler, our approach is able to capture dynamic control flow. Ideally, future work would capture PTX instruction streams and also dynamic control flow. Collecting and processing more data will certainly need more processing time and if the possibly better predictions are worth the effort needs to be discovered

in the future.

5.7.3 Limitations

Our proposed approach has some limitations which we would like to discuss:

Training Data: Even though four benchmark suites were used, a larger and more diverse training data set would be helpful. There is a bias towards short-running kernels. Possibly, due to the age of the benchmarks and GPU simulators as execution targets. Recent GPUs do not utilize all their SMs fully for all benchmarks. For the GTX1650 14.59% of the kernels do not use all available SMs and for the V100 the percentage increases to 56.08% (ignoring register usage). This shows a lack of highly parallel workloads. Regarding the power measurement method, the applied method may not reflect actual power usage accurately for short-running kernels. The power sampling frequency is just too low. We believe that our workaround utilizing loops for longer execution solves this problem to some degree, but it cannot be used for all kernel types.

Model Features: reduced precision in arithmetic (16bit and 8bit floating-point operations) would currently be treated like 32bit floating-point operations. The instructions could be weighted by the bit width or processed into a separate feature. In the case of separate features, there may be a lack of training data to ensure good predictions. Additional features reflecting the degree of optimization would certainly be helpful. Uncoalesced memory accesses for example can impart kernel performance by a large factor. The model is currently not able to directly detect such factors. Other performance bugs like shared memory bank conflicts, bad cache hit-rates or branch divergence should ideally also be detected and quantified. Mapping these properties of a kernel into features is non-trivial and may require serious effort.

Model Training: The model could benefit from a larger hyper-parameter space and regularization of some parameters such as the number of estimators. There needs to be a balance between prediction latency and accuracy as well. Additionally, the model could predict the throughput of instructions for example instead of kernel execution time. This may improve the situation of predicting kernels outside the unseen execution time range. In our experiments, this heavily impacted the accuracy and thus was not implemented.

CHAPTER 6

Conclusions for Automated Multi-GPU Partitioning

In this work, several key findings have emerged, shedding light on important aspects of multi-GPU computing:

1. *Models built on compute graph-based simulations are sufficient for automated partitioning decisions.*

   Using compute graph-based simulations as a foundation for building models can effectively automate partitioning decisions. By simulating only necessary aspects of the application, execution time estimates can be computed at a low computational cost. It has been shown, that this simulation can be used to create models that approximately resemble real application behavior. Even with the very simple regression model, good partitioning decisions could be done. On average the partitioning decisions are only 10.17% slower than the optimal partitioning, over all simulated scenarios.

2. *Instrumentation-based profiling is a valuable tool and offers immense performance benefits.*

   Using the PTX assembly for profiling kernels gives deep insight into what is happening in kernels and since PTX is portable across different GPUs the profiling results can be transferred. The average overhead on the application level is 13.20 times the normal execution time. Using NVIDIA's profiler nvprof the average overhead is 171.01. By enabling the collection

of detailed data on kernel execution with low overhead, researchers can gain insights into the behavior and bottlenecks of CUDA kernels at a larger scale. This helps to identify areas for improvement and optimization, which emphasizes the importance of this finding.

3. *GPU-independent metrics are viable as input features for performance models.*

   The static input features used by the GPU Mangrove approach produce results comparable to similar work. The median average percentage error ranges from 8.86-52.0% for time and 1.84-2.94% for power prediction. Consumer GPUs with a wider dynamic frequency range are more difficult to predict. Given that the model does not consider core and memory frequencies as input and these cannot be fixed for consumer-grade GPUs, this is no surprise.

In this work, we asked how multi-GPU programming framework can be made partitioning agnostic. We believe to have found one feasible way by leveraging partitioned compute graphs that have been automatically generated. The compute graphs allow reasoning about the application as a whole and its execution on a specific system. Delivering good performance by optimizing the partitioning is still a hard task with multiple different challenges to solve. In this work, we showed that kernel time prediction can be solved elegantly for many different GPUs. However, other building blocks like our simulation should be improved further and in the end, all the building blocks of this method need to be integrated.

Unfortunately, this work has a few short-comings:

The number of applications used with compute graph-based workload partitioning is very low. The problem was that there are few toy examples with compute graphs that are not too complex. In addition, the simulated results should be compared to real benchmark results and almost no benchmarks which are using multiple-GPUs and allow to set the number of GPUs and the workload size freely.

Regarding the multi-GPU systems that have been included in this work, there are no systems featuring NVLink. NVLink are direct high-bandwidth connections between GPUs and the finding on those systems could be very different. The higher communication bandwidth lowers the communication cost,

which in term would make the use of more GPUs in communication intense workloads worthwhile.

We would like to end this dissertation with an outlook into future work:

To use simulators like Pick-Sim with many applications, it is currently required to reproduce the compute graph programmatically. Capturing the graph using static analysis or instrumentation of the application would reduce the effort in obtaining more compute graphs drastically. These graphs can not only be used for simulation but also for analysis of the applications, which could help to detect patterns in them that can be leveraged to accelerate the processing. Our implementation generates the compute graphs partitioned for a fixed number of GPUs. Enabling the partitioning of non-partitioned compute graphs would lower the effort for graph generation even more.

The GPU Mangrove approach showed how the kernel execution time estimation could be improved. Regarding the simulation of communication, a similar improvement could be done. Better estimation of the memory transfers would make the simulation much more accurate. How do concurrent transfers influence each other and how can the overheads be quantified? A more diverse set of interconnects should be explored.

As mentioned in the limitations of this work dynamic frequency settings have not been considered. It would be a great addition to extend both Pick-Sim and the GPU Mangrove approach to support changing clocks.

This dissertation has resulted in valuable findings and tools which can help advance further research. We hope that the future of multi-GPU computing will become more accessible to developers by letting frameworks take care of the delicate details of partitioning GPU workloads and optimizing their execution for efficiency. Let the immense computational power of GPUs help to improve the future of everyone.

These final words of this dissertation are dedicated to the ever-ongoing pursuit of knowledge and may this be an inspiration to aspiring scholars to start and finish their intellectual exploration. The way may be long and challenging, but it is a worthy cause to contribute to the scientific community.

Acknowledgements

I would like to express my deep appreciation and thanks to all those who have supported me through the journey of completing my doctoral dissertation.

First and foremost, I would like to express my deepest appreciation to my supervisor, Prof. Dr. Holger Fröning. Without him, I would have never started that journey, which helped me to grow personally and professionally. I am very grateful for the freedom and support he gave me during this time and that he was able to excite and motivate me. I cannot understate how helpful it was, that I could always have an open discussion with him, and he was always very dedicated to giving me good advice and giving me quality feedback.

I would like to extend my thanks to all the researchers of the scientific community, with whom I interacted. For this a want to thank Tobias Grosser, who inspired me a lot at the beginning of my PhD. Next, I want to thank Prof. Dr. Vincent Heuveline, Dr. Simon Gawlok and Sotirios Nikas, with whom I had the pleasure of working on a „Bundesministerium für Bildung und Forschung" (BMBF) project together. I would like to acknowledge the financial support provided by the BMBF, for which I am very thankful.

Additionally, many thanks to my fellow PhD Students Alexander Matz, Günther Schindler, Felix Zahn, Benjamin Klenk, Bernhard Klein, Yannick Emonds, Jonas Dann, Hendrick Borras, Vahdaneh Kiani and Kazem Shekofteh for their encouragement and inspiring discussions. Besides them, I want to thank all the students, which I had the pleasure to supervise.

Finally, I would also like to acknowledge the support and encouragement from my family and friends. I thank them for believing in me, for their patience and for the stability they gave me during demanding times.

List of Figures

List of Tables

Bibliography

[1] Oak Ridge National Laboratory. “Frontier”, Oak Ridge Leadership Computing Facility. (2022), [Online]. Available: https://www.olcf.ornl.gov/frontier/ (visited on 02/03/2023).

[2] LUMI consortium. “LUMI’s full system architecture revealed”, LUMI. (2021), [Online]. Available: https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/ (visited on 02/03/2023).

[3] CINECA. “Leonardo HPC System | Leonardo Pre-exascale Supercomputer”. (2022), [Online]. Available: https://leonardo-supercomputer.cineca.eu/hpc-system/ (visited on 02/03/2023).

[4] Oak Ridge National Laboratory. “Summit”, Oak Ridge Leadership Computing Facility. (2018), [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/ (visited on 02/03/2023).

[5] Amazon Web Services. “Amazon EC2 Instance Types - Amazon Web Services”, Amazon Web Services, Inc. (2023), [Online]. Available: https://aws.amazon.com/ec2/instance-types/ (visited on 02/03/2023).

[6] Google. “GPU platforms | Compute Engine Documentation”, Google Cloud. (2023), [Online]. Available: https://cloud.google.com/compute/docs/gpus (visited on 02/03/2023).

[7] Microsoft. “Pricing - Linux Virtual Machines | Microsoft Azure”. (2023), [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/ (visited on 02/03/2023).

[8] NAS Program, NASA Ames Research Center. “NAS Parallel Benchmarks”, NAS Parallel Benchmarks. (2022), [Online]. Available: https://www.nas.nasa.gov/software/npb.html (visited on 05/17/2023).

[9] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, “Memory access patterns: The missing piece of the multi-GPU puzzle”, ACM Press, 2015, pp. 1–12. DOI: 10.1145/2807591.2807611.

[10] A. Matz, J. Doerfert, and H. Fröning, “Automated Partitioning of Data-Parallel Kernels using Polyhedral Compilation”, in *49th International Conference on Parallel Processing - ICPP : Workshops*, Edmonton AB Canada: ACM, 2020, pp. 1–10. DOI: 10/gjr3rw.

[11] NVIDIA Corporation. "CUDA C++ Programming Guide". (2023), [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 04/08/2023).

[12] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures", in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 863–874. DOI: 10/crdd6g.

[13] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping", in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, New York, NY, USA: Association for Computing Machinery, 2009, pp. 45–55. DOI: 10/fk4qtc.

[14] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems", *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 62–80, 2018. DOI: 10/gcsg75.

[15] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs", *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 277–288, 2011. DOI: 10/df4n2j.

[16] A. Matz and H. Fröning, "Quantifying the NUMA Behavior of Partitioned GPGPU Applications", in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 53–62. DOI: 10.1145/3300053.3319420.

[17] K. Asanovic *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley", p. 56, 2006.

[18] J. Enmyren and C. W. Kessler, "SkePU: A multi-backend skeleton programming library for multi-GPU systems", in *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, ser. HLPP '10, New York, NY, USA: Association for Computing Machinery, 2010, pp. 5–14. DOI: 10.1145/1863482.1863487.

[19] M. Steuwer, P. Kegel, and S. Gorlatch, "Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library", in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012, pp. 1858–1865. DOI: 10/ggmz4r.

[20] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs", in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, New York, NY, USA: Association for Computing Machinery, 2019, pp. 372–383. DOI: 10.1145/3352460.3358307.

[21] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems", in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques - PACT '10*, Vienna, Austria: ACM Press, 2010, p. 353. DOI: 10.1145/1854273.1854318.

[22] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili, "Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures", in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, 2012, pp. 58–67. DOI: 10.1109/ISPASS.2012.6189206.

[23] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator", in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[24] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Chicago, IL, USA: IEEE, 2007, pp. 407–420. DOI: 10.1109/MICRO.2007.30.

[25] S. Collange, D. Defour, and D. Parello, "Barra, a modular functional gpu simulator for gpgpu", *University de Perpignan, Tech. Rep*, 2009.

[26] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing", in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, New York, NY, USA: Association for Computing Machinery, 2012, pp. 335–344. DOI: 10/ggmz5c.

[27] X. Gong, R. Ubal, and D. Kaeli, "Multi2Sim Kepler: A detailed architectural GPU simulator", in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 269–278. DOI: 10.1109/ISPASS.2017.7975298.

[28] M. Stephenson *et al.*, "Flexible software profiling of GPU architectures", in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, Portland, Oregon: ACM Press, 2015, pp. 185–197. DOI: 10.1145/2749469.2750375.

[29] L. Braun, S. Nikas, C. Song, V. Heuveline, and H. Fröning, "A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels", 2020. arXiv: 2001.07104 [cs].

[30] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "GPU Static Modeling Using PTX and Deep Structured Learning", *IEEE Access*, vol. 7, pp. 159 150–159 161, 2019. DOI: 10.1109/ACCESS.2019.2951218.

[31] K. Fan, B. Cosenza, and B. Juurlink, "Predictable GPUs Frequency Scaling for Energy and Performance", in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019, Kyoto, Japan: Association for Computing Machinery, 2019, pp. 1–10. DOI: 10.1145/3337821.3337833.

[32] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing", IEEE, 2009, pp. 44–54. DOI: 10.1109/iiswc.2009.5306797.

[33] A. Danalis *et al.*, "The Scalable Heterogeneous Computing (SHOC) benchmark suite", in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3, Pittsburgh, Pennsylvania, USA: ACM Press, 2010, pp. 63–74. DOI: 10.1145/1735688.1735702.

[34] J. A. Stratton *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing", 2012, p. 12.

[35] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes", in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339595.

[36] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures", *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. DOI: 10/fwnvb4.

[37] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least Angle Regression", *The Annals of Statistics*, vol. 32, no. 2, 2004. DOI: 10/bjvr62. arXiv: math/0406456.

[38] L. Braun and H. Fröning, "CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications", in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 73–81. DOI: 10.1109/PMBS49563.2019.00014.

[39] "CUDA Pro Tip: Nvprof is Your Handy Universal GPU Profiler", NVIDIA Technical Blog. (2013), [Online]. Available: https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/ (visited on 05/09/2022).

[40] CASL Gatech. "Lynx: A Dynamic Instrumentation System for GPGPU Computing | CASL Gatech". (2013), [Online]. Available: https://casl.gatech.edu/research/lynx-a-dynamic-instrumentation-system-for-gpgpu-computing/ (visited on 05/09/2022).

[41] J. Wu *et al.*, "Gpucc: An open-source GPGPU compiler", presented at the Proceedings of the 2016 International Symposium on Code Generation and Optimization, ACM, 2016, pp. 105–116.

[42] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA: IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[43] LLVM Foundation. “Compiling CUDA C/C++ with LLVM — LLVM 3.9 documentation”. (2016), [Online]. Available: https://releases.llvm.org/3.9.0/docs/CompileCudaWithLLVM.html (visited on 03/22/2022).

[44] NVIDIA Corporation. “Parallel Thread Execution ISA Version 7.6”. (2022), [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html (visited on 03/23/2022).

[45] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs”, in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151. DOI: 10.1109/IISWC.2012.6402918.

[46] N. Kondratyuk, V. Nikolskiy, D. Pavlov, and V. Stegailov, “GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP”, *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 312–324, 2021. DOI: 10.1177/10943420211008288.

[47] Advanced Micro Devices. “AMD ROCm Profiler — ROCm 4.5.0 documentation”. (2022), [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html (visited on 03/24/2022).

[48] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, “A Framework for Dynamically Instrumenting GPU Compute Applications Within GPU Ocelot”, in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4, New York, NY, USA: ACM, 2011, 9:1–9:9. DOI: 10.1145/1964179.1964192.

[49] D. Shen, S. L. Song, A. Li, and X. Liu, “CUDAAdvisor: LLVM-based runtime profiling for modern GPUs”, in *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*, Vienna, Austria: ACM Press, 2018, pp. 214–227. DOI: 10.1145/3168831.

[50] P. Bumbulis and D. D. Cowan, “RE2C: A more versatile scanner generator”, *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, pp. 70–84, 1993. DOI: 10.1145/176454.176487.

[51] NVIDIA Corporation. “Nvidia-tesla-p100-PCIe-datasheet.pdf”. (2016), [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf (visited on 04/11/2022).

[52] NVIDIA Corporation. “Volta-v100-datasheet-update-us-1165301-r5.pdf”. (2020), [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf (visited on 04/11/2022).

[53] A. Koike and K. Sadakane, “A Novel Computational Model for GPUs with Application to I/O Optimal Sorting Algorithms”, IEEE, 2014, pp. 614–623. DOI: 10.1109/IPDPSW.2014.72.

[54] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort, "The landscape of GPGPU performance modeling tools", *Parallel Computing*, vol. 56, pp. 18–33, 2016. DOI: 10.1016/j.parco.2016.04.002.

[55] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning", IEEE, 2015, pp. 564–576. DOI: 10.1109/HPCA.2015.7056063.

[56] T. C. Carroll and P. W. Wong, "An Improved Abstract GPU Model with Data Transfer", IEEE, 2017, pp. 113–120. DOI: 10.1109/ICPPW.2017.28.

[57] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness", p. 12, 2009.

[58] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures", in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 382–393. DOI: 10.1109/HPCA.2011.5749745.

[59] S. Hong and H. Kim, "An integrated GPU power and performance model", p. 10, 2010.

[60] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU Performance Modeling Technique Based on Interval Analysis", IEEE, 2014, pp. 268–279. DOI: 10.1109/MICRO.2014.59.

[61] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures", p. 10, 2010.

[62] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures", IEEE, 2013, pp. 673–686. DOI: 10.1109/IPDPS.2013.73.

[63] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters", IEEE, 2010, pp. 115–122. DOI: 10.1109/GREENCOMP.2010.5598315.

[64] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling", IEEE, 2012, pp. 1–11. DOI: 10.1109/SC.2012.20.

[65] J. Chen, B. Li, Y. Zhang, L. Peng, and J. Peir, "Statistical GPU power analysis using tree-based methods", in *2011 International Green Computing Conference and Workshops*, 2011, pp. 1–6. DOI: 10.1109/IGCC.2011.6008582.

[66] B. Johnston, G. Falzon, and J. Milthorpe, "OpenCL Performance Prediction using Architecture-Independent Features", 2018. arXiv: 1811.00156 [cs].

[67] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, "A comparison of GPU execution time prediction using machine learning and analytical modeling", in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, 2016, pp. 326–333. DOI: 10.1109/NCA.2016.7778637.

[68] Y. Arafa, A.-H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "PPT-GPU: Scalable GPU Performance Modeling", *IEEE Computer Architecture Letters*, vol. 18, no. LA-UR-18-30853, pp. 55–58, 1 2019. DOI: 10.1109/LCA.2019.2904497.

[69] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power Modeling for GPU Architectures Using McPAT", *ACM Transactions on Design Automation of Electronic Systems*, vol. 19, no. 3, 26:1–26:24, 2014. DOI: 10/gnhv42.

[70] X. Wang, K. Huang, A. Knoll, and X. Qian, "A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation", in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 506–518. DOI: 10/gnhv6f.

[71] S. Salaria, A. Drozd, A. Podobas, and S. Matsuoka, "Learning Neural Representations for Predicting GPU Performance", in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 40–58. DOI: 10/gnhv6m.

[72] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi, "Dynamic GPGPU Power Management Using Adaptive Model Predictive Control", in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 613–624. DOI: 10.1109/HPCA.2017.34.

[73] P. Reisert, A. Calotoiu, S. Shudler, and F. Wolf, "Following the Blind Seer – Creating Better Performance Models Using Less Information", in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds., Cham: Springer International Publishing, 2017, pp. 106–118. DOI: 10.1007/978-3-319-64203-1_8.

[74] Y. Arafa, A.-H. Badawy, G. Chennupati, A. Barai, N. Santhi, and S. Eidenbenz, "Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles", in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–12. DOI: 10.1145/3392717.3392761.

[75] L. G. Valiant, "A bridging model for parallel computation", *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990. DOI: 10.1145/79173.79181.

[76] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment", in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–10. DOI: 10.1109/HPCA.2010.5463058.

[77] C. Lehnert, R. Berrendorf, J. P. Ecker, and F. Mannuss, "Performance Prediction and Ranking of SpMV Kernels on GPU Architectures", in *Euro-Par 2016: Parallel Processing*, P.-F. Dutot and D. Trystram, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 90–102. DOI: 10.1007/978-3-319-43659-3_7.

[78] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "GPGPU Power Modeling for Multi-domain Voltage-Frequency Scaling", in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Vienna: IEEE, 2018, pp. 789–800. DOI: 10.1109/HPCA.2018.00072.

[79] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction", in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08, New York, NY, USA: Association for Computing Machinery, 2008, pp. 368–377. DOI: 10.1145/1375527.1375580.

[80] Q. Wang and X. Chu, "GPGPU Performance Estimation with Core and Memory Frequency Scaling", 2018. arXiv: 1701.05308 [cs].

[81] A. Botchkarev, "A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms", *Interdisciplinary Journal of Information, Knowledge, and Management*, vol. 14, pp. 045–076, 2019. DOI: 10.28945/4184.

[82] V. Adhinarayanan and W.-c. Feng, "An automated framework for characterizing and subsetting GPGPU workloads", IEEE, 2016, pp. 307–317. DOI: 10.1109/ISPASS.2016.7482105.

[83] NVIDIA Corporation. "NVIDIA Management Library (NVML)", NVIDIA Developer. (2011), [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml (visited on 04/14/2022).

[84] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU Power with the K20 Built-in Sensor", in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7, New York, NY, USA: Association for Computing Machinery, 2014, pp. 28–36. DOI: 10.1145/2588768.2576783.

[85] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface.* Rev. 4. ed. Amsterdam; Heidelberg: Elsevier Morgan Kaufmann, 2012.

[86] S. Varma and R. Simon, "Bias in error estimation when using cross-validation for model selection", *BMC Bioinformatics*, vol. 7, no. 1, p. 91, 2006. DOI: 10.1186/1471-2105-7-91.

[87] R. J. Tibshirani and R. Tibshirani, "A Bias Correction for the Minimum Error Rate in Cross-Validation", *The Annals of Applied Statistics*, vol. 3, no. 2, pp. 822–829, 2009. JSTOR: 30244266.

[88] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, no. 12, pp. 2825–2830, 2011.

[89] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters", in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 977–987. DOI: 10.1109/ICDCS.2017.262.

APPENDIX A

Simplifying Multi-GPU Programming with Unified Memory Usage

Can the programmability problem be eased by a simplified programming model? Unified memory allows paging of memory on the GPU. The developer needs not worry about moving data to and from the GPU as this can be triggered automatically when a page fault occurs.

This possible aid was examined in some preliminary experiments and a bachelor thesis by Georg Weisert (advised by Lorenz Braun). The starting point of this consideration is a single GPU application, which should be accelerated by using multi-GPUs. Using unified memory is as simple as replacing memory allocations with `cudaMallocManaged` and omitting copy operations to and from the GPU. For usage of multiple GPUs, the kernel needs to be split up, which is done at thread block level. Kernels just need to be given an offset such that each kernel launch on each GPU can work on a different subset of the thread blocks. Because this task can be automated source-to-source transformation, which was shown by Weisert, this solution is a potential low-hanging fruit for the programmability problem.

In experiments, a reduction, matrix multiplication and n-body application were transformed into multi-GPU applications using unified memory.

The results in figure A.1 show that unified memory is faster for some applications, but the performance impact, in general, is quite high. Table A.1 shows the

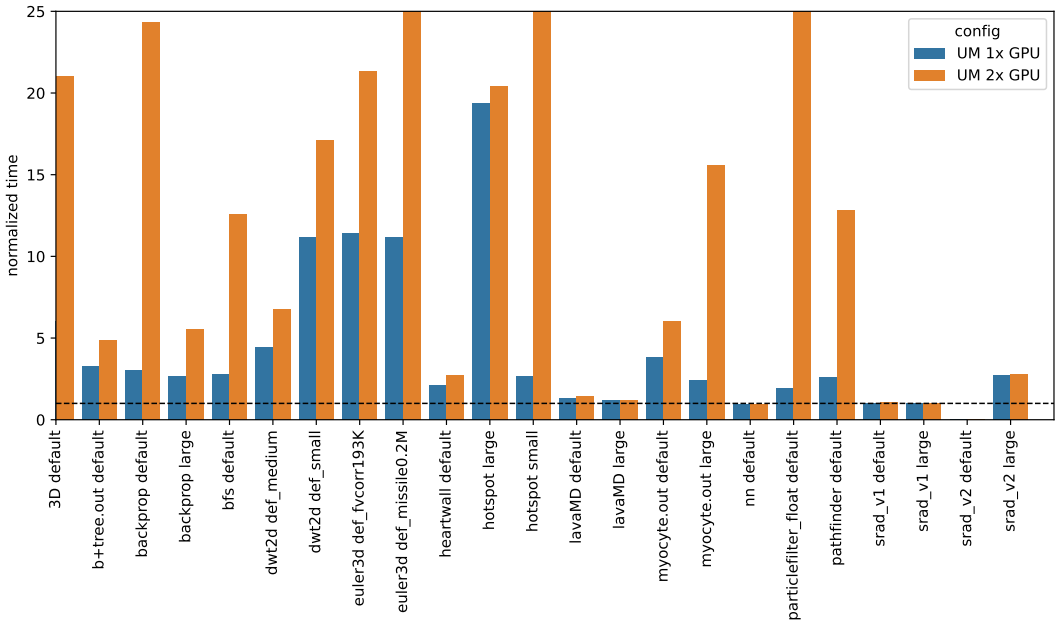

Figure A.1 Benchmark execution time of selected applications from the Rodinia benchmark suite [32] using unified memory, normalized to unmodified benchmark execution time.

App	Dataset	Um 2x GPUs
euler3d	def_missile0.2M	73.285187
hotspot	small	73.359306
particlefilter_float	default	54.344262

Table A.1 Overhead results for selected application-dataset combinations.

results of three application-dataset combinations, which did not fit in the plot on the figure Since the overhead is usually higher than the performance gained by using two GPUs, performant and energy-efficient multi-GPU applications need another solution.