DISSERTATION

submitted to the

Combined Faculty of Natural Sciences and Mathematics

of the

Heidelberg University, Germany

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Alexander Matz

born in
Görlitz, Sachsen

Heidelberg, 2020

Exploiting BSP Abstractions for Compiler Based Optimizations of GPU Applications on multi-GPU Systems

Advisor: Professor Dr. Holger Fröning

Oral examination: .....

I dedicate this dissertation to my loving parents

Gabriele and Torsten Matz

Abstract

Graphics Processing Units (GPUs) are accelerators for computers and provide massive amounts of computational power and bandwidth for amenable applications. While effectively utilizing an individual GPU already requires a high level of skill, effectively utilizing multiple GPUs introduces completely new types of challenges. This work sets out to investigate how the hierarchical execution model of GPUs can be exploited to simplify the utilization of such multi-GPU systems.

The investigation starts with an analysis of the memory access patterns exhibited by applications from common GPU benchmark suites. Memory access patterns are collected using custom instrumentation and a simple simulation then analyzes the patterns and identifies implicit communication across the different levels of the execution hierarchy. The analysis reveals that for most GPU applications memory accesses are highly localized and there exists a way to partition the workload so that the communication volume grows slower than the aggregated bandwidth for growing numbers of GPUs.

Next, an application model based on Z-polyhedra is derived that formalizes the distribution of work across multiple GPUs and allows the identification of data dependencies. The model is then used to implement a prototype compiler that consumes single-GPU programs and produces executables that distribute GPU workloads across all available GPUs in a system. It uses static analysis to identify memory access patterns and polyhedral code generation in combination with a dynamic tracking system to efficiently resolve data dependencies. The prototype is implemented as an extension to the LLVM/Clang compiler and published in full source.

The prototype compiler is then evaluated using a set of benchmark applications. While the prototype is limited in its applicability by technical issues, it provides impressive speedups of up to 12.4x on 16 GPUs for amenable applications. An in-depth analysis of the application runtime reveals that dependency resolution takes up less than 10% of the runtime, often significantly less.

A discussion follows and puts the work into context by presenting and differentiating related work, reflecting critically on the work itself and an outlook of the aspects that could be explored as part of this research. The work concludes with a summary and a closing opinion.

Zusammenfassung

Grafikkarten sind Beschleuniger für Computer, die durch ihre extrem hohe parallele Rechenleistung und Speicherbandbreite für bestimmte Arten von Anwendungen bestechen. Während die effiziente Nutzung von individuellen Grafikkarte bereits hohe technische Fähigkeiten von Entwicklern verlangt, wird die effiziente Nutzung mehrerer Grafikkarte durch völlig neue Arten von Problemen erschwert. Diese Arbeit erforscht, wie das hierarchische Ausführungsmodell der Grafikkarten ausgenutzt werden kann, um den Schritt von einer Grafikkarte zu mehreren parallel verwendeten Grafikkarten zu erleichtern.

Einleitend wird eine Analyse der Speicherzugriffsmuster von populären Grafikkarten Testprogrammen durchgeführt. Die Speicherzugriffsmuster werden zunächst mittels eigens entwickelter Instrumentierung gesammelt und dann in einer Simulation auf implizierte Kommunikation hin untersucht. Diese Analyse zeigt, dass die Mehrheit der untersuchten Anwendungen stark lokalisierte Speicherzugriffe aufweisen und sich auf eine Art aufteilen lassen, bei der das Kommunikationsvolumen langsamer mit der Anzahl Grafikkarten wächst, als ihre aggregierte Bandbreite.

Darauf folgt der Entwurf eines Anwendungsmodells basierend auf Z-Polyhedra, das die Aufteilung von Arbeit formalisiert und die Identifikation von Datenabhängigkeiten ermöglicht. Dieses Modell wird dann für die Umsetzung eines Compiler-Prototypen genutzt, der Grafikkarten Anwendungen so übersetzt, dass sie auf mehreren Grafikkarten ausführbar sind. Der Prototyp nutzt statische Analyse um die Speicherzugriffsmuster in das Anwendungsmodell zu übersetzen und generiert code, der in Kombination mit einem dynamischen Trackingsystem Datenabhängigkeiten effizient auflöst. Als Grundlage für den Prototyp dient das LLVM Projekt und der gesamte Quelltext is frei zugänglich veröffentlicht.

Der Compiler-Prototyp wird anschließend mit Hilfe von extra angefertigten Anwendungen ausgewertet. Obwohl technische Probleme den Anwendungsbereich des Compilers einschränken, liefert er bei Anwendungen, die der statischen Analyse zugänglich sind, beeindruckende Beschleunigung von bis zu 12.4x bei Ausführung auf 16 Grafikkarten. Eine tiefergehende Analyse der Programmlaufzeit kommt zu dem Schluss, dass die Auflösung von Datenabhängigkeiten maximal 10% und meist deutlich weniger beträgt.

Im Anschluss folgt eine Diskussion, besteht aus einem vergleich mit verwandten Arbeiten und einem Ausblick auf interessante zukünftige Arbeiten. Die Arbeit schließt mit einer Rekapitulation der wichtigsten Punkte und einer Stellungnahme zu den Ergebnissen ab.

Contents

CHAPTER 1

Introduction

Computers as the defining invention of the information age have enabled breakthroughs in a variety of fields. They execute computations that are so massive in scale that even large groups of humans simply can not perform the same computations within a reasonable time-frame. Equations that describe real-world phenomena, such as Maxwell's equations behavior of electro-magnetic fields or Euler equations for the motion of non-viscous fluids, are computationally so expensive that their manual application typically was limited to more easily solvable corner cases. The invention of powerful computers suddenly provided the opportunity to perform these calculations kick-started the field of computational science that focuses on computers primarily as tools for scientific research. The wild success of computers has made them pervasive in virtually every industry, from the characterization and prediction of financial markets, 3-dimensional x-ray imaging and gene sequencing in medical sciences, to the simulation of galaxies and analysis of the cosmos' background radiation in astronomy. In all of these examples, having access to more computational power provides more accurate results and as a result. As a result, since the invention of the first computer, the manufacturers of their components have been in a steady result to provide ever better-performing products.

A common way to improve the performance of a computer system is the addition of specialized chips that have a limited set of functions but outperform the Central Processing Unit (CPU) at these functions. These chips are called accelerators, and a very successful example of them is the Graphics Processing Units (GPUs). They were originally invented to speed up the synthesis of (three-dimensional) computer images and

are designed to compute a very large number of simple linear algebra problems, being both faster than and more energy-efficient than CPUs for this purpose. They quickly stirred the interest of the scientific community, which began using them for scientific purposes other than image synthesis, creating the General Purpose Programming on Graphics Processing Units (GPGPU) community. Their focus on computer graphics has led to different design decisions than those made for CPUs. Instead of aggressively optimizing the execution of an individual computation to be as fast as possible, they are optimized to execute a large batch of similar computations in a short time, sacrificing the execution time of any individual computation. They require special software to be programmed and the two most important GPGPU frameworks, OpenCL and CUDA, rely on the so-called Bulk-Synchronous Parallel (BSP) programming model as their primary abstraction [1]. In the BSP programming model, an application is broken down into sections, so-called "super steps", which contain a high number of small programs (called threads) that can be executed independently of each other and synchronization is only possible at the boundaries of these super steps. Both OpenCL and CUDA provide a relaxed implementation of this that allows limited communication between parallel threads in small groups. By providing many more threads than can be executed on the hardware, while also limiting communication between any two threads, memory latencies can be hidden and performance is portable between different generations and designs GPUs. This is a very important feature of data-parallel languages such as OpenCL and CUDA, as it hides architectural aspects from the user while providing consistent performance, independent of actual hardware configurations.

For applications that require large amounts of parallel computations, GPUs often provide the majority of the computation power and turn into the bottleneck of the system. While installing additional GPUs into the system increases the available raw processing power, applications need to be aware of this, which introduces additional complexity. Efficiently utilizing an individual GPU already requires high skill and adding the need to synchronize GPUs and manage data movements between them is a significant burden. Although more recent CUDA versions are tackling this complexity by allowing different GPUs to directly access each other's main memory, creating a Virtual Shared Memory (VSM), accessing memory located on another GPU is orders of magnitudes slower. The additional complexity required to efficiently utilize multiple GPUs in a single node is comparable to utilizing multiple nodes in traditional cluster systems. While the distribution of work across multiple GPUs benefits from the BSP programming model and its relaxed guarantees between multiple threads, data distribution and collection requires the insertion of communications code around all

BSP super steps. This transformation requires careful changes throughout the whole application, a process that is tedious and error-prone.

This work analyzes the challenges of utilizing multiple GPUs and derives a way to simplify the transition from one to multiple GPUs. It relies on the key observation that both the BSP programming GPU hardware favor programs that contain limited control flow and access memory in a predictable way. By creating a model of the memory accesses, data dependencies can be identified and the data transfers required by a partitioned application can be computed automatically. Using this information, the application can then be safely partitioned over multiple GPUs, each computing a subset of the solution and communication where necessary. This work can be performed automatically by a compiler that translates a given GPU program from source code to an executable file. Such an automatic partitioning provides a massive increase in productivity as it transparently allows the utilization of multiple GPUs while keeping the relative simplicity of the single-GPU programming model.

An analysis of the memory accesses performed by applications taken from popular GPU-benchmarks suites suggests that GPU applications are expected to be amenable to such an automated partitioning. The analysis uses the memory accesses to determine the minimum amount of communication that the given application would require to compute correct results when being split into multiple partitions. The primary takeaway of this analysis is that most applications can be split in such a way that communication volume grows sublinear with the number of partitions, suggesting that the aggregated bandwidth of additional GPUs can more than balance the additional volume.

This work presents a concept and prototype implementation for an automatically partitioning CUDA compiler that relies on a hybrid optimization scheme utilizing static code analysis and a runtime support system. The static code analysis extracts a mathematical model of the application's memory accesses in GPU memory and a transformation step distributes work across multiple GPUs and inserts code for synchronization and communication. The communications code is generated using optimizing libraries to efficiently describe data dependencies to minimize runtime overheads. A runtime system determines the target GPU of data dependencies and minimizes data transfers by implementing a simple coherence scheme. A fall-back solution provides support for non-partitionable GPU kernels. The prototype compiler is explained in detail and evaluated using a set of custom benchmarks. It is based on the gpucc CUDA compiler that is integrated into the LLVM compiler project [2]. Although CUDA has been chosen as the sole input language for pragmatic reasons,

there are no conceptual differences when replacing CUDA with OpenCL. While this work focuses on the distribution of work across the GPUs of a single node, the approach can be transparently applied to GPU-clusters or cloud environments using existing software [3].

Contributions

This work makes the following contributions:

1. *Instrumentation for gpucc to capture GPU memory access on CTA granularity.* As a precursor to the memory access analysis of GPU applications, an LLVM module has been developed that instruments the kernels in CUDA applications to log their memory accesses. Memory accesses are logged in Cooperative Thread Array (CTA) granularity and written to disk for off-line analysis. The design of instrumentation pass and capturing infrastructure is explained in this work and the source code is published for open access.

2. *A trace-based analysis of the inter-CTA communication exhibited by single-GPU applications.* The applications are a selection of benchmarks from three established GPU benchmark suites. Using the custom instrumentation pass, memory accesses are collected and analyzed off-line using a simple simulation. The simulation creates virtual partitions of the kernel and uses a memory tracking system to determine the interactions between CTAs that imply communication during the application's lifetime. Different partitioning schemes are used and their effect on the CTAs' communication is analyzed. The methodology and implementation of the analysis are the successors to a publication in the *12th Workshop on General Purpose Processing 2019* [4].

3. *Design and implementation of a prototype for an automatically partitioning compiler for CUDA.* An open-source CUDA compiler is modified to automatically partition GPU applications based on an analysis of its kernels memory access patterns is presented. First, the concept is discussed in the context of existing similar solutions with their benefits and drawbacks. The tasks required to partition an application are identified, potential solutions evaluated in theory, and a suitable one is selected. The resulting compiler uses a combination of polyhedral compilation and a dynamic runtime system to minimize data transfers while not requiring to fully model inter-kernel dependencies. The different parts

of the implementation are grouped into functional units and explained in detail. The work implements an improved concept of the prototype presented at the *9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2016).* [1].

4. *An evaluation of the automatically partitioning compiler prototype.* The compiler prototype is evaluated using a set of custom workloads representing typical computation profiles seen in scientific applications. The evaluation includes a discussion of the technical limitations of the implementation, an analysis of the speedups that have been achieved on a single-node system with 16 NVIDIA GPUs, and an evaluation of the overhead introduced by the runtime system. It consists of a quantitative analysis of the measured performance of the generated binaries and a qualitative analysis comparing the application's computational and space complexity to their scaling behavior when executed in multiple partitions.

Organization of this Work

The remainder of this work is structured as follows. Chapter 2 provides the information required for the understanding of this work: the architecture and programming model for GPUs, the open-source compiler infrastructure used in this work, trace collection, and the polyhedral model. Chapter 3 presents the trace-based analysis of single-GPU memory access patterns. It covers the toolchain implemented to collect the memory traces as well as the benchmark selection and evaluates the traces by replaying them in a virtually partitioned application. The results are discussed and interpreted. The concept of an automatically partitioning compiler, as well as the details of its prototype implementation, are explained in Chapter 4. The prototype is implemented partially as a compiler plugin and partially as a runtime library. Most of the complex code is found in the static analysis and code transformations, while the runtime library implements most of the algorithms that are used for work partitioning and buffer synchronization. An in-depth evaluation of the prototype implementations is presented in Chapter 5. The evaluation is based on three custom benchmarks with different characteristics and memory access patterns. A functional evaluation discusses technical limitations, followed by an analysis of the speedup achieved on a multi-GPU test system and measurements of the overhead that is introduced by the dynamic parts of the runtime

[1] The automatically partitioning compiler is part of the "Mekong" umbrella project. The name is taken from the Mekong delta in Vietnam, a natural formation in which the Mekong river branches out into a large number of smaller rivers.

support system. In Chapter 6.1, the work is put into context in its field by discussing and differentiating work related to the individual aspects of this work or the general approach in its entirety. The work concludes in Chapter 7, which critically discusses the result of this work, provides an outlook, and parts with final words.

CHAPTER 2

Background

This chapter provides background on the major topics that have been relevant as part of this work. The sections are written to familiarize the reader enough with each topic to be able to follow this work. Each topic represents a field of research in its own right and is too vast to provide an in-depth introduction within the scope of this thesis. Interested readers are encouraged to use the provided references as starting points for further reading.

2.1 GPU Architecture and Programming Models

The basic choices in the hardware designs for GPUs and CPUs differ significantly due to the different requirements set for their performance. CPUs traditionally are designed to optimize single-thread performance, where the goal is to execute an individual computation in the shortest possible time [5]. In contrast, graphics computations typically require massive amounts of linear algebra computations that are all independent of each other and only the total time to finish all of these computation matters. GPUs are therefore designed to optimize throughput for massively parallel computations, sacrificing single-thread performance[6]. Although general-purpose GPU programming has become pervasive today in computationally intensive areas, it started as a "creative misappropriation" of the hardware through specially prepared textures and filter settings in graphics computations [7]. The introduction of programmable shader units allowed the implementation of complex filters and algorithms, greatly simplifying general-purpose GPU programming and result in a demand for GPUs in scientific

computing [8, 9, 10]. This high demand led to the creation of tools for explicit GPGPU programming, such as CUDA and OpenCL [11, 12].

Statements made in this section about NVIDIA hardware and the CUDA programming model are taken primarily from official white papers and online documentation released by NVIDIA unless cited otherwise [13, 14, 12]. While this work focuses on NVIDIA GPUs and the programming model implemented by CUDA, the general architecture and programming models of GPUs from other vendors are similar and the general statements apply to them as well.

The differences in the design between GPUs and CPUs can be summarized like this: GPUs contain thousands of fairly simple cores instead of a few very powerful ones found in CPUs. Many of the more detailed design decisions follow as a consequence to facilitate this overarching premise. GPU cores are simplified to such a degree that they don't even contain an individual scheduler but share a single one within a group (of size 32 in current NVIDIA hardware), called a *warp*. As a consequence, the cores can not be programmed individually but instead present a *single instruction multiple thread* system, where all cores of the system execute the same instruction with different operands in lockstep [15, 13]. The main memory follows a *relaxed consistency* model with severely limited guarantees about the visibility of modifications, reducing synchronization overhead and increasing parallel throughput. Caches are not implicit but explicit, simplifying hardware at the cost of requiring manual management. Host-system synchronization and I/O-facilities are severely limited to further increase parallel throughput. While some of these missing features are beginning to be implemented in more recent versions of the CUDA hardware and software (CUDA 9.0 and onwards), they are still severely limited compared to CPUs.

CUDA is a proprietary GPGPU programming language based on C++, developed and endorsed by NVIDIA to program their hardware. The GPGPU programming model used by CUDA is an implementation of the Bulk Synchronous Parallel (BSP) programming model [14]. In this programming model, a parallel program is divided into *super-steps* that execute a large number of threads in parallel without synchronization, followed by a synchronization point for all threads in-between super-steps [1]. In CUDA terminology, the super-steps are called a *kernel* and execute in a two-level hierarchy of threads. The first level of the hierarchy is called the (thread) *grid*, a large, 3-dimensional grid containing a thread block on each node of the grid. Each of these *thread blocks* itself consists of a large number of individual threads that are organized into another 3-dimensional cuboid, forming the second level of the hierarchy.

Within a thread block, threads share an explicit cache, can synchronize execution

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

int main() {
  int n = /* problem size */;
  float scale = /* scaling factor */;
  float a_dev = cudaMalloc(sizeof(float) * n);
  cudaMemcpy(a_dev, /* host buf */, sizeof(float) * n,
             cudaMemcpyHostToDevice);
  float b_dev = cudaMalloc(sizeof(float) * n);
  dim3 block(512, 1, 1);
  dim3 grid((int)ceil((double)n / block.x), 1, 1);
  kernel_scale <<< grid, block >>> (b, a, scale, n);
  cudaMemcpy(b_host, b_dev, sizeof(float) * n,
             cudaMemcpyDeviceToHost);
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

Figure 2.1: A simplified example code in CUDA showing code intended to run on the device and the host-code syntax for the configuration and launch of a kernel.

with each other, and provide guarantees about progress. Thread blocks are the atomic scheduling unit in CUDA: threads in a thread-block can only be scheduled for execution as a whole, not individually. Communication or synchronization between threads of different thread blocks is generally not possible [1]. The weak/non-existent guarantees about the visibility of memory accesses and lack of synchronization effectively mean that thread blocks execute fully independently of each other.

Applications written in CUDA are targeting two different architectures: the host CPU and the NVIDIA GPU executing the kernels. Although the code for both architectures can, and often is, written in a single source code file, the functions targeting each architecture can be clearly distinguished from each other and provide slightly different semantics. Definitions and functions intended for the GPU are called *device code* and are prefixed by the macros `__global__` or `__device__`. The code in a CUDA kernel describes the code for an individual thread, which is then executed by a potentially very large number of actual threads. In order to differentiate both control flow and data for each thread, a 3-dimensional index for each thread is provided that

[1] The exception is a memory fence across the whole system, which is rarely used due to dramatic performance implications and a weak implementation [16].

uniquely identifies the thread in the thread hierarchy. This unique global index is calculated in figure 2.1 in line 2 and then first used to avoid an out-of-bounds error in line 3 and to index a unique element in the arrays `a` and `b` in line 4. This usage pattern for thread indices can be found in almost every CUDA application. OpenCL provides a built-in command that returns the global unique index directly, skipping the equivalent formula `threadIdx.x + blockIdx.x * blockDim.x` used in CUDA.

The threads of each thread block in a GPU can be configured to have accesses to a portion of the manually managed cache, called *shared* memory in CUDA terminology. While a consistent view of this cache is also not enforced between threads by default, CUDA provides memory fences that enforce consistency. Shared memory is located on-chip and therefore much faster than the GPU's main memory, called *global* memory. The lower latency is particularly beneficial for repeatedly accessed values shared by multiple threads, empirical proof of which is the high popularity of tiling optimizations on the granularity of thread blocks [17, 18].

Device code is compiled down into an intermediate Instruction Set Architecture (ISA) that is portable between different iterations of NVIDIA GPUs, called Parallel Thread eXecution (PTX). It is then translated into machine code for the actual GPU found in the system by the driver before kernel execution.

Analogously to device code, definitions and functions intended for the host architecture are referred to as *host code* and are implemented in regular C++, except for a special notation that configures and launches a CUDA kernel on a GPU. In addition to launching kernels, the host code also takes care of the auxiliary code required for the kernels to run, in particular allocation and management of memory buffers on the GPU. The code in figure 2.1 from line 10 to line 20 shows a simplified version of the code required to launch a kernel. First, a buffer is allocated on the device using the `cudaMalloc` API call, which is then filled in from a regular host buffer by the `cudaMemcpy` API call. Then, the thread block and the thread grid sizes are computed in such a way that the total number of threads is guaranteed to be at least as large the number of array elements, a pattern commonly found in CUDA code. Finally, the kernel is launched using the special kernel launch syntax, followed by another `cudaMemcpy` call to copy the result from GPU memory back to host memory. While the kernel launch is an asynchronous operation and might return before the kernel has finished execution, operations on a given GPU are sequential, so the synchronous `cudaMemcpy` call waits for the kernel to finish before it starts copying. The application finishes by releasing the GPU buffers.

The host and device code found in a CUDA source files are compiled as a unit and

produce a single binary that executes on any system with a compatible driver and CUDA enabled hardware. The GPU architectures are backward-compatible, i.e. newer GPUs can always execute code compiled for older architectures and any translation steps are handled transparently by the driver.

The CUDA API has supported the parallel utilization of multiple GPUs in parallel since its release. Multi-GPU support is implemented by providing a function that allows setting any available GPU as the "active" GPU that will be the target of all subsequent CUDA API calls. If only a single GPU is to be utilized, this function can be simply ignored as the API is initialized to use the "first available" GPU in the system by default.

2.2 The LLVM Compiler Infrastructure Project and Clang

LLVM (originally "low-level virtual machine" but not used as an acronym anymore) is a collection of compiler components for program analysis and optimization that is designed to provide high reusability and customizability [19]. It achieves this goal by defining a common Intermediate Representation (IR) that is shared by all analysis and transformation components and used as the basis for all optimizations. At first glance, this IR looks like a form of assembly and provides an ISA for a theoretical computer architecture. However, programs translated into the IR are not intended to be directly executed like this as no processor implementing this ISA exists, instead it just provides a model to represent programs in a simpler form than the original programming language. Differences to assembly languages targeting actual processor architectures include:

- LLVM IR is strongly typed with a language-independent type-system that supports compound data-types such as arrays and structures. Types are never automatically coerced and any non-trivial casts require an explicit instruction with clearly defined semantics.
- An instruction dedicated to type-safe address arithmetic, enabling transparent integration of pointer calculations into the type-system.
- LLVM IR is in Static Single Assignment (SSA) form, a way to represent programs that share many similarities with functional programming [20]. This requires

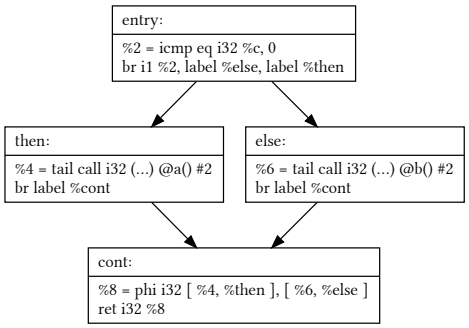
that the majority of all instructions are side-effect free. Exceptions to this rule exist, the most prominent ones being load and store instructions.

Some of the information presented in this section is specific to LLVM/Clang while some applies to compiler construction in general. There are multiple sources about the basics and details of compiler design [21, 22, 23]. All of these cover the general architecture of compilers (which components they can (or should) consist of), as well as specific information about program parsing, optimization pipelines, and code generation. Unless supported by a specific citation, statements made about general compiler design made in this section are based on the information these sources. LLVM and Clang specific information is based on more informal sources. While there are peer official and/or peer-reviewed publications about LLVM and Clang [19, 24], the projects are moving at a very fast pace and reference its online documentation and in particular its source code as the authoritative sources [25, 26]. Statements made in this section about Clang and LLVM, unless cited otherwise, are based primarily on information from online documentation.

The powerful type system can implement many high-level features of popular programming languages. Directly representing these high-level features and their constraints in IR, allows more aggressive optimizations by the compiler by providing additional semantics for instructions and reducing the number of assumptions required about a code section. Additionally, integrating pointer arithmetic directly into the IR simplifies reasoning about memory safety of applications written in typically unsafe languages, such as C [27].

LLVM IR being in SSA form is the largest difference to regular assembly languages. Assembly languages are typically a direct translation of machine-code into a human-readable format and represent the specific code to run on a particular processor with a fixed amount of registers. Programs represented in assembly heavily rely on side-effects to implement a given algorithm: inputs and outputs of functions are communicated using specific registers and memory locations defined by an Application Binary Interface (ABI). Program representations in SSA, on the other hand, are designed to be mostly side-effect free: inputs and outputs to functions are passed using named or anonymous values without specifying their storage.

Although these values are often described as “registers” of a machine with an unlimited number of registers, this description is a simplification. Values in SSA form (as used by LLVM) are not limited to the word size of any specific processor and can only be assigned a name (“written to a register”) by exactly one instruction, hence the name “Static Single Assignment”. This creates a problem when trying to represent



(a) The control flow graph of the LLVM IR code.

```
1  entry:
2    %2 = icmp eq i32 %c, 0
3    br i1 %2, label %else, \
4                  label %then
5  then:
6    %4 = tail call i32 (...) @a()
7    br label %cont
8  else:
9    %6 = tail call i32 (...) @b()
10   br label %cont
11 cont:
12   %res = phi i32 [ %4, %then ], \
13                  [ %6, %then ]
14   ret i32 %res
15 }
```

(b) The LLVM IR generated for code that chooses between two values based on a variable "%c".

Figure 2.2: Example program that requires a phi-instruction to represent divergent control flow.

values that evolve over the runtime of the program, such as loop induction variables. SSA solves this problem by introducing the phi-instruction (also ϕ-instruction/node), a pseudo-instruction that "magically" selects between different values depending on previous control flow. Figure 2.2 illustrates the Control Flow Graph (CFG) of a function that selects between two values, for example from a statement such as `return c ? a() : b();`. The code first diverges at the condition in the "entry" block and then converges in the "cont" block, returning the result of either the "then" or the "else" block. Since it is illegal for programs in SSA form to assign two different values to the same name, a new value is created in each block: the value `%4` in the then-block and `%6` in the else-block. They are merged into a single value by the phi-instruction in the cont-block by assigning either the `%4` or `%6` value to `%res`, depending from where the cont-block was entered.

The phi-instruction is not intended to be executed on real hardware and only used to simplify data flow analysis during optimization stages by providing a side-effect free way of representing changing values in SSA. During code generation, the phi-instruction is translated into one or more instructions on the target hardware that together provide identical semantics. As an example, when generating code for x86-based architectures, the code shown in 2.2 can be translated into mov instructions that store the value of `a()` and `b()` directly into the e.g. `EAX` register in the respective branch. Depending on

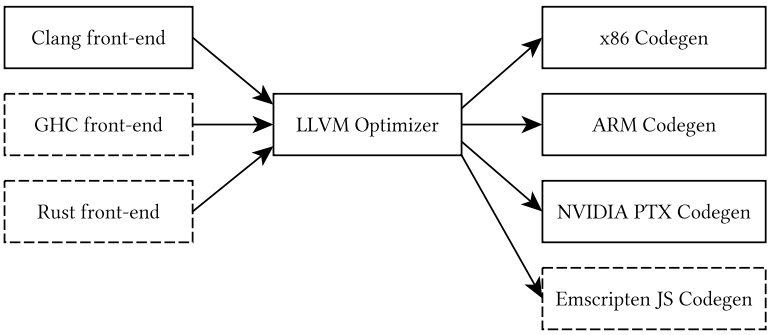


Figure 2.3: Simplified architecture of the Clang compiler. Examples for related toolchains that reuse parts are depicted in dashed outlines.

which path is taken, the correct value can be found in the `EAX` register and returned as the result. This implements the logic represented by the phi-instruction at the cost of introducing state, since the `EAX` register is not freely available for computations anymore.

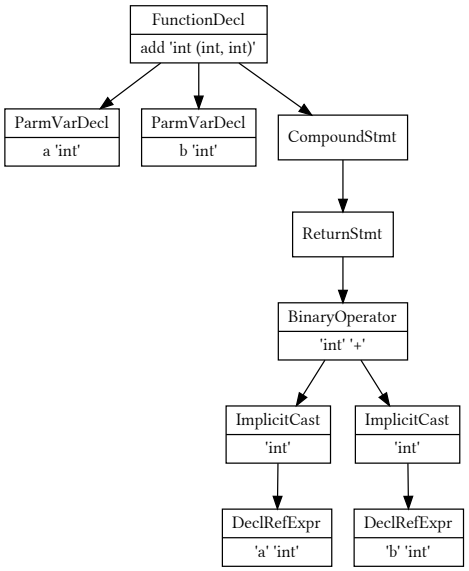
By providing this well-defined intermediate representation as the primary interface between the components of the LLVM toolchain, the individual components can be very loosely coupled and designed in a highly reusable way. Since the majority of the core LLVM project is implemented as a set of mostly independent analysis and transformation passes, the core project is rarely used on its own and instead typically integrated into a fully-featured compiler toolchain. One such toolchain that is maintained as part of the LLVM project is the Clang project, a compiler for languages of the C-family of languages [24].

Clang implements the generic design of an optimizing compiler by being split into the following three basic components: front-end (parser), middle-end (optimizer), and back-end (code generator). This design simplifies porting a compiler to new input languages or target architectures by decoupling input, processing, and output. A simplified view of the LLVM architecture is shown in figure 2.3 and includes toolchains other than Clang that are based on LLVM [28, 29, 30].

First, the *front-end* reads a program in the input programming language and generates an easier processable intermediate representation of the program. This is done by first parsing the source code into an Abstract Syntax Tree (AST), which is a structured high-level representation of the input. Parsing is often done in a two-step process: A lexer splits the input data stream into individual tokens such as identifies, numbers, parentheses, and others, while ignoring insignificant parts such as white-space or comments. The tokens are then fed into a parser that uses rules (based on a grammar

```
1  int add(int a, int b) {
2    return a + b;
3  }
```

(a) C-Code.



(b) Clang AST.

```
1  define i32 @add(i32, i32) {
2    %3 = alloca i32, align 4
3    %4 = alloca i32, align 4
4    store i32 %0, i32* %3, align 4
5    store i32 %1, i32* %4, align 4
6    %5 = load i32, i32* %3, align 4
7    %6 = load i32, i32* %4, align 4
8    %7 = add nsw i32 %5, %6
9    ret i32 %7
10 }
```

(c) Unoptimized LLVM IR.

Figure 2.4: A function adding two numbers in its different forms as it processed by the Clang front-end.

describing the structure of the programming language) to assemble the tokens into an AST that is returned as the result. If it encounters an unexpected token, it simply aborts with an error message. The AST is then translated into unoptimized LLVM IR. Since AST is considered a high-level representation containing complex program structures and IR is much simpler and can not model all of the high-level information, this step is called "lowering". Figure 2.4 illustrates the process for a function that adds two numbers and returns the result. Each translation of the input into a different form performs a lowering from a representation with complex semantics into an equivalent representation using simpler semantics.

Second, the *middle-end* takes unoptimized LLVM IR from the front-end and passes it through a series of optimizations. This optimizer stage represents the heart of the LLVM project in the context of the Clang compiler. Optimizations range from fairly simple such as common subexpression elimination to very complex such as aggressive restructuring and parallelization of loop nests [31, 32]. The output of the middle-end is again LLVM IR representing a semantically identical, but optimized, form of the input. Although certain optimizations rely on others as a precondition (in particular, most optimizations rely on the code being in proper SSA form), the optimizations and the

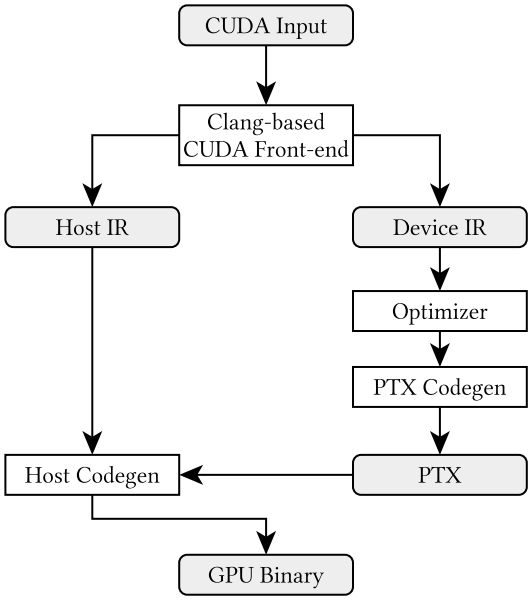


Figure 2.5: The modified Clang pipeline to support CUDA via the gpucc project.

code generator are largely independent of each other. This allows reusing the majority of the optimizations for almost arbitrary input languages or target architectures.

Lastly, the *back-end* consumes the (optimized) LLVM from the middle-end and produces code for a specific target architecture from it. Similar to the front-end, this step again lowers the program representation, from the LLVM IR into either assembly or machine code for a specific architecture, e.g. x86 or ARM. Target-specific optimizations are also applied at this stage, such as instruction combining (e.g. fused multiply-add) or removal of redundant instruction sequences [33, 34, 35]. The output of the back-end is an object-file containing machine-code specific to the target architecture.

In addition to providing all the components required by a modern compiler, the Clang project also provides an interface that connects these components and provides a convenient interface for the end-user, called the compiler driver. While the compiler driver primarily benefits the user experience, the complexity of a modern optimizing compiler is often overwhelming and necessitates such a driver for anyone but expert users.

Although Clang primarily targets the languages C, C++, and Objective-C, it also supports the proprietary GPGPU language CUDA in the form of the *gpucc* project [2]. It initially started as a separate effort by Google, but was soon fully integrated into the project and is now maintained as an official part of the LLVM and Clang projects. The combination of both host and device code in a single source code file requires

drastic changes to the regular Clang pipeline used for C or C++. Host and device code are split and then compiled separately, with the device code being compiled down to NVIDIDA PTX code first, which is then embedded into the host code. Figure 2.5 depicts an overview of the pipeline implemented by gpucc.

In the context of this work, both the front-end and the back-end of Clang are left unchanged. Some functionality is implemented as pre-processing before the front-end is executed, but the majority of the components developed in this are implemented to purely work on LLVM IR.

2.3 Memory Trace Collection on GPUs

Memory traces are a log containing a list of some or all memory accesses performed by a program, usually collected by executing an application and recording the memory accesses. Such traces are often used to classify applications, analyze their performance, or reveal potential bugs. Several ways exist to automatically gather memory traces of applications, with varying degrees of accuracy and speed.

The first approach, *static analysis*, describes an application's memory accesses using a mathematical model that is then used throughout the rest of the compilation process. These models are usually used to enable aggressive optimization, but it is possible to extract the model and use it purely for analysis purposes. This approach produces results very quickly since the analysis is typically fast enough to not prohibitively increase compile times [32]. The mathematical model must both be powerful enough to describe a wide variety of memory accesses and simple enough to provide an abstraction that can be represented by a mathematical description. While a mathematical model is not the same as a memory trace, a perfectly accurate model of an application's memory accesses can be used to infer memory traces and provides the same capabilities in the context of this work. Some common ways of representing memory accesses using a mathematical model are described in detail in section 2.4. Since all models are simplifications of the real world, there are applications they can not represent, which are typically handled by over-approximation. This limits the applicability of these approaches for the use of memory traces.

Simulation follows a different approach. Instead of analyzing an application's source code, the binary application is executed on a virtual model of the target hardware. The virtual model provides a detailed insight into the function of the lower levels of the hardware. The degree of accuracy between different simulators varies significantly. Some simulators operate on the instruction level and consequently can only extract

information at the ISA level. The Barra simulator is based on the simulator framework UNISIM and provides a functional simulator based on the CUDA PTX ISA without targeting a specific hardware target [36, 37]. A more detailed simulator is Multi2Sim, which simulates a system containing both a generic x86 CPU as well as an Advanced Micro Devices (AMD) Evergreen GPU on the instruction level [38, 39]. The highest level of detail can be extracted from simulators modeling the full architecture of a specific hardware. One such project is a reincarnation of the Multi2Sim project that simulates an NVIDIA GPU based on the Kepler architecture to a high degree of accuracy [40]. Another project is the GPGPU-Sim, which models large aspects of an NVIDIA GPU, including individual cores, memory controllers, and the central interconnection network [41]. As a consequence of this high degree of detail, executing an application on this simulator is slowed down by a factor of $170,000x$ to $2,000,000x$ [42]. For the collection of memory traces that tie a CUDA thread block to the addresses of its memory accesses, an ISA-level simulation is accurate enough since it does model the execution grid and the instructions contain the addresses in question.

The third option for the extraction of memory accesses is *instrumentation*, the compiler-based insertion of specific instructions that emit information about the application. Instrumentation is often used to collect statistics about the execution profile of an application, e.g. to identify time-consuming code regions, compute test coverage, or locate memory leaks [43, 44]. The changes required for instrumentation are usually not complex and easily automated. Two different types exist: source instrumentation and binary instrumentation. Source instrumentation refers to the insertion of instrumentation code during the normal compilation of the program [23, 45]. Depending on where in the compilation pipeline the instrumentation code is inserted, it can interfere with the optimizations performed by the compiler, so instrumentation code should be added late in the pipeline for accurate results. While there are approaches that insert the corresponding code extremely early during the preprocessor stage [46], compilers often already include some instrumentation facilities [47]. An obvious requirement for source instrumentation is access to the source code of the application that is to be instrumented. The alternative, binary instrumentation, targets compiled applications and inserts either fixed or user-defined code into an application by carefully transforming the binary file, providing very accurate results but offering limited portability. Some projects go a step further and provide facilities for the modification of application during execution time [48]. Adding instrumentation to an application always increases its runtime, whether the code is inserted during compilation or as part of a binary translation [49, 44, 50, 51]. Although the amount of

overhead depends on the specific type of instrumentation added, the slowdown observed for popular instrumentation suites is between one and two orders of magnitude [52, 44]. Memory traces collected using instrumentation are accurate on the level of individual instructions. Although the lack of access to hardware internals might prevent the user from seeing the effects of techniques such as read-combining [53], the traces reproduce the exact sequence of memory accesses as seen by the application. Instrumentation on GPUs is not as common as on CPUs, but frameworks exist. GPU Lynx is a dynamic instrumentation tool and hooks into the CUDA runtime system to install user-specified code written in a C-like language [54]. Unfortunately, the project does not support any CUDA releases after version 5.0 and is limited to producing results of a fixed size, while memory traces are unbound in the general case. SASSI is a framework for static instrumentation of CUDA kernels and fully supports CUDA to specify instrumentation hooks [55]. However, it is a closed source solution that does not seem to be actively maintained. BARRACUDA is a custom binary instrumentation tool for CUDA that is used to develop instruction traces for race detection [56]. This project comes closest to fulfilling the requirements for the analysis in this work and a patched version of this framework could have been used instead.

2.4 Polyhedral Compilation

Polyhedral compilation is a set of techniques that is primarily used to optimize highly parallel numerical codes. The common ground for these techniques is the use of a rigorous mathematical model to describe the semantics of a program during the different stages of its optimization. At the core of this model are integer sets, called Z-polyhedra and relations between them [57, 58, 59]. The base functionality is implemented by several different libraries [60, 61, 62]. Other program representations with overlapping functionality exist, but are often less expressive or require more difficult proofs of correctness [63].

The polyhedral model is particularly useful to describe the behavior of loop nests and the memory accesses of instructions contained in them. Polyhedral sets can be used to accurately model the dynamic iterations of loops, with each iteration being represented by an n-dimensional integer tuple in a set containing all possible loop iterations, n being the depth of the loop nest. In the polyhedral model as used in the context of this work, polyhedral sets are described by formulas using presburger arithmetic [64]. A set can be described by either enumerating all points in the sets, describing the conditions to be in the set using a symbolic notation or combining

existing sets. The conditions, sets, and relations used in the polyhedral model are considered to be quasi-affine [65]. An example of a possible description for the set

$$S = \{0, 1, 2, 3, 4, 8, 9, 10\}, \tag{2.1}$$

using the notation used throughout this document, is

$$\{\ S[i] : 0 \leq i \leq 5 \vee 8 \leq i \leq 10\ \}. \tag{2.2}$$

The list $[i]$ after the name of the set S is used to both state the dimensionality of the set and assign names to the dimensions. All quasi-affine conditions that define the tuples in the set are listed after the colon. The dimensions of a set are identified by their index and their names can be chosen arbitrarily, so the following is true:

$$\{\ S1[i, j] : i = 0 \wedge j = 1\ \} = \{\ S1[a, b] : a = 0 \wedge b = 1\ \} \tag{2.3}$$

Conditions can be parametrized by referencing named parameters that are declared by listing them before the set description. An example of a set with a parameter n is

$$[n] \rightarrow \{\ S[i] : 0 \leq i < n\ \}. \tag{2.4}$$

Given the parameter $n = 2$, the resulting set is

$$S = \{0, 1\}. \tag{2.5}$$

In contrast to dimensions, parameters are identified by their name instead of their position, so the following equivalence is also true:

$$[a, b] \rightarrow \{\ S[i] : a \leq i < b\ \} = [b, a] \rightarrow \{\ S[i] : a \leq i < b\ \}. \tag{2.6}$$

While naming polyhedral sets is convenient, the name can also be omitted, creating an unnamed, or anonymous, set.

Figure 2.6 shows a simple loop in C, containing two statements, S_1 and S_2. To a human programmer, it is obvious that statement S_1 is executed ten times, one time each for $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 9\}$, and statement S_2 is executed five times, one time each for $i \in \{0, 2, 4, 6, 8\}$. The execution of such a statement during run-time is considered its "dynamic instance". The polyhedral model represents these dynamic instances of a statement as the statement's **domain**. Using polyhedral sets, the domains of the

```
1       for (int i = 0; i < 10; ++i) {
2  S1:    f(i);
3         if (i % 2 == 0) {
4  S2:      f(i);
5         }
6       }
```

Figure 2.6: A simple loop containing two statements with different domains.

corresponding statements can be described as:

$$\{\ S_1[i] : 0 \leq i < 10\ \}, \text{and} \tag{2.7}$$

$$\{\ S_2[i] : 0 \leq i < 10 \wedge i \bmod 2 = 0\ \}. \tag{2.8}$$

This example show cases one of the main benefits of a polyhedral program representation: high-level semantics are retained and expressed in a mathematical sound way. Notice how loop bounds and conditional branches can be translated into simple conditions that concisely and accurately describe the loop nest without relying on any of the control flow constructs of the original programming language. An equivalent while loop would result in the same polyhedral representation.

Memory accesses in the polyhedral model are represented as linear functions that map one polyhedral set to another. The domain of such an access is the domain of the statement it belongs and the image, the output, of this polyhedral map describes a set containing the array elements that are accessed. The notation for this is an extension from the notation for polyhedral sets:

$$[n] \rightarrow \{\ [i, j] \rightarrow [o] : 0 \leq i, j < n \wedge o = i + j\ \} \tag{2.9}$$

describes an access from a 2-dimensional loop nest to a one dimensional array (both anonymous). The quasi-affine function that describes the array index is $i + j$. Analogously to the dimensions in a set, the dimensions of a map's domain and image are identified by their position, and can be arbitrarily named, resulting in the following sets all being equal

$$\{\ [i, j] \rightarrow [o1, o2] : o1 = i + j, o2 = j\ \}=, \tag{2.10}$$

$$\{\ [i, j] \rightarrow [o1 = i + j, o2] : o2 = j\ \} \quad =, \tag{2.11}$$

$$\{\ [i, j] \rightarrow [i + j, i]\ \} \tag{2.12}$$

```
1      int a[2*n];
2      int b[n];
3      for (int i = 0; i < n; ++i) {
4  S:    b[i] = b[2*i] + b[2*i + 1];
5      }
```

Figure 2.7: A loop partially reducing an array by combining adjacent elements.

Notice how the domains of these accesses are unconstrained. If these accesses were part of a program they would be executed for all points in an infinite 2-dimensional plane.

Figure 2.7 shows an example code in C that memory accesses on two different arrays. The statement S has the domain $[n] \rightarrow \{ S[i] : 0 \leq i < n \}$ and performs the following memory accesses:

$$\text{Write} : [n] \rightarrow \{ S[i] \rightarrow b[i] : 0 \leq i < n \} \tag{2.13}$$

$$\text{Read 1} : [n] \rightarrow \{ S[i] \rightarrow b[2i] : 0 \leq i < n \} \tag{2.14}$$

$$\text{Read 2} : [n] \rightarrow \{ S[i] \rightarrow b[2i+1] : 0 \leq i < n \} \tag{2.15}$$

Since polyhedral sets and maps can be manipulated using operators from set theory, the read accesses can be combined into a single map. Unions are represented by describing all sets in the union separated by semicolons, resulting in the set:

$$\begin{aligned} \text{Read} : [n] \rightarrow \{ \; & S[i] \rightarrow b[2i] : 0 \leq i < n; \\ & S[i] \rightarrow b[2i+1] : 0 \leq i < n \; \} \end{aligned} \tag{2.16}$$

This union can be simplified by coalescing the accesses, resulting the equivalent set:

$$\text{Read} : [n] \rightarrow \{ S[i] \rightarrow b[o] : 0 \leq i < n \wedge 2i <= o <= 1 + 2i \} \tag{2.17}$$

Polyhedral maps can also be used to describe dependencies between statements. For this, the domain of the map represents the dependent loop nest and the map's image represents the loop nest it depends on, both of which can be identical for dependencies within a loop nest. The edges in the map then reveal dependencies between iterations. As an example, the map

$$\{ [i] \rightarrow [o = i - 1] : 0 \leq i, o < 10 \} \tag{2.18}$$

can be used describes dependencies in a loop, where the first ten iterations must

occur in ascending order. Although this information is critical for the typical kind of optimizations performed using polyhedral frameworks [66, 67], a polyhedral model for dependencies between loop nests is not required for this work.

This representation then allows extensive analyses on the program including, but not limited to, queries such as the following [62]:

- The minimum and maximum indices of a memory access in each of its dimensions, which loosely corresponds to the bounds of an array. Both the lexicographic extremes as well as bounds for an individual dimension can be computed.
- Different kinds of hull volumes, e.g. a convex hull that potentially over-approximates the result. Such a hull volume can be used to estimate the upper bounds for the memory area affected by a memory access.
- Set theory operations on arbitrary polyhedra, for example, the intersection of the write set of one loop nest with the read set of another one as a simple way to compute data dependencies between the two.

The polyhedral model not only allows extensive analysis, but it also simplifies complex transformations. The simplification stems from the model's ability to accurately and concisely represent data dependencies and correctly applying transformations to a set to all dependent objects. The rigorous mathematical model behind polyhedral compilation greatly simplifies proving the corrects of a given transformation by reducing it to a problem in set theory. Examples for these transformations are reordering of loop nests, or tiling of a loop nest by introducing additional loops, both of which are typically used to improve the locality of memory accesses resulting in a better utilization of CPU caches. However, this work uses polyhedral transformations only in a limited way to bring the application model into a canonical form that allows reasoning about the properties of a partitioned GPU kernel.

Translating an application from the polyhedral representation back to LLVM IR or any program representation requires advanced code generation techniques [68, 69, 70, 71]. The code generation tools provided by the popular polyhedral compilation libraries typically generate highly optimized code in a high-level representation such as an AST. Control flow in the AST is represented using the primitives of structured languages, in particular, for-loops and conditionals. The polyhedral optimizer then takes the high-level representation, lowers it into IR (or assembly), and inserts the statements of the original loop iterations to produce an optimized program. After code generation, the polyhedral model of the application is not required anymore and can be discarded, making it a transient program representation.

CHAPTER 3

GPU Memory Access Patterns

This chapter covers the collection of traces containing the memory accesses of GPU applications, inference of the resulting communication if the application was partitioned among multiple GPUs, and an analysis of this communication. [1] Multi-GPU systems are typically used for workloads with very large requirements on computational power and memory bandwidth. Examples include high-performance computing (HPC) [72] and training of deep neural networks [73, 74]. Multi-GPU solutions have also been proposed to overcome the scalability limitations of monolithic GPUs [75, 76]. However, compared to single GPUs, multi-GPU systems show strong Non-Uniform Memory Access (NUMA) effects, as on-card vs. off-card throughput differs by more than one order of magnitude [77, 72]. This drastic difference in throughput requires careful data placement and memory management, complicating the programming of multi-GPU systems.

The purpose of the analysis is to estimate certain characteristics of automatically split single-GPU applications, in particular, two properties are of interest:

1. the overall expected amount of communication, and

2. the scaling of communication with the number of partitions.

Using the behavior of a single-GPU application to infer properties of multi-GPU applications might seem unintuitive at first; after all, observing multi-threaded CPU applications to learn about distributed CPU applications is unlikely to yield useful results. However, the hierarchical memory and execution models of GPUs make this

[1] This section contains parts of already published work of the author as well as original research [4].

approach feasible, as the introduction of additional GPUs just introduces another level in this hierarchy. Reliable communication between thread blocks during kernel execution is only possible in the form of atomic operations and their bad performance characteristics typically prohibit extensive use. As a result, most applications forgo the use of atomic operations and the applications are written in a way so that thread-blocks are fully independent of each other. The analysis in this chapter exploits this fact by first checking for the use of atomic operations and, if absent, virtually distributing the thread grid of single-GPU applications across multiple GPUs.

The contributions of this chapter are a tool that collects memory traces from GPGPU applications, a communication model to identify the internal communication of GPG applications, as well as an analysis based on this model of the traces of several popular benchmarks. Trace collection relies on a custom source instrumentation of the applications and is implemented as a plugin for an existing CUDA compiler. The resulting traces are then post-processed, summarizing all accesses occurring in a thread block into two sets: a read set and a write set. The summarized traces are analyzed using a simple communication model in combination with a virtual partitioning mechanism, revealing the communication that would be expected to occur if the given application was partitioned.

The analysis will be divided into five parts. First, design and implementation of the trace collection framework is presented. Then, the application model underlying the analysis is presented and it's capabilities and limitations discussed. The third part gives an overview of the popular benchmarks suites and lists the benchmarks that are part of the analysis and provides explanations for the benchmarks that had to be left out. Fourth, the collected traces of these benchmarks are analyzed and the results discussed. The chapter concludes with a summary and puts the result into its context within this work.

3.1 GPU Memory Trace Collection

This section explains the design and implementation of the custom framework that has been used to collect memory access traces. It starts by shortly arguing the merits of instrumentation of other approaches of collecting information about memory accesses and then follows with details on the instrumentation framework used in this work.

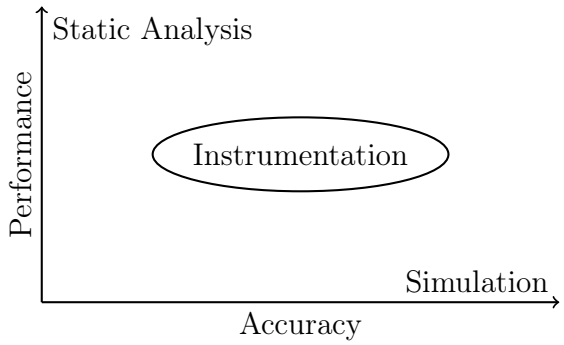


Figure 3.1: Broad comparison of different approaches to extract memory patterns based on accuracy and speed.

3.1.1 Extracting Memory Access Patterns from Applications

Several options exist to automatically extract memory access patterns from applications, as described in section 2: static analysis, instrumentation, and simulation. Figure 3.1 shows a simplified comparison of these approaches based on their two most basic characteristics accuracy and speed. Instrumentation acts as a middle-ground between the other two approaches. It does not require extensive static analysis and instead collects traces of memory accesses at runtime, similar to the simulation approach. Instead of executing the application on a simulator and using hooks inside the simulator to emit information about the application and processor state, hooks are inserted directly into the application, which is then executed on the target hardware. The hooks then call into code that emits the desired information. Inserting additional code naturally creates a runtime overhead. However, since the application is executed on real hardware instead of being simulated, this overhead is typically much lower (for CPUs about one to two orders of magnitude [52]). It has a rich history on CPU and many tools are available [49, 50, 51, 48], and any compiler can be extended to include some custom form of instrumentation. Although instrumentation can not provide the same level of detail in its traces, the available information is sufficient for the analysis, which only cares about the mapping of thread blocks to memory addresses of loads and stores.

Static analysis, on the other hand, might not be accurate enough. Although it is typically quicker than executing an instrumented application or simulating it on virtual hardware, static analysis often needs to resort to approximations for complicated memory accesses. How quickly it needs to resort to approximation depends entirely on the model in use. Most models are designed for the optimization of simple loop nests and so are usually limited to fairly simple linear, quadratic, or exponential expressions,

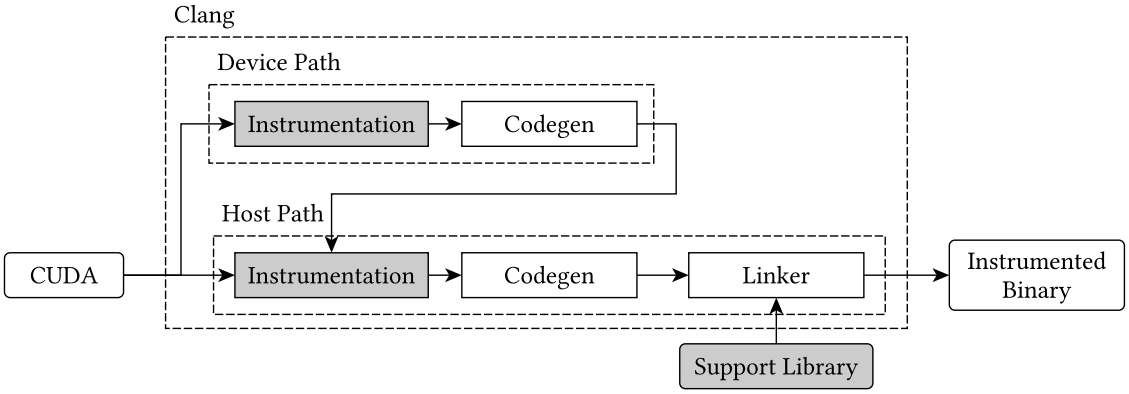


Figure 3.2: CUDA Compilation pipeline of Clang (gpucc) with tracing. Components required for memory tracing are highlighted in gray.

limiting their accuracy for general expression. Although the code of GPU applications often represents the inner loop nest and uses the thread grid as an implicit outer loop nest, this assumption can not be made in general. The accuracy of static analysis is therefore deemed too low. This leaves instrumentation as the most reasonable choice for the collection of memory access patterns.

3.1.2 CUDA Instrumentation using LLVM and Clang

The instrumentation framework used in this work is tailored specifically to the goals of the analysis: it is limited to global memory accesses only and does not distinguish between different threads of the same thread block, also called Cooperative Thread Array (CTA). Preserving the order of memory accesses within a kernel is, while a welcomed feature, not required and the resulting framework only preserves the order of memory accesses within a warp. Warps are the smallest unit of work that can be executed by a GPU, so the memory accesses in a trace file should be regarded as unordered within a kernel.

As the basis for the implementation of the instrumentation, the CUDA compiler gpucc from the Clang/LLVM project has been chosen. This greatly simplifies both the identification of insertion points for hooks and the code generation for these hooks and the code they call into. As Clang provides both the front end as well as all code generation, the instrumentation can work purely on the IR level. In contrast, binary instrumentation additionally requires parsing the machine code and updating it while keeping all relative references intact.

Figure 3.2 shows a simplified view of the original pipeline [2] and how the new

components (highlighted in gray) are integrated into it. The framework [2] is designed as a plugin for Clang/LLVM and a static library that is linked into the final application. This architecture provides a simple usage model that requires only a few compiler flags to be set to enable instrumentation. When the instrumented application is executed, a trace file is produced that, for each kernel, contains a list of records that describe each memory access of the application. Each record includes among other things the base address, kind, and origin of each memory access (the full data layout is described in figure 3.7). The instrumentation is split into three parts that roughly correspond to the components shown in figure 3.2. Device instrumentation enables trace collection and writes them into a queue that is setup using the host instrumentation; the queue implementation itself is part of the support library.

Device Code Instrumentation

The main part of the instrumentation framework is the device code instrumentation, which is broken up into three steps. The first step is preparation and force-inlines all inline-able function calls. If any un-inlined function calls remain, the instrumentation can not proceed and compilation aborts. The next step is the set-up of the context required to emit memory traces, which mainly consists of a set of queues set up by the host. Pointers to these queues are located using global device variables following a naming convention. Lastly, all load or store instructions targeting global memory are collected and instrumented. The instrumentation consists of the collection of all interesting information (CTA index, address, operand size) and pushing this information into the queue to host memory. Once the information is passed to the host, all work required by the device side is done.

Figure 3.3 illustrates a simplified version of this process. The original code simply contains to write accesses, one to some address in shared memory and one to some other address in global memory. Line 1 of the instrumented code contains the device variable declaration that will contain the pointers to the queue at runtime (notice how the name contains the kernel name, not mangled here for clarity, as part of the naming scheme). The next line shows the injected helper function `__trace` that provides access to the queue. Inside the kernel, existing instructions are left untouched but right before the store to global memory a call to the `__trace` function has been inserted (note that no such call was inserted before the write access to shared memory). The arguments to this helper function have been collected from the kernel and the store itself: the pointer to the queue, the memory access' base address, the size of the operand and

[2] Liberally licensed under MIT and hosted on Github at https://github.com/UniHD-CEG/cuda-memtrace

```
1
2
3
4
5
6
7  __global__ K(float* arr) {
8    __shared__ float shmem[512];
9    shmem[address_a] = f();
10
11
12   arr[address_b] = g();
13 }
```

(a) Original kernel code.

```
1  __device__ uint8_t *__queue_K;
2  __device__ __trace(
3      uint8_t* ptrs[2], void* addr,
4      int32_t size, uint8_t flags)
5    { ... }
6
7  __global__ K(float* arr) {
8    __shared__ float shmem[512];
9    shmem[address_a] = f();
10   __trace(__queue_K, address_b,
11       sizeof(float), WRITE);
12   arr[address_b] = g();
13 }
```

(b) Instrumented kernel code.

Figure 3.3: Simplified illustration of the device code transformation for memory access tracing on a simple CUDA code snippet.

the kind of access (read or write). Information about the CTA is gathered inside the `__trace` function.

Host Code Instrumentation

The task of the host code instrumentation is to set up the environment for the device code to produce traces and to process these traces. Most of the more complex functionality is implemented externally in the support library, reducing host code instrumentation to a two-step process:

1. Locate kernel calls, and
2. Install setup and tear down code for the support library.

Locating kernels in the host code requires a surprising amount of analysis on the IR level. LLVM IR is a low-level program representation that lacks the higher semantics of most programming languages, including CUDA kernel launches. In IR, CUDA kernel launches are lowered into a sequence of function calls into the (stateful) CUDA Runtime API. Figure 3.4 shows control flow graphs for the two possible structures of IR generated for a kernel launch. The graph on the left illustrates the inlined version of a kernel launch and the right graph illustrates the extracted version using a wrapper. The process is best illustrated using the version with inlined code. The first IR generated for the kernel launch (in the `entry` basic block) is a call to the `cudaConfigureCall` API function, configuring the execution grid and shared memory of the next kernel launch. If this call has a non-zero value, indicating an error, the code branches to a landing pad basic block (called `kcall.end`). Next, one basic block for each of the arguments of the

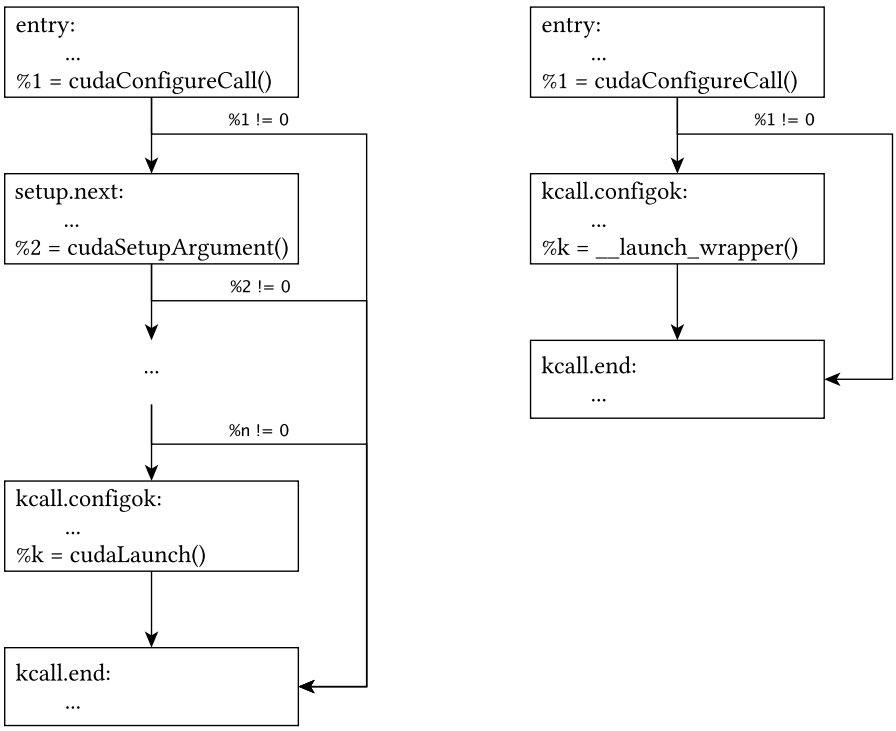


Figure 3.4: Kernel calls after lowering into LLVM IR by Clang. The left shows the inlined version, the right shows the wrapped version.

kernel is created. A call to `cudaSetupArgument` instructs the CUDA API to prepare the corresponding argument with the given value, a non-zero return value again results in aborting the kernel launch by branching to the landing pad. After the execution grid, shared memory, and arguments are set up, the kernel is launched in a basic block with the prefix `kcall.configok` using the (non-blocking) `cudaLaunch` API function. The kernel launch finishes by branching to the landing pad, regardless of the call's return value. If the code is not inlined, the argument preparation and kernel launch are extracted into a wrapper function, but the overall control flow is very similar. The pair of basic blocks (`entry`, `kcall.end`) describes a single-entry single-exit (SESE) region that encapsulates the kernel launch. This fact is exploited for kernel launch location by first location calls to `cudaConfigureCall` and then following the CFG to a basic block where the name is prefixed with `kcall.configok`. The last call in this basic block must be either a call to `cudaLaunch` or a call to the wrapper function.

Installing the setup and tear-down code then is simply a matter of inserting some function calls into a kernel launch region that has been identified. Figure 3.5 shows a simplified version of the transformation in CUDA source code. First, the CUDA stream

```
1
2
3
4 K <<<... , stream>>> (...);
5
6
```

(a) Original kernel launch.

```
1 __queue_prepare(stream, "K");
2 cudaStreamAddCallback(stream,
3   __queue_start, "K");
4 K <<<... , stream>>> (...);
5 cudaStreamAddCallback(stream,
6   __queue_stop, NULL);
```

(b) Instrumented kernel launch.

Figure 3.5: Simplified illustration of the host code transformation on a simple kernel launch.

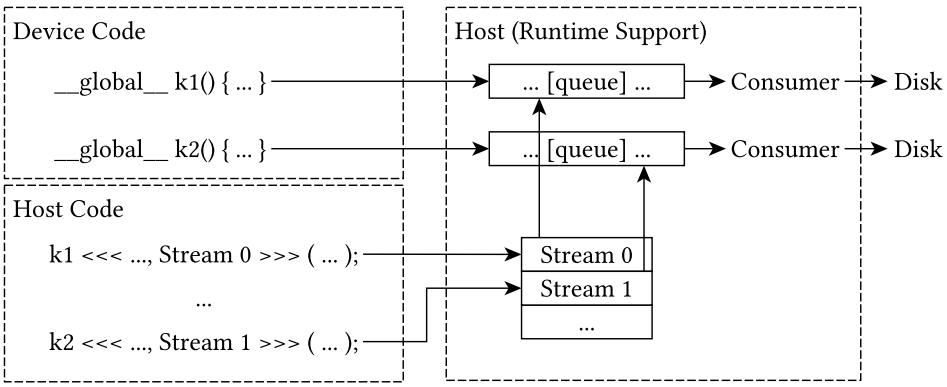


Figure 3.6: Schematic overview of the interaction of the Runtime Support System with the other components. Host Code indirectly manages the queues that the device code writes into using CUDA stream IDs as handles.

ID and the name of the kernel are extracted. The stream ID can be extracted from the call to `cudaConfigureCall` and the kernel name from either the argument `cudaLaunch` or the kernel wrapper function directly. This information is then used to insert calls into the support library (`__queue_*`) and register callbacks on the CUDA stream to start and stop a host-side queue consumer. Registering these functions as callbacks into the CUDA stream automatically handles the asynchronous behavior of CUDA streams correctly.

Runtime Support Library

The Runtime Support Library is the last component and provides the link between device code and host code instrumentation. Its primary purpose is the implementation of the shared device-host queue and a multi-threaded consumer that reads data from the queue and flushes it to disk. Figure 3.6 provides a high-level overview of the interaction between host code, device code and the Runtime Support library. The host code manages queues on a per-stream basis, using CUDA stream IDs as handles for

	0		16		32		48	60
Bit	16		16		16		12	4
0	SM ID				Size			Type
64	Address							
128	CTA X				CTA Y		CTA Z	
192	Count (disk only)							

Figure 3.7: Record format for memory accesses. The record format in the queue does not includes bits 192 .. 207, they are only used for run-length encoding on the disk.

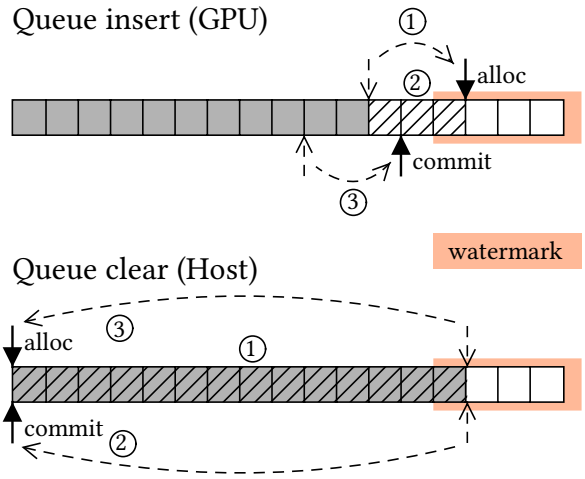


Figure 3.8: Two-phase operation of the queue: first the GPU inserts until allocations cross a watermark, then, once commit also crosses the watermark, the host clears and resets the queue.

the runtime system. It initiates the queue set up, as well as starting and stopping the consumer threads. On the device code, queue access is implemented using a convenience function that implements the queue protocol described below.

The queue is a lock-free linear buffer that contains records with a size of 24 bytes each. The binary encoding of a record is illustrated in figure 3.7 and contains the following information:

1. a 32 bits ID of the Streaming Multiprocessor issuing the memory operation,
2. a 28 bits field containing data size of the access,
3. 4 bits describing the type of access (read, write, or atomic),
4. a 64 bits field containing the base address,
5. the CTA ID (encoded in 32 bits for x and 16 bits each for y and z),

```
1  uint32_t lane_id = /* thread id within warp */;
2  uint32_t active = __ballot(1); // mask of active threads in warp
3  uint32_t n_active = __popc(active); // number of active threas in warp
4  uint32_t leader   = __ffs(active)-1; // leader = lowest active id
5
6  uint32_t slot = -1;
7
8  if (lane_id == leader) { // leader only
9    do {
10     while (*alloc > watermark) ((void)0); // wait for available slots
11     slot = atomicAdd(alloc, n_active); // get allocation
12   } while (slot > watermark) // protect against data races
13 }
14
15 // find relative id within active threads
16 uint32_t rlane_id = __popc(active << (32 - lane_id));
17 // communicate slot among from leader to rest of warp
18 uint32_t offset = __shfl(slot, lowest) + rlane_id;
19 records[offset] = /* record information */;
20
21 __threadfence()_system; // wait for reads to be visible by host
22 if (lane_id == leader ) { // leader only
23   atomicAdd(commit, n_active); // commit write accesses
24 }
```

Figure 3.9: Warp synchronized device access to the queue (simplified).

6. and a 16 bits field used for run-length encoding for the disk-version of this format.

Instead of implementing the queue using a circular buffer, a simpler, linear implementation operating in two phases is used, as shown in figure 3.8. Access to the loop is coordinated using a pair of pointers: an *alloc* pointer and a *commit* pointer. The *alloc* pointer is updated atomically (①) to allocate slots in the queue for new records and to communicate the number of records that have been inserted into the queue (pending and finished insertions). Then the allocated number of records is written into the corresponding slots (②) and finally, the *commit* pointer is incremented (③). Therefore, the difference $alloc - commit$ is exactly equal to the number of outstanding insertions. Unfortunately, the order in which insert operations finish is not necessarily the same order in which they are started. As a consequence, there is no guarantee that data in slots before the *commit* pointer has been safely written. On the device, write access is coordinated in warps to reduce synchronization by electing a leader thread within the warp, which then performs the atomic operations on behalf of all active threads. The maximum number of active threads in a warp is 32, which makes this number the largest number of slots that can be atomically allocated. A watermark is

```
while (true) {
  // wait for queue to be ready
  while(*commit < watermark) ((void)0);

  uint32_t nrecords = *commit;
  record_t record = records[0]; // we know there is at least 1 record
  uint16_t counter = 1; // initialize counter
  for (int i = 1; i < nrecords; ++i) { // iterate over all records
    record_t ref = records[i];
    // run-length encoding for similar accesses
    if (ref.size == record.size && ref.smid == record.type &&
        ref.type == record.type && ref.cta == record.cta &&
        ref.address == record.address + counter * ref.size) {
      counter += 1;
    } else {
      // if this is a new record, flush record + counter to disk and
          reinitialize
      write_record(record, counter);
      record = ref;
      counter = 1;
    }
  }
  write_record(record, counter); // flush last record
  *commit = 0; // reset commit first
  *alloc = 0; // then alloc, reopening queue for device.
}
```

Figure 3.10: Consumer thread code running on the host including run-length encoding(simplified).

declared at the position $N - 32$, with N being the total number of slots, and write access by the device is prohibited once the *alloc* pointer passes the watermark. This secures the queue against overflows and guarantees that all outstanding operations have finished once the *commit* pointer has passed the watermark. Figure 3.9 shows a simplified version of the device helper function that orchestrates access to the queue, acting as the producer of this producer-consumer relationship.

On the host side, the consumer thread starts by waiting for the *commit* pointer to cross the watermark, since the queue is linear instead of circular. Flushing the queue earlier may yield incorrect results, since increments to the *commit* pointer do not necessarily follow the order of increments to the *alloc* pointer, resulting in potentially unfinished insertions into the queue in slots before the *commit* pointer. Once the queue is full, a running record and counter are initialized using the first record found in the queue. Then each record in the queue is compared to the running record: if it is an access of the same size and type on a contiguous memory address, a counter is

increased, otherwise, the running record and counter are flushed to disk and reset to the record under inspection. After all records have been read from the queue (①), the remaining running record is flushed to disk and the *alloc* and *commit* pointers are reset. Resetting the *alloc* pointer immediately switches control back to the GPU so the *commit* pointer needs to be reset first to remain a consistent state (② and ③). This process is then repeated until the consumer thread is killed. Run-length encoding for the applications analyzed in this work provides compression ratios of up to 25 : 1 depending on the regularity of the application, with the worst case being an increase in file size of about 8 %. Due to its simplicity, virtually non-existent run-time overhead, and small potential disk space overhead it is applied to all traces regardless of the regularity of the application.

3.1.3 Trace Compression as a Post-Processing Step

The raw compressed traces are still very large and easily reach several Gigabytes of data. This increases the processing time and renders an iterative data exploration prohibitively time-consuming.

Additionally, much of the data found in the traces is redundant: 1) repeated identical accesses and the order of accesses within a kernel are irrelevant, and 2) the run-length encoding only considers consecutive records in the queue (when reordering might expose additional compressible data). Instead of introducing additional complexity into the runtime, it is kept light-weight and a high-quality compression scheme is implemented as a post-processing step.

The compression implemented in post-processing is lossy and discards information about both ordering of accesses within a kernel and the data-size of accesses. It translates application traces from a sequential format logging individual memory accesses to a large lookup table that maps CTAs within a kernel to the regions in memory they accessed. Each consecutive region in memory is described by a half-closed range containing the bytes that have been accessed. Let $(addr, size)$ be a read access of size $size$ to base address $addr$, if the threads in a CTA now perform the set of read accesses $((0,4),(4,4),(16,4),(20,4))$, they are transformed into the two byte intervals (not $(addr, size)$ accesses) $[0,8) \cup [16,24)$.

Definition 1. ***Read/Write Set****. The read set $R(B)$ of a thread block B is the set of all addresses in global memory read at by B at least once. Analogously, the write set $W(B)$ is the set of all addresses in global memory written to by B at least once.*

This transformation is performed by keeping a list of the regions each CTA has read, written or accessed atomically, iterating through the trace, and incrementally updating the byte intervals. Atomic accesses are a combination of a read and a write access with special semantics. Since these semantics do not matter for the analysis, no list of atomic accesses is built but they are simply treated as a read-write access. The list of intervals for read and write accesses describe sets of bytes accessed by a given CTA and are respectively called that CTAs *read set* or *write set* (see definition 1. A memory access from the trace is then inserted into the sorted list at the corresponding position, or merged with either one or two existing intervals if the access overlaps or adjoins them. For example, consider some CTA with a read set $[0, 8) \cup [16, 24)$. A read access to the interval $[8, 8)$ would result in a new read set $[0, 24)$, while a read access to $[4, 8)$ would instead result in the new read set $[0, 12) \cup [16, 24)$.

The output of the post-processing step is a map

$$\begin{aligned} K_{\vec{n}} :&[0, n_1 - 1] \times [0, n_2 - 1] \times [0, n_3 - 1] \to \mathcal{P}(\mathbb{N}_0) \times \mathcal{P}(\mathbb{N}_0) \\ &(x, y, z) \mapsto (R(x, y, z), W(x, y, z)) \end{aligned}$$

that provides the read set R and the write set W for a given CTA with the 3-dimensional index (x, y, z), part of the thread grid $\vec{n} \in \mathbb{N}^3$. Such a map is created for all kernels in the application trace in the order they were launched. This summary is in-line with the CUDA consistency model, which for this analysis allows ignoring the order of loads and stores during a single kernel execution.

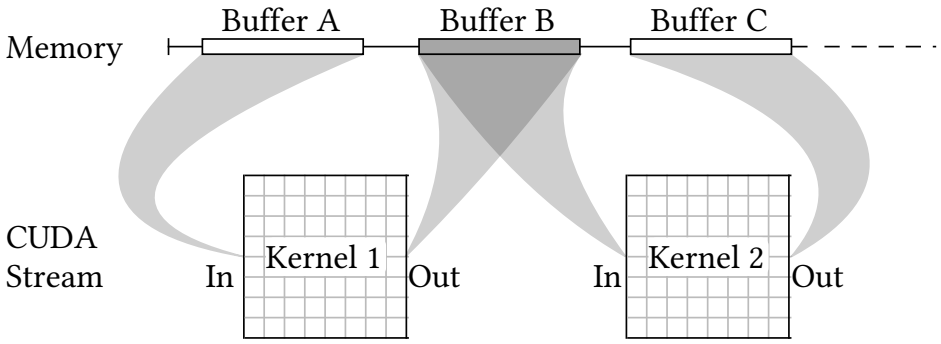
The map is stored in a relational database of about 5.3 GB for the summaries of the benchmarks presented in 3.3, compared to the about 29 GB of the compressed traces. A relational database is used due to its proven high maturity compared to a custom solution as well as the possibility to issue flexible and sophisticated queries.

Having the data collected, post-processed, and stored, enables extensive analysis on this data to gather insights on the implicit communication of CUDA applications.

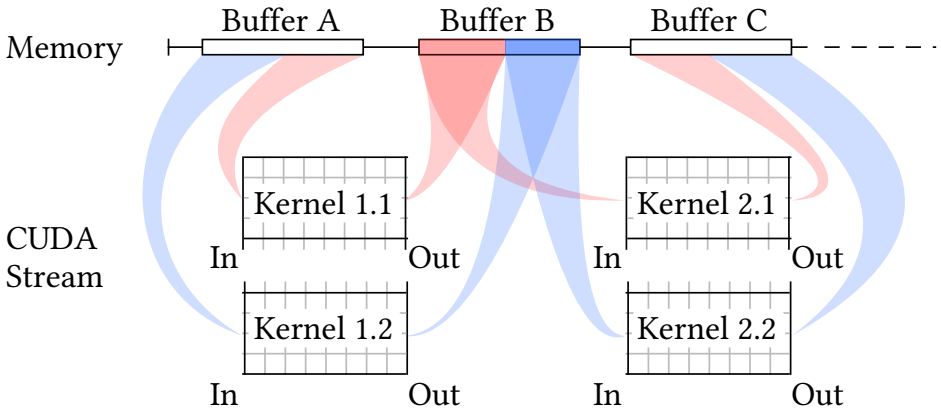
3.2 Analysis Model

This section explains the model that was used to analyze the internal communication of GPU Applications. It is designed to work with traces collected according to the description in 3.2. The model is split into two largely independent parts.

First, a communication model is devised that differentiates memory accesses that are considered “communication” from those that are not. It formalizes the meaning of



(a) Buffer usage across multiple kernels in a single-GPU setting. Buffer B is used to communicate results from kernel 1 to kernel 2, it therefore acts as a communication channel.



(b) Buffer usage across multiple kernels split over multiple GPUs. Notice the similarity to the single-GPU setting except that each partition of a kernel read or writes only half of each buffer. Buffer B is now split into two communication channels without cross communication.

Figure 3.11: Communication between kernels in unpartitioned and partitioned kernels.

"communication" and is used to detail the transformation steps that are required to reveal internal communication in GPU applications.

Then, the partitioning model that is used to virtually split the kernels of GPU applications is explained. The model contains a placeholder that allows limited customization of how thread blocks are divided into partitions. Several functions to fill in the placeholder are presented and their properties regarding the distribution of thread blocks are discussed.

3.2.1 Communication Model

The main goal of the communication model is to formalize communication in the context of memory accesses of single-GPU applications. The first step for this is to identify the communication partners. While individual threads are entity that performs

any actual work on a GPU, they have limited autonomy and operate in lock-step with other threads in their thread block. For this reason, GPU code is typically in a way that minimizes control-flow divergence within thread blocks, making thread blocks the base unit that executes a given task. Therefore, the end points of communication, as understood by this analysis, are thread-blocks as a whole.

Next is the mechanism of communication used by the thread blocks. Similar to regular programs that are written to run on Operating Systems with a memory management system, memory on GPUs is organized in units of buffers, which are allocated before being used. In contrast to regular programs, however, these buffers are typically not allocated by the program running on the GPU, but by the supporting host program. These buffers are then used as communication channels to provide the results of a kernel launch to either subsequent kernel launches or to the host. Figure 3.11a depicts the communication of two kernel launches. Buffer A is a read-only buffer (regarding kernels) and buffer C is a write-only buffer, only buffer B is both written to and then read from. In this case, all of buffer B acts as a single communication channel between kernel 1 and 2.

If the thread grid of such a kernel were to be split and executed in the same shared memory system, the resulting communication would have a finer granularity, as shown in figure 3.11b. All kernels are split into two partitions across the Y-axis and in this hypothetical case, the memory access patterns of each kernel's partitions match this split (i.e. a partition only accesses half of each buffer, also split across the Y-axis). This splits the buffer into two communication channels, one between the kernel 1.1 and 2.1 and another channel between kernel 1.2 and 2.2. Kernels 1.1 and 2.2 do not communicate, likewise for kernels 1.2 and 2.1. While this type of memory access was chosen for the example, there is no guarantee that splitting a kernel results in such a clear split of the communication channels (what happens in real world applications is precisely the topic to be investigated by the analysis).

Definition 2. *A **kernel launch** is largest scheduling unit in CUDA and corresponds to a super step in the BSP programming model [1]. Kernel launches are scheduled as part of a CUDA stream and exhibit a total ordered within their respective stream. The statement $K_1 < K_2$ holds if K_1 and K_2 are part of the same stream and K_1 is scheduled before K_2. There is no defined order between kernel launches of different streams.*

Definition 3. *A **thread block** is a user-defined group of CUDA threads, executing concurrently on a GPU, with limited means of communication among themselves. It is CUDA's atomic scheduling unit and part of a specific kernel launch's thread grid. A thread block's index refers to its position within the 3-dimensional thread grid of*

a particular kernel launch. Thread blocks with identical index but executed as part of different kernel launches are considered different from one another. Thread blocks within the same kernel are ordered lexicographically by their indices. There is no order between thread blocks from different kernel launches.

Using these two concepts, a formal model for the internal communication of GPU applications can be designed that is in-line with the major design choices in GPU programming models. The model defines communication on the atomic unit of execution in CUDA, thread blocks, and interprets the term as described in definition 3. This definition differs from the one used by CUDA in that it answers the question of the identity of thread blocks over multiple kernels: thread blocks from different kernel launches are always considered different from one another, even if the executed kernel code and all kernel parameters are identical [14]. While for these cases it seems easily possible to assign an identity to a thread block that is persistent over multiple kernel launches, it can not be generalized to kernels executing different code or with different thread grid configurations. Furthermore, communication between thread blocks can only if they are part of different kernel launches and is directed from launches and is directed from earlier launches to later ones, using kernel launches from definition 2. This is in-line with the CUDA consistency model lacking guarantees about the visibility of write accesses to main memory.

Given

$$b_s \in K_s \text{ and } b_d \in K_d, \text{ two thread blocks of different kernels,} \tag{3.1}$$

$$W(b_s), \text{ the write set of thread block } b_s, \tag{3.2}$$

$$R(b_d), \text{ the read set of thread block } b_d, \tag{3.3}$$

$$M \coloneqq W(b_s) \cap R(b_d), \text{ the set of addresses both written by } b_s \text{ and read by } b_d, \tag{3.4}$$

the thread block b_s is sending a message to b_d using the memory locations in M if all of the following statements hold:

$$K_1 < K_2 \tag{3.5}$$

$$M \neq \varnothing \tag{3.6}$$

$$\forall b \in K_i (M \cap W(b) = \varnothing), s < i < d \tag{3.7}$$

$$\forall b \in K_s (b \leq b_s \vee M \cap W(b) = \varnothing) \tag{3.8}$$

In other words, the kernel of b_s must be different from and precede that of b_d (3.5), they need at least one shared memory location between the write set of b_s and the read

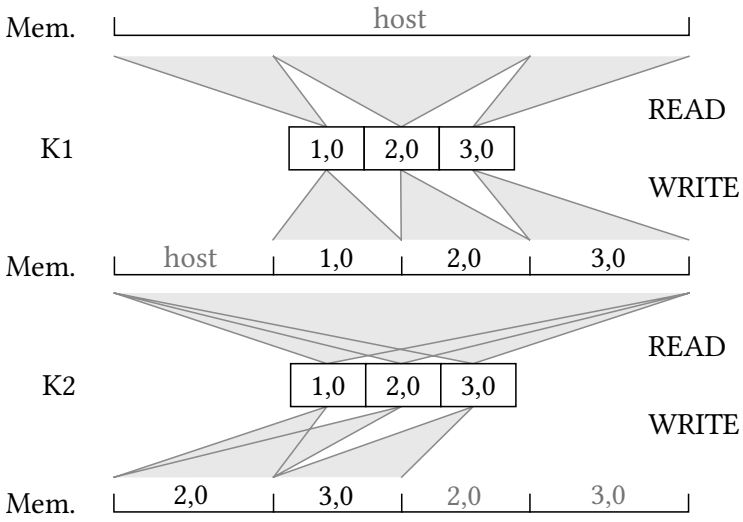


Figure 3.12: Example of the evolution of the tracker state (denoted as "Mem.") over an application trace containing two kernel launches with three thread blocks each.

set of b_d, and b_s is the only writer to these locations in all kernel launches between K_s and K_d (3.7). Within kernel launch K_s, other thread blocks either do not write to the shared memory location or have a higher index than b_s (3.8). Another interpretation of the statement 3.8 is that the highest thread block index wins if multiple thread blocks of the same kernel launch perform competing writes to the same location. As a result of the lack of coupling between CUDA streams, each stream is regarded as independent from different streams and no communication is possible between them.

This formal definition is used as the basis for an efficient algorithm that computes all communication within a GPU kernel in a single pass over all kernels. This is done by using a map that keeps track of all write accesses throughout the kernels of an application and can be used to lookup the last writer of each address. Because of the size of the GPU address space and the fact that memory accesses very rarely interleave on individual bytes, an interval tree can be used to compress the data [78]. The interval tree used for this algorithm has two extensions modified: a) intervals are tagged with a value, and b) intervals can not overlap. If an interval is inserted that overlaps with an existing interval, two cases are considered: if the existing interval has a different tag, it is split, shrunk, or removed to create room for the new interval and if it has the same tag as the new interval, they are merged into a single interval. In this work, such a segment is called a "memory tracker" when used for this kind of analysis.

The algorithm 1 illustrates how communication is efficiently extracted from an application trace using a memory tracker. Note that this algorithm is designed for the

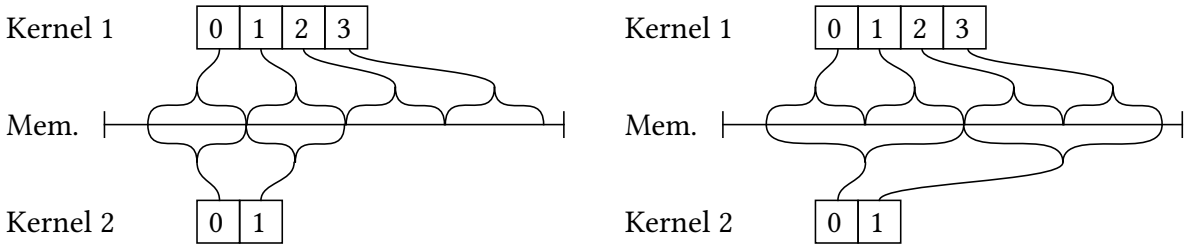
Algorithm 1 Algorithm to extract communication from post-processed traces containing read and write sets for each thread block of a kernel launch.

$T \leftarrow insert\_interval(T, 0, \infty, (HOST, \varnothing)$
for K **in** $kernels$ **do**
  for R **in** $reads(K)$ **do**
    $overlaps, rest \leftarrow intersect\_interval(T, start(R), end(R))$
    for i **in** $overlaps$ **do**
      $from \leftarrow$ start address of i
      $to \leftarrow$ end address of i
      $(SrcKernel, SrcBlock) \leftarrow$ tag of i
      $DstBlock \leftarrow$ thread block issuing R
      emit as communication $(from, to, (SrcKernel, SrcBlock), (K, DstBlock))$
    end for
  end for
  for W **in** $writes(K)$ **do**
    $from \leftarrow$ start address of W
    $to \leftarrow$ end address of W
    $tag \leftarrow (K,$ thread block issuing $W)$
    $T \leftarrow insert\_interval(T, from, to, tag)$
  end for
end for

read and write sets produced by the post-processing step of the trace collection. The first step is the initialization of a new interval tree (called a "tracker") such that all memory locations are tagged with the special value $(HOST, \varnothing)$ that precedes all kernel launches. What follows is an iterative process that is repeated for each kernel in the order they are launched: All read sets from the kernel trace are tested for intersections with existing intervals in the tracker. Each intersection corresponds to communication, either from a previous kernel launch or from the host and is emitted as such. After all read sets have been processed, the write sets are used to update the tracker. Each interval in the tracker is tagged with a pair containing the thread block index of the access and the thread blocks kernel launch index. Since the tracker does not permit overlapping intervals, inserting an interval that would overlap an existing one results in one of the following scenarios:

1. The insertion is ignored if: the new kernel launch index is lower than the old one, or the kernel launch indices are equal and the new thread block id is lower than the old one.

2. Both intervals are merged if: both the kernel launch indices and the thread block indices are equal.



(a) Kernel 1 and 2 have identical working sets sizes per thread block.

(b) Kernel 1 and 2 have identical working sets sizes per kernel.

Figure 3.13: Two common combinations of kernel types of arbitrarily many ones. No general rule exists that reliably matches thread blocks that communicate with each other using only their thread indices.

3. The old interval is shrunk, split or removed in all other cases to make room for the new interval.

Figure 3.12 illustrate the evolution of the tracker state throughout an application trace. Memory is initially tagged and then updated by write accesses from kernel launches. Note the conflict resolution after K2, where the two thread blocks $(1, 0)$ and $(2, 0)$ both write to the same address and the higher thread block index $(2, 0)$ wins.

3.2.2 Partitioning Model

Based on the communication model described above, the communication between thread blocks of different kernel launches can be analyzed. Computing the communication on the granularity of individual thread blocks results in large amounts of data that are difficult to interpret. This is caused by the lack of identify of thread blocks, which can easily be illustrated using two kernels with differently sized thread grids as shown in figures 3.13. Although the communication can easily be computed, useful and stable metrics are difficult to defined when thread blocks pop in and out of existence with each kernel launch. The partitioning model aims to solve this problem by introducing partitions as a new concept, which do have an identity that persists over multiple kernel launches. Using partitions then allows to compute statistics for individual partitions or groups of partitions that have meaningful semantics over multiple kernel launches.

This introduces an interesting problem: the analysis aims to approximate a realistic multi-GPU execution of single-GPU applications, so the assignment of thread blocks into partitions should approximate that of a realistic multi-GPU application reasonably well. Authors of multi-GPU applications can exploit their knowledge of the algorithm that is implemented to find an efficient partitioning that maximizes performance.

However, this knowledge about the algorithm is unavailable for the analysis performed with this model. The only input to the partitioning is the thread grid configuration of each kernel launch.

Providing a good partition assignment for thread blocks using only this information provides challenges. Figure 3.13 illustrates the primary issue using two common cases of memory access patterns in CUDA kernels. The first case, shown on the left, has two kernel launches of different sizes but with memory accesses of the same size for each thread block. A explanation is that both launches execute the same kernel code, just on different grid sizes. In this case, a partition assignment that results in a high spatial locality between kernels should have a fixed mapping of indices to partitions that does not depend on the overall kernel size, for example, a round-robin approach. In the second case, shown on the right, the amount of data accessed per thread block varies between kernels, but that of the full thread grid is the same size. This can be the result of kernel launches executing different code, or executing the same code, but with different thread block sizes. Here, a partition assignment with high spatial locality should be based on the thread blocks index relative to the total thread grid size. Both these cases are very common and there is no obvious solution.

The problem is side-stepped by only defining an interface for the partition assignment and then performing the analysis with a range of different partitioning schemes. This interface is defined as a mapping

$$M_{\vec{n}} : [0, n_1 - 1] \times [0, n_2 - 1] \times [0, n_3 - 1] \to [0, 1) \in \mathbb{R}, \vec{n} \in \mathbb{N}^3 \tag{3.9}$$

that maps a three-dimensional thread ID to the half-open real interval from (inclusive) zero to (exclusive) one. The 3-dimensional vector $\vec{n}$ describes the total thread grid size in x, y, and z. If this function is applied to all thread blocks in a particular kernel launch, the resulting set of thread blocks within the $[0, 1)$ range is called the called its thread block range.

This approach allows splitting assigning a thread block to a partition based on its position in the 1-dimensional thread block range using the function

$$\begin{aligned} P : [0, 1) &\to \{1, 2, \ldots, p - 1\}, p \in \mathbb{N} \\ u &\mapsto \lfloor u \cdot p \rfloor \end{aligned} \tag{3.10}$$

This function distributes thread blocks into one of p partitions by cutting the range into equally sized segments and assigning all thread blocks in one segment to the same partition. Although the two functions could be combined into one, splitting them up

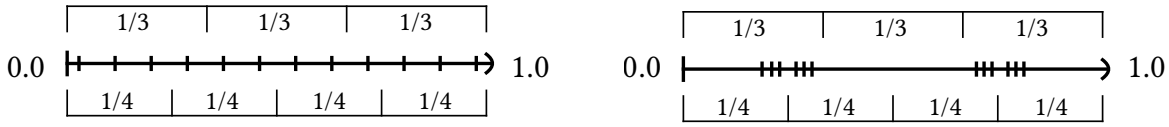


Figure 3.14: Two examples of thread block ranges. The left example has perfect uniformity while the example on the right is heavily clustered, resulting in unevenly sized partitions.

has benefits. First, function 3.10 does not change for any mapping, so it can be kept separate. Second, the distribution of thread blocks on the range $[0, 1)$ can be used to judge the quality of a mapping.

Cutting the range into equally sized segments for the partition assignment is susceptible can result in unequally sized partitions of the partition mapping produces an uneven distribution in the $[0, 1)$ range. Two examples are the existence of multiple groups of points in proximity to each other within a group (clustering) and not utilizing the full $[0, 1)$ range (bounding). Both of these irregularities in the result of the mapping function can result in unequally sized partitions and lead to inaccurate results for the communication analysis. Figure 3.14 illustrates the importance of this property. Both examples contain a total of 12 thread blocks mapped to a thread block range. The left example with perfect uniformity has partitions of sizes $(4, 4, 4)$ for $p = 3$ and partitions of sizes $(3, 3, 3, 3)$ for $p = 4$. For numbers of partitions that are factors of the number of thread blocks, this distribution always produces perfectly evenly sized partitions and only produces errors that are off by one for other values of p. The right example, on the other hand, is highly clustered with two large clusters the $1/4$ and $3/4$. For $p = 4$ the partitions are again evenly sized with $(3, 3, 3, 3)$ but for $p = 3$, the clustering produces heavily imbalanced partitions with the sizes $(6, 0, 6)$.

A reliable way to identify both irregularities in a mapping function is by comparing it to an ideal linear mapping that produces perfectly equally sized partitions. Given a kernel K_n containing n thread blocks, this approach starts the sequence r of the outputs of M_K for all thread block indices in K_n. Then, a permutation k is required where the elements are in their natural order, possibly destroying the order created by M_K:

$$k = (M_K(id_1), M_K(id_2), ..., M_K(id_n)), n = |K_n| \tag{3.11}$$

$$k_s = (\pi(k_1), \pi(k_2), ..., \pi(k_n)), \tag{3.12}$$
$$\text{such that } \pi(k_1) < \pi(k_2) < ... < \pi(k_n)$$

As an example, let $n = 3$, $K_n = ((2,0,0),(1,0,0),(0,0,0))$ and $M_K(id) = id_0/3$. This results in $k = (2/3, 1/3, 0)$ and its permutation in natural order is $k_s = (0, 1/3, 2/3)$. This approach explicitly allows M_K to map multiple thread blocks to the same real number. The ideal linear mapping is represented by a sequence computed as

$$l_n = \left(\frac{0}{n}, \frac{1}{n}, ..., \frac{n-1}{n}\right). \tag{3.13}$$

Finally, the sorted sequence k_s is compared to l_n by computing the average (of the magnitudes) of their differences in each element:

$$\varepsilon_M = \frac{1}{n}\sum_{i=1}^{n} |k_{si} - l_i|.$$

Taking up the example above, $l_n = (0, 1/3, 2/3)$ and is identical to k_s, resulting in an average difference of $\varepsilon_M = 0$, indicating a perfectly uniform matching. Since this metric represents the error between an ideal distribution and a real one, it should be interpreted as **lower is better**. Using an ideal linear mapping as reference exposes both clustering as well as bounding.

This can be illustrated with examples for each of the extremes. The first example is a mapping with extreme clustering, that maps all thread blocks to a single real number:

$$\begin{aligned}
&n = 3, K_n = ((0,0,0),(1,0,0),(2,0,0)) \\
&M_K(id) = \frac{1}{2} \\
&k = k_s = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right) \\
&\varepsilon_M = \frac{1}{3}\left(\left|\frac{1}{2} - 0\right| + \left|\frac{1}{2} - \frac{1}{3}\right| + \left|\frac{1}{2} - \frac{2}{3}\right|\right) \approx 0.28
\end{aligned}$$

The average difference is quite high at $\varepsilon_M \approx 0.28$, correctly indicating a very bad thread block mapping. The second example is one of very strong bounding, where all

values fall into exactly one half of the available range:

$$
\begin{aligned}
&n = 3, K_n = ((0,0,0),(1,0,0),(2,0,0)) \\
&M_K(id) = id_0/6 \\
&k = k_s = \left(0, \frac{1}{6}, \frac{2}{6}\right) \\
&\varepsilon_M = \frac{1}{3}\left(|0-0| + \left|\frac{1}{6} - \frac{1}{3}\right| + \left|\frac{2}{6} - \frac{2}{3}\right|\right) \approx 0.17
\end{aligned}
$$

This example results in a fairly bad average error of $\varepsilon_M \approx 1.7$. While this is still indicating a fairly bad thread block mapping, it is better than the previous one, matching the intuitive result when comparing both. The highest possible average error would be achieved by mapping all thread blocks to one end of the scale and tends towards $\varepsilon_M = 1/2$ for large values of n.

Notice that this metrics treats the mapping as a black box and judges the uniformity for a single combination of a mapping and a thread grid. As a result, a single result typically cannot accurately describe the overall uniformity of a mapping, especially in the presence of potentially skewed thread grids with vastly different dimension sizes. In order to provide a fair comparison, all selected mappings are judged using thread blocks with 2048 thread blocks in total and the following ratios of X and Y: $\frac{1}{256}, \frac{1}{128}, \frac{1}{64}, \frac{1}{32}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{4}{1}, \frac{8}{1}, \frac{16}{1}, \frac{32}{1}, \frac{64}{1}, \frac{128}{1}, \frac{256}{1}$. The Z-dimension is always of size 1, since all relevant effects are already visible in 2 dimensions. The different results for the uniformity of a mapping are then summarily presented by their quintiles.

The other important metric is the preservation of locality. The locality is preserved when thread blocks that are close to each other in the thread grid are mapped to real numbers that are also close to each other. Intuitively, a mapping that scores high on this metric should lead to more intra-partition communication and less inter-partition communication, lessening the demands on the interconnect between devices. Whether that is the case is one of the research questions of this analysis.

Two approaches are taken to judge the preservation of the locality of a given mapping. The first one is a qualitative analysis of the general behavior of the mapping algorithm. Although this approach does not provide a quantitative measure, it provides a valuable intuition about the expected communication.

The second approach is a stencil that assumes communication between adjacent thread blocks and computes the mean value of the worst-case distances for all thread blocks. As such, the way to interpret this metric is that **lower is better**. Let $\vec{n} \in \mathbb{Z}^3$

be the thread grid size, $M\colon \mathbb{Z}_0^3 \to \mathbb{R}$ the mapping in question, then

$$\begin{aligned}
\hat{d}&\colon (\mathbb{Z}^3, \mathbb{Z}^3) \to \mathbb{R} \\
\hat{d}(\vec{x}, \vec{s}) &= \begin{cases} |M(\vec{x}) - M(\vec{x} + \vec{s})|, & \text{if } \vec{0} \leq x \dot{+} s < \vec{n} \\ 0, & \text{otherwise} \end{cases} \qquad (3.14) \\
\hat{d}_{max}&\colon \mathbb{Z}^3 \to \mathbb{R} \\
\hat{d}_{max}(\vec{x}) =& max(\hat{d}(\vec{x}, (-1,0,0)), \hat{d}(\vec{x}, (1,0,0)), \qquad (3.15) \\
& \quad \hat{d}(\vec{x}, (0,-1,0)), \hat{d}(\vec{x}, (0,1,0)), \\
& \quad \hat{d}(\vec{x}, (0,0,-1)), \hat{d}(\vec{x}, (0,0,1))) \\
d_M&\colon \mathbb{Z}^3 \to \mathbb{R} \\
d_M(\vec{n}) =& \frac{1}{n_1 \cdot n_2 \cdot n_3} \sum_{i=0}^{n_1} \sum_{k=0}^{n_2} \sum_{l=0}^{n_3} \hat{d}_{max}(i,k,l) \qquad (3.16)
\end{aligned}$$

The function $\hat{d}$ returns the distance between the mapped values of $\vec{x}$ and $\vec{x}$ shifted by $\vec{s}$, provided that the shifted vector is within the bounds of the thread grid, otherwise it returns the distance 0. $\hat{d}_{max}$ is the stencil operator, determining the maximum distance between the vector $\vec{x}$ and its six adjacent neighbors. The *max* operator assigns high weight to communication over longer distances in the resulting mean value. This produces a pessimistic estimate of the possible communication, which intuitively mirrors real-world scenarios where the total runtime of a bulk synchronous operation is the maximum runtime of each of its threads. Finally, the average communication distance d_M is defined as the mean value of the stencil-operator being applied to the full thread grid. Similar to the linear error metric for uniformity, this metric depends on a specific thread grid and might produce inconclusive results for a single evaluation. To combat this effect, it is evaluated with the same number of thread blocks and ratios as the linear error metric.

Intuitively, preserving locality should lead to less communication between partitions and is, therefore, assumed to be a positive trait of a mapping. Instead of an equation that defines locality preservation, this property is discussed qualitatively for each of the selected mappings.

3.2.3 Selected Thread Block Mappings

This subsection presents and discusses four mappings for the placeholder of the partitioning model defined 3.9. Each mapping is defined and discussed qualitatively,

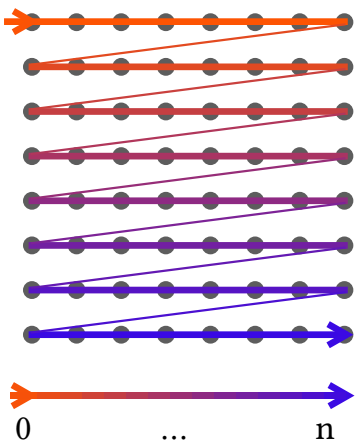


Figure 3.15: Visualization of the lexicographic mapping on an 8x8 grid. Thread blocks are lexicographically ordered by comparing their position in in dimension individually, starting in X, then Y and lastly Z.

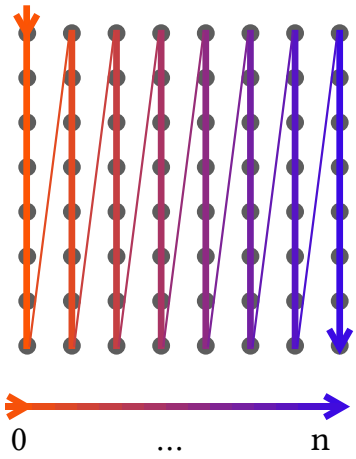


Figure 3.16: The colexicographic mapping, the counterpart to the lexicographic one. All characteristics are the reverse of its sister mapping.

followed by the quantitative analysis on the uniformity and locality preservation as defined in the previous chapter. Since both the average linear error ε_M and the average communication distance d_M are calculated for a variety of different thread grid configurations, their results are reported as the five-number summary with as (min, first quartile, median, third quartile, max). If all the values are identical for all thread grid configurations, only the singular result value is reported.

Lexicographic

The first mapping is based on the **lexicographic** order of the thread blocks. Here, thread blocks are ordered by comparing each dimension individually, starting with the X-, then Y-, and lastly Z-dimensions. This order is identical to the row-major order used in most C-programs to store multi-dimensional arrays in linear memory and shares a virtually identical definition. Given a thread block index $veci \in N_0^3$ in a thread grid of size $\vec{n} \in N^3$, the lexicographic mapping is defined as

$$M_{\vec{n}}(\vec{i}) = \frac{i_1 + i_2 \cdot n_1 + i_3 \cdot n_1 \cdot n_2}{n_1 \cdot n_2 \cdot n_3}. \tag{3.17}$$

This mapping produces perfectly uniform thread block ranges for thread grids of arbitrary sizes. The average difference to the ideal linear mapping on the thread grid

sizes specified in 3.2.2 is zero in all cases. The linear error for this mapping is $\varepsilon_M = 0$ and the number of thread blocks in a partition differs at most by one for arbitrary numbers of partitions and thread grid shapes.

Locality in this mapping is not preserved uniformly for each dimension. The distance between thread blocks and its influence on distances in the thread block range can be considered individually and their ratios can be taken directly from the mapping's definition. These ratios are $X : Y : Z = 1 : n_1 : n_1 \cdot n_2$, meaning that distances in X have the smallest impact (locality is preserved best), distances in Y produce a distance that is n_1 times larger, and distances in Z produce distances that are larger by a factor of $n_1 \cdot n_2$ (locality is preserved worst). Calculating the communication distance estimate d_M for this mapping results in the following five-number summary of distances: $(0.0014, 0.0078, 0.031, 0.12, 0.5)$.

Colexicographic

The **colexicographic** mapping is closely related to the lexicographic one and can be seen as its inverse mapping. It is defined as

$$M_{\vec{n}}(\vec{i}) = \frac{i_1 \cdot n_2 \cdot n_3 + i_2 \cdot n_3 + i_3}{n_1 \cdot n_2 \cdot n_3}$$

The general properties are identical to that of the lexicographic mapping, however, locality is preserved in the opposite order of dimensions. It has perfect uniformity as well dimension-dependent locality preservation, with the ratios $X : Y : Z = n_2 \cdot n_3 : n_3 : 1$.

Z-Order curve

While the first two mappings favor one dimension over the other regarding locality, the **Z-order curve** is a mapping from N-dimensional points to a one-dimensional range that roughly preserves locality across all dimensions. It is used in many areas in computer science, examples are storing multi-dimensional GPU textures in Z-order to improve cache hit rates in [79] and efficient indices for multi-dimensional data for use in databases and similar [80]. It belongs to the class of space-filling curves and is popular due to its simple calculation. Calculating the one-dimensional index of an N-dimensional one in Z-order simply requires interleaving the bits of each dimension in lexicographic order. For a three-dimensional point in $\mathbb{B}^b \times \mathbb{B}^b \times \mathbb{B}^b$, $\mathbb{B} := \{0, 1\}$, described by three binary numbers of length b, the Z-order index can be calculated by

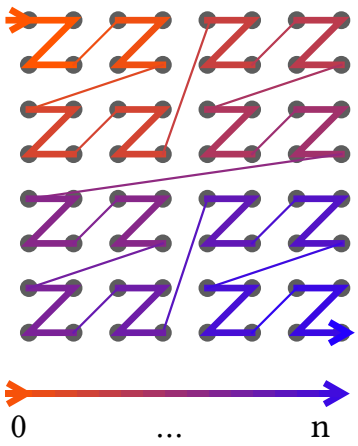


Figure 3.17: A Z-order curve over a thread grid of size $8 \times 8 \times 1$. Notice the recursive partitioning.

interleaving bits of its three components in lexicographic order. This mapping

$$\begin{aligned} Z : \mathbb{B}^b \times \mathbb{B}^b \times \mathbb{B}^b &\to \mathbb{B}^{3b} \\ (\vec{x}, \vec{y}, \vec{z}) &\mapsto z_b, y_b, x_b \ldots z_2, y_2, x_2, z_1, y_1, x_1 \end{aligned}$$

describes a walk through $\mathbb{B}^b \times \mathbb{B}^b \times \mathbb{B}^b$ in the shape of multiple Zs, giving the curve its name. The Z-order curve is defined for an arbitrary number of dimensions and always calculated by interleaving the bits of its components in lexicographic order. Figure 3.17 illustrates the Z-order curve in two dimensions, displaying the distinctive Z-pattern.

Dividing the Z-order index of a point by the maximum number of points in the cube (2^{3b}) results in a value in the $[0, 1)$ range. Due to the recursive nature of this approach, it only works for thread grids that are cubes with a side length that is a power of two (Z-cube). For any other thread grid shape, no perfectly fitting cuboid with points in Z-order can be created directly from this definition. As an example, a thread grid of the dimensions $8 \times 4 \times 4$ can only be ordered assigned an order by mapping its points into that of a 8^3 cube, for which a Z-order is defined. Using an identity function to map the thread grid to the cube would result in significant bounding (and clustering to some degree), as the whole last quadrant would not be populated by the thread grid. The more skewed the dimensions of a thread grid are, the more pronounced this effect. These issues are partially solved by scaling the thread grid up to the dimensions of the cube, in the case of this example by multiplying every point with the factor $(1, 2, 2)$. Degenerate dimensions (dimensions of size 1) still produce bounding since all points have the same value in this dimension. They are handled by falling back on

a 2-dimensional Z-order curve (one degenerate dimension) or a linear mapping (two degenerate dimensions). The full formula for the mapping M then results in

$$M_{\vec{n}}(\vec{i}) = \frac{Z(\vec{j})}{2^{3b}},$$
$$\vec{j} = (\lfloor i_1 \frac{2^b}{n_1} \rfloor, \lfloor i_2 \frac{2^b}{n_2} \rfloor, \lfloor i_3 \frac{2^b}{n_3} \rfloor)$$

which maps thread block IDs of any given thread grid to the range $[0, 1)$. A three-dimensional ID of non-negative integers $\vec{j}$ is interpreted as a three-tuple of binary numbers when applied to Z and the resulting Z-order index is interpreted as a non-negative integer for the division. Overall, uniformity is preserved fairly well, with calculated average linear error ε_M having the five-number summary $(0, 0.0024, 0.01, 0.034, 0.083)$.

Qualitatively speaking, this mapping preserves locality roughly equally well in all dimensions. The compromise between an easily computable solution and the reasonable locality between consecutive points in the ordering is the primary reason it is used in computer science [80]. However, the recursive definition creates comparatively large jumps at the higher levels, the largest ones being $\frac{1}{2}$ in each dimension highest level as illustrated in figure 3.17. The average communication distance d_M has the same maximum as that of the lexicographic and colexicographic values but worse results overall, with a five-number summary of $(0.036, 0.046, 0.1, 0.2, 0.5)$.

An important thing to note about all three of these mappings is that they are identical for one-dimensional thread grids, where they collapse into the linear mapping:

$$(i, n) \mapsto i/n, i \in [0, n-1], n \in \mathbb{N},$$

Such a linear mapping preserves locality very well and provides a uniform distribution of points in the image. Using these mapping functions, the resulting simulated communication between these partitions can be analyzed.

Hashed

The fourth mapping is a deliberately chaotic one that does not optimize locality in any or all dimensions. The **hashed** mapping acts as a deterministic, quasi-random mapping. The use of a hash function instead of a true or pseudo-random assignment has primarily two benefits:

1. the mapping produces repeatable results that eliminate transient effects on the communication analysis, and

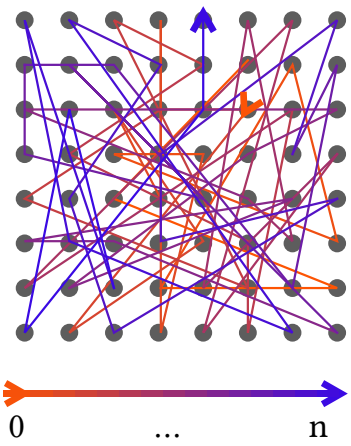


Figure 3.18: The thread block order of the hash-based mapping in a thread grid of size $8 \times 8 \times 1$.

Algorithm 2 The algorithm for the hash-bashed mapping using SHA-256.

procedure $M_{hash}(x, y, z)$
  $packed \leftarrow uint32(x).uint32(x).uint32(x)$ ▷ a 12 byte buffer
  $hash \leftarrow SHA256(packed)$
  $front \leftarrow uint64(hash[0{:}8])$
  return $float(front/2.0^{64})$ ▷ scale down to range $[0, 1)$
end procedure

2. the hash function is, in contrast to random number generators, state-less and therefore a true function of only its inputs.

This particular implementation is based on the SHA-256 hashing algorithm from the SHA-2 family of cryptographic hashing functions [81]. Algorithm 2 illustrates the approach. The assignment of a thread block index is computed by first packing the X, Y, and Z coordinates of a thread block index into a 12 byte buffer in that order as unsigned 4 byte integers each. The choice of 4 bytes per integer is based on the maximum number of thread blocks in any dimension of the thread grid and can be extended to 8 or more bytes should this value change in the future [14, section H.1. Features and Technical Specifications]. Then the SHA-256 hash with a size of 32 bytes is computed, the first 8 bytes are interpreted as an unsigned integer and divided as by 2^{64} to scale the value down into the range $[0, 1)$. In contrast to the previous three mappings, this one does not depend on the thread grid size.

Considering the quasi-random behavior of the hashed mapping, systematic clustering or bounding effects are unlikely to occur. The likelihood of a thread index to be mapped into a specific sub-range of $[0, 1)$ is directly proportional to the size of that sub-range

and independent of its position. The five-number summary of the linear error ε_M is $(0.0048, 0.0057, 0.0074, 0.0093, 0.012)$.

The quasi-random behavior also suggests a fairly high distance for the virtual stencil. The average communication distance d_M for the thread grid sizes specified before results in the following five-number summary: $(0.51, 0.55, 0.56, 0.56, 0.57)$. The minimum value is already larger than the worst case for any of the other mappings, suggesting that it would map thread blocks highly unfavorable in virtually all settings and confirming the intuition that a random or quasi-random mapping should not produce an efficient thread block distribution.

The list of mappings presented in this section is by no means exhaustive. There is a theoretically unlimited number of different mappings and more research into this area could provide helpful insights into automatic distribution schemes for multi-GPU applications.

3.3 Workload Selection

The traces used for the analysis are collected from popular benchmark suites as well as custom applications that were used to develop and debug the tracing software and this section provides an overview of each. The selected benchmark suites are Rodinia [82], SHOC [83], and Parboil [84]. Although any given benchmark or even benchmark suite can not realistically cover all types of applications in the real world, the benchmark suites selected in this work cover a wide range of common computational kernels found in GPGPU applications [85].

The first benchmark in the list, Rodinia, was released in 2009, only two years after the initial release of CUDA, and is one of the first benchmark suites targeting scientific workloads for GPUs [82]. Previous benchmarks were primarily geared towards by the gaming industry and focused on graphics performance as well as the interplay with the system's CPU, ignoring GPGPU capabilities [86]. The workloads it contains are based on scientific computational kernels, identified by the Berkeley Dwarfs [87].

The Scalable Heterogeneous Computing Benchmark (SHOC) from 2010 is designed to target heterogeneous systems in different configurations, namely multiple nodes as well as multiple accelerators per node [83]. It can be considered a refinement of Rodinia as it aims to provide a wider variety of application types and more differentiated measurements. It offers performance measurements of three different levels, each providing results for different kinds of characteristics. The first level contains benchmarks providing raw characteristics from the hardware under ideal conditions,

such as memory bandwidth and peak FLOP/s. Level two measures the performance of basic parallel algorithms and is the most direct counterpart to other benchmark suits. Lastly, level three contains complex applications with different computation kernels and complex system-accelerator-interactions designed as stress tests to identify flaws in the hardware itself, the system setup, and cooling.

The third benchmark, Parboil, was released as another improvement on Rodinia in 2012 and aims to provide benchmarks that can easily be adapted to the quickly changing landscape of accelerators [84]. The name stems from the process of pre-cooking rice to create a product that is already partially cooked, but requires to be finished by the consumer, known as parboiling. The name was chosen because each benchmark in the suite is provided in multiple stages of "done-ness". It generally provides three levels of implementations: fully optimized CUDA and OpenCL implementations, less hand-optimized versions using OpenMP, and lastly naive CPU implementations. When new hardware is released, the benchmarks can be quickly adapting by choosing the most appropriate starting point and then porting to and optimizing for the new hardware. In the context of this analysis, only the CUDA implementations are relevant.

Finally, the custom benchmarks are included as a baseline. They only contain the most basic optimizations and are written in a way that is friendly towards static analysis and transformations. This means they are implemented in a single translation unit with little program logic and no dynamic behavior changes (e.g. swapping kernels based on program arguments). The custom benchmarks are comprised of the following applications:

1. A chained matrix multiplication $\boldsymbol{D} = (\boldsymbol{A} \cdot \boldsymbol{B}) \cdot \boldsymbol{C}$, executed in two kernels. The first kernel computes $\boldsymbol{AB}$, the second reused this result directly for the second multiplication with $\boldsymbol{C}$, resulting in $\boldsymbol{D}$.

2. A fast-fourier-transformation implementing the Cooley-Tukey algorithm [88]. It is implemented by first reordering all elements and then launching one kernel for each level of the recursive DFTs.

3. A 2D heat relaxation implemented using a 5-point jacobi stencil on a static grid. Each iteration of the stencil is executed in an individual kernel launch.

4. A discrete gravity-based n-body simulation using euler integration and constant time steps. Bodies are modeled as point masses without collisions and static weights. No clustering optimizations are applied.

5. An in-place vector sum using two phases in a polymorphic kernel. The first phase reduces the vector size by a factor of two by computing partial sums until a threshold for the vector size is reached. Each reduction step is implemented as a kernel launch. The second phase then reduces the partial result into a final sum in another kernel launch.

Unfortunately, not every benchmark from the three official benchmark suites could be included in the analysis. There are several reasons for exclusion, ranging from implementation details that prohibit trace generation to conceptual issues that render any analysis of the traces useless. Each problem encountered as part of the analysis was sorted into an error category based on the underlying root cause. The following types of root causes have been identified in the analysis:

- **Feature**. The benchmark uses CUDA features that are not supported by gpucc. This means primarily the use of texture memory, a common optimization for read-only data.
- **Kernels**. The benchmark contains only a single CUDA kernel launch. For an analysis of the communication between kernels, a benchmark must contain at least two CUDA kernels. Communication through the GPU's global memory is unreliable within a single kernel and therefore illegal in the context of this work. This category also catches applications where kernels do not read from global memory.
- **Libraries**. Many common computational kernels, such as BLAS routines or fast-fourier-transformations are already provided by the CUDA SDK in the form of libraries [89]. These libraries are compiled ahead-of-time without and their source code is inaccessible, prohibiting the source instrumentation used for this analysis.
- **Bugs**. LLVM IR is unaware of any CUDA concepts, requiring the tracing plugin to reverse engineer gpucc to include all functionality. This category catches all bugs that could not be fixed during the development of the instrumentation framework and manifests primarily in crashes during compile time or run time.

The benchmarks that had to be excluded due to one of these reasons, as well as the specific reason, are listed in figure 3.19. An interesting case is the md5hash benchmark of the SHOC benchmark. The traces of this benchmark contain kernels that have written to main memory, but not read anything from it. The kernel input in this

Benchmark	Error	Benchmark	Error
Parboil - bfs	Features	Rodinia - nn	Bugs
Parboil - sad	Features	Rodinia - pathf.	Bugs
Parboil - sgemm	Kernels	Rodinia - streamcl.	Bugs
Parboil - tpacf	Kernels	SHOC - fft	Bugs
Rodinia - cfd	Bugs	SHOC - gemm	Libraries
Rodinia - heartwall	Bugs	SHOC - level 0	Kernels
Rodinia - hotspot	Bugs	SHOC - md	Features
Rodinia - huffman	Bugs	SHOC - md5hash	Kernels
Rodinia - hybridsort	Features	SHOC - neuraln.	Libraries
Rodinia - kmeans	Features	SHOC - spmv	Features
Rodinia - lavamd	Bugs	SHOC - stencil	Bugs
Rodinia - leukocyte	Features	SHOC - triad	Bugs
Rodinia - mummergpu	Features		

Figure 3.19: Excluded benchmarks and reasons for the exclusion.

benchmark is hidden by passing it as a scalar variable instead of putting it in global memory. Its input is a single 128 bit MD5 hash that is passed as four 32 bit integers via kernel parameters. The tracing infrastructure only traces array accesses and therefore does not see any read accesses.

After the exclusions, a total of 27 benchmarks is left, listed in figure 3.20. Each benchmark is assigned a shorthand code that is used to refer to the benchmark in the rest of this analysis. The shorthand is mostly used to provide graphs with easier readable names. Although some of the algorithms used in the benchmarks have significant overlap (e.g. Parboil spmv and Rodinia bfs), they are often implemented in slightly different ways and might exhibit interesting differences.

3.4 Trace Evaluation

This section presents the analysis of the trace files containing memory accesses that have been collected from the benchmarks listed in section 3.3. The analysis is performed by computing a variety of quantitative metrics, which are then used to classify the applications. Each metric is explained in two parts: the first part describes the algorithm used to compute it and the second part provides an interpretation of its implications for the application. It is then computed for all benchmarks and the results are discussed and interpreted right away.

The metrics are divided into two large groups which are presented one after the

Benchmark	Code	Benchmark	Code
Custom - 2mm	C2M	Rodinia - btree	RBT
Custom - fft	CFF	Rodinia - dwt2d	RDW
Custom - hotspot	CHS	Rodinia - gaussian	RGA
Custom - n-body	CNB	Rodinia - hotspot3d	RHS
Custom - reduction	CRE	Rodinia - lud	RLU
Parboil - cutcp	PCC	Rodinia - myocyte	RMY
Parboil - histo	PHI	Rodinia - nw	RNW
Parboil - lbm	PLB	Rodinia - particlef.	RPF
Parboil - mri-grid.	PMG	Rodinia - srad	RSR
Parboil - mri-q	PMQ	SHOC - bfs	SBF
Parboil - spmv	PSP	SHOC - reduction	SRE
Parboil - stencil	PST	SHOC - scan	SSC
Rodinia - backprop	RBA	SHOC - sort	SSO
Rodinia - bfs	RBF		

Figure 3.20: Benchmarks included in the analysis and three letter codes used in figures.

other. First, kernels are analyzed as a whole, without splitting them into partitions, using the communication model from section 3.2.2. Then, the second group focuses on the interactions between different partitions and how the communication is influenced by the partitioning mapping used.

The analysis does not process raw traces due to their large size and corresponding long processing time, but relies on the summaries that are computed as explained section 3.1.3. To summarize, the summary of a kernel contains a list of all threads blocks executed as part of the kernel. Each thread block, in turn, contains two big sets, one with all addresses that have been read by it (called the read set) and one with all addresses that have been written by it (called the write set). Although this way of summarizing memory accesses loses some of the information available in the traces, name in-warp order of memory accesses and the operand size of memory accesses, they are sufficient for this analysis.

3.4.1 Kernel-Level Analysis

Kernel-level analysis does not partition the thread grid and instead treats all thread blocks of a kernel launch as one big partition. The memory accesses of all thread blocks in this large partition are combined into one read and one write set. This simplifies the analysis of applications that contain kernels of greatly varying size and dimensionality.

This simplification does not, however, provide very useful information about the

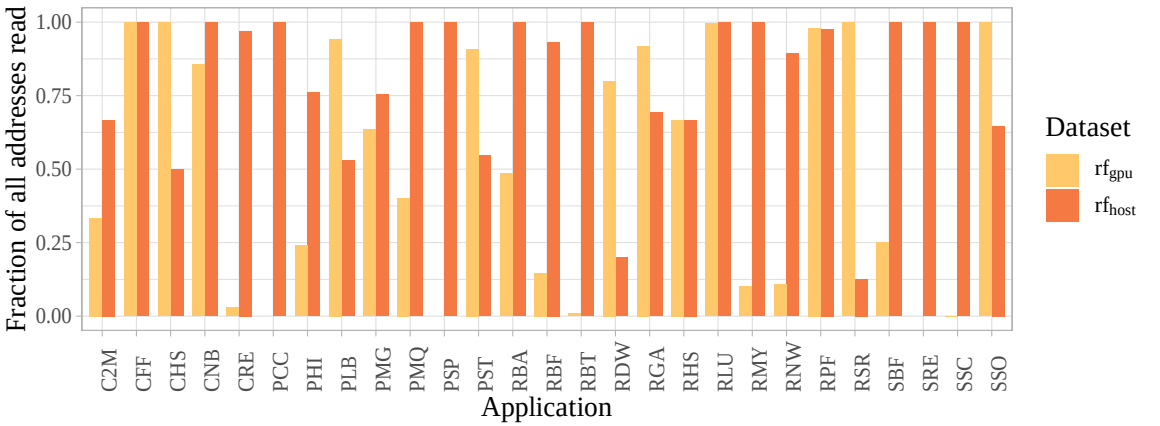


Figure 3.21: Addresses read from the GPU (rf_{gpu}) and addresses read from the Host (rf_{host}), both normalized to the number of all addresses read by the application's kernels. Many addresses fall in both categories.

behavior of any individual kernel launch, which is reduced to just the contiguous region of memory it accesses and their size. To extract useful information, multiple kernel launches throughout the application are analyzed e.g. to identify the amount of data that is communicated between different launches. The different metrics are then normalized so that they can be fairly compared between different applications. This generality of the results if the main benefit of analyzing kernels as one big partition.

$$R_{app} := \bigcup_{i=1}^{i=n} R(k_i) \tag{3.18}$$

$$R^i_{gpu} := R(k_i) \cap \bigcup_{k=1}^{k=i-1} W(k_k) \tag{3.19}$$

$$rf_{gpu} := \frac{\left| \bigcup_{i=1}^{i=n} R^i_{gpu} \right|}{|R_{app}|} \tag{3.20}$$

$$R^i_{host} \coloneqq R(k_i) \setminus \bigcup_{k=1}^{k=i-1} W(k_k) \tag{3.21}$$

$$rf_{host} \coloneqq \frac{\left|\bigcup_{i=1}^{i=n} R^i_{host}\right|}{|R_{app}|} \tag{3.22}$$

The first metric computes the sources of all read accesses. In this context, the "source" of a read access is the source of the last write access to the same address. It can be either a kernel executed on the GPU or the host system and often changes for any given address over multiple kernel launches. Formally, given a kernel K_i, the source of an address in its read set $a \in R(K_i)$ is considered to be a kernel (counted towards R_{gpu} from 3.19), if any of the previous kernels $K_k, k \in \{1, ..., i-1\}$ contains the address a in its write set $W(K_k)$. If the address a is not in the write set of any previous kernels (i.e. has not been written to by a kernel), it is considered to be read from the host (counted towards R_{host} from 3.21). As an example, the value at address a can be read from the host in kernel K_0, followed by a kernel K_1 that updates the value, and finally read from GPU by a third kernel K_2. This analysis is performed for the full read set of each kernel. These read sets of kernels differ wildly in their sizes between different applications. Therefore, the final metrics rf_{gpu} and rf_{host} are the sizes of these sets normalized to the size of all addresses read throughout the application.

Figure 3.21 shows the computed metrics for all applications, note that since an address can belong to both rf_{host} and rf_{gpu}, the bars do not necessarily add up to one. Three major observations are made on this illustration:

1. All but two applications read 50%+ of their data from the host at least once.
2. Of the 27 applications, 13 read 50%+ of their data from the GPU at least once.
3. Three applications (PCC, PSP, and SRE) do not read any data from the GPU and two applications (RBT, SSC) only very small amounts.

This data emphasizes the importance of the host-device interconnection. The large amount of data that is read from the host at least once implies large transfers from host to device for virtually all applications, making these transfers a lucrative target for optimization. One such optimization could come automatically when splitting the application. If a partitioning in e.g. 2 partitions results in each partition only reading half of the original data from the host, the larger aggregated bandwidth is expected to speed up this transfer.

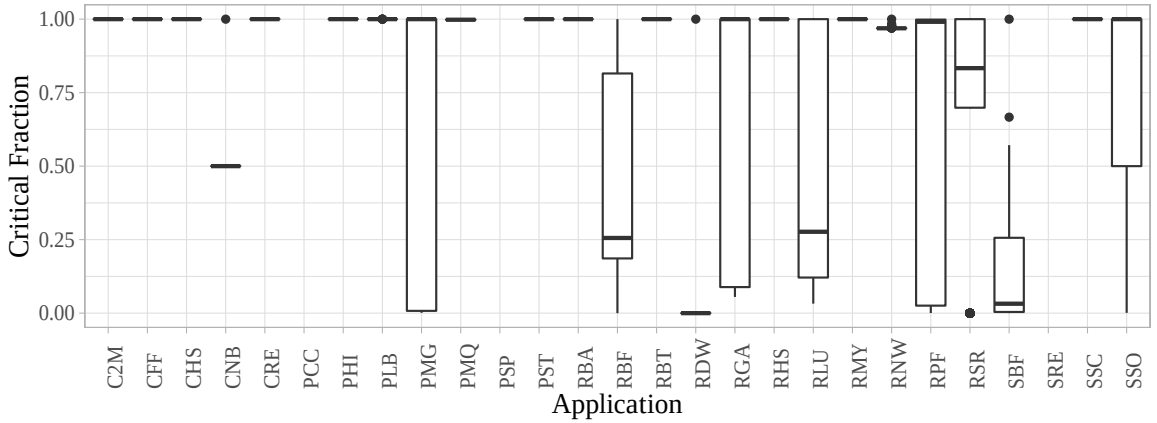


Figure 3.22: The critical GPU data volume R^i_{crit}, normalized to the total amount of data read from the GPU R^i_{gpu}, for all kernels of analyzed applications. A high fraction implies the majority of a kernels input data has been generated in the previous kernel launch.

$$R^i_{crit} \coloneqq R(k_i) \cap W(k_{i-1}) \tag{3.23}$$

$$rf^i_{crit} \coloneqq \frac{|R^i_{crit}|}{|R^i_{gpu}|} \tag{3.24}$$

Another interesting metric is the amount of data that is being communicated on the critical path. An address is considered to be on the critical path for a given kernel k_i if it was written to by the previous kernel k_{i-1}. In other words, data is on the critical part if it is produced by one kernel and immediately required by next kernel. The volume of critical data R^i_{crit} from equation 3.23 is then normalized to the kernels total amount of data read from the GPU R^i_{gpu}, resulting in the critical fraction rf^i_{crit} of the kernel. A large critical fraction implies that large amounts of data need to be quickly communicated between kernels. This is irrelevant for single-GPU applications, since both kernels share the same memory and no transfer has to take place. However, for multi-GPU applications data on the critical path might well be subject to transfers between GPUs. If data stays local to a single partition between kernels (and is thus only accessed by the same GPU), no data needs to be transferred. However, if data is written by one partition and read by another, data transfers between partitions (and therefore devices) are necessary, for example using gather or scatter semantics.

Figure 3.22 illustrates the critical fraction of the kernels in each application, aggregated per kernel using a box plot. The key insight of this plot is the high amount of

applications with large critical fractions: 19 out of the 27 analyzed applications have a critical fraction of over 75%. Although this high number suggests very large communication overhead for partitioned application, not all data on the critical path necessarily requires communication. Considering the total amount of data of an application is fixed, communication grows linearly with the number of partitions (GPUs) in the worst case. This can likely be compensated by the aggregated bandwidth of multiple-GPUs. Another important insight is the uniformity of kernels within each application: in more than half (15 out of 27) virtually all kernels behave identically with at most 1 outlier.

3.4.2 Partition-Level Analysis

While the kernel-level analysis provides some interesting insights about the general communication between kernels, it lacks in detail. The missing pieces are filled in by the partition-level analysis, which is more sophisticated and analyzes communication on a finer granularity.

CUDA's and OpenCL's programming model is based on the grouping of threads into thread blocks. Each thread block is assigned a 3-dimensional thread block index, determining its position in the thread grid. Each thread within a thread block is then assigned a 3-dimensional thread index determining its position within the thread block. The combination of these two indices uniquely describes each thread's position within the execution hierarchy. While communication between threads of the same thread block is possible and almost always utilized, communication between threads of different thread blocks is typically not reliably possible and strongly discouraged. An exception to this is the use of atomic operations, which massively reduces performance and is often avoided entirely. As a result, the natural way of splitting a GPU application is along the boundaries of thread blocks and should be possible in a transparent manner. This is the model of partitioning used in this analysis and described in more detail in section 3.2.2. All analysis is performed using the four mappings presented in section 3.2.3.

The first partition-level analysis computes the accumulated volume of all communication between the partitions of an application. The number of partitions the application is split into is always 16, which is a realistic estimate of high-end real-world systems [3]. The analysis is performed for all combinations of applications and mapping, but within a single simulated application run, the mapping is not changed. Each thread

[3] At the time of this writing, 16 installed GPUs is the typical upper bound of the most common cloud providers (Amazon AWS, Google GCP), as well as AI development systems (NVIDIA DGX-2)

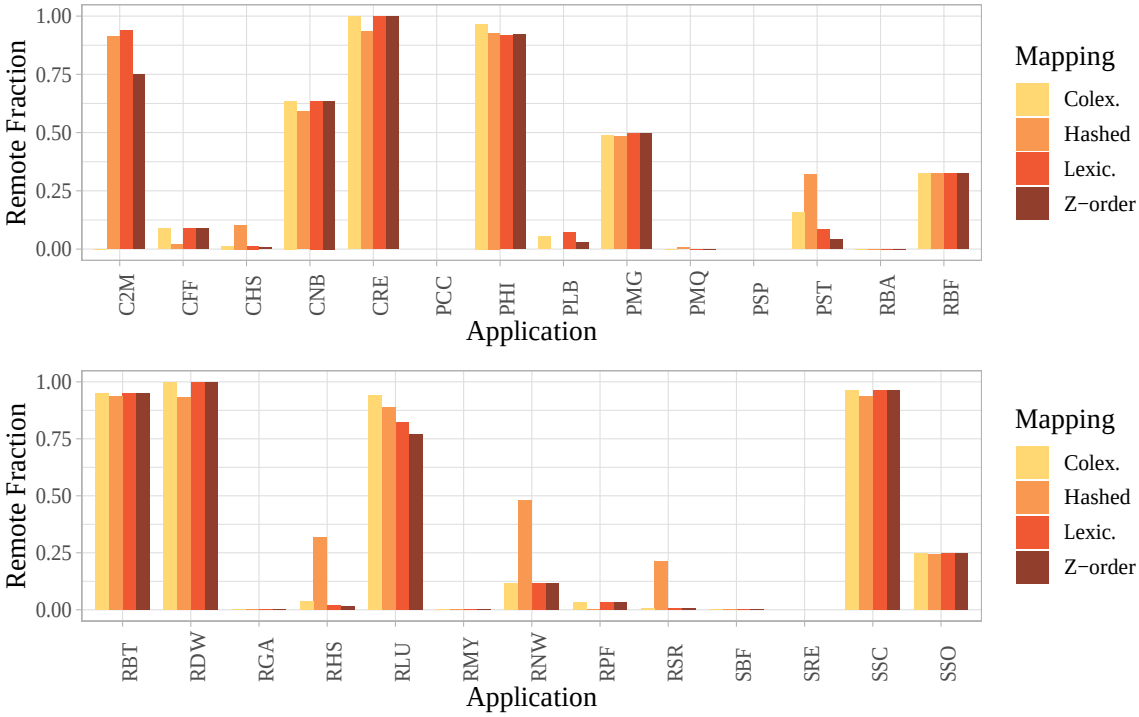


Figure 3.23: Sum of all data read from remote device of all partitions, normalized to the sum of data read from any non-host device.

block of a kernel is assigned to one of the 16 partitions and all addresses in its read set are assigned to one of the following categories:

Host. The data at this address has not been updated by any previous GPU kernel.

Local. The data at this address has been updated most recently by a thread block of the *same* partition.

Remote. The data at this address has been updated most recently by a thread block of a *different* partition. This data needs to be communicated between partitions on different devices.

The number of addresses in each category is accumulated over all kernels and finally, the sum of all remote data is normalized to the sum of local and data, host data is ignored for this metric. This analysis complements the one about the source of data presented in figure 3.21 as it provides more information about the fraction of data that results in communication.

Figure 3.23 reports the results for this analysis for all combinations of applications and mappings. The first observation is the variance between different applications:

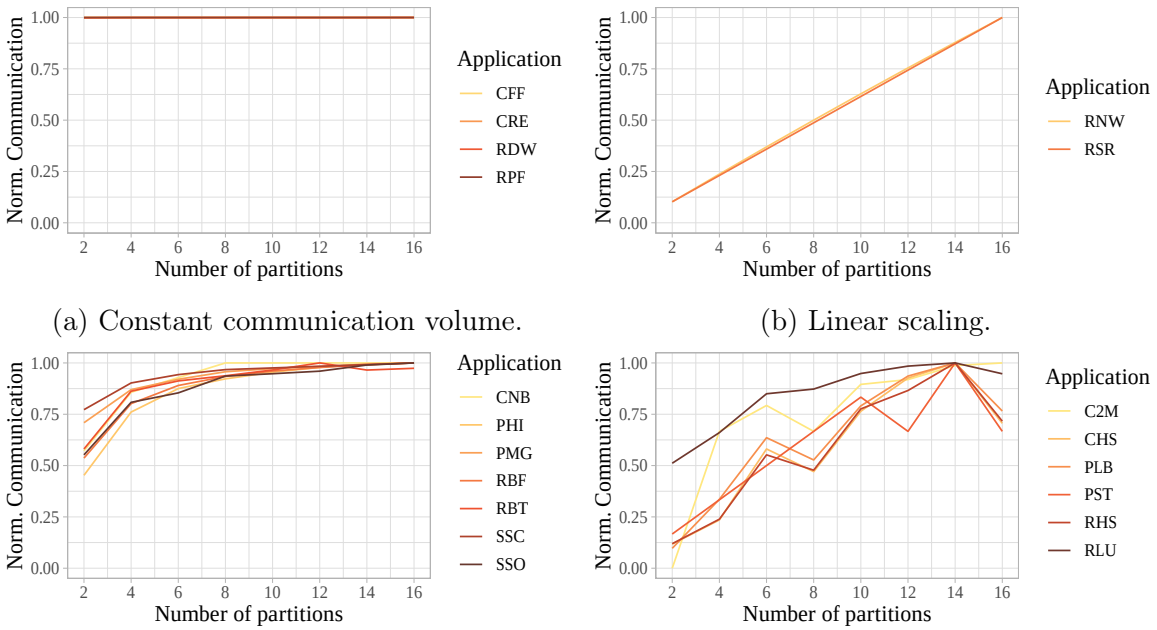
most applications read either almost all their data from a different partition or very little, only four applications fall into the range of 25% - 75% for three or more mappings. For the majority of mappings, 16 applications require less than 25% of their GPU-data to be communicated between partitions and only 7 applications require 75% or more to be communicated. This is an indicator of the high spatial locality of the algorithms implemented in GPU applications.

Another insight is the irregular behavior of the hashed mapping. It was intended as a random alternative that does not optimize locality in any dimension and, as intended, produces chaotic results. It scores the worst, best, and average in roughly the same number of applications. Interestingly, it seems to perform worst for applications with low overall communication. This can be attributed to these applications exhibiting high spatial locality, which is not exploited by the hashed mapping.

Many of the applications behave identical with the lexicographic, the colexicographic, and the Z-order mapping. This is caused by those applications only containing kernels with 1-dimensional thread-grids. In these cases, all three mappings fall back to a linear mapping. For the other applications, the Z-order mapping generally performs best, with three exceptions where either the lexicographic or colexicographic mapping win. Why some applications have a clear preference for one of the mappings can be illustrated using the chained matrix multiply (C2M). If a colexicographic mapping is used, the first matrix multiply splits its result into horizontal chunks among the 16 partitions. The second matrix multiply then reads the result in rows to compute the final matrix. The distribution of the read accesses matches that of the colexicographic partitioning and no data needs to be transferred between partitions. If instead the lexicographic mapping is used, the read pattern is directly opposed to the distribution of the intermediate matrix (vertical chunks), leading to the worst performance. The Z-order mapping and the hashed mapping, not distributing in horizontal or vertical chunks, fall in-between as expected.

The second analysis removes the constraint of using a fixed number of partitions and focuses instead on the behavior of the applications with varying numbers of partitions. It only uses the Z-order mapping to partition thread blocks due it providing the overall best behavior of all mappings. Some applications contain only kernels without any communication and are excluded: PCC, PMQ, PSP, RBA, RGA, RMY, SBF, SRE.

The metric used to analyze the scaling behavior is a modification of that used in the previous analysis. First, the total amount of data read from remote partitions is computed normally, but it is not normalized to the total amount of data read from any partition. Instead, it is normalized to the maximum amount of data read from



(a) Constant communication volume.

(b) Linear scaling.

(c) Asymptotic growth.

(d) Irregular growth.

Figure 3.24: Total communication between partition vs number of partitions, normalized to maximum per application. Categorized into the four different types of scaling behavior.

remote partitions over different numbers of partitions. The result is the change in the total amount of remote data read when varying the number of partitions. As an example, consider a scaling analysis for some application that returns the total number of remote bytes read for different numbers of partitions and produces the following pairs of $(n_{partition}, n_{bytes})$: (2, 4), (4, 8), (6, 12), (8, 16). In this sequence, 16 is the maximum amount of remote data and the reference to normalize against, resulting in the normalized sequence (2, 0.25), (4, 0.5), (6, 0.75), (8, 1.0).

The figures 3.24 present the result of this scaling analysis for all applications with communication between their kernels. Four basic classes of scaling behavior can be identified:

1. The first class contains applications exhibiting a constant amount of communication. As this is the total communication over all partitions, it implies that the communication per partition sinks linearly with the number of partitions.

2. The second class contains applications where communication grows linearly with the number of partitions used. The most likely reason for this behavior is a constant amount of communication per partition, for example, if every thread

block reads the full input data set.

3. The third class of scaling behavior is an offset asymptotic growth, that can be approximated for example using some configuration of the logistic function $\frac{1}{1+e^{-k(x-x_0}}$, with k describing the growth rate and x_0 the midpoint of the sigmoid function (in these cases always on the negative side of the x-axis). Applications falling into this category have more complex memory access patterns as the first two, but are still highly regular, resulting in the predictable behavior.

4. The last class is a catch-all for applications that behave differently. They primarily distinguish themselves from the linear and logistic growth classes due to their irregular jumps between certain numbers of partitions. While the communication volume scales irregularly, the memory access patterns might not necessarily also be irregular. One cause for the irregularities might be thread grid sizes of the kernels that are not amenable to the Z-order mapping and cause bounding or clustering issues as described in 3.2.3.

The key observation from this scaling analysis is that the communication volume grows at most linear with the number of partitions. For most applications, the communication volume even grows less than linear. In contrast, the total aggregated bandwidth in a multi-GPU system grows linear with the number of GPUs (assuming identical GPUs and a non-blocking interconnection network). This, again, is an encouraging result for automatic partitioning, as it suggests that the communication requirements arising from the partitioning can be covered by multiple GPUs.

3.5 Summary

This chapter presented an instrumentation framework that allows extracting the memory access patterns of GPGPU applications, a methodology for the analysis of these patterns with regards to the NUMA behavior of a multi-device execution of the applications, as well as a fully performed in-depth analysis of 27 applications of three benchmark suites, using four partitioning strategies.

The extraction of memory accesses is performed using an instrumentation system that is embedded into the application during its compilation and does not require any modifications to its source code [4]. Application instrumentation is implemented using a set of simple transformations on both the host and the device side. Memory

[4]The plugin can be used completely independently of the rest of the trace analysis and is available for download at https://github.com/UniHD-CEG/cuda-memtrace

accesses are first written into a queue in host memory and then flushed to disk, where they can be processed further. The transformation is packaged into a plugin for the Clang/LLVM compilation framework and a static library. This allows a simple usage model that only requires one additional flag each for the compilation step and the linking step.

The analysis relies on summaries of the memory accesses performed by thread blocks. The memory accesses of all threads in a thread block are combined to form one read set and one write set. These sets are then used to identify NUMA effects in virtually partitioned single-GPU applications, by splitting the thread grid into partitions using a selection of four simple mappings. Each of the mappings has different properties regarding the locality and distribution of the partitions, which are explained in-depth. The memory accesses of each partition are then combined and interactions between partitions interpreted as communication. Due to the relaxed consistency guarantees of the GPU memory model, communication between partitions can only occur between different kernel launches.

This analysis methodology is then used to compute different metrics, some of which ignore partitions and focus on communication between kernels as a whole, while others focus on inter-partition communication. A total of 27 applications have been analyzed and classified with regards to their locality. The findings are then put into context by discussing their implications for an automatic partitioning that is executed on real hardware. The analysis results in three key findings:

1. GPGPU applications differ dramatically between each other with regards to the amount of data read from the GPU but all read large amounts of data from the host.

2. More than half of all benchmarks require less than 25% of their data to exchanged between devices to synchronize between iterations. The exception is the semi-random hash mapping, which performs significantly worse.

3. The Z-order mapping creates partitions in such a way that the total communication volume grows slower with the number of GPUs than the aggregated bandwidth of these GPUs.

Splitting a grid of independent units of computation is easy, but communication typically turns out to be a bottleneck. However, the second end third key findings are very encouraging and suggest that an automatic way to partition GPU applications is feasible.

This chapter produces three artifacts: a reusable instrumentation framework to collect memory traces of CUDA applications, an analysis methodology based on these traces that reveals the internal communication of a GPU application, and the analysis applying these tools to a selection of popular GPU benchmark suits. Although the analysis focuses on the feasibility of automatic partitioning schemes, the instrumentation framework and analysis methodology can be applied to other investigations, such as the identification memory-related performance bottlenecks in GPU applications.

CHAPTER 4

Automatically Partitioning CUDA

The chapter presents the design and implementation of an automatically partitioning compiler for CUDA. It starts in section 4.1 by identifying the required tasks, presenting techniques and components that can perform these tasks, and discussing the most viable option that was chosen for this project. Then, an overview of the toolchain is provided in section 4.2, followed by explanations of the transformations it performs to translate single-GPU applications into multi-GPU binaries in sections 4.3 to 4.7.

4.1 Concept

This section provides a high-level overview of the challenges that arise when automatically partitioning GPU applications and their solution. It builds on the results and insights of previous work of the author [90]. Potential implementation details are not discussed at this point except where these details are critical for the feasibility of an implementation.

The basic premise used to automatically partition CUDA applications is the fact that thread blocks in a CUDA are independent of each other and cannot communicate with each other [14]. Due to communication being possible in a reliable way between threads of the same thread block, threads in a thread-block typically cooperate with each other (also giving them the name CTA), making it impossible to split thread blocks while guaranteeing correct results. In this regard, they share some similarities with tasks in task-based programming models such as X10 or HPX, albeit far simplified [91, 92]. The main similarity between the two is that tasks can be executed independently of each

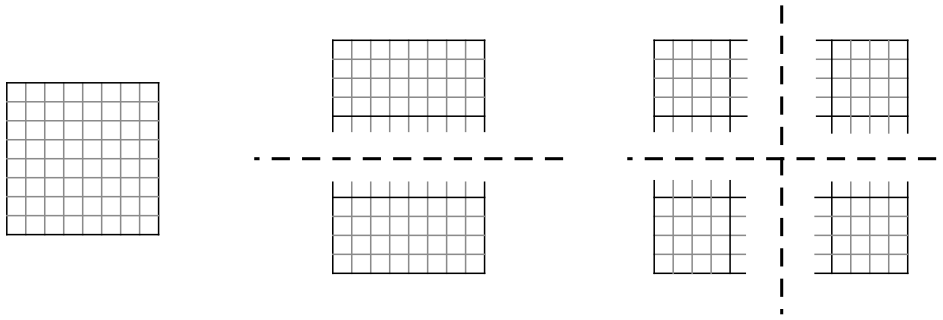


Figure 4.1: Subdivision of a 2-dimensional thread grid using straight cuts across its axes.

other as pure functions, taking some input and producing some output without side-effects. There are significant differences, however, as task-based models often contain sophisticated scheduling algorithms with complex dependency graphs and support nesting, which is not the case for CUDA thread blocks. In the CUDA execution model, all thread blocks of a kernel are considered ready for execution when the kernel is scheduled.

Distributing the execution of a CUDA kernel across multiple devices requires three steps:

1. splitting the thread blocks of a CUDA kernel into multiple partitions,
2. identifying the data dependencies of all thread blocks in each partition, and
3. synchronizing input and output data to satisfy data dependencies.

All illustrations in this section use a 2-dimensional stand-in for the thread grid for clarity, without loss of generality. The illustrated concepts apply directly to the 3-dimensional thread grid of CUDA.

4.1.1 Work Partitioning

Splitting computation, i.e. the thread grid of a kernel, in CUDA is simple: the thread grid can be subdivided along thread block boundaries by cutting along any of its three axes. The CUDA consistency model assumes that thread blocks do not communicate with each other by any means except for atomic operations. Consequently, applications that use atomic operations cannot be automatically split using this approach. This step of cutting the thread grid into half is then repeated until the desired number of partitions is created. Figure 4.1 illustrates this process on a 2-dimensional grid.

One way of limiting the thread blocks that are executed on a GPU is called an "execution guard" and implemented by putting a conditional return in front of the kernel code [93]. The execution guard tests whether the current thread index is part of active partition and aborts if the test fails. This requires scheduling all thread blocks, even if only a small portion of them pass the execution guard. These thread blocks that do not pass the execution guard still consume resources on the GPU, preventing new thread blocks from being scheduled. Alternatively, the execution guard can be eliminated by instead shrinking the thread grid for each kernel launch down to the size of its partition and adjusting the calculation of the thread block index to reflect its position in the original thread grid. However, this way of splitting a thread grid is only possible in this case because the thread grid is split by cutting along its axes, resulting again in an axis-aligned cuboid. Sub-sections of a thread grid in any other shape require an execution guard.

The best dimension to partition along depends on the optimization goal and the application in question. Since thread blocks are assumed to be executed independently of each other, the most interesting factor to optimize appears to be the volume of communication. Extensive work has been done in this area to optimize the partitioning as a function of both input and output memory access patterns [94]. This work uses a significantly simpler approach that aims to reduce fragmentation in the input data by cutting across the dimension that causes the largest stride in the access patterns of the input data. As an example, consider a 2-dimensional kernel accessing a 2-dimensional array stored in row-major order using the formula `float value = array[y * n + x];`, with `x` and `y` being the current thread's respective global X and Y index and `n` the size of the array in the X-dimension. Here, consecutive elements in the Y-dimensions exhibit a stride of n, while consecutive elements in the X-dimension only have a stride of 1. The thread grid should be horizontally split across the Y-axis, resulting in the accesses of each partition forming two large consecutive chunks in memory.

4.1.2 Identification of Data Dependencies

The key challenge when partitioning a GPU application is the satisfaction of all data dependencies of the partitioned kernel.

Although powerful tools exist to help identify the data dependencies of a particular piece of code, this is not always necessary and the problem can be partially side-stepped. Two simple solutions do not require the identification of data dependencies:

1. Fully replicate all arrays on all devices, automatically satisfying all data depen-

dencies at the cost of large upfront data transfers. Partial results need to be merged, which can be successfully done at runtime.

2. Place all buffers in memory that is shared between the host and all GPUs. Data dependencies are then solved by directly accessing (remote) memory. This approach is available for devices that support CUDA 6 and above.

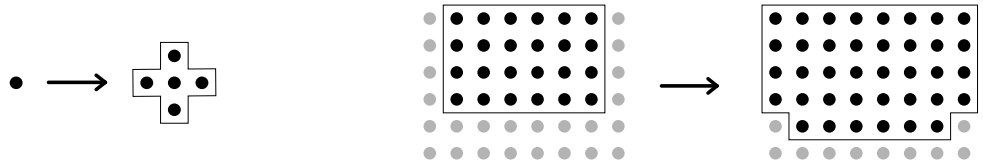
Although both solutions work and do not require any data flow analysis, they have suboptimal performance. In the first case, always copying the full buffer although often only a small portion is required wastes several Gigabytes of data transfers for larger problems. In the second case, the high latencies between different PCIe devices (tens of microseconds [95]) destroy any potential speedup from distributing the computational work.

An approach that only uses minimal static analysis is an application of the "merge kernels" of the Single Kernel Multiple Devices (SKMD) project [93]. In this approach, a kernel is stripped from all non-index related computation and only writes a bitmap of all read accesses performed by the kernel. This bitmap of the data dependencies can then be used to copy the corresponding data to the GPU. The VAST project uses a page-based version of this approach with a granularity of 4 KB [96]. However, the required round trip to the GPU and before data distribution as well as the required fine-tuning of the page size render the approach unsatisfactory for this work.

Many approaches for dependency analysis use static analysis to build a mathematical model of the data dependencies. The details of such a model are designed to match the given problem and represent a compromise between usefulness and simplification on one hand and expressive power and complexity on the other hand.

The SKMD project uses a custom model that represents memory accesses as a linear function of the global thread index in any of the three dimensions (represented as $a \cdot (W_i \cdot w_i + l_i)$, with a as a constant or loop induction variable, and W_i, w_i, and l_i the thread block size, thread block index, and thread index respectively) [93]. Although simple and robust, this model is severely limited in the type of access patterns it supports and even a constant offset can prohibit using this approach.

A more sophisticated approach is the polyhedral model, where loop iterations are modeled as sets of multi-dimensional points and memory accesses as maps from between such sets. This model can represent complex applications as long as loop boundaries and indices memory of accesses can be described using presburger relations [97]. It's An in-depth description that contains all of its aspects required for this work can be found in 2.4. Since extracting the maximum performance from GPUs requires very regular



(a) Memory access pattern of a stencil, mapping a single 2-dimensional point to 5 2-dimensional points.

(b) The polyhedral memory access pattern applied to a partition of a 2-dimensional thread grid. Notice the correctly handled out-of-boundary condition at the top of the grid.

Figure 4.2: Illustration of polyhedral memory access patterns as used in this work.

control-flow and memory accesses, this model is expected to be a good fit for GPU applications. Although it also primarily relies on memory accesses being described by linear functions, the model can be parametrized with arbitrarily many parameters and supports arbitrarily many constraints on each memory access, allowing to model complex memory accesses. Being based on a rigorous mathematical foundation makes it easy to use and its generality should make it easy to adapt to GPU applications.

From the available options, polyhedral compilation seems to fit the requirements of this project well and is chosen as its foundation. The model describing a CUDA kernel relies primarily on two abstractions:

- The CUDA thread grid and the partitions are represented by 3-dimensional polyhedral sets. Due to the constraint that the thread grid can only be partitioned by straight cuts along the three axes, the resulting sets will always by axis-aligned cuboids.

- Memory accesses are represented by linear function mapping a thread of thread grid or a partition to an element in a n-dimensional polyhedral set representing the kernel's arrays. Multiple memory accesses to the same array can then be combined into one union map for simplification. Conditional accesses in the kernel are modeled by applying the corresponding constrains to these maps.

Figure 4.2 illustrates the use of polyhedral memory access patterns. Subfigure 4.2a on the left illustrates the actual memory access pattern as a mapping from a single 2-dimensional point to a union of 5 different 2-dimensional points. The subfigure 4.2b on the right is the application of this memory access pattern to a partition of a 2-dimensional thread grid. The partition is described by the constraints $x \geq 1 \wedge x \leq 6 \wedge y \leq 3$ (starting to count from zero). The resulting access in a 2-dimensional array of the same size is the union of the memory access pattern applied to each individual

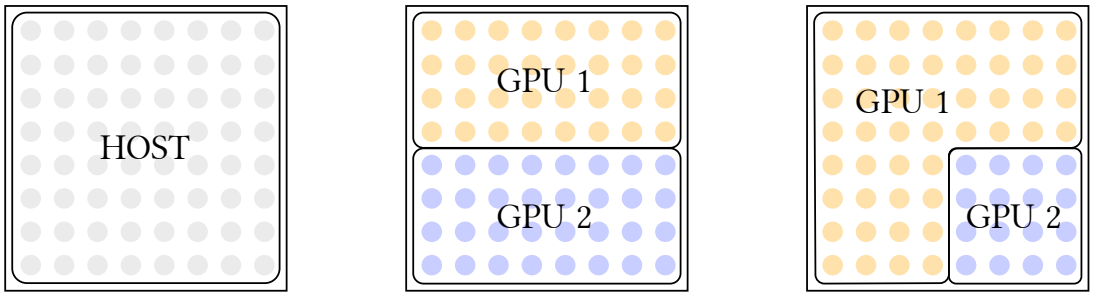
of the partition and handles boundary conditions, such as array sizes, correctly. Such a memory access pattern is created for the read and write accesses for each of the arrays used in a CUDA kernel and allows the reliable identification of input data that potentially requires communication between GPUs.

4.1.3 Synchronization of Input and Output Data

Identifying which areas of a buffer are read by the threads of a specific partition of a kernel is an important step in satisfying data dependencies, but not sufficient on its own. Once the data dependencies are identified, they must be satisfied by either confirming a local copy of them already exist or transferring the most recent copy of them to the local GPU.

This requires matching the output of previously executed kernel launches against the input of the next kernel launch. One approach is to rely on static analysis to fully model the data-flow between different kernels. For single-device code without accelerators, such a data-flow analysis can be performed fairly reliable and has been used to in the past [67]. The data-flow analysis also needs to correctly model potentially opaque use of pointers (e.g. buffer swapping, potentially as a side effect using `std::swap`) in addition to matching read and write patterns. In CUDA applications, GPU kernels and the host are only loosely coupled via auto-generated boilerplate code that retains only little semantic information. For an approach fully relying on static analysis, all kernels that might have been launched in the past and their order have to be taken into account to accurately model data dependencies. Due to these difficulties with static analysis, this work instead relies on a dynamic approach, trading the conceptual difficulties for runtime overhead.

The dynamic approach relies on a tracker that keeps track of the state of the elements in a buffer and can be queried to look up the most up to date location for each element. The tracker is associated with its buffer using the buffer's base address. This eliminates the need for any data-flow analysis to identify buffers. The tracker itself is similar in concept to the one used to track ownership of elements in the trace analysis in section 3.2. It is a dynamic component that manages the buffer state as a set of non-overlapping ranges storing the last writer to a set of addresses in the buffer. A simple version of this approach is illustrated in figure 4.3 for an application with two kernel launches. The first kernel has a simple write pattern and is split into an upper and lower partition, each located on a different GPU. The second kernel has a more complicated write pattern that leaves three-quarters of the most recent data on GPU 1 and only one quarter on GPU 2. Whether the second kernel has updated all



(a) Buffer state before any kernel launches. All data was last updated (initialized) by the host.

(b) Buffer state after a kernel with two partitions was executed. The top half was updated by GPU 1, the bottom by GPU 2.

(c) Buffer state after another kernel with a different memory access pattern. Regions in the buffer do not need to be polyhedra.

Figure 4.3: Possible evolution of buffer state over the runtime of an application with two launches of different kernels.

the data or only the lower-left quarter cannot be statically determined, but is resolved at runtime using the tracker.

The read pattern of a kernel on a particular buffer can then be used to query for the location of all required data and issue the corresponding data transfers. Polyhedral compilation provides facilities to generate very optimized code to query the shape of polyhedra, which can be used to both update the tracker and query it, reducing runtime overheads [71].

4.2 Toolchain Overview

This section provides a quick overview of the architecture of the prototype compiler, to help to guide the reader in the following sections that provide in-depth explanations of all its components. The project is built on top of gpucc, a CUDA compiler that is integrated into the Clang and LLVM projects [2]. This avoids having to re-implement a CUDA compiler, which is strongly preferred given the complexity of modern compilers. Additionally, the LLVM project is designed to be very modular and provides interfaces that allow an easy extension of its functionality. However, not all required functionality could be reasonably implemented as part of LLVM directly.

Most applications written in C or C++ target a single architecture. In contrast, CUDA is a language designed to include computational kernels for GPUs that are embedded into a host program running on the main CPU. The dual nature of these applications requires to compile code for the architectures of both systems, typically

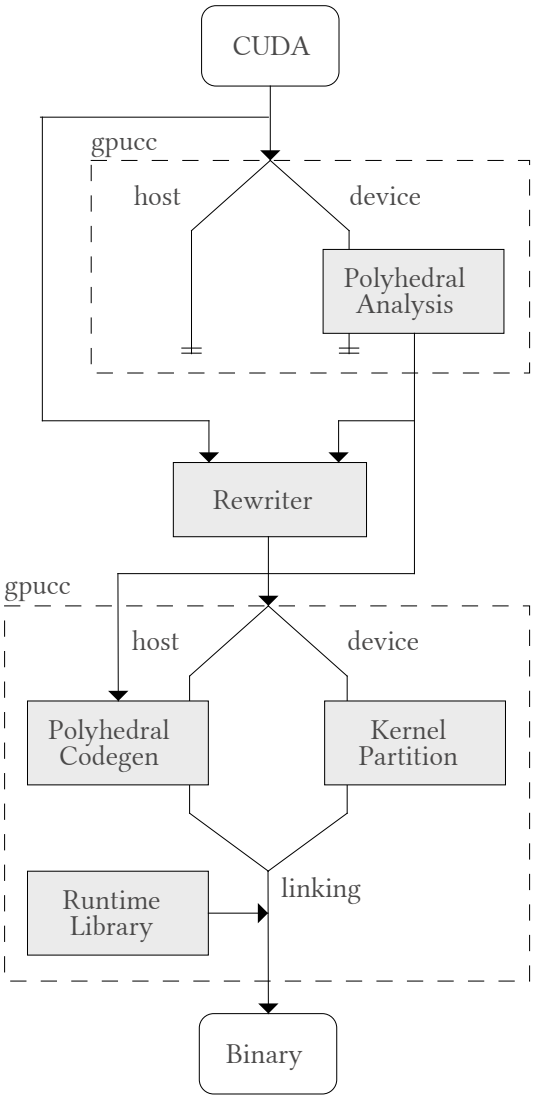


Figure 4.4: Toolchain overview

x86/ARM for the host and PTX for GPU related code. In gpucc, this is achieved by implementing two separate pipelines. GPU code is compiled first, ignoring all CPU code and resulting in an artifact that contains the generated code. Then, all host-related code is compiled, GPU-related code is parsed to collect information about the kernels that are present, and the artifact created from the first pipeline is embedded into the resulting compiled object.

The automatically partitioning prototype compiler keeps this model mostly intact. However, during GPU compilation, memory access patterns are extracted and the kernels are modified to be relocatable within the thread grid. Information about the memory access patterns or transformations can not be reliably passed from the

GPU-pipeline to the host-pipeline without large changes to the components. Therefore, both steps were implemented as pre-processing steps, requiring two passes to compile the application. Figure 4.4 illustrates the modified toolchain implemented by the partitioning compiler, highlighting the differences to the toolchain of the unmodified gpucc compiler [2].

Overall, the automatic partitioning implements the following additional steps, each detailed in their corresponding section:

1. Transformation of GPU kernels to implement work splitting into rectangular partitions of the thread grid (section 4.3).

2. Analysis of GPU kernels and creation of a polyhedral model of their behavior regarding buffer usage and memory access patterns (section 4.4).

3. Code generation for efficient queries of memory access patterns, used to identify data dependencies and redistribution of partial results (section 4.5.

4. Insertion of a static runtime library implementing communication primitives and kernel management (section 4.6.

5. Source-to-source transformation of host code based on the analysis, inserting kernel management and communications code from a static runtime and generated code for memory access patterns (section 4.7).

The following sections explaining the individual components have been ordered in a way that the author feels requires the least jumping between sections, which is not the order in which they are called used in the compiler. For example, the transformation of GPU kernels for work splitting is performed late in the pipeline (in the second gpucc invocation) but does not depend on any of the other steps; therefore, it is explained first. Host code transformations depend on essentially all previous steps and are therefore explained last.

4.3 Kernel Transformations for Work Splitting

The execution model of CUDA provides a two-level hierarchy, the thread grid containing thread blocks, and thread blocks containing individual threads. When configuring a CUDA kernel launch, the full thread grid is scheduled for execution, without a built-in mechanism to only execute a subset of kernels. As detailed in section 4.1, work should be split into a cuboid subset of the thread grid to avoid the need for execution

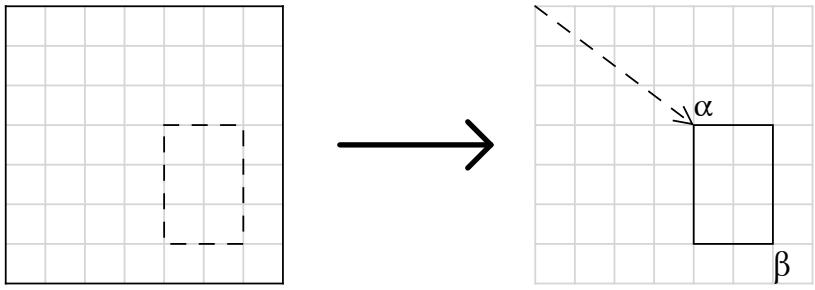


Figure 4.5: Illustration of the work splitting as implemented in this work. Partitions are created by shrinking and shifting the thread grid to the size specified by α and β.

guards. The resulting partitions therefore are axis-aligned bounding boxes (AABBs) and sufficiently described by a pair $(\vec{\alpha}, \vec{\beta})$, with $\vec{\alpha} \in \mathbb{Z}^3$ describing the minimum point and $\vec{\beta} \in \mathbb{Z}^3$ the maximum point in X, Y, and Z respectively. The 3-dimensional interval describe by these points is half-open (i.e. $\vec{\alpha}$ is the first point outside of the interval), so the size of the partition can be calculated with the formula $\beta_i - \alpha_i$ with $i \in \{x, y, z\}$ for any of the three dimensions.

Executing the thread blocks of such a partition requires two steps that are mostly independent of each other. First, schedule the kernel with a thread grid of the same size as the partition and then forward information about the partition to the kernel. Second, manipulate all index and thread grid size computations in such a way that boundary conditions and array index calculations are the same as if the kernel was executed with its original thread grid. In other words, the thread grid needs to be shrunk to the size of the partition and then shifted to its position, as illustrated in figure 4.5. Shrinking the thread grid is the responsibility of the host code when launching the kernel, leaving only the transformation of thread-grid-related computations inside the kernel.

First, a copy of the kernel code is made and modified to accept additional arguments that describe the partition as the pair $(\vec{a}, \vec{b})$. The kernel copy is named using a convention that allows it to be found by other components of the compiler.

Then, calculations based on the thread grid can be updated according to the following to substitution rules, which are applied for each of the three dimensions:

$$blockIdx_i \rightarrow blockIdx_i + \alpha_i \tag{4.1}$$

$$gridDim_i \rightarrow \beta_i \tag{4.2}$$

The first rule (equation 4.1) updates all occurrences of the thread block index to their new position inside the partition by adding the minimum point of the partition. The second rule (equation 4.2) updates computations using the thread grid size to instead

use the maximum point of the partition. These substitutions only keep the semantics of the original CUDA code when the thread grid size the kernel is launched with matches the size of the partition.

Since work is always split along the boundaries of thread blocks, leaving them intact as the atomic scheduling unit, the size of thread blocks remains unchanged. The same is true for all calculations performed with a thread block's size inside the kernel. This is desirable because CUDA applications often rely on thread blocks having a specific size for optimizations and changing this result can lead to incorrect results, an example being tiling optimizations [98].

4.4 Polyhedral Analysis

Several polyhedral compilation frameworks exist and they all require a polyhedral analysis to build a model of code region they try to optimize. Such an analysis could have been repurposed for this analysis, but they are typically designed to the exact requirements of the optimizations they are used for [59, 32]. Instead, the analysis used in this work largely based one that was created as a reaction to the growing demand of polyhedral analysis as a stand-alone tool (referred to as low-level polyhedral analysis from here on) [99]. The analysis is implemented as a set of LLVM passes that provide a general-purpose polyhedral analysis with few dependencies. Although the low-level polyhedral analysis and the ones used by popular polyhedral optimizers share similarities in that both try to create a polyhedral representation of the application's behavior, there are significant differences. The analysis stage of polyhedral optimizers is geared towards their specific purpose, the optimization of loop nests, and as such typically include heuristics to select promising candidates and discard less promising ones as well as early exits if a code region is less amenable for analysis. In contrast, the low-level polyhedral analysis is less restricted in its applicability and can generate (approximative) results for any reducible control flow graph, which covers virtually all applications written in structured languages such as CUDA [100]. This work is intended as a prototype and low-level correctness issues, e.g. integer overflows, are currently ignored. This is a simplification that is often used for research projects and does not result in a loss of generality [101].

The application model is built around the memory access patterns exhibited by GPU kernels. Read and write accesses to arrays in global memory are the primary means of communicating input and output data between the host and the GPU. Scalar kernel arguments, which are not located in global memory, are also used to feed

information to a kernel, but their use is usually restricted to configuration parameters as opposed to input data sets.

The low-level polyhedral analysis divides all accesses into one of two categories: *must* and *may* accesses. While *must* accesses are those load and store operations that are performed unconditionally, *may* accesses are part of a conditional branch and may or may not be executed. Although this distinction is required for high accuracy of the analysis, this work currently discards the information and over-approximates accesses by treating all load and store instructions to global memory as *must* accesses. The over-approximation does not impact correctness but may lead to unnecessary transfers.

In polyhedral compilation, memory accesses are typically expressed as a function that maps the induction variables of a loop nest to an index in some (multi-dimensional) array. In CUDA kernels, the thread grid is an implicit loop nest iterating over all thread blocks, which in turn represent a loop nest iterating over all threads in a thread block. The coordinates of these implicit loop nests, thread-block indices and thread indices, are typically used as the parameters for domain decomposition. As a result, array indices are often a function of thread block indices and thread indices.

Neither the thread block index nor the thread index within the block can individually be used to uniquely identify the current thread on their own. Instead, the unique global index of a thread is a combination of both and computed using

$$threadIdx_i + blockIdx_i \cdot blockDim_i \tag{4.3}$$

for the dimension $i \in \{x, y, z\}$. Both *blockIdx* and *blockDim* are unknown at compile-time, resulting in the non-linear term that can not be represented in the polyhedral model. However, since the block size is known at the launch of a kernel, the product can be replaced by a virtual "block offset" that encapsulates the non-affine multiplication [102]. The result of this substitution is the affine expression

$$threadIdx_i + blockOff_i. \tag{4.4}$$

As described in 4.1, each memory accesses is modeled as a function that maps a thread to the array element it accesses. Since a thread is uniquely identified by a combination of its thread block index and its thread index, this should be used as the domain of the polyhedral map that models the memory access. Incorporating the thread block offset to avoid non-affine expressions in memory accesses, this results in a total of nine dimensions in the domain of a memory access. This results in a total of nine dimensions, each of the thread grid dimensions x, y, and z represented

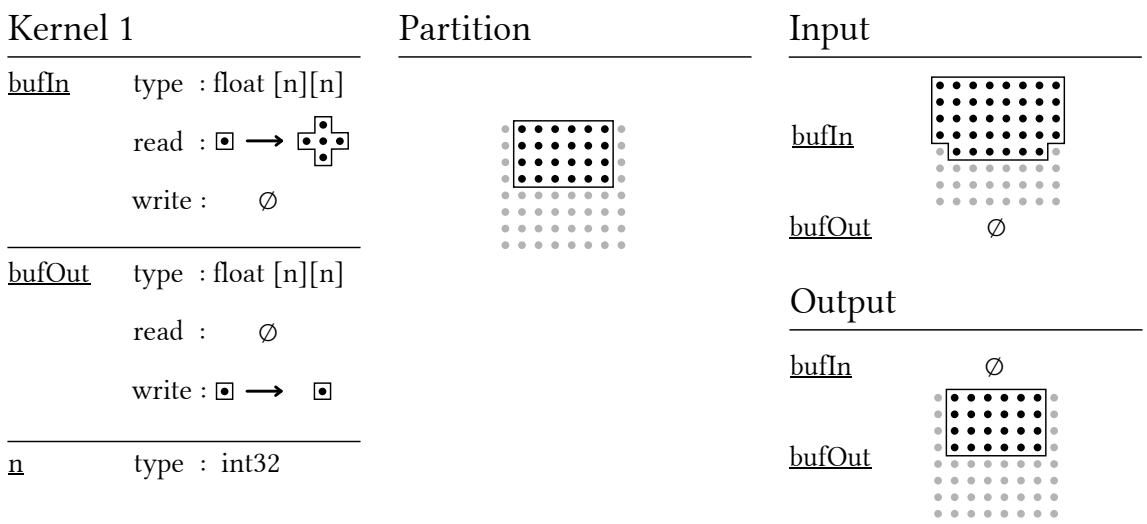
by their components *blockOff*, *blockIdx*, and *threadIdx*. Based on CUDA's execution model, there is no benefit in modeling individual threads since all threads of a thread block are scheduled in one batch. The domain can be simplified by eliminating all dimensions for the thread index. This is achieved by adding the additional constraint $0 \leq threadIdx_{\{x,y,z\}} < blockDim_{\{x,y,z\}}$ to all memory access maps and then projecting out the thread indices.

The resulting memory access maps then are functions of $\mathbb{Z}^6 \to \mathbb{Z}^d$, $d \in \mathbb{N}$ being the number of dimensions of the array that is being accessed. The languages C, C++, and CUDA all perform some kind of array-to-pointer conversion when arrays are passed to functions or used in expressions, which is called "array decay" and loses information about the arrays dimensions and size [103, 104, 14]. As a result, multi-dimensional arrays are manually linearized using expressions such as $i_2 \cdot n_1 + i_1$ for access to a 2-dimensional array, with i_1 and n_1 being the respective index and size in the innermost dimension and i_2 being the index in the outer dimension. This is not an affine expression and can not be represented in the polyhedral model. The solution employed by the low-level polyhedral analysis is to specifically search for expressions of this form in array indices and recover the original semantics by marking the array as potentially multi-dimensional. If all accesses to an array suggest the same number and sizes of its dimensions, the memory accesses can safely be transformed into multi-dimensional ones, often becoming linear in process. As a convention, dimensions are ordered outer-most to inner-most from left to right for all polyhedral expressions. Taking up the example above, the 2-dimensional access could be represented by the polyhedral map $\{ [i_2, i_1] \to [o_2 = i_2, o_1 = i_1] \}$, which consists exclusively of affine expressions. The recovery of array dimensions is only valid if arrays are addressed using compatible linearized indices for all accesses, e.g. they are invalid if i_1 can not be proven to always be smaller than n_1. Memory access maps then have the form

$$\{ [blockOff_{\{x,y,z\}}, blockIdx_{\{x,y,z\}}] \to [o_n, ..., o_1] \}, \tag{4.5}$$

$o_n, ..., o_1$ representing the array indices in each dimension. This approach is not specific to the low-level polyhedral analysis and used in some form in most polyhedral projects using a C-based language as input [105, 106, 107].

By extracting these maps with a CUDA-friendly domain and recovering multi-dimensional arrays, arbitrary shapes of partitions can be supported. Given some read access to an array represented by a polyhedral map and a thread grid partition represented by a polyhedral set, the memory read by that particular partition can be computed by intersecting the memory access' domain with that of the partition.



(a) The application model created for the stencil kernel. The model contains data types, array sizes and scalar arguments of the kernel.

(b) The partition the model is applied to.

(c) The input and output data sets for the involved buffers for the required partition.

Figure 4.6: An illustration of an example application model for a 2-dimensional, out-of-place Jacobi stencil (5-point), applied to a 2-dimensional partition of the thread grid.

For axis-aligned cuboids as used in this work, the intersection can be performed by constraining each dimension of the memory access' domain with an upper and lower limit. While less expressive than arbitrary polyhedral sets, partitions can now be described using a fixed set of numbers. The corresponding parameters are introduced as additional constraints, following the recipe

$$
\begin{aligned}
&[\ boffmin_i, boffmax_i, bmin_i, bmax_i\ ] \rightarrow \\
&\{\ :\ boffmin_i \leq blockOff_i < boffmax_i\ \wedge\ bmin_i \leq blockIdx_i < bmax_i\ \}, \\
&boffmin_i \coloneqq bmin_i \cdot blockDim_i \\
&boffmax_i \coloneqq bmax_i \cdot blockDim_i
\end{aligned}
\tag{4.6}
$$

for each of the X, Y, and Z dimensions of the thread grid. The parameters $boffmin_i$ and $boffmax_i$ are introduced to avoid non-affine expressions and must be calculated using the provided formula.

As a final preparation before constructing the application model, the maps of write accesses are tested for injectivity. In this context, an injective write map corresponds

to memory being accessed in a way so that no two thread blocks try to write to the same memory location. This proves the absence of write-after-write (WAW) hazards contained in the application, breaking the CUDA consistency model. While single-CUDA applications might tolerate this behavior when executing on a single-GPU by relying on eventual consistency, updates from different thread blocks will never be visible to kernels executing on other GPUs before synchronization.

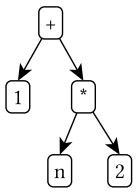
Figure 4.6a illustrates the model extracted a CUDA application containing a 2-dimensional Jacobi stencil. The kernel has three arguments `float *bufIn`, `float *bufOut`, and `int n`. For scalar arguments, only the data type is stored. For array arguments, the data type, the number, and sizes of dimensions, as well as the read and write maps are stored. In this case, both `bufIn` and `bufOut` contain a square grid of floats, with one array acting as input and the other as output respectively. Arrays are not required to be either pure input or pure output as shown in the illustration, they can be both input and output at the same time. Figure 4.6b presents a partition with the specification $\alpha = (1, 0, 0), \beta = (7, 4, 1)$, using the notation from 4.3 and starting to count from zero. The input and output data sets of this partition, based on the Jacobi kernel's model, are illustrated in figure 4.6c. The input array `bufIn` is exclusively read and contains a halo area. Out-of-bounds memory accesses are avoided by conditional code in the kernel, which is preserved in the model that only adds the halo area where appropriate. The output array `bufOut` has a $1:1$ access map and does not create WAW hazards; the resulting application model is therefore valid and can be used to partition the application.

4.5 Polyhedral Code Generation

The polyhedral library that implements the foundation of the application model allows querying the properties of sets and maps directly from their polyhedral representation. For most queries (e.g. extreme values) this involves solving systems of linear equations, which has a complexity of between 2^3 and 2^2 in the general case [108, 109]. Employing such complex algorithms at runtime to identify data dependencies is likely to become a bottleneck.

Fortunately, polyhedral compilation tools are designed to generate parametrized solutions for queries at compile time that can then be evaluated very efficiently at runtime. The solution to any query is either a set, a map, or a polyhedral expression. Computing the extrema of a convex set in a given dimension produces two polyhedral expressions, each representing one extreme value. Such an expression can be interpreted

$$[n] \rightarrow \{\ 2n + 1\ \}$$



(a) The original polyhedral expression and the generated AST.

```
1 %1      = mul 2 %n
2 %result = add 1 %1
```

(b) The LLVM IR created to compute this expression.

Figure 4.7: Code generation example for a polyhedral expression. Expressions always produce a single scalar value and do not contain control flow.

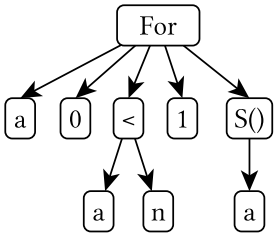
and evaluated as is, providing a correct, albeit slow, result. Similarly, enumerating the points of a set (e.g. resulting from the intersection of two other sets) can also be performed by interpreting the set description, computing the polyhedrons corners, and interpolating from there. However, a polyhedral compilation library provides a way to create a representation of the set that, when evaluated, enumerates the points directly, e.g. in the form of a 2-dimensional loop nest. This generation of highly optimized parameterized solutions is, in fact, one of the core features of polyhedral libraries and research to find further optimizations is ongoing [68, 71, 110, 69].

4.5.1 LLVM IR Generation from isl ASTs

The *isl* library is a polyhedral compilation library and as such provides facilities to aid the code generation for polyhedral queries. Its code generation facilities are designed to provided ASTs for highly optimized solutions for polyhedral sets and expressions. Depending on the type of input, the resulting AST has one of two forms. ASTs generated from polyhedral expressions produce a single scalar value that is represented by a tree of the required computations. The ASTs generated from polyhedral sets, however, produce a loop nest that calls a payload function for each element in the set. This is mirrored by the type system for *isl* ASTs, which provides different types for each type of AST.

The first kind is an *AST expression* generated from closed-form polyhedral expressions. They contain a computation tree, where the node represents operators (such as addition or multiplication) and leaves reference either constant values or variables. Most operators (except some specialized types of division) have an LLVM instruction that directly implements the operator. Code for these trees can be generated by performing a post-order traversal on the tree and emitting an instruction for each

$$[n] \rightarrow \{\ [a] : 0 \leq a < n\ \}$$




```
for (int a = 0; a < n; a += 1)
  S(a);
```

(a) The polyhedral set, the corresponding AST generated by *isl* and the equivalent C-Code.

```
Entry:
  ; code generation starting point
  br label %ForHead

ForHead:
  %a = phi i32 [0, %Entry], \
               [%nexta, %ForBody]
  %cond = icmp slt i32 %a %n
  br i1 %cond, label %ForCont, \
               label %ForBody

ForBody: ; loop body insertion point
  call void(i32) @S(i32 %a)
  %nexta = add i32 %a 1
  %br label %ForHead

ForCont:
  ; code generation continues here
```

(b) The LLVM IR generated for the loop.

Figure 4.8: Code generation example for polyhedral set. The generated code iterates over all elements in the set by creating loops.

operation. The operands of each operator are either leaves (constant values or variable names) or nodes representing the result of preceding operations, which are already emitted due to the post-order traversal. *AST expressions* always generate straight-line code without any control flow and execute in constant time. Figure 4.7 illustrates this approach using a simple example with two operations.

The second kind of object is an *AST node*, created from polyhedral sets. They represent a loop nest that iterates over all elements in a set and call a user-defined payload function for each element. Since the *isl* library does not assume an order between dimensions and requires the user to provide a schedule that defines the order of traversal over the set. Once a schedule has been used to create an *AST node* for a polyhedral set, each of its nodes represents one of five constructs, implementing different control flow: blocks, for-loops, if-else-branches, callbacks, and marks. Together they again form a tree that can be traversed recursively to generate the corresponding code. In order to set up place holders for the contents of each of the nodes, this AST is traversed in pre-order, creating a basic block for each node. The following segments describe each of the available types of control flow and how to generate the corresponding code.

Mark nodes are used to attach arbitrary information to a schedule and retrieve it

during code generation, they are not required for this work and can be ignored.

User nodes represent the payload callback that is called for each element of the set. The n-dimensional index of the specific element is provided by several *polyhedral expressions*, which can be passed as arguments to the callback. Code generation for user nodes requires first generating the index expressions and then calling a function.

Block nodes represent a sequence of other nodes. They directly correspond to a basic block in LLVM IR.

If nodes correspond to an if-else-branch in structured programming languages. An if-node contains three members: the condition and the bodies of the "then" and "else" branches. It can be implemented by first generating the *AST expression* for the condition, and creating three basic blocks, the "then" block, the "else" block, and the continuation. The "then" and "else" blocks are used as insertion points for the corresponding body *AST node* and both terminate by an unconditional branch to the continuation. The continuation is used as the insertion point for *AST nodes* in the same block as the if-node.

A **For** node is the most complex type of node and represents a for-loop. For-nodes have five members: the name of the iterator, its starting value, a condition, the increment, and the loop body. They can be generated by creating three empty basic blocks, the header, body, and continuation, and an unconditional branch to the loop header. Two structures are inserted into the loop header: a phi-node that selects between either the initial value of the following values of the iterator, and code for the expression of the loop condition. The loop terminates by evaluating the loop condition and branching either to the loop body or the continuation. The start of the loop body is used as the insertion point for the body *AST node* and terminates by adding the increment to the iterator and unconditionally branching back to the loop header. The continuation is used as the insertion point for *AST nodes* following the for-node.

The code generation process for *AST nodes* is illustrated in figure 4.8 using a simple for loop. Instead of producing a scalar value, the result of the code generated for a polyhedral set is the side effect produced by calling the function "S" with the index of the set element passed as the argument.

4.5.2 Code-Generation for Memory Access Maps

The requirements for the generated code can be specified as follows: *enumerate all elements in a memory access map for a particular partition.* Using memory maps as defined in the application model in 4.4, the cuboid partition is described by its lower and upper bounds in both its *thread block offset* and *thread block index* and maps to a

set of multi-dimensional array indices. Interpreted literally, the generated code for a memory access map should implement a function

$$f\colon (\mathbb{Z}^3, \mathbb{Z}^3, \mathbb{Z}^3, \mathbb{Z}^3) \to \mathcal{P}(\mathbb{Z}^d), \tag{4.7}$$

with d being number of dimensions of the array. However, the thread block offset was only introduced for compatibility with the polyhedral model and contains redundant information (it is the thread block index multiplied by the thread block index). It is more convenient to specify the partition as its upper and lower bounds for the thread block index and the corresponding thread block size. Enumerating each buffer element of a read or write set individually results in very inefficient code, so the generated code is optimized to avoid this:

1. GPU applications often contain very regular memory access patterns and it is unlikely that a partition of the thread grid produces access maps that contain holes. Therefore, memory access maps are simplified by replacing them with their convex hull, resulting in simpler shapes and simpler loop nests.

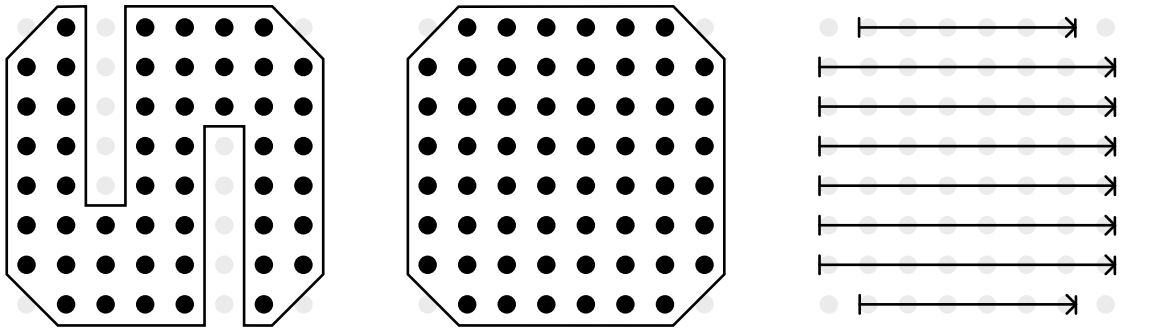
2. In the memory access maps produced by the low-level polyhedral analysis, the inner-most dimension of multi-dimensional arrays is stored in memory consecutively. This can be exploited by not enumerating individual elements of the inner-most dimension but only computing the lower and upper bound of these ranges.

3. Array reshaping is not in the scope of this project. As a consequence, multi-dimensional indices are not useful beyond enabling the polyhedral model to represent the otherwise non-affine expressions of array indices. All results should be returned as linearized accesses, using the array sizes from the application model.

Incorporating all these optimizations leads to the generated code representing functions

$$f\colon (\mathbb{Z}^3, \mathbb{Z}^3, \mathbb{Z}^3) \to \mathcal{P}(\mathbb{Z}, \mathbb{Z}), \tag{4.8}$$

expecting a partition's lower and upper bound as thread block indices, the thread block size, and returning a set of pairs with each pair describing the start and end of an interval of array elements. Figure 4.9 illustrates the different levels of optimizations of the memory access map, from its original state in the application model on the left to the simplified intervals on the right.



(a) Unoptimized access map. (b) Simplified as convex hull. (c) Rows replaced with intervals.

Figure 4.9: The different optimization levels (excluding linearization of accesses) for the memory access map before code generation.

Since only polyhedral sets and expressions can be used to generate ASTs with *isl*, the memory access maps have to be broken down into sets before generation. Fortunately, the information that is required for data dependency analysis can be extracted from the memory access map's image. When isolating the image of the map, it represents the memory access of all elements (thread blocks) in its domain. Therefore, the domain of the map should be constrained in such a way that it contains exactly a (parameterized) partition. The domain can then be dropped, translating all its constraints into equivalent constraints on the image, resulting in a set representing only the map's image.

Taking the optimized approach described above, memory access maps should be translated into code that performs the following steps:

1. Iterate through *all but the inner-most* dimension of the image of the access map, effectively enumerating all rows of the memory access.

2. For each row at index $(x_n, ..., x_2)$,

   (a) compute the minimum value min and maximum value max at position $(x_n, ..., x_2)$ as closed-form expressions.

   (b) Linearize the two indices $min_{lin} = (x_n, ..., x_2, min)$ and $max_{lin} = (x_n, ..., x_2, max)$ as closed-form expressions.

   (c) Return the pair (min_{lin}, max_{lin}).

Except for the array index linearization, the *isl* library provides the tools to create polyhedral expressions and sets required to implement the algorithm. Overall, the

algorithm suggests the following structure for the generated code: A loop nest with $n-1$ loops for the outer dimensions of the memory access, the innermost of which calls another function that computes the bounds of the memory access and, in turn, calls a user-defined callback.

Step 1 mainly involves setting up a loop nest that embeds the payload of step 2. The order in which the image of the memory access map is traversed is specified by the schedule that is passed to *isl* at code generation. Although selecting the best schedule for a particular set is generally non-trivial when optimizing loop nests, the choice easily falls on the lexicographic order for the identification of data dependency. This way, outer loops represent outer dimensions of the array (larger distances in memory) and inner loops represent the inner dimensions (smaller distances in memory). The *isl*-version of this lexicographic schedule is called an identity schedule. The loop nest can then be created using the following steps:

1. Project out inner-most dimension of the memory access map
2. Extract image (output) of outer dimensions of the map
3. Create identity schedule for the image
4. Generate code for this schedule, invoking the callback for each element (i.e. each row).

The callback function implementing step 2 contains a bit more logic. Its input are the indices of the $n-1$ out dimensions and as output it produces the (linearized) upper and lower bounds of the memory access. It is implemented using the following approach:

1. Equate the $n-1$ outer-most dimensions of the memory access map to the $n-1$ indices received as the arguments $(i_n, ..., i_2)$.
2. Compute the minimum value in the n^{th} dimension as a closed-form expression using the existing *isl* functions and generate the corresponding code.
3. Repeat for the maximum value.
4. Generate code for the closed-form expressions of the arrays dimension sizes $s_n, ..., s_1$.

5. From the multi-dimensional index $(i_n, ..., i_2, min)$ compute the linearized minimum index min_{lin} by generating code for the recursive formula:

$$o_1 = min$$
$$o_k = (i_k + o_{k-1}) * s_{k+1}$$
$$o_n = i_k + o_{k-1} = min_{lin}$$

6. Repeat the approach to compute the linearized maximum value max_{lin}.

7. Emit the result by invoking the function of form $(i64, i64)$ with the values (min_{lin}, max_{lin}).

The function containing this code is automatically inserted as the call-target of the surrounding loops. It is factored out as a separate function to avoid accidental interference between the two code segments and to avoid code duplication. The code generation relies on the LLVM infrastructure to inline this code if its heuristics indicate it as the more efficient option.

Since this code will be used as the interface to the rest of the system, it needs a flexible way to receive the data it needs and pass on its result. The resulting function that enumerates a memory access' elements is called an iterator and has the following prototype:

```
1   typedef void(*__me_itfn_t)(int64_t grid[], int64_t param[], __me_cbfn_t
        callback, void* user);
```

The partition information is passed using the `grid` argument as a fixed sized array containing lower and upper bounds of the partition and the thread block size. Kernel arguments are passed using the `param` argument, containing all integer arguments to the kernel that might be used by memory access maps. Although this list can have arbitrary sizes for each kernel, its size is known at compile time, avoiding out-of-bounds issues. The third argument is a pointer to a callback function with the prototype:

```
1   typedef void(*__me_cbfn_t)(int64_t lower, int64_t upper, void *user);
```

As explained above, it receives the access' index in all outer dimensions and the bounds of the access in the inner dimension. Additionally, it receives a pointer to user-data, that is simply passed through from the original call. These functions can then be used by the runtime system to identify data dependencies of the individual partitions of a kernel.

4.6 Runtime Library

The runtime library contains high-level, static functions that are used by all partitioned applications generated by the compiler. These functions do not need to be customized to each application and can be defined and compiled in advance. This allows their implementation in a high-level language such as C++ instead of having to generate LLVM IR for all the required code. The library provides functions for several tasks that are interconnected to varying degrees:

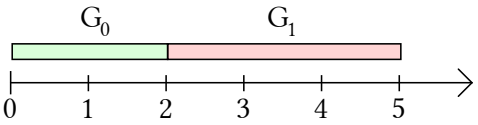
- An interface layer that enables the automatic integration of the runtime library into multi-GPU applications.
- A buffer management system that allocates, and cleans up buffers as well as managing their meta-data.
- Multi-GPU memcopy implementations that distribute and gather data between the host and GPUs.
- A buffer synchronization engine that enforces coherence between buffers before kernel launches.

4.6.1 CUDA Interface Layer

The CUDA interface layer is a wrapper that provides the functionality of the runtime system functions with an API that is compatible with the CUDA Runtime API. It follows a simple naming scheme by replacing the prefix `cuda` with `__me`, for example the multi-GPU equivalent for the `cudaMalloc` function is called `__meMalloc`. Each function in the wrapper is matched with a function of the runtime library that implements the same semantics as the single-GPU CUDA function, just in a multi-GPU context. Currently, only the most popular functions are wrapper directly: `__meGetDeviceCount`, `__meDeviceSynchronize`, `__meMalloc`, `__meFree`, `__meMemcpy`, and `__meMemcpyAsync`. Additional wrappers are added as required.

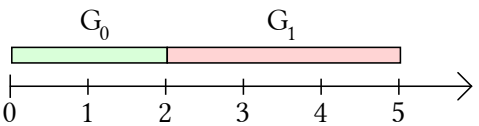
4.6.2 Buffer Management System

Executing a partitioned kernel on multiple GPUs requires some way to provide memory for data input and output to the kernel. This work uses a distributed memory model that replicates buffers on each GPU and enforces coherence on the live data set. The

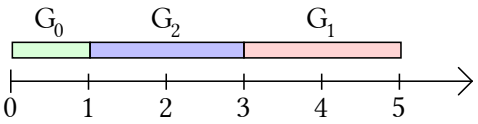


Query: $[0, 4) \longrightarrow \{ ([0, 2) - G_0), ([2, 4) - G_1) \}$

(a) Range query on a tracker covering two intervals. Two new intervals fitting the query are returned providing their exclusive owners.



Update: $([1, 3), G_2) \longrightarrow$ OK



(b) An update on a tracker containing two intervals. The existing ones are updated to provide space for the new one.

Figure 4.10: Two example use cases of the tracker component tracking the exclusive owners of all array elements. All ranges are specified as half-closed intervals.

buffer management system is tasked with allocating and freeing the corresponding buffers, as well as setting up and maintaining auxiliary data structures.

Where the original application allocates a single buffer, the partitioned application allocates one buffer per GPU and the auxiliary data structures and returns a "virtual buffer" that refers to the compound object.

Buffer allocation is simple: for each GPU available to the application, one buffer of the requested size is allocated on the corresponding GPU. As the number of GPUs does not vary throughout the runtime of the application, they can be kept in an array that is addressed using the GPU index, e.g. position 0 in the array contains a reference to the buffer of GPU 0.

The type of memory system created by GPUs in this work can best be described as a "cache-only memory architecture" (COMA) and coherence is enforced using a relaxed and simplified version of the DDM [111]. The first simplification is that the model is reduced to only the *invalid* and *exclusive* states. The second simplification is that the "caches" (GPU memory) do not communicate directly via coherence broadcasts and probes, but all cache state is managed centrally using a tracker component. For each element in the buffer, the tracker keeps track of the owner of the exclusive copy of the element. Given n GPUs, each element is in the *exclusive* state on exactly one GPU and in the *invalid* state on all others. The lack of a *shared* state immediately indicates to potential redundant transfers for data that has not been updated between.

Figure 4.10 illustrates the tracker usage with two examples, a query of the exclusive owners over a range and an update on a particular range, declaring a new exclusive

owner for this range. These two operations are the only ones required for the coherence scheme used in the partitioning compiler. Usage is not systematically biased towards either queries or updates, requiring both to exhibit acceptable performance.

A naive implementation of the tracker could be a bitmap that stores the index of the GPU that exclusively owns the element. However, this approach has prohibitive space requirements and poor performance for queries about large regions in the buffer. Instead, the tracker employs two optimizations.

The first optimization is to store all elements as non-overlapping intervals that share the same exclusive owner. For example, instead of storing the list (G_0, G_0, G_0, G_0) to indicate that the first four elements are owned by GPU 0, it stores $([0, 4), G_0)$, the half-closed interval $[0, 4)$ containing the numbers $(0, 1, 2, 3)$ and the exclusive owner G_0. To avoid fragmentation and keep storage requirements stable, neighboring intervals with the same exclusive owner must be merged into a single one representing the union of both. This optimization exploits the domain decomposition using the thread grid that is often employed for GPU applications. Threads that are neighbors in the thread grid often write to neighboring elements in the output buffer. In the worst-case scenario, each element must be represented by an interval of length one and has a storage requirement of 3 numbers (start, end, owner) instead of only one (owner). On the other hand, if the average size of intervals with the same owner is larger than three, this approach is space-efficient. For the common $1:1$ write pattern, where all threads write to consecutive addresses (after linearization), the space requirements for the tracker using intervals are 3 numbers per GPU.

The second optimization is storing the intervals in a sorted, self-balancing tree using. The start of an interval is used as the key and they are small and quick to compare, so a B-Tree has been chosen to improve performance over e.g. a red-black tree [112]. For a tracker containing n intervals and comparing against a sorted list of intervals, this keeps the time complexity of a lookup of $\mathcal{O}(n)$ but reduces the time complexity for inserts from $\mathcal{O}(n)$ (caused by shifting the following elements) down to $\mathcal{O}(n)$.

4.6.3 Multi-GPU CUDA Memcopies

Similar to GPU buffer allocations, memcopy operations involving GPU buffers are also adjusted to the multi-GPU context. In a single-GPU application, CUDA memcopies always have a single source buffer and a single destination buffer. In the partitioned application, however, any of the buffers involved might be a virtual buffer, representing several actual buffers replicated over all GPUs. There are four unique combinations

that each require a different approach for the translation from a single-GPU to a multi-GPU context.

Host-to-Device memcopies turn into a $1 : n$ data movement. The single source buffer is distributed among multiple GPUs. Static analysis can not be guaranteed to accurately predict the required data distribution for the next kernel launch involving the corresponding buffer. An alternative approach to relying on memory access patterns for the data distribution is to distribute the data in some predefined way. In the best case, the distribution pattern matches the next kernels memory access patterns and no data needs to be transferred between kernels. Otherwise, the buffers are synchronized to resolve the data dependencies on all kernels. In addition to the data transfers, the tracker needs to be updated to reflect the new distribution, where each GPU now exclusively owns the part that it received. This is the approach chosen for this work, which currently simply uses a linear distribution pattern. [1] [2]

Device-to-Host memcopies are a $n : 1$ data movement in the translated application. Many source buffers are gathered into a single host target buffer. Performing a range query on the tracker of the virtual source buffer over the full size of the buffer returns the current data distribution. Using this information, the buffer can be assembled in the host memory by a series of memcopies that collect the most up to date copies of all elements from their exclusive owners.

Device-to-Device copies turn into $n : n$ data movements. It can be implemented as a combination of the *Host-to-Device* and *Device-to-Host* strategies. First, the full data distribution is queried from the tracker. Then, the data is linearly distributed over all GPUs with this information. Single-GPU applications typically avoid these copies since it simply creates a copy of data that is already existent on the GPU. For this reason, this type of CUDA memcopy has not been implemented yet.

Host-to-Host data movements are left unmodified as $1 : 1$ data movements.

4.6.4 Virtual Buffer Coherence and Synchronization

As mentioned earlier, the GPUs of an automatically partitioned CUDA application essentially create a COMA system and use a simplified and relaxed version of the DDM coherence protocol. The simplification is the existence of only the *exclusive* and *invalid*

[1] With additional engineering efforts, memory access patterns from static analysis could be as the best-case distribution in cases where they are proven correct.

[2] An earlier iteration of the runtime deferred Host-to-Device transfers until the start of the kernel, where memory access patterns are known. However, Modifying the host buffer between the memcopy and the kernel start introduces Write-After-Read conflicts that can only be avoided by either extensive static analysis or an additional copy of the host buffer.

states. The relaxation is the requirement to enforce coherence only at specific points in the program, immediately after a kernel has finished execution on all partitions. This matches the coherence guarantees made by the CUDA execution model [14] and is completely transparent for applications adhering to these guarantees and not relying on architecture-specific exceptions to it. In other words, enforcing coherence only after kernel launches is indistinguishable from enforcing it at every point in the application.

Algorithm 3 Algorithm for the synchronization of a single virtual buffer for a single partition.

Input: *kernel* - a CUDA kernel
buffer - a buffer of *kernel*
GPU - the current GPU
Output: none

```
pattern ← read set for GPU for buffer of kernel
intervals ← query tracker of buffer with pattern
for each (start, end, owner) in intervals do
    if owner ≠ GPU then
        size ← end − start
        obuf ← instance of buffer of owner
        gbuf ← instance of buffer of GPU
        cudaMemcopy(gbuf + start, obuf + start, size, DeviceToDevice)
    end if
end for
```

However, for a kernel to execute successfully and compute the correct result, the buffers on each GPU need to contain valid (up-to-date) copies of all elements in the read set of the GPU's partition. The process of identifying this data and communicating it between the different GPUs is referred to as *buffer synchronization*. Since the tracker of a virtual buffer does not support a shared state for any elements, all data that is not exclusively owned by a specific GPU is considered out-of-date. Using the code generated for the read accesses of the buffers of a kernel, this results in the algorithm depicted in 3 for the synchronization of a single virtual buffer for a single GPU. The execution of this algorithm is repeated for each partition for each virtual buffer used in a kernel. Since no exclusively owned data is overwritten, none of the memory transfers have any dependencies between them and they can all be executed concurrently.

Updating the tracker after kernel execution is similar to buffer synchronization, but simpler. Tracker state is not queried at all but simply overwritten using the write accesses computed using the generated code. The write accesses to a buffer from a particular GPU are simply traversed and the tracker is updated to set the owner for

Algorithm 4 Algorithm that updates the coherence information in the tracker of a buffer for a single partition, after kernel execution.

Input: *kernel* - a CUDA kernel
buffer - a buffer of *kernel*
GPU - the current GPU
Output: none

pattern ← write set for *GPU* for *buffer* of *kernel*
for each (*start*, *end*) **in** *pattern* **do**
    update tracker of *buffer* with (*start*, *end*, *GPU*)
end for

each interval to the GPU in question. Algorithm 4 shows this algorithm. The buffer updates are also repeated for each buffer for each partition.

4.7 Host Code Transformations

The host transformations are what ultimately connects the other steps and produces the partitioned application. The translation from single-GPU to multi-GPU applications is divided into two smaller steps that can be performed independent of each other:

1. Substitute all calls to the regular CUDA Runtime API with their wrappers from the runtime system.

2. Replace all CUDA kernel launches with code for partitioning, buffer synchronization, and kernel orchestration.

Due to their different levels of complexity, they are implemented in different stages of the compilation using different abstractions.

The first transformation, the replacement of calls to the CUDA Runtime API with the wrappers provided by the static runtime library, is implemented on the IR level. All wrappers in the runtime library have been designed to have prototypes that are compatible with that of their CUDA Runtime API counterparts. The replacement logic in LLVM can then be implemented with a map from CUDA API calls to their substitute and following this recipe:

1. For each function in the CUDA Runtime API, perform a look-up in the translation unit's symbol table.

2. If the symbol is not found, continue with the next API function.

3. If the symbol is found and

   (a) a substitute *does* exist: iterate over all its uses and if the use is a function call, replace the called function (the CUDA Runtime API function) with its substitute from the partitioning runtime library.

   (b) a substitute *does not* exist: check if any of the uses is a function call, abort compilation with an error message about the unsupported CUDA Runtime API call.

Aborting the compilation when unsupported CUDA API calls are encountered is necessary to guarantee correct semantics of the resulting partitioned application. Any CUDA API calls that can not be translated keep their original semantics and expect device buffer reference to directly point into device memory instead of the compound virtual buffer object. This can lead to undefined behavior and data corruption.

The second transformation, implementing the partitioning, buffer synchronization, and kernel orchestration, is complex enough to warrant its implementation in a high-level language. While this transformation seems to be a good fit for some form of template instantiation in LLVM IR, there is an impedance mismatch because LLVM IR does not have a semantically equivalent representation of a CUDA kernel launch. It is lowered into a set of setup instructions and function calls that requires a complicated matching heuristic to identify, as described in section 3.1.2. Ideally, the transformation would be implemented as a source-to-source translation or as an AST-level transformation. However, the LLVM framework in general and Clang (as the C-family front-end) are not intended for source-to-source translation. Although frameworks for these purposes exist, the learning curve and engineering efforts involved with the integration of a new technology into the toolchain were deemed too high [113, 114, 115]. Instead, all transformations are implemented using regular expressions substitutions before the application is passed to the compiler [116]. This implements a poor man's source-to-source transformation that does not require a CUDA capable compiler frontend but is unaware of the semantic structure of the program.

Lua has been chosen as the implementation language due to its availability, expressiveness, and string processing facilities (including built-in support for a flavor of regular expressions) [117]. The translator written in lua expects the application model and a CUDA file as input and then performs the substitutions in two steps.

First, type definitions and prototypes are inserted at the top of the translation unit. This prepares the translation unit so that external functions from the runtime

system and generated code are recognized by the compiler. Specifically, the following information is inserted:

- A definition for the data type of partitioning information.
- Definitions for function pointers that represent the code generated for memory access patterns as described in section 4.5 and the callbacks used to emit intervals from the memory access.
- Prototypes for the functions generated for all memory access patterns from section 4.5.
- Prototypes for the runtime library functions for partitioning, buffer-distribution and -collection, synchronization, and updates of coherence information.
- Prototypes for the partitioned kernels created in section 4.3.

The second step is the transformation of CUDA kernels. For this, all CUDA kernel launches are located by using the regular expression defined by `launch`:

```
launch = id ws "<<<" ws config ws ">>>" ws "(" ws params ws ")" ws ";"
id = [_a-zA-Z][_a-zA-Z0-9]*
ws = [ \t\f\n]*
config = .* / ws ">>>"
params = @balanced{"(", ")"}
```

where `"s"` matches the string literal s, `[a-xyz]` matches a single letter contained in the character class $a...x, y, z$, `R*` matches zero or more occurrences of R, `R / S` matches R followed by S without consuming S, and `@balanced{"(", ")"}` matches anything between a balanced number of occurrences of (and). This expression allows matching CUDA kernel launches with reasonable accuracy. However, using regular expressions instead of an actual parser is vulnerable to malicious input since it ignores comments and some other structures available in the language. Each match for this pattern is then replaced with code that generated by instantiating the template shown in algorithm 4.11.

The first line extracts all scalar arguments used in the kernel launch. It identifies them by comparing them to the arguments stored in the application model by position. The resulting array `params` contains all kernel arguments that influence the kernel's read or write accesses. Then, the thread grid of this kernel is split into partitions, using the kernel name (which contains the predicted best partitioning based on its read accesses), the grid configuration, the scalar kernel arguments, and the number of GPUs as input.

```
1  params = [arg in args | arg is parameter]
2  partitions = model.kernel.partitioning(grid, params, GPUs)
3
4  for gpu in GPUs:
5    partition = partitions[gpu]
6    reads = [arg in args | arg is array and arg is read]
7    for array in reads:
8      pattern = pattern_for(model.kernel, array)
9      buffer_synchronize(array, pattern, partition, params)
10 all_devs_synchronize()
11
12 for gpu in GPUs:
13   partition = partitions[gpu]
14   newGrid = partition.max - partition.min
15   newArgs = []
16   for arg in args:
17     if arg is array:
18       newArgs += [instance_for_gpu(arg, gpu)]
19     else:
20       newArgs += [arg]
21   partitioned_kernel<<<newGrid, blocks>>>(newArgs, partition)
22
23 for gpu in GPUs:
24   partition = partitions[gpu]
25   writes = [arg in args | arg is array and arg is write]
26   for array in writes:
27     pattern = pattern_for(model.kernel, array)
28     buffer_update(array, pattern, partition, params)
```

Figure 4.11: Pseudo code of the kernel launch replacement that is inserted by the source-to-source rewriter.

Next is the buffer synchronization that is performed by the loop in line 4. For each GPU, it retrieves a list of arrays that are read by at least some threads in the kernel. It then retrieves the code generated for the read access map of the array by this particular kernel. A reference to the virtual buffer, the memory access map, the GPUs partition, as well as the kernel scalar arguments are then passed to the runtime function implementing the buffer synchronization. It uses this information to distribute all elements among the GPUs that are required by them as input. After the loop, a device synchronization call waits for all GPUs to finish their pending operations to ensure all memory is up to date before launching any kernels.

After synchronization, the partitioned kernel can be launched on each GPU in the loop starting at line 12. The first step is to compute an updated thread grid size, to limit the kernel's execution to only the threads of the current GPU's partition. Next, an updated list of arguments is created using the application model. Scalar arguments

are copied unmodified. Array arguments are replaced with virtual buffers. The buffer instance located on the current GPU is retrieved and its pointer copied to the argument list. Last, the partitioned kernel is launched using the new grid configuration, new list of arguments, and the partition information that is required by the relocatable kernel's index calculations as described in section 4.3. No device synchronization is required after launching the kernels to match the semantics of the regular CUDA kernel launch, which is an asynchronous call as well.

The last major operation in the replacement for kernel launches is the update of coherence info, which is performed in the loop starting at line 23. The program semantics are unaffected by whether the update is performed directly before the kernel launch or directly after it. But since kernel launches are launched asynchronous and most GPU applications simply wait on the kernel to finish, this is an opportunity to hide the latency from the update in the execution time of CUDA kernel launches. The coherence information in the tracker is updated using the write access maps in the application model for the corresponding kernel. It follows the same recipe as the buffer synchronization but retrieves write maps instead of read maps and calls the runtime library function that updates the tracker. The injectivity of write maps (by the polyhedral analysis in section 4.4) ensures that the order in which the write maps are traversed does not influence the resulting distribution of exclusive owners.

The loop selecting and processing kernel arrays and arguments are unrolled during the template instantiation. This produces simpler code that is easier to debug and reduces the depth of the loop nests created by this code. The pseudo-code in algorithm 4.11 contains the corresponding regions as loops for clarity.

The host code transformation has now combined the individual pieces of the partitioning compiler. By first analyzing the code to produce the application model, then translating both host and device code, and generating the code for memory access maps, a partitioned binary is been created that utilizes all GPUs in the system to jointly compute the solution of the program.

CHAPTER 5

Prototype Evaluation

This chapter evaluates the concept and prototype implementation for the automatically partitioning compiler presented in this work and discusses the results. It focuses on two aspects of the performance of the produced application binaries: measured performance of the generated binaries, and overhead introduced as a side-effect of the transformation. The evaluation will be based on three proxy applications with memory access patterns that can be successfully extracted by static analysis, thus enabling the generation of and multi-GPU binaries.

This chapter first details the evaluation methodology, which includes detailed descriptions of the proxy applications, the test system used, and how execution times are measured. Then, the speedup achieved with multiple GPUs is calculated and explanations for the behavior of the individual proxy applications are discussed. Next, the overheads introduced by the individual parts of the runtime system are presented and discussed. These overheads can be assigned to both essential and accidental complexity, and their individual contributions will be examined. Lastly, the chapter concludes with a summary of the findings and their consequences for the prototype implementation and the underlying concept.

5.1 Functionality

This section evaluates the functionality of the partitioning compiler. Functionality includes both the range of applications covered by the compiler as well as the correctness of applications that can be optimized by it.

The range of applications supported by the compiler is limited due to a variety of reasons. The first limitation stems from the approach itself: polyhedral compilation does not attempt to be a model that can represent arbitrary applications. Polyhedral compilation as understood by the "Polly" project of within LLVM is limited to Static Control Parts (SCoPs), which are simple regions containing only well-structured control flow and no unknown side effects [118]. Applications expressed in high-level languages, CUDA in this case, are generally translated into code containing well-structured control flow or can be coerced into such using the available transformation passes [19]. CUDA programs also tend to be very structured and rarely contain instructions with unknown side effects. However, some constructs that are often used in GPU kernels cannot be represented in the polyhedral model:

- Non-affine memory accesses. Examples of this are tree-based reductions with exponential array indices and graph computations with indirect memory accesses [119, 120].
- Control flow dependent on the state of the GPU memory, if the content of the memory cannot be represented in the polyhedral model [121]. This is the case for complex convergence criteria, as used for example in different string matching algorithms [122] or simply loops with non-affine bounds.

Issues with these and other constructs are typically side-stepped in polyhedral compilation by focusing on sub-regions that can be fully represented in the polyhedral model. However, for the automatic partitioning approach in this work, the control flow of the full kernel must be modeled. There are efforts to relax these constraints by over-approximating the problematic regions, in the worst case with loops from zero to infinity [123]. This simplification severely limits the usefulness of the memory access patterns used in this work and cannot be adopted. Many algorithms of GPU applications rely on either indirect memory accesses and loops with non-affine bounds, or introduce them with various optimizations.

The analysis used in this work has been repurposed from another research project and is designed as a general purpose analysis instead of bespoke for GPU code. It requires external hints about the semantics of the CUDA programming model to successfully create a model of the application. An example of this is a common formula used in CUDA that computes the global index of a given thread by calculating `blockIdx.x * blockDim.x + threadIdx.x`, which results in non-affine accesses that can not be handled by polyhedral analysis without help. The issue is resolved by creating a parameter for this expression that can later be located and replaced with actual

values when a memory access pattern is resolved. Although the solution works for this very common special case, it requires the global thread index to be computed in this exact way, mixing dimensions or any other non-trivial modification breaks the analysis. Additionally, the analysis itself is experimental and does not cover all possible edge cases, resulting in crashes due to violations of invariants [99]. Overall, only a fraction of applications that might generally be amenable to polyhedral analysis can be represented by the application model used in this work.

For this reason, the evaluation is limited to a comparatively small set of applications. They are minimal implementations of the workloads they represent and have been written in such a way as to avoid any of the issues mentioned above. Naive implementations of the algorithms themselves do not contain indirect memory accesses or non-affine loops and only optimizations that are possible within these constraints have been implemented. They cover both iterative and non-iterative applications, host-side buffer swapping and a variety of different kinds of memory access patterns. The compiler can analyze these applications and generate partitioned binaries for them that keep the original semantics intact, producing the correct results for all of them in all tested configurations.

5.2 Evaluation Methodology

For credibility and reproducibility, the guidelines this evaluation is based on must be established. This includes not only the applications that have been used to perform the evaluation but also the system that the applications were executed on, as well as which and how execution times have been measured.

The selection of workloads the approach can be tested on is limited to those that exhibit memory access patterns that can be accurately predicted using our polyhedral model. The three benchmarks chosen are taken from the computational dwarfs identified by Berkeley in [87]. They are a subset of the applications used in the memory access pattern analysis in chapter 3 but exist in different development stages for both analyses and are not necessarily identical. In the context of this analysis, only computation and communication directly or indirectly related to the GPU workload are explained and considered.

5.2.1 2mm

The *2mm* benchmark computes two chained matrix multiplications that reuse the result of the first computation as input for the second: $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}, \boldsymbol{E} = \boldsymbol{C} \cdot \boldsymbol{D}$. Both multiplications are executed in individual kernel launches of the same kernel code. The problem size n of this application to the side length of the quadratic matrices of size $n \times n$. The matrix multiplication code used here is a fairly simple implementation with the only optimization being tiling in shared memory with a tile-size of 32. The sequence of actions performed by the application is the following:

1. Copy the host-initialized matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ from host memory to GPU.
2. Execute kernel $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ on the GPU.
3. Copy the host-initialized matrix $\boldsymbol{D}$ from host-memory to the GPU.
4. Execute kernel $\boldsymbol{E} = \boldsymbol{C} \cdot \boldsymbol{D}$.
5. Copy the GPU-computed matrix $\boldsymbol{E}$ from the GPU to the host-memory.

5.2.2 Hotspot

The *hotspot* benchmark is a heat relaxation, implemented as a 5-point stencil iteration on a static grid with a fixed time-step. The grid has size $n \times n$, with n being the problem size. Each iteration is performed in a separate kernel launch and executes the same kernel code, which is a naive 2D-Jacobi iteration. In particular, no adaptive mesh refinement approach is used, which necessarily introduces dynamic buffers and prohibits a program analysis using the polyhedral model [124].

1. Copy host-initialized data to a 2D-buffer $\boldsymbol{A}$.
2. Compute the result of the Jacobi iteration from the input $\boldsymbol{A}$ to a buffer $\boldsymbol{B}$.
3. Switch device $\boldsymbol{A}$ and $\boldsymbol{B}$.
4. Repeat steps 2 and 3 a total of 96 times.
5. Copy result from buffer $\boldsymbol{A}$ (after switch in step 3) back to host memory.

5.2.3 N-body

The *n-body* benchmark is a gravitational simulation of point masses with a fixed time-step. The problem size n refers to the number of bodies in the simulation of this benchmark. Each body is initialized with a random position, mass, and velocity, and the simulation then updates velocities and positions using a semi-implicit euler integrator. First, velocities are updated in one kernel by calculating the forces of each body onto every other body. Then, another kernel updates the positions of all bodies by integrating the velocities over a fixed time-step. Most optimizations typically applied to n-body simulations introduce dynamic memory accesses via indirections, making them inappropriate for polyhedral analysis. In particular, no clustering optimizations are applied, where bodies in close proximity are grouped into a cluster. Within each cluster, the contributions of each body are computed accurately, but the contributions of bodies in other clusters are simplified as a single point-mass per cluster. The simulation performs the following steps:

1. Copy the host-initialized positions, masses, and velocities from host-memory to the GPU.
2. Launch kernel A to update velocities for all bodies.
3. Launch kernel B to update positions for all bodies using the update velocities (semi-implicit euler integration).
4. Repeat steps 2 and 3 a total of 96 times.
5. Copy positions and velocities from the GPU back to host-memory.

5.2.4 Test Setup and Time Measurement

The total runtime of an application is the result of many different factors, including hardware, software, and environmental conditions to different degrees. While it is impossible to fully specify all possible conditions that could influence the runtime, specifying the major factors increases reproducibility significantly.

The system used for the tests is a Supermicro X10DRG equipped with two Intel Xeon E5-2667 Processors, eight NVIDIA K80 GPUs, and 256GiB of DDR4 RAM running at 2133MHz. The operating system is CentOS Linux release 7.4.1708. As the benchmark applications do not contain CPU intensive operations in the relevant code paths, the CPU scaling governor was kept in its default value `powersafe`.

All applications have been compiled twice, once with an unmodified CUDA compiler for single-GPU reference and once with the partitioning compiler to measure execution times and overhead. Both variants have been compiled with the same LLVM/Clang compiler, a development build of version 7.0.0 of January 8, 2018 and the CUDA SDK in version 8.0. The single-GPU binaries have been compiled with this combination as is, for GPU architectures `sm_30` and with the optimization flag `-O3`. The multi-GPU binaries have been compiled with the same compiler and settings, but with the automatic partitioning passes added to the toolchain via an external LLVM module.

Each benchmark was executed at least 16 times in each configuration and all execution times are provided as mean values with the corresponding standard deviation.

Application runtime can be measured in several ways. This analysis exclusively uses wall-clock time instead of CPU time. The applications used here simply go to sleep when calculations are performed on the GPU, which is not captured by CPU time. Additionally, since this work focuses on time spent with GPU-related activities, only these activities are captured. This primarily includes memory copies between the host and the device and the execution of CUDA kernels. Driver initialization, host-buffer initialization, and result verification are not counted towards the runtime. Driver initialization time can be up to several seconds and is a function of the CUDA driver, number of GPUs and amount of system memory. The benchmark applications are written in such a way that all activities that are counted towards the runtime are executed back to back without any time-consuming operations in-between (in particular the activities mentioned before).

Time is measured using the `gettimeofday` call, which has a mean resolution of less than 1 μs and a worst-case resolution of about 700 μs on the test system, measured using the approach from Finney [125]. The time of each runtime phase has been measured separately and added up to the total runtime, which introduces an average error of less than a microsecond.

5.3 Speedup of Partitioned Applications

This section examines the raw achieved speedup for each application on the test system with different numbers of GPUs and discusses the results based on the properties of the applications. The speedup S is calculated using the standard formula from Amdahl's Law as

$$S = \frac{T_s}{T_p}, \tag{5.1}$$

the ratio between the runtime of the unpartitioned binary T_s and the runtime of of partitioned binary T_p [126]. Both runtimes are measured based on the time spent in GPU activities as detailed in section 5.2. As an additional indicator of the scaling behavior, its efficiency η is calculated from the speedup S and the number of GPUs (partitions) P for that speedup using the formula

$$\eta = \frac{S}{P}. \tag{5.2}$$

The efficiency describes the contribution to the speedup per additional GPU. As Amdahl's Law demonstrates with the inevitable sequential parts of any application, the marginal contribution per GPU should shrink with each additional GPU.

In addition to the speedup and efficiency, inherent properties are discussed to explain the results observed experimentally. Two properties regarding the scaling behavior of the amount of both work and space required for an algorithm are discussed [127]:

- The **computational complexity** is an abstract measure for the asymptotic behavior of the execution time as a function of the algorithms problem size. It does not intend to provide a specific estimate of the expected runtime but captures only how the relative runtime for different problem sizes behaves.
- The **space complexity** is a related measure that describes the relative amounts of memory required to solve the algorithm for different problem sizes.

Both types of complexities are defined as a function of the problem size n as defined in section 5.2 and the number of iterations I if appropriate.

Both the computational and space complexity are properties of the underlying algorithm of the unpartitioned application. This does not always directly translate to the corresponding requirements for the partitioned application, in particular considering that additional communication to synchronize data between GPUs might be necessary. Therefore, for the partitioned application, the **computational complexity per partition** and the **communication complexity per partition** are calculated analogous to the two properties above, as a function of the problem size n, the number of GPUs (partitions) P and the number of iterations I where appropriate. Note that the communication complexity is *directly dependent on the partitioning* used to distribute the application over multiple GPUs and must be mentioned when discussing the communication complexity. Communication complexity also does not capture

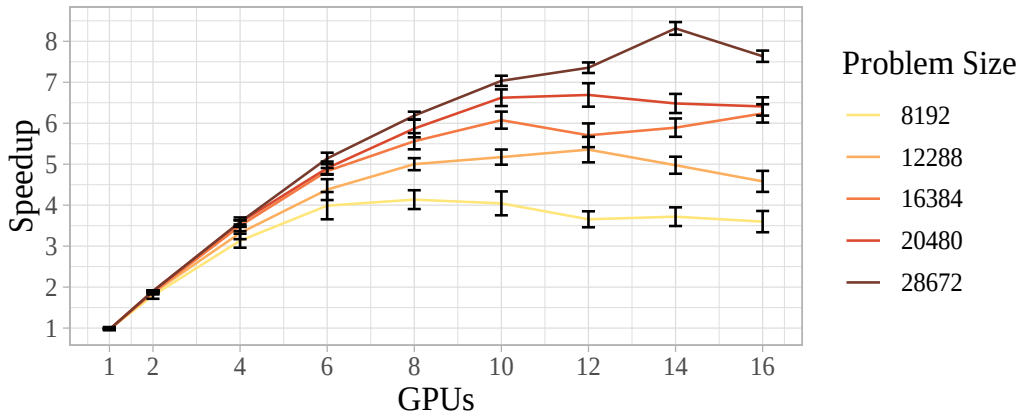


Figure 5.1: Speedup of the 2mm workload for up to 16 GPUs.

additional latencies that can arise from data being laid out sub-optimally, prohibiting batch transfers and instead requiring many small transfers.

Figure 5.1 presents the speedups measured for the *2mm* workload. The highest speedup achieved is $S = 8.31$ (with a standard deviation $\sigma = 0.155$) with 14 GPUs, which corresponds to an efficiency $\eta = 59.4$ %. The highest scaling efficiency achieved is $\eta = 95.2\%$ ($\sigma = 0.86\%$) with two GPUs, resulting in a speedup of $S = 1.90$. Both maximum values are observed for the largest problem size of $n = 28672$.

While the computational complexity of the chained square matrix multiplication is $\mathcal{O}(n^3)$, the space complexity is only $\mathcal{O}(n^2)$. Computation can be perfectly distributed between multiple GPUs independent of the partitioning and results simply in $\mathcal{O}\left(\frac{n^3}{P}\right)$. The partitioning heuristic in the compiler suggests a linear distribution across the Y-axis to allow batch transfers or contiguous chunks in memory. As a result, the left-hand matrix in the first multiplication $\mathbf{A}$ is laid out correctly across the GPUs using the linear distribution for host-to-device memcopies. Similarly, the intermediate matrix used on the left-hand side of the second multiplication $\boldsymbol{C}$ and each partition computes the partial result it later reuses. On the other hand, the linear data distribution does not correctly layout the right-hand matrices $\boldsymbol{B}$ and $\boldsymbol{D}$ which are required in full by each GPU, resulting in some synchronization between the devices. The communication complexity per partition, therefore, results in $\mathcal{O}(n^2)$. This large communication complexity independent of the number of partitions limits the speedups that can be achieved despite the massive amounts of computation.

The speedup achieved on the test system for the *hotspot* benchmark is shown in figure 5.2. Similar to the 2mm benchmark, the highest speedup is achieved with 14 GPUs with $S = 7.59$ ($\sigma = 0.35$), with an efficiency of $\eta = 54.2\%$ that is slightly

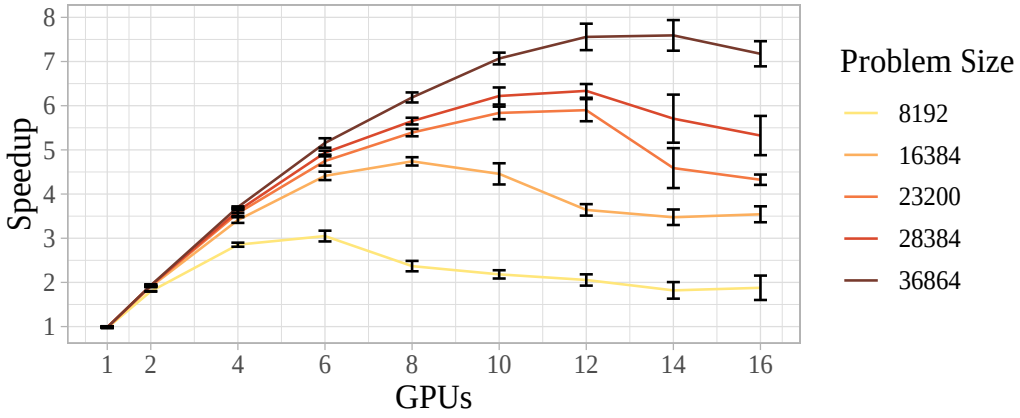


Figure 5.2: Speedup of the hotspot workload for up to 16 GPUs.

lower than that of the highest speedup of 2mm. The highest efficiency of $\eta = 96.6\%$ ($\sigma = 1.2\%$) is also measured when using two GPUs with a speedup of $S = 1.93$ and is slightly higher than the highest efficiency of the 2mm benchmark.

The computational complexity of the hotspot benchmark is $\mathcal{O}(n^2 I)$, while the space complexity is $\mathcal{O}(n^2)$, indicating the 2-dimensional grid and a constant amount of work per element in the grid per iteration. When partitioned across multiple GPUs, work can again be perfectly distributed independent of the partitioning and computational complexity per partition is $\mathcal{O}\left(\frac{n^2 I}{P}\right)$. Similarly to the 2mm benchmark, the partitioning heuristic suggests a linear distribution across the Y-axis to allow few large transfers of chunks that are contiguous in memory. The initial data distribution as well as the write-pattern in every iteration match the read pattern almost exactly and only require to synchronize a small static amount before each partition, resulting in a communication complexity of $\mathcal{O}\left(\frac{n^2}{P} + nI\right)$. Although the small synchronization transfers between iterations introduce a sequential lag that limits parallelization, increasing the number of iterations causes the computation to grow faster than the communication, creating a potential for a speedup, as illustrated by figure 5.2.

Figure 5.3 shows the speedup measured on the test system with the *n-body* benchmark for up to 16 GPUs. The highest speedup of $S = 12.07$ ($\sigma = 0.02$) was measured with 16 GPUs and results in an unusually high efficiency of $\eta = 75.4\%$. The highest efficiency of $\eta = 99.1\%$ ($\sigma = 0.1\%$) was observed with when using two GPUs and has a speedup of $S = 1.98$.

The computational complexity of the unpartitioned n-body benchmark is $\mathcal{O}(n^2 I)$ and its space complexity is $\mathcal{O}(n)$. As with the previous two benchmarks, work can be shared perfectly between GPUs, resulting in a computational complexity per partition

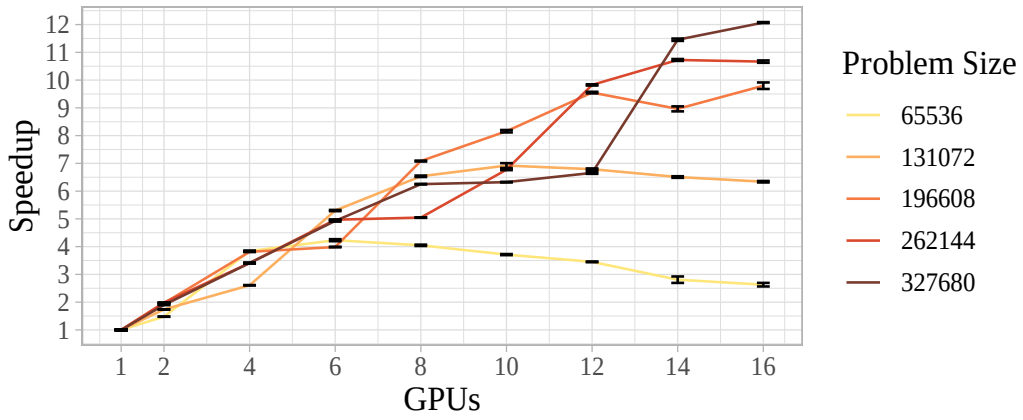


Figure 5.3: Speedup of the n-body workload for up to 16 GPUs.

of $\mathcal{O}(\frac{n^2 I}{P})$. As the main data structures in this simulation are flat arrays of particles, the partitioning heuristic suggests a linear distribution across the X-axis. During any iteration, each partition needs the full list of positions and masses and a fraction of the list of velocities. However, the tracker is implemented in a way that does not allow shared unmodified copies of data on multiple GPUs, requiring the positions and masses to be fully synchronized in each iteration. The array containing velocity values, however, requires only a single transfer during the first host-to-device memcopy and then stays local to its GPU. This results in a communication complexity per partition of $\mathcal{O}(nI)$. Although the n-body benchmark has the same computational and communication complexities per partition as the hotspot benchmark, the very small space complexity allows scaling to much larger problem sizes. Large problem sizes exploit the higher order of the computational complexity per partition and so enable higher speedups, as is visible from figure 5.3.

5.4 Partitioning Overhead Analysis

Although the analysis of the speedup of the produced binaries is the ground truth of the effectiveness of the partitioning compiler, further investigation can provide valuable insight into its potential and limitations. Due to the extensive transformations applied to the input application, the specific semantics of many details of the original application change. In particular, transfers between host memory and GPUs in the original application do not necessarily exactly match those in the partitioned one, as most likely more additional transfers need to be introduced. Although the code that was generated to enumerate memory access patterns for the dependencies of specific

partitions was designed to be lightweight, nonetheless it might incur overhead that limits scalability. For this reason, this chapter investigates these overheads that have been introduced by the partitioning process. This analysis focuses on the overheads introduced by the data transfers involving GPUs and the enumeration of memory access patterns for data dependencies.

Measuring these overheads is not trivial as many of the activities on the host and those on the GPU overlap to exploit concurrency. As a workaround, the runtime is equipped with a mode switch that puts the application into one of the three following modes:

1. In **regular** mode, the default mode, all access patterns are enumerated and memory is transferred normally. The partitioned application behaves as expected.

2. In **no-transfers** mode, no CUDA memcopies are issued. However, data dependencies are computed using the access pattern enumerators.

3. The last mode **no-patterns** disables enumeration of memory access patterns altogether. This mode implies no-transfers, as no data dependencies are computed and essentially disables all inner loops in the runtime system.

None of the proxy applications contain data-dependent control flow. While disabling transfers does change the result that is computed, the algorithm itself is unaffected by it. Identical to the speedup analysis, the benchmarks have been executed in each configuration and each mode for at least 16 times and all results are presented by their average and standard deviation. For clarity, only the smallest and the largest problem sizes are examined, with one plot for each. The fraction of the runtime that is taken up by communication c is calculated as

$$c = \frac{T_{regular} - T_{notransfers}}{T_{regular}}, \tag{5.3}$$

and the fraction of the runtime spent enumerating memory access patterns e is calculated as

$$e = \frac{T_{notransfers} - T_{nopatterns}}{T_{regular}}. \tag{5.4}$$

These calculations are bound to be imperfect representations of their corresponding times due to the overlap of CPU and GPU activities. But since they are collected from binaries executing one algorithm from the same binary, additional compiler

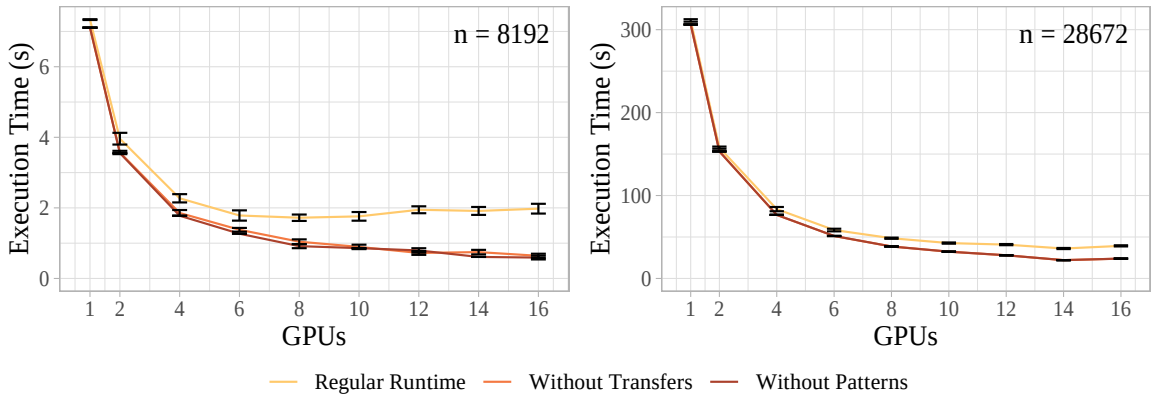


Figure 5.4: Composition of the application runtime by type of activity for the 2mm-workload.

optimizations are avoided, producing a realistic insight into the performance of the runtime system.

The *2mm* benchmark has a fairly high communicational complexity per partition of $\mathcal{O}(n^2)$, as explained in section 5.3. The large overheads of these transfers were expected to severely limit the application's scalability, which is confirmed by the results shown in figure 5.4. For both problem sizes, the runtime of modes no-transfers and no-patterns continues to decrease for larger growing numbers of GPUs, while the gap between them and the runtime in regular mode increases. This suggests the growing portion of the runtime is taken up by data transfers, which is clearly visible for both problem sizes. The highest largest portion of time spent transferring data is $c = 67.6\%$ ($\sigma = 7.6\%$), completely negating any time savings from distributed computation. The largest fraction of time spend in pattern enumeration is $e = 7.3\%$ ($\sigma = 5.1\%$) and is fairly high, but has a standard deviation in the same order of magnitude. The corresponding values for the large problem size are $c = 39\%$ ($\sigma = 1.8\%$) and $e < 1\%$.

In general, the *hotspot* benchmark behaves similarly to the 2mm benchmark: for small problem sizes, communication grows large enough to destroy any reduction in runtime by work distribution and communication is still a significant factor even for the large problem size. However, for the small problem size $n = 8192$, the fraction spent transferring data is even more exaggerated than with 2mm. The maximum value for the communication fraction is $c = 78.1\%$ ($\sigma = 14.1\%$), more than 10% higher than 2mm. The maximum time spent enumerating patterns is $e = 6.8\%$ ($\sigma = 2.6\%$), which is comparable to 2mm. For large problem sizes, these values are $c = 41.2\%$ ($\sigma = 3.9\%$) and $e = 3.6\%$ ($\sigma = 0.2\%$).

The *n-body* benchmark behaves significantly differently than the other two bench-

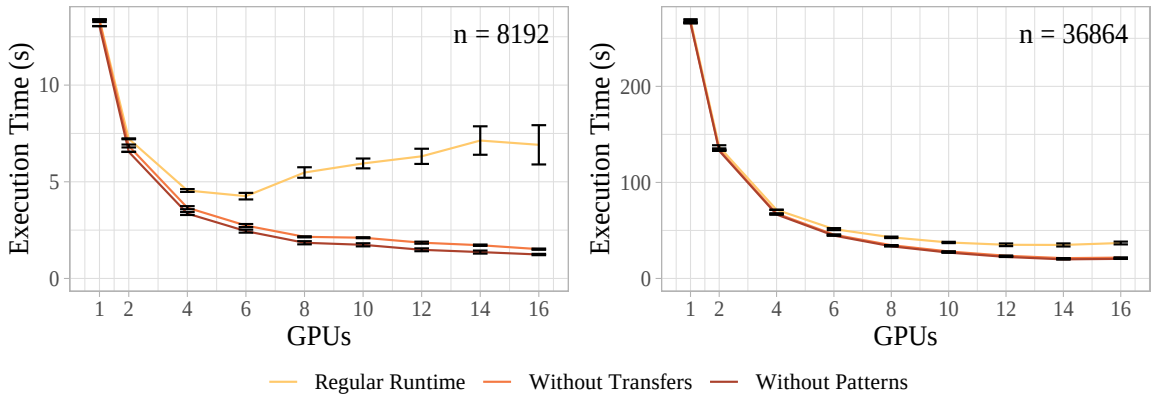


Figure 5.5: Composition of the application runtime by type of activity for the hotspot-workload.

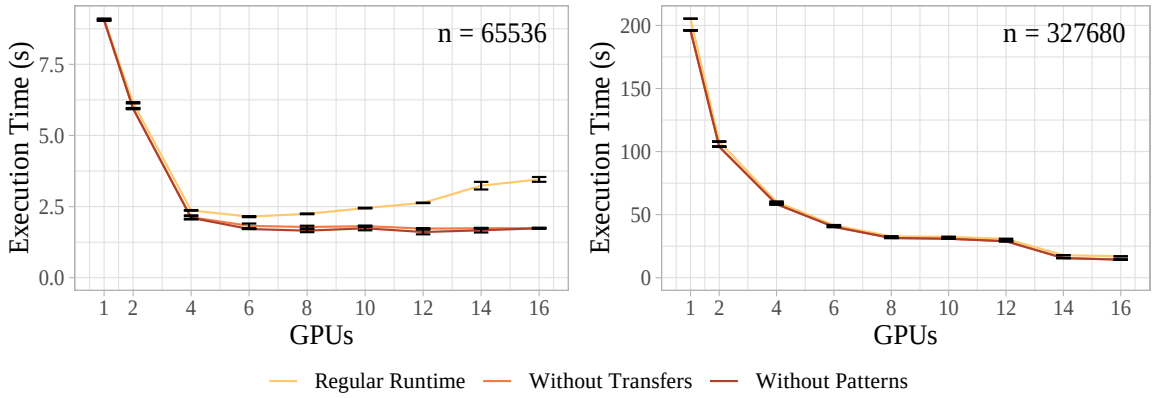


Figure 5.6: Composition of the application runtime by type of activity for the n-body workload.

marks, although transfers still ruin scalability for small problem sizes. However, communication takes up at most $c = 49.6\%$ ($\sigma = 2.5\%$) of the total runtime, far less than for 2mm or hotspot. Pattern enumeration is also far less pronounced with a maximum value of $e = 5.7\%$ ($\sigma = 3.2\%$). For the large problem size, the highest fraction of time spent transferring data is $c = 16\%$ ($\sigma = 0.2\%$), and patterns a negligible taking up less than 1% in the worst case. This behavior matches the predictions made in section 5.3 about the communication behavior of the application.

The necessity to communicate is an inherent problem when distributing computation and while optimized algorithms can limit the amount of communication required, it can rarely be avoided altogether. Overheads like the enumeration of data dependencies, however, are accidental complexity introduced only by the particular approach chosen to partition applications. This kind of overhead should be avoided or kept insignificant

wherever possible. The analysis performed in this section shows that the accidental overhead introduced by the runtime system is always less than 10% of the total runtime and in many cases negligible for larger problem sizes. This is an acceptable result and validates this aspect of the automatic partitioning solution.

A different, but not insignificant type of overhead is the compile-time overhead. The automatic partitioning requires certain repeated execution of certain steps in the compilation pipeline and adds significant static analysis and code generation. On the benchmarks evaluated here, using the two-socket machine described earlier, compilation times have been increased by a factor of $1.9x$ to $2.2x$. While a factor of 2 is a large increase, this cost is amortized over multiple runs of the application and is appropriate considering the extensive restructuring of the application.

CHAPTER 6

Discussion

This chapter critically reflects on the research presented in this work. It is put into context using related work from past research as well as the current state of the art, providing an estimate for the impact this work might have on the scientific community. The comparison with related work is then followed by discussing future work that could not be explored as part of this research project but is highly interesting and could improve its results.

6.1 Related Work

Although GPGPU computing is a comparatively recent field of computer architecture, it has been the subject of considerable research. Many of these efforts are related to the analyses, ideas, and techniques presented in this work, which this chapter presents and discusses. It mirrors the general structure of this work as a whole. First, research focusing on the patterns and effects of communication between multiple GPUs is presented. Both the analysis of dedicated multi-GPU benchmark suites as well as an analysis of single-GPU benchmarks re-framed in a multi-GPU context are discussed. Then, approaches aiming to simplify multi-GPU programming using various abstractions are reviewed. Finally, multi-GPU programming solutions that keep the single-GPU programming model by leveraging compiler-based partitioning are examined.

6.1.1 Multi-GPU Communication Analysis

Communication in-between multiple GPUs has received surprisingly little attention yet. Most related works in the context of data transfers and GPU systems focus on the interconnection between the host system and the GPUs. While communication between multiple GPUs often requires utilizing the host-device interface (i.e. Peripheral Component Interconnect Express (PCIe)), this is not the focus of the analysis in this work. Also, hardware support for direct peer-to-peer communication between GPUs takes pressure from the host-device interface [128]. Victor Lee et al. analyze the performance of GPUs as a component integrated into the host system instead of as an isolated component for a comparison of real-world performance between CPU and GPU-based applications [129]. Fujii et al. focus on the details of transfers between host system and the GPU in order to identify performance bottlenecks and optimizations that reduce latencies of host-device transfers [130]).

One such work analyzing transfers between GPUs, in particular, is from Milic et al., which investigates the implications of NUMA effects of multi-Socket, multi-GPU system with shared GPU memory and proposes hardware and software modifications to mitigate the NUMA effects [131]. Sun et al. use the multi-GPU benchmark suite MGMark to test their multi-GPU simulation framework MGSim and shortly report on the cross-GPU traffic observed for three different multi-GPU configuration [39].

Li et al. propose the Tartan benchmark suites which they use to evaluate four different types of GPU interconnects: PCIe, two versions of NVLink, and an InfiniBand solution [72]. Tallent et al. also evaluate the performance of different GPU interconnects, PCIe and NVLink, but use a multi-GPU training algorithm for different Deep Neural Networks (DNNs), where each GPU independently improves the model based on a subset of the training set and the partial updates to the model are collected and redistributed between iterations [73]. Shi et al. performed an extensive analysis of different distributed deep-learning frameworks in system configurations ranging from single-node single-GPU to multi-node multi-GPU systems, identifying the bottlenecks in communication and overall performance [74]. While this type of analysis naturally provides the most accurate results, it focuses primarily on the final effects of the communication on the actual performance, which depends on a multitude of factors, instead of the pattern and volumes of the communication itself. In addition, this approach is limited by the existing hardware as well as the number of available multi-GPU applications.

The work done in this direction is primarily focused on identifying bottlenecks of the GPU memory hierarchy. In a paper by Li et al., the spatial and temporal

locality of groups of thread blocks in the GPU's L1 cache is analyzed and then use as the motivation for a software-based clustering of thread blocks for better cache utilization [132]. More explicitly focused on communication between thread blocks is the analysis of Wang et al. which specifically investigates the network traffic of on-chip networks between Streaming Multiprocessors (SMs) caused by inter-thread-block locality [133] and propose hardware extensions to minimize redundant network traffic. In a work proposing a software-based extension to the thread block scheduler that leverages user-provided locality information, Vijaykumar et al. perform an analysis of the NUMA effects between SMs arising from spatial and temporal locality between different thread blocks [134]. The approach of Wang et al. is the most closely related to the methodology of the analysis in this work. Both subdivide kernels launched on a single GPU into partitions and then analyze the communication between them as a function of their locality. While Wang et al. use SMs as the granularity, the analysis in this work uses user-defined groupings of thread blocks and lifts them into a completely separate virtual device. Neither of these approaches utilize memory traces, allowing to replay arbitrary system configurations.

6.1.2 Multi-GPU Programming

After the promising speedups achieved by utilizing GPUs in many applications, the next step was increasing the number of GPUs per host system to reduce the host-systems energy overheads [135]. Although CUDA has supported multiple GPUs since its original release, the Application Programming Interface (API) only provides low-level abstractions and requires the user to manually manage all of the additional complexities, including resource allocation, workload distribution, and buffer distribution and synchronization between GPUs.

The simplest approach to simplify multi-GPU programming is to forgo work and data distribution by the user altogether and instead rely on libraries implementing the operations required by the user. This approach does not require any assistance from compilers or hardware and can be implemented purely in libraries. It is an optimization often found in Basic Linear Algebra Subprograms (BLAS) libraries, as implemented by Wang et al. or the cuBLAS library [136, 137]. This approach minimizes engineering efforts required by the user and potentially provides the best performance due to the possibilities for fine-tuning by the developers. However, the limited set of predefined operators limits its applicability to only certain applications.

The unmodified CUDA API provides abstractions similar to that of Message Passing Interface (MPI) implementations, with each GPU representing an MPI-rank

and Memcopies representing messages [138]. However, the similarities between MPI and pure multi-GPU CUDA semantics are superficial as this approach does not follow the Single Program Multiple Data (SPMD) approach of MPI and the CUDA API does not provide any collectives or other advanced features. There exist MPI extensions that extend MPI to support memory located in GPUs, so-called CUDA-aware MPI, but all messages still originate from the host code [139]. A spiritually closer fit to MPI programming model is achieved by projects providing an implementation for GPU-initiated messages, such as the Gravel project proposed by Orr et al. [140].

Another approach that alleviates the user from manual buffer management and synchronization but still requires work distribution is using shared host memory between GPUs. CUDA Unified Virtual Addressing and Unified Memory both transparently provide such a shared memory environment through [14]. Both solutions require no engineering effort from the user, which can simply access any memory in the system from any GPU, albeit with dramatic consequences for application performance [141]. Oden et al. propose GGAS, a Global Address Space (GAS) implementation that provides direct access to memory even on remote GPUs by providing GPUs direct access to an interconnect and forwarding memory requests through the network [142].

An extension of the approach of providing fully implemented operators for the user to combine (e.g. BLAS) is to only provide operator-patterns and let the user fill in the operator-kernel. This approach is taken from functional programming where function composition is generally favored over explicitly defining control flow One of the most popular approaches is Google's MapReduce [143], which has been adapted to GPUs multiple times [144, 145], and is named after the two primary types of computation it provides: map (one-to-one relation between elements) and reduce (many-to-one relation). In the context of this work, projects such as Spark-GPU, Gunrock, and Copperhead can be interpreted as an extension of this approach that primarily changes the operator-patterns (with Copperhead leveraging the patterns of the built-in functional operators in python) [146, 147, 148]. A generalization of this is the Groute project which provides facilities to create graphs describing custom operator patterns and then provide the operator implementation [149]. This approach most closely resembles the one taken in this work with the exception that the programming model is only partially kept intact (it is usually significantly restricted) to simplify implementation and improve performance.

The projects SnuCL by Jungwon Kim et al. and rCUDA Duato et al. both provide the infrastructure to transparently scale out a single node system to multiple nodes by projecting GPUs on remote nodes into the local system, effectively providing a

form of GPU virtualization [150, 3]. Although these works are related in that they, too, address transparently scaling out GPU applications, they are considered to be complementary, since they only address the transition from single-node to multi-node systems on the API level without providing a way to partition applications.

6.1.3 Executing single-GPU applications on multiple GPUs

Several techniques that enable the execution of single-GPU applications on multiple GPUs have been proposed in the past. Although they share a similar goal, they substantially differ in their concept and details.

Pai et al. propose a compiler-assisted approach to manage the memory of accelerators that tracks memory usage and automatically inserts coherence checks that enforce a coherent view on buffers for the X10 programming language [151, 91]. While this approach does leverage static analysis, this analysis is limited to binary flags that specify whether a buffer is read and/or written during a kernel without considering memory access patterns, and a dirty flag for buffers during runtime to track whether a buffer has been modified since its last read access.

Janghaeng Lee et al. first explores optimized scheduling and mapping of workloads across the devices in heterogeneous systems without splitting kernels in the Multiple Kernels Multiple Devices (MKMD) project [152]. He then partitions kernels by prefixing kernels with guarding code while enforcing a coherence between buffers at kernel launch boundaries in the SKMD project and vast projects [93]. Partial results computed on multiple devices are merged using a specially prepared "merge kernel" that is reduced to the index calculation of the original kernel and used to copy memory to a shared buffer, which is taken from the project's predecessor: VAST [96]. If the indices in the kernels memory accesses are a simple linear function of the thread id, memory is distributed according to this function, otherwise buffers are copied in full to satisfy data dependencies. The approach proposed by Pandit et al. follows a similar implementation for the execution guard, but enforces coherence by computing a diff between the kernels original buffer and modified buffer and merging the buffers according to this diff [153]. This work differs in that it extensively utilizes information that is collected with static analysis and tracks memory usage at byte granularity without employing merge kernels or diffs.

Ben-nun et al. distribute both work and data of an application written using the MAPS framework by utilizing user-supplied memory access patterns [94]. Different base types of memory access exist and are refined using a template-based syntax and then exploited to correctly distribute data and minimize transfers. It differs from our

work primarily in that it is based on a fixed set of memory access patterns and requires implementing the application in a specific framework.

Static analysis and in particular the memory access patterns collected from it have been used to manage data dependencies and distribute workloads in the past. The work of Jang et al. uses a fairly simple model that can be interpreted as an extension of the linear model of SKMD, but the results are used for optional optimizations on GPU code without partitioning [154].

More powerful models exist, such as scalar evolutions, which allow modeling non-linear memory accesses [155, 63]. They are used as parts of various optimizations throughout the LLVM optimizations and are used as a transient analysis to calculate the polyhedral representation of a program in the Polly optimizer [32]. However, their use for data distribution in Distributed Memory Systems (DMSs) seems to be unpopular.

Jungwon Kim et al. implement an approach that exhibits some similarities with the one taken by us [156]. They utilize a combination of a runtime system and compile-time transformations to translate a single-GPU OpenCL application to execute as a multi-GPU CUDA application. However, they use a custom model for memory access patterns that relies on sampling array indices of individual threads and is limited to strictly affine, unconditional in well-behaved loops accesses. Additionally, each device buffer image is backed by a synchronized buffer in host memory instead of using a tracker and no host-code transformations are required since the full framework is implemented as a modified OpenCL Runtime.

The polyhedral model is a popular program representation for aggressive transformations, including workload and data distribution, and originates from the desire to accurately model deep loop nests [58, 157]. While it is tempting to assume that any polyhedral transformations based on CPU code can be directly applied to GPU projects, there are enough differences in the technology stacks, execution model and memory model to consider these two separate [158]. The model has been used to predict the performance of OpenCL kernel by creating a database of memory access patterns and their measured performance, without attempting to distribute either work or data [159]. It has also been used to optimize data transfers within GPU kernels. Fauzia et al. optimize accesses to global memory by coalescing them and promoting them to registers and shared memory where possible [160]. Similar work has been done by Baskaran et al. that automatically manages shared memory as a cache for data with high reuse [161]. The Apollo project by Martinez Camaaño et al. utilizes a combination of polyhedral compilation and instrumentation to predictively parallelize

loop nests for multi-core CPUs, supporting even non-linear loops through speculation and a runtime system [162]. Moll et al. exploit memory accesses in the polyhedral model to improve vectorization and GPU utilization by reorganizing kernels into regions with less divergent control flow [102]. While kernels are split into sub-kernels in a transient transformation, these sub-kernels are not targeted at workload distribution and no data distribution across distributed memories is performed.

Polyhedral compilation has also been successfully used to partition applications within specific domains, often written in Domain Specific Languages (DSLs). Denniston et al. propose an extension to Halide, a DSL for complex pipelines of stencil computations, which enables automatic partitioning of the stencil codes across distributed memory systems with minimized data transfers [67]. Bondhugula et al. extend this for general-purpose programming languages in their work that partitions loop nests and the corresponding data across distributed memory systems, relying on MPI for the generated communication code [66]. While influential, their work focuses solely on CPU based systems with no CPU-GPU interplay and relies solely on polyhedral compilation for the resolution of data dependencies, instead of using a dynamic tracker that allows arbitrary reshaping of arrays and data distributions.

6.2 Future Work

The automatic partitioning scheme and prototype implementation presented in this work are primarily intended to decrease the complexity of programming multi-GPU systems, increase developer productivity, and transparently optimize legacy applications to exploit the available hardware in such systems. While these goals have generally been achieved with acceptable performance, there are aspects, both technical and conceptual, that would significantly improve the work.

This low coverage is caused by the use of experimental analysis, targeted at nested loops running on CPUs, and applying it on CUDA code with the requirement that a full kernel must be fully modeled. Polyhedral analysis is typically not applied to code region this large and used for optional optimizations. Future iterations of this project need to either improve the polyhedral analysis or provide alternative models as a fallback solution.

The rewriter applies all transformations on the text level instead of the language level and is susceptible to seemingly malformed but perfectly acceptable code, e.g. through the use of the preprocessor or unfortunately placed comments. Implementing the rewriter on the language level was a decision made purely for productivity reasons.

The transformation is hard to implement in LLVM IR directly because of the concept of a kernel launch (or offloading to an accelerator in general) cannot be reversibly mapped to IR, which is unlikely to change. A transformation on the language level (i.e. the AST) would require either modifying clang or adding another AST-transformation framework to the project. However, this effort is required to provide a correct solution, as the current one is an approximation and only correct under certain circumstances. Considering that the gpucc project already requires modifications to Clang to support the kernel launch syntax of CUDA, modifying Clang to transform the code on the AST level before IR is generated appears to be the cleanest solution.

Currently, all GPUs available in the system are used to execute a partition of the kernel each. While this may be acceptable for some types of kernels and problem sizes, there generally is an upper limit on the number of devices to split across. If this threshold is exceeded, the overhead introduced as communication and synchronization exceeds the performance gained (more details in Section 5.3). One way to identify the maximum number of devices is to quickly compute estimates of the kernel runtime as a function of the number of GPUs used as well as the resulting time required for communication. Based on these estimates, the optimal number of GPUs can be chosen for each kernel individually. If such a performance model of a kernel can be reliably created, it can and should also be used as the bases to identify the optimal partitioning strategy. However, creating reliable performance models is not a trivial task and requires extensive static analysis as well as a model of the individual characteristics and interplay of the system's hardware.

The application model and transformations proposed in this work support any number of arbitrarily sized rectangular partitions. Instead of creating one partition per device, all equally sized, smaller partitions could be created to open up opportunities for overlap of computation and communication. As an example, a 5-point Jacobi stencil can be partitioned into halo regions and the core region. Then, the halo regions (4 strips of size $1 \times N$ or $N \times 1$ per device) are computed first, allowing to communicate the results very early and simultaneous to the computation of the core region. The halo regions can be computed in an application-agnostic way by intersecting read and write sets of the single large "super partitions" on each device.

The prototype presented in this work leverages a dynamic memory tracker that negates the requirement for a correct data flow analysis of all GPU buffers in the host code. Due to the interplay of transfers between host and device buffers and kernel accesses to device buffers, such an analysis is often difficult to perform reliably, hence the need for an efficient alternative. But to improve performance, it might be

worthwhile to still attempt the data flow analysis for buffers and utilize the knowledge if it succeeds. Two optimizations immediately come to mind:

1. Data transfers can be issued earlier, as soon as the data is computed, instead of at the time of the kernel launch. This opens up opportunities for overlap of communication and computation, as described for the finer partition granularity, which requires this kind of extensive static analysis for accurate results.

2. Unnecessary transfers caused by the lack of a shared state of the tracker can be avoided for kernels between which the analysis is successful. For kernels with read-only data that is shared between all devices, such as the planet's positions in N-Body benchmark, this approach can significantly reduce the amounts of transferred data.

This list of improvement ideas is not exhaustive, but instead contains the most promising candidates for optimizations according to the best of the author's knowledge.

CHAPTER 7

Conclusion

This work has set out to investigate how the BSP programming model can be exploited to provide an abstraction of the GPU setup in a given system. It was motivated by the impressive computational power and memory bandwidth of GPUs, which is enabled by a fairly complex programming model that requires highly trained programmers to fully exploit these benefits.

The main challenges in exploiting the full power of GPUs are their sensitivity to complex control flow, the specific ordering of memory accesses to utilize the full bandwidth, and the severely limited communication between threads. Therefore, the first step in this analysis was to collect memory accesses of different existing applications and analyzing them with regards to how well they fit these requirements if they were to be partitioned. A custom instrumentation framework was developed to collect the memory accesses of commonly used CUDA benchmark suits. After collecting the memory traces, they have then been used as input for a simulation that executes a simplified version of these benchmarks, containing no computation, on multiple GPUs by partitioning them. The partitioning is based on four simple mapping functions that do not require any knowledge about the input application. The output of the simulation is the data locality exhibited by thread blocks, with any non-local data accesses being considered communication between either thread-blocks or GPUs, depending on where the communication thread blocks are scheduled. The conclusion of this analysis is promising and two-fold: first, the majority of GPU applications in common benchmarks exhibit highly localized data accesses, both on the thread-block level and the GPU level, and second, for the majority of applications, there exists a simple mapping where

the volume of communication grows slower than the total aggregated bandwidth with the number of GPUs in a system.

This information was then taken as the basis to design and implement a prototype for an automatically partitioning compiler prototype for CUDA applications. The design of the prototype was influenced by the model used to represent work and data dependencies. Due to the regular nature of GPU applications, the polyhedral model was considered an appropriate choice and was selected to model thread blocks and the memory accesses of these thread blocks. Then, the thread grid of the application is interpreted as a super grid, in which each GPU is responsible for computing only a subset of the original grid. The polyhedral model accurately models the memory locations that are read and written by each partition, allowing the identification of data dependencies that require communication. Communication can be optimized by creating a simple coherence model based on CPU caches, which each GPU corresponding to a cache. After designing this model, it was implemented using the LLVM compiler project, which provides a modular design that eases experimentation and contains an almost fully compliant CUDA compiler, which was reused for this analysis. The prototype first performs a polyhedral analysis of the GPU kernels in an application and then modifies the kernels to be position-independent in the super grid. Then, code is generated that efficiently computes data dependencies between GPUs. A runtime library implementing work splitting and the simple, cache-inspired coherence protocol is embedded. Lastly, the main application code is augmented to use the runtime library and the generated to split kernels across GPUs and issue optimized data transfers to satisfy data dependencies.

The prototype was evaluated using custom benchmarks, as using the previously chosen benchmark suites was prevented by technical limitations. Functionally, the prototype's results are mixed. The severe technical limitations of the approach prohibit a straight forward adaptation of the prototype for general-purpose use, which is primarily caused by the reliance on a full-scope polyhedral analysis of the GPU kernels of an application. However, in cases where polyhedral analysis can provide a full-scope analysis, the results are accurate and allow a straight forward and correct partitioning of the applications. Performance-wise, the results are uplifting. In all applications, speed-ups of up to 12.4x on 16 GPUs are achieved, and the overheads introduced by the identification of data dependencies and the resulting communication are typically small. While the small selection of applications certainly can not cover the full spectrum of different types of GPU applications, care was taken to provide a fair selection.

In the context of the current state of research, this work falls into a niche. There

are efforts to simplify the utilization of multiple GPUs within a single node, but they typically only provide partial abstraction and leave data and/or work distribution to the end-user. This work is the first that uses static analysis in the form of polyhedral analysis to fully abstract the underlying GPU setup and provide a fully intact single-GPU programming model: users need not be concerned with whether multiple GPUs are installed in the current system or how many. Work is automatically split where possible and communication inserted in a way that minimizes transfers.

The reliance on static analysis as a critical step results in interesting benefits and drawbacks. On one hand, it allows the assumption that every memory access is accurately represented in the application model, opening the opportunity to generate very efficient code that does not need to approximate or handle many corner cases. On the other hand, it prevents a large set of applications from being compatible with this approach. Supporting gradual degradation of the static analysis could have lessened the impact of this problem, although in practice the analysis quickly degrades to extreme over-approximation.

The work presented in this thesis resulted in several artifacts. The memory access pattern analysis produces two reusable components: the instrumentation framework and the simulation which can be used independently of each other and this work. They enable further research regarding the locality behavior of GPUs, which is likely to stay a relevant topic considering the physical limitation of data transport on accelerators. The primary software artifact is surely the prototype implementation of the automatic partitioning scheme. While it is limited by technical issues, it nonetheless shows impressive results for the cases where static analysis succeeds and showcases the possibilities of the approach with impressive speed-ups of up to 12.4x on 16 GPUs. This is an uplifting result and encourages further research in this area. These artifacts have been made publicly accessible and can be used either as-is or as the basis for new research efforts. In conclusion, the artifacts in combination with the research are valuable contributions to the field of heterogeneous computing and the supporting community.

Acknowledgements

First, I would like to thank my supervisor, Prof. Dr. Holger Fröning, for the opportunity to execute this project under his supervision, which was not in small part responsible for my decision to join this doctoral program. His guidance throughout this process has helped my professional and personal growth tremendously and made this work possible.

The research project described in this thesis did of course not exist in a vacuum, but enjoyed feedback and input from countless like-minded researchers. For this, I want to thank Sudhakar Yalamanchili who, although no longer with us, continues to inspire by his example and dedication throughout his career. I also want to thank Sebastian Hack and Johannes Doerfert both for their discussions and technical advice, as well as for their software contributions.

I would have not been able to finish this project without my fellow PhD students, Benjamin Klenk, Felix Zahn, Günther Schindler, and Lorenz Braun, who I want thank for their endless encouragement and discussions.

Heartfelt thanks goes to my family who has supported me throughout this journey and helped me with all the smaller and larger life issues I have encountered on the way.

This list could not possibly include all the people I am indebted to thanks. Therefore, want to collectively express my gratitude for the countless people I met and formed lasting friendships throughout my time as a Ph.D. student.

List of Figures

List of Algorithms

List of Abbreviations

ABI Application Binary Interface

AMD Advanced Micro Devices

API Application Programming Interface

AST Abstract Syntax Tree

BLAS Basic Linear Algebra Subprograms

BSP Bulk-Synchronous Parallel

CFG Control Flow Graph

CPU Central Processing Unit

CTA Cooperative Thread Array

DMS Distributed Memory System

DNN Deep Neural Network

DSL Domain Specific Language

GAS Global Address Space

GPGPU General Purpose Programming on Graphics Processing Units

GPU Graphics Processing Unit

IR Intermediate Representation

ISA Instruction Set Architecture

MKMD Multiple Kernels Multiple Devices

MPI Message Passing Interface

NUMA Non-Uniform Memory Access

PCIe Peripheral Component Interconnect Express

PTX Parallel Thread eXecution

SKMD Single Kernel Multiple Devices

SM Streaming Multiprocessor

SPMD Single Program Multiple Data

SSA Static Single Assignment

VSM Virtual Shared Memory

Bibliography

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, 1990.

[2] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: An open-source gpgpu compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016.

[3] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proceedings of the 2010 International Conference on High Performance Computing and Simulation*, ser. HPCS '10. IEEE, 2010.

[4] 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*. ACM, 2019, pp. 53–62.

[5] W. Aspray, "The intel 4004 microprocessor: What constituted invention?" *IEEE Annals of the History of Computing*, vol. 19, no. 3, pp. 4–15, 1997.

[6] K. Akeley, "Reality engine graphics," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 109–116.

[7] M. Hopf and T. Ertl, "Accelerating 3d convolution using graphics hardware (case study)," in *Proceedings of the conference on Visualization'99: celebrating ten years*. IEEE Computer Society Press, 1999, pp. 471–474.

[8] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *IEEE Transactions on nuclear science*, vol. 52, no. 3, pp. 654–663, 2005.

[9] T. Klein, S. Stegmaier, and T. Ertl, "Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids," in *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.* IEEE, 2004, pp. 186–195.

[10] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 258.

[11] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[12] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[13] N. Corporation, "Whitepaper - NVIDIA's next generation CUDA compute architecture: Fermi," https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009, accessed: 2019-07-15.

[14] "Cuda toolkit documentation," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, accessed: 2019-05-18.

[15] G. T. Davis and S. Ventrone, "Method and apparatus for substantially concurrent multiple instruction thread processing by a single pipeline processor," 1994, uS Patent 5,357,617.

[16] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: Weak behaviours and programming assumptions," *ACM SIGPLAN Notices*, vol. 50, no. 4, 2015.

[17] C. Xu, S. R. Kirk, and S. Jenkins, "Tiling for performance tuning on different models of gpus," in *2009 Second International Symposium on Information Science and Engineering*. IEEE, 2009, pp. 500–504.

[18] B. Van Werkhoven, J. Maassen, and F. J. Seinstra, "Optimizing convolution operations in cuda with adaptive tiling," in *A4MMC'11: Proc. Workshop on Applications for Multi and Many Core Processors*, 2011.

[19] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04, Mar. 2004.

[20] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, 1998.

[21] R. Wilhelm, D. Maurer, and S. S. Wilson, *Compiler design*. Springer, 1995.

[22] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[23] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern compiler design*. Springer Science & Business Media, 2012.

[24] C. Guntli, "Architecture of clang," *Analyze an open source compiler based on LLVM*, 2011.

[25] L. Project, "LLVM compiler infrastructure v7.0.0 - documentation," https://releases.llvm.org/7.0.0/docs/index.html, 2018, accessed: 2019-07-15.

[26] ——, "Clang 7 documentation," https://releases.llvm.org/7.0.0/tools/clang/docs/, 2018, accessed: 2019-07-15.

[27] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 69–80, 2003.

[28] "Rust blog - introducing MIR," https://blog.rust-lang.org/2016/04/19/MIR.html, accessed: 2019-07-15.

[29] A. Zakai, "Emscripten: an LLVM-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 301–312.

[30] D. A. Terei and M. M. Chakravarty, "An LLVM backend for GHC," in *ACM Sigplan Notices*, vol. 45, no. 11. ACM, 2010, pp. 109–120.

[31] J. D. Ullman, "Fast algorithms for the elimination of common subexpressions," *Acta Informatica*, vol. 2, no. 3, pp. 191–213, 1973.

[32] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly - polyhedral optimization in LLVM," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, ser. IMPACT '11, vol. 2011, 2011.

[33] A. S. Tanenbaum, H. v. Staveren, and J. W. Stevenson, "Using peephole optimication on intermediate code," 1982.

[34] J. W. Davidson and C. W. Fraser, *Automatic generation of peephole optimizations*. Citeseer, 1984, vol. 19, no. 6.

[35] J. W. Davidson, "Simplifying code generation through peephole optimization," 1981.

[36] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for GPGPU," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2010, pp. 351–360.

[37] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani, "UNISIM: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, 2007.

[38] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2012, pp. 335–344.

[39] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGSim + MGMark: A framework for multi-GPU system research," *arXiv e-prints*, p. arXiv:1811.02884, Oct. 2018.

[40] 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)*. IEEE, 2017, pp. 269–278.

[41] A. Bakhoda, G. L. Yuan, W. W. 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*. IEEE, 2009, pp. 163–174.

[42] S. Lee and W. W. Ro, "Parallel GPU architecture simulation framework exploiting work allocation unit parallelism," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 107–117.

[43] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1996, pp. 46–57.

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[45] Z. Wang, A. Sanchez, and A. Herkersdorf, "SciSim: a software performance estimation framework using source code instrumentation," in *Proceedings of the 7th international workshop on Software and performance*. ACM, 2008, pp. 33–42.

[46] R. Venkatapathy, "Preprocessor-based source code instrumentation," 2009, uS Patent 7,484,205.

[47] L. Project, "LLVM compiler infrastructure v7.0.0 - XRay instrumentation," https://releases.llvm.org/7.0.0/docs/XRay.html, 2016, accessed: 2019-07-17.

[48] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering ..., 2004.

[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[50] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[51] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 175–183.

[52] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari, "Analyzing dynamic binary instrumentation overhead," in *WBIA workshop at ASPLOS*. Citeseer, 2006.

[53] T. M. Conte and A. Wolfe, "Hierarchical read-combining local memories," May 15 2012, uS Patent 8,180,963.

[54] 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*. IEEE, 2012, pp. 58–67.

[55] "SASSI instrumentation tool for NVIDIA GPUs," https://github.com/NVlabs/SASSI, accessed: 2019-05-09.

[56] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky, and J. Devietti, "BARRACUDA: Binary-level analysis of runtime races in CUDA programs," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 126–140.

[57] M. Griebl and J.-F. Collard, "Generation of synchronous code for automatic parallelization of while loops," in *European Conference on Parallel Processing*. Springer, 1995.

[58] L. Lamport, "The parallel execution of do loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, 1974.

[59] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and*

Implementation, Tucson, AZ, USA, June 7-13, 2008, 2018. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[60] P. Feautrier, "Parametric integer programming," *RAIRO-Operations Research*, vol. 22, no. 3, pp. 243–268, 1988.

[61] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The omega library interface guide," 1995.

[62] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*, vol. 6327. Springer, 2010.

[63] S. Pop, A. Cohen, and G.-A. Silber, "Induction variable analysis with delayed abstractions," in *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 2005, pp. 218–232.

[64] S. Verdoolaege, "Presburger formulas and polyhedral compilation," 2016.

[65] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second International Workshop on Polyhedral Compilation Techniques*, ser. IMPACT '12, 2012.

[66] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[67] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016, p. 5.

[68] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004, pp. 7–16.

[69] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong, "Improving polyhedral code generation for high-level synthesis," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 2013, p. 15.

[70] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, 2013.

[71] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST generation is more than scanning polyhedra," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 4, p. 12, 2015.

[72] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: Evaluating modern GPU interconnect via a multi-GPU benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 191–202.

[73] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating on-node GPU interconnects for deep learning workloads," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Cham: Springer International Publishing, 2018, pp. 3–21.

[74] S. Shi and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on GPUs," *CoRR*, vol. abs/1711.05979, 2017. [Online]. Available: http://arxiv.org/abs/1711.05979

[75] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-chip-module GPUs for continued performance scalability," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080231

[76] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-GPU system design with memory networks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 484–495.

[77] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying NUMA and contention effects in multi-gpu systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 11:1–11:7. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964194

[78] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[79] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, May 2012.

[80] J. K. Lawder and P. J. King, "Using space-filling curves for multi-dimensional indexing," in *British National Conference on Databases*. Springer, 2000, pp. 20–35.

[81] Q. H. Dang, "Secure hash standard," NIST, Tech. Rep., Jul. 2015. [Online]. Available: https://doi.org/10.6028/nist.fips.180-4

[82] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[83] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[84] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *IMPACT Technical Report*, 2012.

[85] V. Adhinarayanan and W. Feng, "An automated framework for characterizing and subsetting GPGPU workloads," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 307–317.

[86] F. N. Sibai, "3d graphics performance scaling and workload decomposition and analysis," in *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*, July 2007, pp. 604–609.

[87] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[88] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: http://www.jstor.org/stable/2003354

[89] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[90] A. Matz, M. Hummel, and H. Fröning, "Exploring LLVM infrastructure for simplified multi-GPU programming," in *Proceedings of 9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2016)*, 2016.

[91] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: An object-oriented approachto non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.

[92] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proceedings of the*

8th International Conference on Partitioned Global Address Space Programming Models. ACM, 2014, p. 6.

[93] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration," *ACM Transactions on Computer Systems*, vol. 33, no. 3, 2015.

[94] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: The missing piece of the multi-GPU puzzle," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807611

[95] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA communication via PCI express," *Cluster Computing*, vol. 17, no. 2, pp. 339–348, 2014.

[96] J. Lee, M. Samadi, and S. Mahlke, "VAST: The illusion of a large memory space for GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628075

[97] W. Pugh and D. Wonnacott, "Static analysis of upper and lower bounds on dependences and parallelism," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1248–1278, 1994.

[98] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in cuda," *Nvidia CUDA SDK Application Note*, vol. 18, 2009.

[99] J. Doerfert and S. Hack, "Polyhedral value & memory analysis," 2017, 2017 US LLVM Developers' Meeting.

[100] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 188–202, 1972.

[101] J. Doerfert, T. Grosser, and S. Hack, "Optimistic loop optimization," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049864

[102] S. Moll, J. Doerfert, and S. Hack, "Input space splitting for OpenCL," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2892208.2892217

[103] "Information technology — programming languages —- c," ISO, Standard, 2011.

[104] "Information technology — programming languages —- c++," ISO, Standard, 2017.

[105] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 351–360.

[106] P. Chatarasi, J. Shirako, and V. Sarkar, "Polyhedral optimizations of explicitly parallel programs," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 213–226.

[107] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. Zaks, "Automatic loop parallelization via compiler guided refactoring," 2011.

[108] D. S. Watkins, *Fundamentals of matrix computations*. John Wiley & Sons, 2004, vol. 64.

[109] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*. Springer, 1996, pp. 79–103.

[110] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *International Conference on Compiler Construction (CC)*, ser. LNCS, vol. 3923. Vienna: Springer, 2006. [Online]. Available: http://icps.u-strasbg.fr/~bastoul/research/papers/VBC06-CC.pdf

[111] E. Hagersten, A. Landin, and S. Haridi, "Ddm-a cache-only memory architecture," *Computer*, vol. 25, no. 9, pp. 44–54, 1992.

[112] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[113] D. A. Plaisted, "Source-to-source translation and software engineering," *Journal of Software Engineering and Applications*, vol. 6, no. 04, p. 30, 2013.

[114] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus - an extensible compiler infrastructure for source-to-source transformation," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 539–553.

[115] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.

[116] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE transactions on Electronic Computers*, no. 1, pp. 39–47, 1960.

[117] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, "Lua — an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[118] T. Grosser, "Enabling polyehdral optimizations in LLVM," Master's thesis, University of Passau, 2011.

[119] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012.

[120] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[121] J. Doerfert, S. Sharma, and S. Hack, "Polyhedral expression propagation," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: ACM, 2018, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/3178372.3179529

[122] C. S. Kouzinopoulos and K. G. Margaritis, "String matching on a multicore GPU using CUDA," in *2009 13th Panhellenic Conference on Informatics*. IEEE, 2009, pp. 14–18.

[123] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *International Conference on Compiler Construction*. Springer, 2010, pp. 283–303.

[124] G. F. Carey and D. L. Humphrey, "Mesh refinement and iterative solution methods for finite element computations," *International Journal for Numerical Methods in Engineering*, vol. 17, no. 11, pp. 1717–1734, 1981.

[125] S. A. Finney, "Real-time data collection in linux: A case study," *Behavior Research Methods, Instruments, & Computers*, vol. 33, no. 2, pp. 167–173, May 2001. [Online]. Available: https://doi.org/10.3758/BF03195362

[126] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[127] J. Hartmanis and R. E. Stearns, "On the computational complexity of algorithms," *Transactions of the American Mathematical Society*, vol. 117, pp. 285–306, 1965.

[128] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[129] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[130] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ser. ICPADS '13. Washington,

DC, USA: IEEE Computer Society, 2013, pp. 275–282. [Online]. Available: http://dx.doi.org/10.1109/.46

[131] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: NUMA-aware GPUs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 123–135. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124534

[132] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware CTA clustering for modern GPUs," *SIGPLAN Not.*, vol. 52, no. 4, pp. 297–311, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093336.3037709

[133] L. Wang, X. Zhao, D. Kaeli, Z. Wang, and L. Eeckhout, "Intra-cluster coalescing to reduce GPU NoC pressure," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018.

[134] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 829–842.

[135] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[136] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, "BLASX: A high performance level-3 BLAS library for heterogeneous multi-GPU computing," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 20.

[137] N. Corporation, "PCUDA toolkit documentation - cuBLAS," https://docs.nvidia.com/cuda/cublas/index.html, 2017, accessed: 2019-07-18.

[138] W. Gropp, W. D. Gropp, A. D. F. E. E. Lusk, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[139] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 257, 2011.

[140] M. S. Orr, S. Che, B. M. Beckmann, M. Oskin, S. K. Reinhardt, and D. A. Wood, "Gravel: fine-grain GPU-initiated network messages," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. ACM, 2017.

[141] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[142] L. Oden and H. Fröning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[143] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[144] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU clusters," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 1068–1079.

[145] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2008, pp. 260–269.

[146] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, p. 3, 2017.

[147] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," *ACM SIGPLAN Notices*, vol. 46, no. 8, 2011.

[148] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, "Spark-GPU: An accelerated in-memory data processing engine on clusters," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 273–283.

[149] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018756

[150] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304623

[151] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. ACM, 2012.

[152] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating multiple data-parallel kernels on multiple devices," in *International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '15, vol. 24, 2015.

[153] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544163

[154] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, Jan. 2011.

[155] M. Wolfe, "Beyond induction variables," in *ACM SIGPLAN Notices*, vol. 27, no. 7. ACM, 1992, pp. 162–174.

[156] 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.

[157] C. Lengauer, "Loop parallelization in the polytope model," in *International Conference on Concurrency Theory*. Springer, 1993, pp. 398–416.

[158] S. Baghdadi, A. Größlinger, and A. Cohen, "Putting automatic polyhedral compilation for GPGPU to work," in *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, 2010.

[159] J. Fang, H. Sips, and A. Varbanescu, "Aristotle: a performance impact indicator for the OpenCL kernels using local memory," *Scientific Programming*, vol. 22, no. 3, Jan. 2014.

[160] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. IEEE Computer Society, 2015.

[161] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345210

[162] J. M. Martinez Caamaño, M. Selva, P. Clauss, A. Baloian, and W. Wolff, "Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, p. e4192, 2017.