UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving Data-Dependent Parallelism in GPUs Through Programmer-Transparent Architectural Support

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Amir Ali Abdolrashidi

September 2021

Dissertation Committee:

Dr. Daniel Wong, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Laxmi N. Bhuyan
Dr. Zizhong Chen

The Dissertation of Amir Ali Abdolrashidi is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This dissertation marks the end of a long journey of research and learning in graduate school, but also the beginning of a new journey of research and learning with additional invaluable experiences, knowledge, insight, and wisdom obtained during my tenure as a graduate student and a Ph.D. candidate.

First and foremost, I would like to extend my deepest gratitude to my advisor, Prof. Daniel Wong. Not only did I learn many things from him in my field of study, but also how to stay motivated, overcome various challenges throughout our research, and incorporate new ideas into it as well. He was always helpful and supportive of us inside and outside of school, guiding us in our quest with great patience and energy. I could have never accomplished this without him, and I am most glad and grateful to have known and worked with him in the past six years. I would also like to express my thanks to all the members of my committee, Prof. Nael Abu-Ghazaleh, Prof. Laxmi N. Bhuyan, and Prof. Zizhong Chen, for their incredible help and support, including during my qualifying exam, dissertation proposal and dissertation defense.

I would also like to thank all my amazing friends, lab-mates and colleagues throughout my Ph.D. career, including, but not limited to, Devashree Tripathy, Bashar Romanous, Kiran Ranganath, Marcus Chow, Ali Jahanshahi, Mohammadreza Rezvani, Shahriyar Valielahiroshan, Hodjat Asghari Esfeden, and Shervin Minaee. It was a pleasure to work with them on various projects, and their company made this journey a much easier and more delightful experience. I shall treasure every moment I have spent with them, either in person or virtually, and wish the best for them at their every step.

Sometimes, when on top of a mountain looking upon the mesmerizing scenery, one can easily forget what they are standing on or how they even got to that spot. Any change along the path, no matter how small it may seem, could change the entire course ahead. Therefore, I would like to use this opportunity to thank all the professors, mentors, co-workers, teachers, students, friends, and others throughout my life for being there for me, believing in me, sharing their knowledge and experience with me, or helping me in any way to get to this point. I shall be forever thankful for them all.

Finally, most importantly, I would like to express my warmest appreciation to my wonderful family for their endless love, nurture, and support from the very beginning. I would not have been here had it not been for their great care, motivation, and affection. They always stood by my side through thick and thin, despite being on the other side of the world, and even through times when all hope seemed to be fading. No word can properly describe how grateful I am for them for everything. They will always be in a special place on my mind and in my heart.

This dissertation contains content published in the following proceedings:

- Abdolrashidi, AmirAli, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. "Wireframe: Supporting Data-dependent Parallelism Through Dependency Graph Execution in GPUs." In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 600-611. 2017.

- Abdolrashidi, AmirAli, Hodjat Asghari Esfeden, Ali Jahanshahi, Kaustubh Singh, Nael Abu-Ghazaleh, and Daniel Wong. "Blockmaestro: Enabling Programmer-transparent Task-based Execution in GPU Systems." In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 333-346. IEEE, 2021.

To my parents for all their love and support.

ABSTRACT OF THE DISSERTATION

Improving Data-Dependent Parallelism in GPUs Through Programmer-Transparent Architectural Support

by

Amir Ali Abdolrashidi

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2021
Dr. Daniel Wong, Chairperson

As modern GPU workloads become larger and more complex, there is an ever-increasing demand for GPU computational power. Traditionally, GPUs have lacked generalized data-dependent parallelism and synchronization. In recent years, there have been attempts to introduce a more sophisticated form of synchronization between different kernels in an application to control the flow and ensure the correctness of the outputs. However, coarse synchronization between such kernels can significantly reduce GPU utilization. Moreover, with hundreds or thousands of kernels in a workload, the overhead can be consequential. Due to GPU's massive parallel design, data can be split among thread blocks, which allows us to manage the data dependencies on a more fine-grained level between the thread blocks themselves rather than the kernel containing them. In this dissertation, we propose several methods to improve the performance of data-dependent GPU applications in this fashion.

In our first method, Wireframe, we propose a hardware-software solution that enables generalized support for data-dependent parallelism and synchronization. It allows

dependencies between the thread blocks in the GPU kernel to be expressed through a global dependency graph, which is then sent by the GPU hardware at kernel launch, which then enforces the dependencies in the graph through a dependency-aware thread block scheduler.

Our second method, BlockMaestro, is aimed at improving the user transparency in the process of determining the inter-kernel thread block dependencies through static analysis of memory access patterns at kernel-launch time. During the runtime, BlockMaestro enables kernel launch hiding by launching multiple kernels on the GPU and utilizes a thread block scheduler in hardware to schedule the thread blocks with satisfied dependencies for execution.

In our third method, SEER, we aim to expand our support for data-dependent applications to those with non-static memory accesses, which can only be known during runtime. Seeking a solution to this problem, we use a machine learning model in an effort to estimate the memory addresses accessed in global load and store instructions in a kernel, and using that information to predict the inter-kernel dependency pattern among thread blocks using such accesses in order to improve the performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphics Processing Units (GPUs) have come a long way in the past few decades. Once used merely as display adapters, GPUs are now capable of performing computations on a massive amount of data through a single-instruction, multiple-thread (SIMT) paradigm, and with the advent of general-purpose GPUs (GPGPUs) and programming interfaces such as CUDA [71] and OpenCL [87] in the early 21st century, their uses have only increased faster. Now, with workloads and their data growing larger, the need for faster and more efficient GPUs are further emphasized. GPUs can execute kernels, i.e., functions programmed by the user, on many threads grouped into *thread blocks* (in NVIDIA terminology, which we will use here) or *wavefronts* (in AMD GPUs). These thread blocks (TBs) run on the GPU's streaming multiprocessors (SMs) and produce outputs when the kernel is finished.

The GPU hardware is designed to run many threads at the same time, which makes it ideal for simple tasks, and especially, *embarrassingly parallel* applications, where there is no notion of *data dependency* among the threads. GPUs nowadays can also utilize *streams*

in order to run multiple kernels at the same time. Such kernels are usually called one after the other in the code, i.e., adjacent kernels, which would constitute a *control dependency* between each kernel pair, had the streams not been utilized.

However, workloads are also getting more complex every day, and data dependencies are now very common in various applications, such as deep learning, stencil operations, and other high-performance computing tasks. Data dependency among threads can occur either within a kernel or between two kernels, i.e., *intra-kernel* and *inter-kernel dependencies* respectively. In both cases, some thread blocks may require the results of other thread blocks for their work (e.g., heat transfer). In traditional GPUs, this requires synchronization of the thread blocks in order to ensure the correctness of the application's output. This forces the dependent thread blocks to wait, despite occupying space on the GPU's execution unit. This can prevent other ready thread blocks to be scheduled on the GPU, reducing the GPU's performance. In addition, as the application grows in size and number of kernels, so does the amount of overhead from launching the kernels.

In many cases of coarse-grained synchronization, a task's data dependencies come from few prior tasks, but it has wait for all the prior tasks to finish to be able to continue. For example, in a wavefront application [8,19,20,73,137], a TB may only depend on two TBs from the previous "wave". However, it has to wait for that entire wave to finish executing before being issued. This will get even worse as the number of TBs in the wave grows.

Inter-kernel dependencies can also affect a GPU application using streams. In this case, if a kernel requires the output of the kernel before it, scheduling TBs from both kernels at once will no longer be possible, and the two kernels should still synchronize to maintain

correctness. However, ready TBs from the waiting kernel should also be delayed until all TBs from the running kernel have finished, resulting in *dependency stalls*. This also results in GPU under-utilization and decrease in performance.

Ideally, by using a fine-grained data dependency support, we can improve the GPU utilization, minimize the dependency stalls, and speed up the running application. This can be achieved using a task-based execution model. Currently, mainstream GPUs lack a generalized solution to tackle the aforementioned issues. Many works exist in literature, each utilizing a variation of such a model [12, 20, 25, 39, 46, 54, 56, 65, 103, 135, 148]. However, many of them require significant user (programmer) intervention, meaning that, in many cases, it is up to the user to specify the whole application as a task graph or re-write the code to fit into the new paradigm, which can be quite burdensome with more complex or irregular applications. In addition, many works have a large software-based management overhead, with little to no management tasks offloaded to the hardware.

The work in this dissertation aims to improve the performance of data-dependent GPU applications by proposing a combination of hardware and software support, stepping towards a fine-grained generalized data dependency management of thread blocks in a GPU application. In short, our main goals are to:

- mitigate the overheads, such as kernel launch, management, etc.;
- minimize the amount of programmer intervention; and
- provide a more generalized framework for dependency resolution.

In Chapter 2, we introduce *Wireframe* [8], a hardware/software solution that enables the user to express the data dependencies of the thread blocks within an application

converted to a single mega-kernel, which will be transferred to the GPU as a *dependency graph*. The dependencies will then be used by a dependency-aware thread block scheduler to schedule the TBs in a way that both ensures correctness and improves the application's performance. With a small hardware overhead, Wireframe is able to achieve an average speedup of 45% in data-dependent applications with a wavefront pattern, which will be explained. However, the user needs to convert the workload into a single-kernel format to use it, thereby highlighting the user burden and low flexibility of Wireframe.

In Chapter 3, we propose *BlockMaestro* [7], another hybrid solution, but focusing on reducing the user burden and fine-grained inter-kernel dependencies. Through modifying the command queue, BlockMaestro can launch multiple adjacent kernels at once, and manage the scheduling of their TBs in hardware through the use of a TB scheduler. In addition to kernel pre-launching aimed at masking kernel launch overheads, BlockMaestro also tries to minimize user intervention by deducing the static data dependencies through the analysis of the kernel codes in the intermediate representation (PTX). In addition, unlike in Wireframe, the user no longer requires to modify the application to run it on BlockMaestro. Thus, we can use it for applications with multiple kernels and various dependency patterns. With two kernels, BlockMaestro achieves an average speedup of 51% on data-dependent benchmarks, with minimal hardware overhead.

Finally, in Chapter 4, we propose *SEER*, a machine learning-based framework in an effort to expand BlockMaestro. One of BlockMaestro's major limitations is that it cannot be used where the data dependencies are not known before runtime, e.g., if the kernel uses indirect memory accesses, and the access pattern is related to the memory content rather

than memory address. We seek to mitigate this issue by building a prediction model to estimate the read and write sets of each TB by analyzing the code on the PTX level, extracting the necessary data and context regarding each global load and store operation, and for each, classify each memory index into either the 'accessed' or 'not accessed' group. Once the accessed list is produced for adjacent kernels, they can be used to generate a dependency graph, which would then be used for fine-grained thread block scheduling in order to improve the performance and utilization of the GPU.

Chapter 2

Wireframe: Supporting Data-Dependent Parallelism Through Dependency Graph Execution in GPUs

2.1 Introduction

GPUs have played a remarkable role in the evolution of scientific computing in the last decade. The massive parallelism offered by thousands of compute cores has led developers to redesign traditional CPU applications to run on the massively parallel hardware. Despite the rapid adaptation of GPGPU computing with an enlarging number of application classes, the GPU hardware has failed to evolve fast enough to account for the increasing

complexity of such applications. A major deficiency in the modern CUDA programming paradigm is a lack of fine-grained support for data-dependent parallelism and synchronization. Typically data dependencies require algorithms to be redesigned and mapped to intra-SM barriers (using `__syncthreads()`) or global barriers via implicit synchronization through consecutive kernel launches. This causes difficulty in programming GPGPUs due to mapping algorithms to these constraints, and more importantly, is responsible for significant inefficiencies in the hardware due to load imbalance and resource under-utilization [31]. Recent studies [19, 144] have shown that, SMs can remain under-utilized and unnecessarily idle as the execution reaches near global barriers, even though there are TBs whose dependencies are already satisfied.

An intermediate level of inter-block synchronization can ease programmer burden by granting programmers flexibility to convey data-dependent synchronization at the thread block (TB) level. Unfortunately, existing GPGPU software and hardware assume that the TBs (in CUDA), or workgroups (in OpenCL), in a given kernel can be executed in any order, since there is no native support for synchronization between TBs.

Prior work [51, 141] has shown that it is possible to implement limited inter-TB synchronization in software via *persistent threads (PT)*. In this approach, the kernels are redesigned to run with limited number of TBs, whose total count is equal to the number of SMs. The threads in different TBs synchronize via global memory-based software barriers as they iterate through the data indices. However, the PT approach may cause deadlocks due to potentially unscheduled TBs and also may increase global memory access contention if the inter-TB synchronization is frequent.

In a step towards supporting data-dependent parallelism, CUDA dynamic parallelism (CDP) was introduced to support nested parallelism [89]. CDP enables parent kernels to launch child kernels, and then optionally synchronize on the completion of the latter. CDP is mainly limited to certain application patterns with recursive nested parallelism and time-varying data-dependent nested parallelism, such as loops [134]. Moreover, CDP introduces additional kernel launch overhead due to in-memory context switching, and also significant effort is required for programmers to efficiently map workloads to dynamic parallelism kernels [40].

Prior work have proposed to avoid the overhead of kernel launches in CDP, by instead launching thread blocks in hardware [123, 134, 135], supporting nested parallelism for loops through code transformation [145] or consolidating kernel launch overheads [27, 40]. In CUDA 9, Cooperative Threads (CUDA-CT) were introduced to enable explicit synchronization between threads within and across thread blocks, which enables an efficient implementation for global barriers [3]. Although CUDA-CT will partially remedy the problems caused by device-level kernel launches, the SM under-utilization problem mentioned above will remain due to bulk-synchronization mechanisms across multiple TBs still present.

In an attempt to enable *true* data-dependent parallelism on GPUs, several task-based software execution schemes have been proposed to enable a producer-consumer model between tasks (i.e., TBs) and SMs. These schemes resemble dataflow execution models [42, 50], but the main computation units are SMs instead of CPU cores. Tzeng et al. [128] proposed a scheme where tasks with resolved dependencies are inserted in a centralized first-come, first-served (FCFS) queue and executed. Belviranli et al. [18] proposed a scheduler-

worker-based solution based on distributed queues, where task dependencies are maintained by a scheduler thread block via an in-memory dependency matrix and updated on-the-fly as the tasks are processed by the worker TBs. However, the major drawback for all these software solutions is their reliance on expensive global memory atomics as well as busy-waiting to handle task insertion & retrieval operations and inter-SM communication.

Fundamentally, there is a lack of support for conveying generalized data-dependent parallelism and inter-SM synchronization. While task-based execution schemes rely on long-latency global memory, others focus on improving CDP-based kernels by compile-time or runtime optimizations to achieve better thread utilization for a specific class of applications (i.e., nested parallelism). Yet none of the aforementioned studies provide a generalized solution for an arbitrary network of inter-block data dependencies. To this end, we propose *Wireframe*[1], a hardware-software approach which provides generalized support for hardware execution of task-based dependency graphs.

Wireframe is built on the abstraction of *Dependency Graph (DG)* execution, where individual thread blocks are represented as *tasks*. These dependency graphs can be generated either through programmer API (*DepLinks*), or compiler profiling [44, 47, 48, 112, 131]. The dependency graph is then enforced in the hardware through a *Dependency-Aware Thread block Scheduler (DATS)*.

In this chapter, we show that Wireframe can be utilized to support a generalized dependency graph-based execution approach to enable programmers to naturally convey data-dependent parallelism. In addition, we show that Wireframe can be used to sup-

[1]The name "Wireframe" stems from the similarities between the graphs utilized in our benchmarks with standard 3D wireframe terrain models used in computer-aided design.

port lightweight barrier and deadlock-free inter-block synchronizations through dependency graph primitives.

This chapter makes the following contributions:

- In Section 2.2, we present a case for Wireframe and show how the dependency graph abstraction can be generalized for data-dependent parallelism and efficient synchronization.
- In Section 2.3, we present DepLinks to support programming data-dependent parallelism. We also present support for run-time dependency graph generation.
- Section 2.4 demonstrates hardware support for dependency graph execution through dependency-aware thread block scheduling (DATS). DATS enforces dependencies with a Dependency Graph Buffer (DGB) and maximizes ready nodes with Level-bounded thread block scheduling.
- Section 2.5 involves evaluation for Wireframe using a range of data-dependent workloads, measuring an average of 45% performance boost, with ∼2KB area overhead.

2.2 Motivation

In this section, we motivate a case for Wireframe. We will make use of various Code Blocks to motivate and drive this section. We use a basic wavefront pattern, a common data-dependent parallel pattern [19, 73, 137], as a running illustrative example due to its simple structure and clarity in conveying concepts in Wireframe. It should be stressed that our proposed technique is generic to all data-dependent parallel patterns and in no way limited to the examples presented here. Figure 2.1 displays a wavefront pattern.

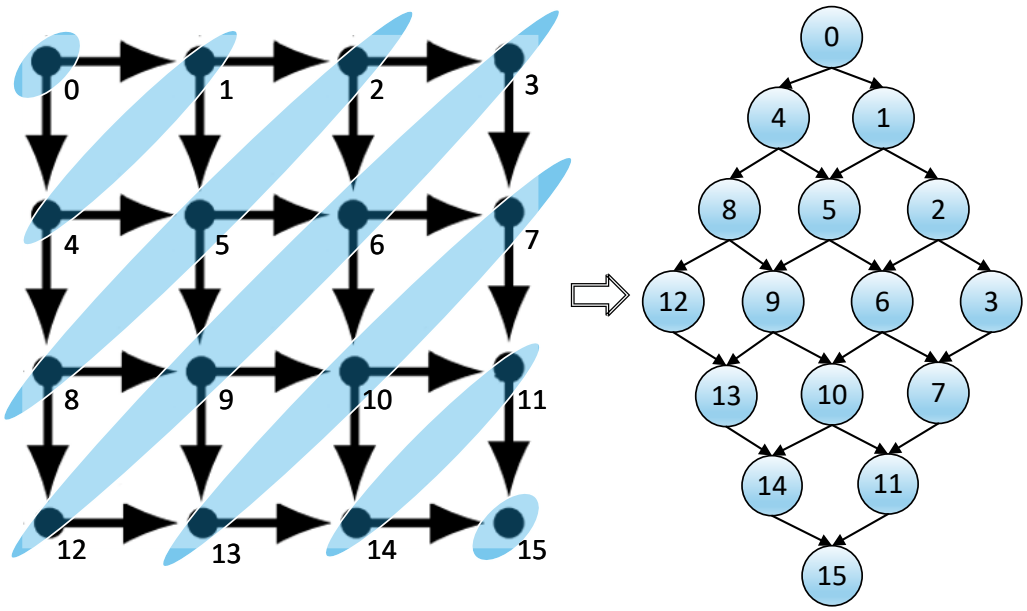


Figure 2.1: Wavefront pattern execution of thread blocks in an application kernel (left) and its equivalent dependency graph (right). The numbers represent the node IDs.

In wavefront parallelism, computation are typically dependent on neighbors, where data dependencies form diagonal "waves" of computation (shown in blue). We define *task* as an abstract unit of computation. In this example, a task can be fine-grained and represent the computation of a single element in the wavefront, or it can be coarse-grained and represent a tile consisting of multiple elements. The dependencies between tasks in this workload is shown on the right, as a directed graph, which we call a *dependency graph*, with each node representing a thread block.

2.2.1 Data-dependent Parallelism

We will now demonstrate CUDA's current support for data-dependent parallelism, and highlight its limitations and challenges. An implementation of wavefront processing

using global barriers is shown in Code Block 2.1. Every wave computation maps to a kernel call, which processes the computation for that wave. As demonstrated in various prior works [18, 19, 124], this limitation of global barriers introduces significant overhead due to multiple kernel launches and requires programmers to map data-dependent parallelism to this rigid constraint. An alternative option so as to avoid multiple kernel launches is to enforce synchronization of waves within the thread block. This requires each wave to be processed entirely within a single thread block, which would severely under-utilize the GPGPU hardware.

Code Block 2.1: Global Barriers

```
int main() {                                                    1
    for (int i=0; i<nWaves; i++) {                              2
      kernel<<<GridSize, BlockSize>>>(args);                    3
      cudaDeviceSynchronize();                                  4
    }                                                           5
}                                                               6
__global__ void kernel(args) { processWave(); }                 7
```

In order to facilitate support for data-dependent nested parallelism, CUDA Dynamic Parallelism (CDP) was introduced. CDP enables device-side kernel launches, avoiding the overhead of host-side kernel launches. Every device-side thread has the ability to spawn a child kernel. CDP typically supports two common implementation methods - recursion and nesting. Code Block 2.2 shows an implementation of CDP using a recursive pattern. Here every wave is still processed by a single kernel and subsequent waves are handled by recursively launching another kernel until every wave has been processed. In lines 9-12, we have thread 0 spawning a single child kernel and wait for its completion. A main limitation of the recursive approach is the recursion depth limitation. In CDP, there

is a maximum nesting depth of 24 [2], i.e 25 waves at most. This pattern works well for algorithms that can be mapped recursively, but otherwise inflexible.

Code Block 2.2: Dynamic Parallelism – Recursive

```
int main() {
     kernel<<<GridSize, BlockSize>>>(0, args);
     cudaDeviceSynchronize();
}
__global__ void kernel(i, args) {
    if(i == nWaves) return;
    processWave();
    if(threadIdx == 0) {
        kernel<<<GridSize,BlockSize>>>(i+1,args);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
```

A more flexible implementation is shown in Code Block 2.3, where nested parallelism is used. In this approach [22], a parent kernel launches a child kernel for every wave. However, unlike recursive parallelism where the child will also spawn a child kernel of its own, the child returns, prompting the parent kernel to launch the next child kernel, which resolves the spawning depth limit issue in the recursive version. This approach is very similar to the global barriers implementation, but with the overhead of device-side kernel launch instead of host-side kernel launch.

Although this implementation has less overhead, the device-side kernel launches still incur non-trivial overhead [134] and there is also the limitation of coarse-grained synchronization across waves. This implicit synchronization introduced by kernel launches limits potential opportunities for nodes to run ahead and execute when ready. For example, during the 4th wave, if nodes 9 and 12 are ready, then node 13 is ready to execute, but

has to stall until nodes 3 and 6 complete the wave. This limitation is mainly due to the 1-parent-m-child representation of CDP, where child kernels can only have a single parent. Therefore, the wavefront pattern has to be mapped to coarse-grained synchronization at wavefront boundaries.

Code Block 2.3: Dynamic Parallelism – Nested

```
int main() {
     parentKernel<<<GridSize, BlockSize>>>(args);
     cudaDeviceSynchronize();
}
__global__ void parentKernel(args) {
    for (int i=0; i<nWaves; i++) {
       if(threadIdx == 0) {
          childKernel<<<GridSize, BlockSize>>>(args);
          cudaDeviceSynchronize();
       }
        __syncthreads();
    }
}
__global__ void childKernel(args) { processWave(); }
```

In order to fully express the data-dependent parallelism of the wavefront pattern, we need a generalized approach to convey n-parent-m-child relationships. In our wavefront example, in addition to 1-parent-1-child (e.g., node 12), there are parent-child relationships such as 2-parent-1-child (e.g., node 9), and 1-parent-2-child (e.g., node 0), all of which need to be expressed properly. To this end, we present *DepLinks* to support expression of generalized data-dependent parallelism. DepLinks is built on the abstraction of dependency graphs between tasks. In our framework, we partition a task as a single thread block (or CTA[2]) in hardware.

[2]We use thread block and CTA interchangeably

Code Block 2.4: Wireframe

```
#define parent1 dim3 (blockIdx.x-1, blockIdx.y, blockIdx.z);
#define parent2 dim3 (blockIdx.x, blockIdx.y-1, blockIdx.z);
void* DepLink() {
    WF::AddDependency(parent1);
    WF::AddDependency(parent2);
}
int main() {
   kernel<<<GridSize, BlockSize, DepLink>>>(args);
   cudaDeviceSynchronize();
}
__WF__ void kernel(args) {
    processWave();
}
```

Code Block 2.4 shows how wavefront parallelism can be expressed using DepLinks. In this scenario, we simply launch a kernel with a sufficient number of thread blocks to represent the entire dependency graph. One of the kernel launch options is a mapping function which defines the graph. This function consists of dependency links which are specified by *dim3* structures and its job is to specify the relative thread block on which any thread block is dependent. The dependency graph will then be generated by running the mapping function on every available thread block. For instance, in our wavefront example, every node is dependent on its north and west neighbors. This dependency graph will then be passed to the GPGPU hardware to enforce data dependency at run-time. In the next section, we will discuss this process in detail. Due to the fine-grained data-dependency that we can convey, individual tasks can run ahead and execute when parent tasks are complete. In this execution pattern, tasks are not constrained to waves. Overall, Wireframe enables a natural and more flexible way to convey data-dependent parallelism.

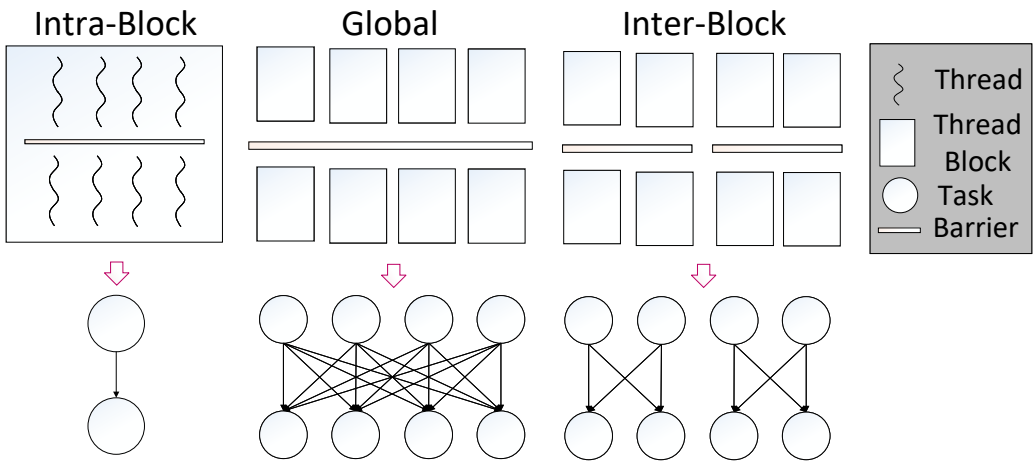


Figure 2.2: Synchronization barrier primitives using dependency graph abstraction: Intra-block (left), Global (middle) and Inter-block primitives (right).

2.2.2 Barrier Synchronization Primitives

As mentioned before, another major challenge of data-dependent parallelism is the lack of support for flexible barrier synchronization. Inter-block synchronization can ease programmer burden by granting programmers flexibility to convey synchronization between TBs, which has limited support in CUDA 9 with Cooperative Groups. Our synchronization primitives have similar support as Cooperative Groups, but we will later showcase how Wireframe can further eliminate stalls due to barrier synchronization by supporting a programming paradigm to avoid barriers completely. In this section, we demonstrate how dependency graphs can be used to form primitives that enable flexible lightweight synchronization across thread blocks. Figure 2.2 shows the supported synchronization primitives.

Intra-block synchronization: As shown in Figure 2.2 (left), intra-block synchronization implements a barrier among threads inside of a single thread block. This is achieved with `__syncthreads()` in CUDA . In our dependency graph abstraction, intra-

block synchronization can be conveyed through a 1-parent-1-child relationship between tasks. Using this dependency graph representation actually imposes greater overhead than `__syncthreads()` due to using 2 thread blocks to achieve this task. Therefore, we still rely on intra-block synchronization using the standard `__syncthreads()` call.

Global synchronization: In Figure 2.2 (middle), a scenario is shown where we assume the kernel consists of 4 thread blocks. Traditionally, in order to globally synchronize all thread blocks, we require implicit synchronization through consecutive kernel calls. This suffers from significant overhead due to the need for host-side kernel launches. Using the dependency graph abstraction, we can represent global synchronization using a dependency graph where each individual task after the barrier is dependent on every task before the barrier. In this example, global synchronization is represented as a 4-parent-1-child relationship. This lightweight global synchronization primitive completely eliminates the unnecessary host- and device-side kernel launches.

Inter-block synchronization: In Figure 2.2 (right), we illustrate inter-block synchronization with a scenario where thread blocks synchronize in pairs. This is similar to the global synchronization primitive where each individual task after the barrier is dependent on every task before the barrier, but constrained to a subset of thread blocks that are synchronizing. Supporting inter-block synchronization is a key component towards fully-supported data-dependent parallelism. What is unique about our approach is that this abstraction is deadlock-free. In prior work [141], a barrier is placed at the end of the thread block and wait for all other thread blocks to reach it. This results in some thread blocks staying in the SM, preventing other thread blocks from being scheduled in, and they will

subsequently cause a deadlock because they never got scheduled to be finished. Unlike [141], our inter-block synchronization primitive does not result in deadlock because parent thread blocks are allowed to complete and exit the SM, with barrier dependencies checked before a new thread block is issued to an SM.

2.3 Wireframe

Figure 2.3 shows an overview of the Wireframe framework. Wireframe consists of three main parts: DepLinks extensions to the CUDA programming model, dependency graph generation, and dependency graph execution in hardware through our dependency-aware thread block scheduler (DATS). The programmer can express data-dependent parallelism and barrier synchronization through CUDA programming model extensions. At kernel-launch-time, Wireframe would then retrieve the dependencies from the programmer via the API, create the dependency graph in Compressed Sparse Row (CSR) format and send it to GPU hardware. Once the CSR is received by the hardware, the GPU will make use of the dependency graph to enforce data-dependent parallelism when scheduling TBs.

The interface between the software and hardware is simply an abstraction of task dependencies represented as CSR. Therefore, our framework is not tied to a specific programming interface. The dependency information between tasks can be in the order of hundreds of MBs, limiting prior dependency-based task scheduling to software run-times [18, 51, 128, 141] with significant overheads. Wireframe, to the best of our knowledge, is the first efficient hardware solution to support and manage dependency-based task scheduling with only 2KB hardware overhead.

Note that the focus of Wireframe is on efficient hardware support of statically generated dependency graphs. There are currently many efforts in various compilers and programming paradigms to convey task dependencies in CPUs [44, 47, 48, 112, 131]. For example, OpenMP [48, 112] contains extensions to define tasks and dependencies using the `depend` clause. This information is utilized to create a directed acyclic graph of the tasks. Till date, there is no software run-time-agnostic API for GPUs to convey task dependencies. This chapter makes an argument for dependency awareness extensions to CUDA, and demonstrates the potential benefits.

As the main focus is on hardware support for dependency graph execution, we propose a simple API in order to convey static task dependencies. It will generate a CSR dependency graph, which enables easy interpolation with any future task dependency programming paradigms. Automatic graph generation is explored as part of BlockMaestro, the follow-up work presented in Chapter 3.

2.3.1 DepLinks API

In this section, we present our DepLinks API and how it maps the TBs to the nodes in the dependency graph. Later, we discuss the in/out dependency concept from OpenMP which could also be used to profile and generate dependencies. DepLinks requires API calls for scheduling policy assignment, inter-block synchronization, and assignment of parents for every thread block. In addition, DepLinks supports executing different kernels with different dependency graphs.

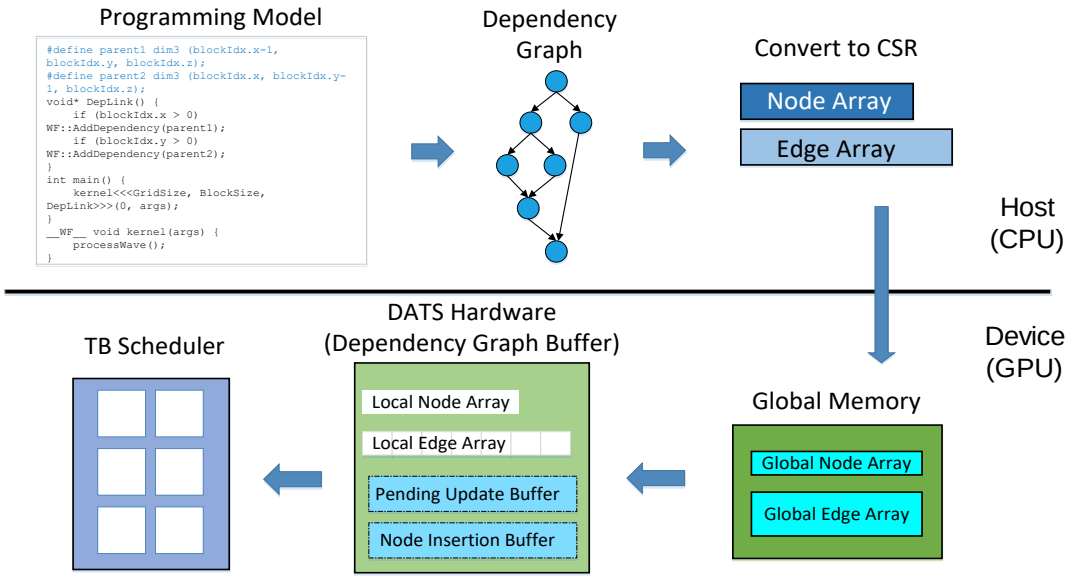


Figure 2.3: Overview of Wireframe; programmer supplied dependency constraints are translated into a dependency graph at run-time and conveyed to the GPU, where it is scheduled for execution through the DATS hardware.

We demonstrate our API in Code Block 2.5. The code block implements a kernel, `kFunction` (line 22). The kernel calls are extended with a mapping function, `DepLink` (lines 4-11). The kernel maps a wavefront dependency graph similar to Figure 2.1. For every thread block in the kernel, it will call the mapping function to identify its parent dependency. In wavefront dependency, each node is dependent on its west (`x-1`) and north (`y-1`) neighbors. We have defined this in lines 1 and 2. However, the thread blocks do not always have identical dependency patterns. In that case, conditional statements could be utilized to differentiate dependencies related to different groups of blocks.

Code Block 2.5: Wireframe API Example

```
#define node1 dim3 (blockIdx.x-1, blockIdx.y, blockIdx.z);                1
#define node2 dim3 (blockIdx.x, blockIdx.y-1, blockIdx.z);                2
                                                                          3
void* DepLink() {                                                         4
```

```
        //Add dependency for every thread block                          5
        WF::AddDependency(node1);                                        6
        WF::AddDependency(node2);                                        7
                                                                         8
        //Set the policy for the hardware                                9
        WF::SetPolicy(WF::LVL,4);                                        10
}                                                                        11
                                                                         12
__WF__ void kFunction(<args>)                                            13
{                                                                        14
   //Do kernel execution                                                 15
}                                                                        16
                                                                         17
void main()                                                              18
{                                                                        19
   //Launch kernel kFunction()                                           20
   dim3 dimGrid(4,4,1), dimBlock(16,16,1);                               21
   kFunction<<<dimGrid, dimBlock, DepLink>>>(<args>);                    22
}                                                                        23
```

The `AddDependency()` call will map the dependencies to the thread block. In addition, in line 10, the thread block scheduling policy is specified as level-bound (LVL) with a range limit of 4, i.e., running TBs in the graph cannot be more than 4 levels apart. Overall, it is possible to declare wavefront dependency pattern in less than 10 lines of code. Similar to OpenMP depend clause, we only need to specify each edge in a dependency graph. Using this simple, yet flexible, API, we can also easily implement any synchronization barrier primitives shown in Figure 2.2.

Note that our API function implements boundary checking to handle invalid arguments. For example in the `DepLink()` function, for block ID (1,0,0), `node1` will have negative elements in which the API will correctly handle and ignore. Similarly, block ID (0,0,0) will have no parent nodes.

Profiling-based generation: The OpenMP depend clause provides a list of dependent inputs and outputs for each task. This dataflow information is then utilized to generate a DAG. Similarly, kernel calls in the global barriers implementation follow a similar pattern, with input and output data to the kernel managed by `cudaMemcpy`. Therefore, it is feasible to extract data dependencies from the global barriers implementation by obtaining the dataflow between the kernels without programmer intervention. Due to the indexing nature of TBs in CUDA programming, we can also profile the dataflow between thread blocks to identify dependencies and generate the dependency graph. In prior work [44] parallel task-based dependencies were extracted from sequential programs using a similar technique.

2.3.2 Dependency Graph Generation

At kernel-launch-time, the program creates a static dependency graph based on the programmer-supplied dependencies. In order to pass this information to the GPU in a compact manner, we chose to represent the dependency graph in modified compressed sparse row (CSR) format. The API gives us a list of nodes and edges from which we can generate the CSR with time complexity of $O(|V| + |E|)$.

Our dependency graph CSR representation is shown in the upper half of Figure 2.6. CSR consists of two arrays: a Node Array and an Edge Array. Every Node Array entry corresponds to a node, with three fields: Edge Start, Parent Count, and Level. How they are used is explained in the Section 2.4.2.

In Figure 2.6, the numbers in the Node Array correspond to the start indices in the Edge Array. For example, node 0 has child nodes 1 and 2, node 2 has child nodes 3

and 4, etc. Our customized CSR array contains the number of nodes in the Node Array, the edge start and edge count for every node (in short, location of child nodes in the edge array and the number thereof), the number of edges and the nodes to which they lead.

A major challenge of using dependency graphs is their size. The size of the dependency graph is arbitrary and can be very large in the order of MBs. So the full CSR should be stored in the global memory (or constant memory if size permitting) of the GPU. However, the thread block scheduler requires dependency information from the CSR in order to schedule thread blocks, which can be very slow with global memory access. To overcome this challenge, we will exploit spatial locality behaviors of actively executing nodes and their immediate child nodes in the dependency graph.

Dependency Graph Execution Properties

We observed that during the execution of data-dependent parallel applications, there exists spatial locality of actively executing nodes and immediately dependent nodes. In our dependency graph, there are no explicit or implicit barriers across different levels of the dependency graph. Due to the fine-grained dependency representation, it is possible for ready nodes to process ahead even when prior levels of the dependency graph are not fully processed.

Despite this freedom, we observe that there exists a narrow *window* of levels in which active nodes are executing. We demonstrate this in Figure 2.4. We ran the HEAT2D application with a dependency graph of 9216 nodes and 191 levels in GPGPU-Sim [16]. During run-time, we measured the level range of active tasks over the course of the application run. We observed that even though the dependency graph has 191 levels, the

level range of the active nodes grows no more than 7. We found this behavior common in data-dependent parallel workloads.

Using this key observation, we can buffer only a small subset of the dependency graph in the thread block scheduler to effectively support a dependency graph of any size, while still enabling the thread block scheduler to quickly keep track of dependency statuses at run-time. We will discuss this hardware mechanism in detail in Section 2.4.

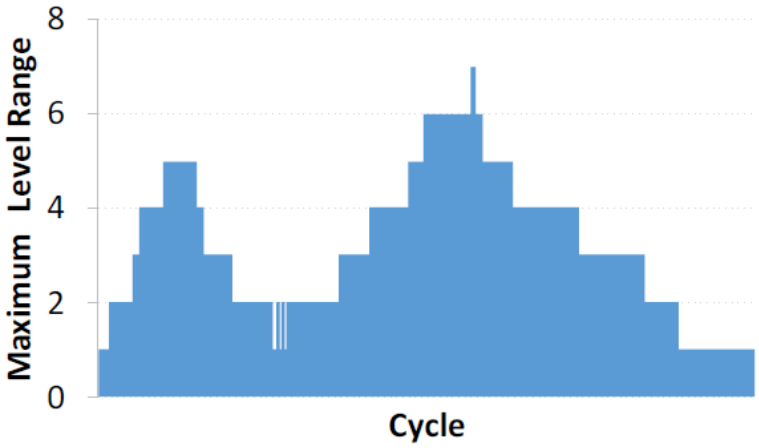


Figure 2.4: Level range during HEAT2D application.

Dependency Graph Node Renaming

By buffering subsets of the dependency graph, we are exploiting level locality. However, the current dependency graph and CSR format may not be amenable to buffering as CSR stores tasks in sequential node ID order (as defined by thread block IDs). In order to efficiently buffer the dependency graph, we need sequential ordering of levels and node IDs. As shown in Figure 2.5 (left), the dependency graph for the wavefront application in Figure 2.1 does not exhibit sequential level-by-level numbering. Therefore, access to

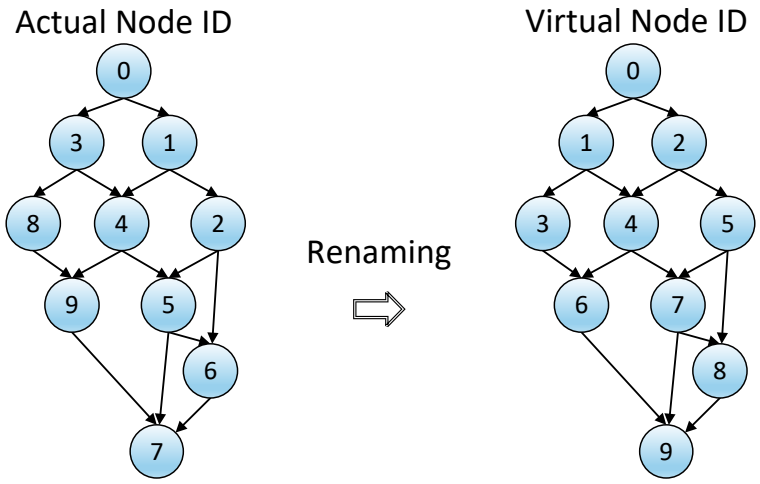


Figure 2.5: Illustrative example of node renaming

the CSR will result in non-contiguous global memory access, which also introduces major complexity issues when fetching nodes to buffer, as well as the management of the buffer. To overcome this, we perform a sequential level-by-level renaming transformation to the dependency graph as illustrated in Figure 2.5 (right).

Rather than changing the actual thread block IDs, we rename the node IDs. The dependency graph will be analyzed, every node's parents, children and level will be determined, and then each node will be assigned a *virtual ID* (VID), which will be used exclusively by the thread block scheduler. The original thread block ID remains intact and is used as normal. The procedure is similar to breadth-first search (BFS), and is performed at runtime. In the beginning, all nodes with no parents will be considered level 0. We then move down the graph and assign the child nodes recursively. For every child node with exactly one parent, the level of the child will be: $lvl_{child} = lvl_{parent} + 1$. For a general case where there are N parents, the level of the child will be: $lvl_{child} = 1 + \max\{lvl_{parent_i}\}, 1 \leq i \leq N$.

When we move from every parent to a child, it increments the *parent counter* in the child node, which represents the number of parents for that node when this process is finished. This will be used in TB scheduling shown in the following sections.

2.4 Dependency-Aware Thread Block Scheduler (DATS)

In the previous sections, we described the DepLinks programmer interface and how dependency graphs are generated. In this section, we describe how the GPGPU hardware enforces dependencies through a dependency-aware thread block scheduler (DATS). As described in the last section, we use the CSR format to store the nodes and edges of a dependency graph, which is generated at runtime and transferred to the GPU's global memory. The CSR representation of the dependency graph in Figure 2.5 is illustrated in the global memory section of Figure 2.6.

2.4.1 GPGPU Architecture Overview

We target an NVIDIA Fermi-like architecture modeled after GTX480. Our architecture comprises of 15 streaming multiprocessors (SM), where every SM in Fermi can execute up to 1536 threads or 8 thread blocks (CTAs). A thread block scheduler is responsible for issuing any ready thread block to an available SM. The technique that we present here is agnostic to the GPGPU microarchitecture and is self-contained within the thread block scheduling mechanism. Kernel parameters are stored in the global memory. There is already a communication path between the global memory and the kernel management and distribution unit [134] to allow transfer of the CSR into the thread block scheduler.

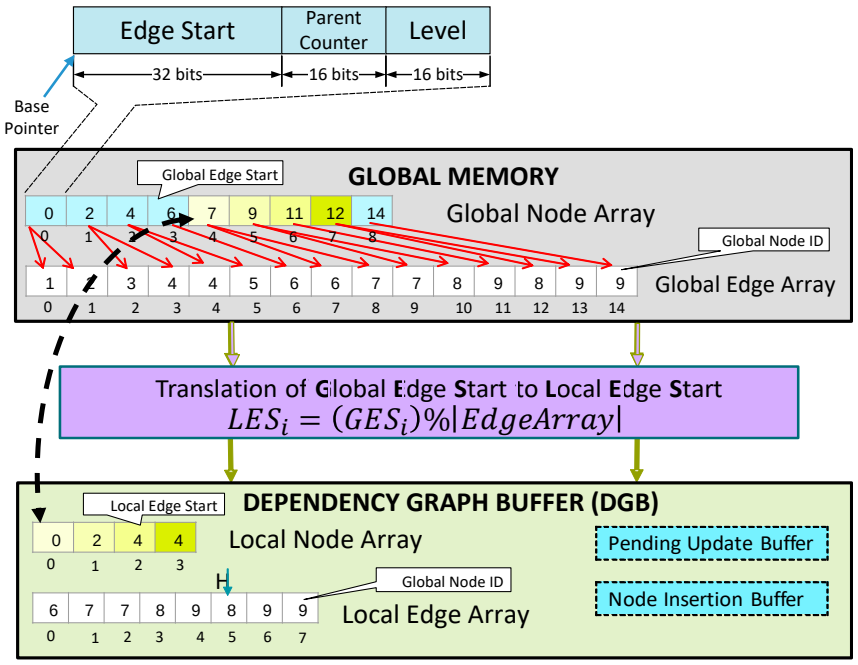


Figure 2.6: Connections between the global memory and local Dependency Graph Buffer for the graph in Figure 2.5.

2.4.2 Dependency Graph Buffer

The Dependency Graph is stored in CSR format in the global memory. It is infeasible for the TB scheduler to keep track of the graph node states in the global memory. Therefore, we propose the addition of a *Dependency Graph Buffer (DGB)* into the TB scheduler to buffer a subset of the CSR, sized large enough to keep the execution flowing and prevent it from stalling. In Section 2.3, we observed that there is spatial locality in the actively executing and immediate dependent nodes of the dependency graph. Therefore, we can buffer this active *window* of nodes of the dependency graph in the Dependency Graph Buffer in order to keep track of dependency states in a low-overhead manner.

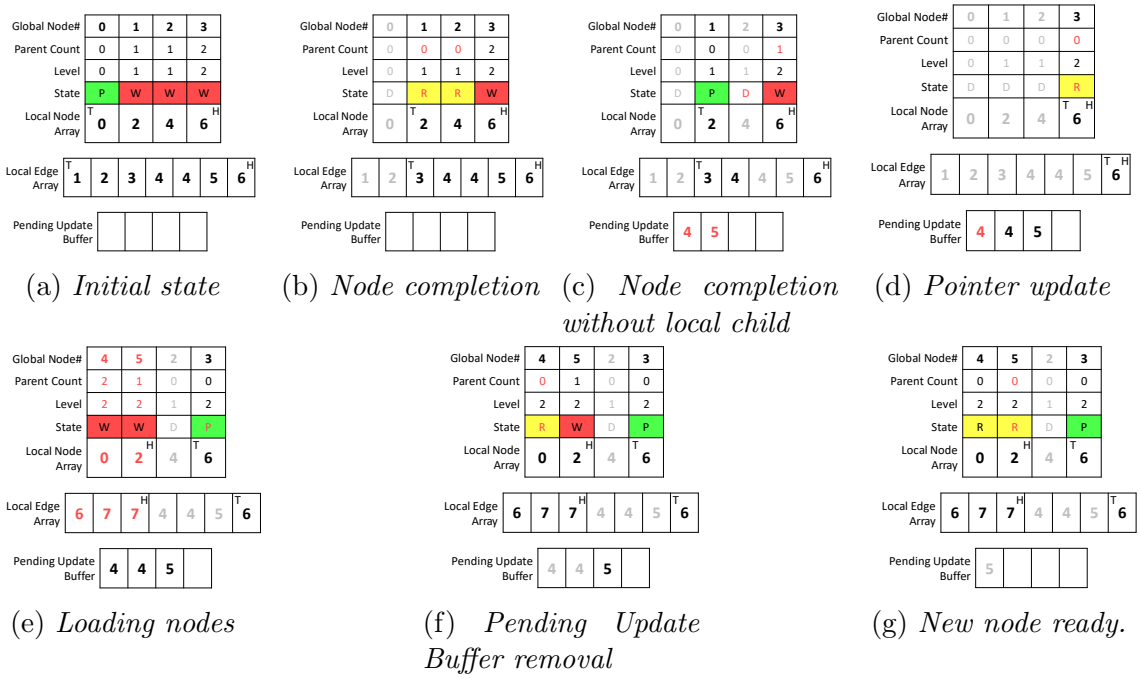


Figure 2.7: Management of Dependency Graph Buffer.

The Dependency Graph Buffer is showcased at the bottom of Figure 2.6. The Dependency Graph Buffer consists of Local Node Array, Local Edge Array, Pending Update Buffer, Node Insertion Buffer, and Node State Table (not shown, details in Figure 2.7). The Local Node Array and Local Edge Array are implemented as circular buffers. The Node State Table is tightly coupled with the Local Node Array, where every node entry of the Local Node Array has a corresponding entry in the Node State Table. The node and edge values for the relevant dependency graph are stored in the global memory in the format shown at the top of Figure 2.6.

When moving portions of the global node/edge array into the local node/edge array, we re-index the global edge start to a local edge start. This is done using a simple modulus-based mapping function to minimize the size of the local node array entries. The

local arrays will be loaded from memory in bursts of 128 bytes, the memory request size, to maximize memory load utilization.

Node State Table

The Node State Table is shown in Figure 2.7 and contains the following fields:

State: Signifies whether the node is Waiting (W), Ready (R), Processing (P), or Done executing (D). Initially all nodes are initialized to the Waiting state, except for the nodes with no dependencies which are set to Ready.

Parent Count: For every node, it shows number of unfinished parent nodes. It is computed in the host and transferred to and stored in the global CSR memory at run-time.

Level: The maximum distance of every node from a root, i.e., a node with no dependency. It is also computed at the host, to be used for thread block scheduling as discussed in Section 2.4.3.

Global Node #: The virtual node ID of the dependency graph, which indexes into a node in the Global Node Array.

Local Edge Start: The address of the first child of the node in the local edge array. If there were no children, it will be set to -1. The number of children can be determined by finding the difference of two consecutive edge starts.

Dependency Graph Buffer (DGB) Management

We will now demonstrate the operation of our DGB management mechanism through the use of the DGB structure shown in Figure 2.7.

Transferring Nodes/Edges from Global Memory to DGB: The hardware fetches *chunks* of nodes and edges from the global memory into the DGB. A chunk is defined as the number of the entries of the Local Node array which fit in a single memory request from the global memory (128B) to the DGB. In our running example, a chunk is 2 node entries, the Node Array size is 4, and Edge Array size is 12. During the transfer, the *edge start* field of the local node array is re-indexed so they point to the local edge array directly. The Local Edge array, on the other hand, will keep the global node IDs so they can be used to update the parent counter of children nodes. To illustrate this, let's look at global node ID 4. In Figure 2.6, this refers to index 4 of the Global Node Array, which contains a Global Edge Start of 7. The neighbor node (ID 5) has a Global Edge Start of 9, which means node 4 has 2 children. At index 7 of the Global Edge Array we see that node 4 has node 6 for a child, and at index 8, node 4 has child 7. Once node 4 is transferred to the DGB, as illustrated at the bottom of Figure 2.6, ID 4 has a transformed local edge start of 0, which points to index 0 of the local edge array. Index 0 and 1 of the local edge array contain the global node ID of children 6 and 7.

Translating Global to Local: The translation of the edge start from the global memory to the local memory is modulus-based: $LES_i = (GES_i)\%|LEA|$, where LES_i is the translated Local Edge Start for node i, GES_i is the Global Edge Starts for nodes i and $|LEA|$ is the size of the Local Edge array. Let us use Figure 2.6 as an example, where $|LEA| = 8$. Suppose that nodes 0 to 3 are already inserted in the DGB along with their edges, so they are both full. Then nodes 0 and 1 finish and, as a result, are invalidated in the DGB. Since the chunk size is 2, nodes 4 and 5 load into the DGB. As

the Global Edge starts for nodes 4 and 5 are 7 and 9, their new Local Edge starts will be $LES_4 = 7\%8 = 7, LES_5 = 9\%8 = 1$ respectively.

The local node address for node i is also modulus-based. At the time of the node's insertion from the global memory into the dependency graph buffer, the operation is performed in the following address: $LNID_i = i\%|LNA|$, where $LNID_i$ is the local address for node i at the time of its insertion and $|LNA|$ is the size of the Local Node array. Since both i and $|LNA|$ are known before the node's insertion into the Local Node Array, the hardware can predict the future location of any node in the said array. In the event of any new node transfer, the hardware will compare the global node ID of the new node and the target location to check if the latter is indeed unused. If the location is occupied by a prior node, it would terminate the memory transfer and put the node in the node insertion buffer until the space becomes available.

Handling Head Pointer Node Once the nodes are inserted into the graph buffer, ready nodes can be issued in any order. The only exception is the last node pointed by the head pointer. When a ready node completes, it decrements the parent counter of each children. In order to do so, we must know the number of children each node has by subtracting its local edge start from that of the next node's local edge start. For the last node pointed by the head pointer, it cannot determine the number of children due to the absence of a next node. We handle this scenario by not scheduling the head pointer node, unless the node is childless, e.g., the last node.

Node Completion Figure 2.7b illustrates the scenario where a node completes execution. This running example starts with nodes 0-3 in the DGB, with node 0 executing

as shown in Figure 2.7a. Since node 0 has no parents, it has level 0 and will be the first to execute. When node 0 completes, we first fetch the children of node 0 (node 1 and node 2). Each child is accessed and their parent counter is decremented. Once a parent counter reaches 0, it indicates that all dependencies are met, and its state is updated to ready.

After decrementing the parent counters, the entry associated with the node which finished is invalidated as shown by the lighter text. Recall that the Local Node/Edge Arrays are circular buffers. As the tail entries are invalidated, we move the tail to the next valid entry. In this case, the tails moved to Local Node Array index 1 and Local Edge Array index 2. In addition, the execution of thread blocks can be completed out of order in the array as shown in Figure 2.7c, where node 2 has finished executing while node 1 is still being processed. We only move the tail if the tail's entry is invalidated.

Pending Update Buffer Insertion: Note that when node 2 finishes, the children nodes 4 and 5 are not in the node status table, and thus we cannot decrement their parent counters. To overcome this overflow, we add a *Pending Update Buffer (PUB)* to handle the situation where the child node is not in the Local Node Array. The PUB stores the global node ID of the child. This is illustrated in Figure 2.7c, where node 2 has finished executing and attempts to update the parent counters of its children, nodes 4 and 5. Since neither of those nodes is in the graph buffer yet, it will use the PUB to save the changes so they can be applied later. Note that if the buffer is full, the hardware cannot mark the node as complete if it has children. Therefore, it has to wait until there is enough space before the node's execution can be finalized.

Loading Local Node/Edge Array Entries As shown in Figure 2.7d, node 1 will complete, and decrement the parent counter for its children nodes 3 and 4. Node 3's parent counter is now at 0 and its state is updated to ready. Node 4 is not in the Node State Table, and is thus put into the PUB. At this point, the entries for node 1 and 2 are invalidated and the tail advances to index 3 (node 3). At this point in time, there is enough empty space in the Local Node Array to load a new chunk of nodes. We can keep track of the available space using the distance between the head and tail pointers.

A memory request is issued to the global memory and the next chunk is fetched from it. From the head pointer in the Local Node Array, we can generate a memory access to load the next node based on its ID:

$$GlobalBaseAddress + NodeID \times NodeEntrySize.$$

The Global Node Array entries contain the global edge start, which points to the Global Edge Array. Memory requests are iteratively issued to fetch the Global Edge Array entries with memory address location calculated similar to accessing Global Node Array.

The hardware loads a new chunk from the global memory into the node array where the head pointer is, followed by the associated data in the edge array, starting from the edge head pointer. This is depicted in Figure 2.7e. A node's insertion is only finalized if there are enough spaces for its edges in the Local Edge Array. Otherwise the node shall be put in a temporary *node insertion buffer* to wait and the loading process halts. The next nodes will not also be loaded until enough space for the node in question and its edges is available in the Local Edge Array, in which case loading will resume. If the node's insertion

into the local memory is successful, the head pointer will then also move. Note that the edges to which the nodes will be pointing have been translated to their local counterparts beforehand as described earlier in this section.

Pending Update Buffer Removal: Figure 2.7f shows the scenario when nodes 4 and 5 are loaded into the Node State Table. At this point in time, the update buffer contains two updates for node 4 (one each from completion of node 1 and 2), and one update for node 5 (from node 2). When a node with a registered ID in the update buffer is loaded into the Node State Table, the parent counter update will be applied and the entry in the update buffer removed. For example, in Figure 2.7f, the two pending node 4 entries decrement the parent counter of node 4 to 0, changing the state of node 4 to ready. Similarly, the update to node 5 will also be applied, marking it as ready (Figure 2.7g).

2.4.3 Level-bound Thread Block Scheduling

Up until now, we have described how dependencies between thread blocks are enforced and managed in hardware. We will now discuss how ready thread blocks are scheduled to SMs. We first present the baseline thread block scheduling policy, and then motivate the need for a thread block scheduling policy for dependency graphs.

The baseline default policy is *Loose Round-robin (LRR)*. It first selects a ready node with the smallest ID, and cycles among all the SMs, selecting the next SM to issue to. If the intended SM is already full, the policy will move to the next available SM. This policy attempts to evenly distribute the workload among the available resources. However, this scheduler is very simplistic and does not account for dependency graph execution dynamics, which leads to performance hindrance.

We will again borrow the wavefront example from Section 2.2 to illustrate dependency graph execution dynamics. In coarse-grained synchronization scenarios (global barrier, CDP), each *level* of the dependency graph is executed until completion, one after another. As a result, nodes ready in subsequent levels cannot be scheduled and must wait until the preceding level is complete, limiting performance. Dependency graph execution allows any ready node to run ahead and execute without having to wait until the prior level is completed.

However, we observed that if nodes run too far ahead, it can end up hampering the performance. This is illustrated in Figure 2.8. Here 8 nodes (marked with 'D') have completed execution. On the left, we depict a potential scenario with the baseline LRR policy where nodes can run ahead unbounded. There is significant run-ahead, with ready nodes ('R') spanning a *level range* of 4 (in levels 3, 4, and 7). Due to a single path running ahead, it can potentially limit the number of ready nodes. In the case of wavefront, there are significant data-dependencies with most nodes dependent on 2 parents from the previous level. If a dependency graph observes a high level range, it means that neighboring nodes may have more dependencies pending. For example, in Figure 2.8, the numerical values within the immediate neighboring nodes of completed and processing nodes represent the levels of dependencies that must be resolved before that node can run. Due to the run-ahead, neighboring nodes have a longer chain of dependencies with less nodes ready in the near future (only 3 nodes have 1-level dependency).

To this end, we propose *Level-bounded (LVL)* thread block scheduling, which extends the LRR thread block scheduler by bounding the level range to satisfy dependencies

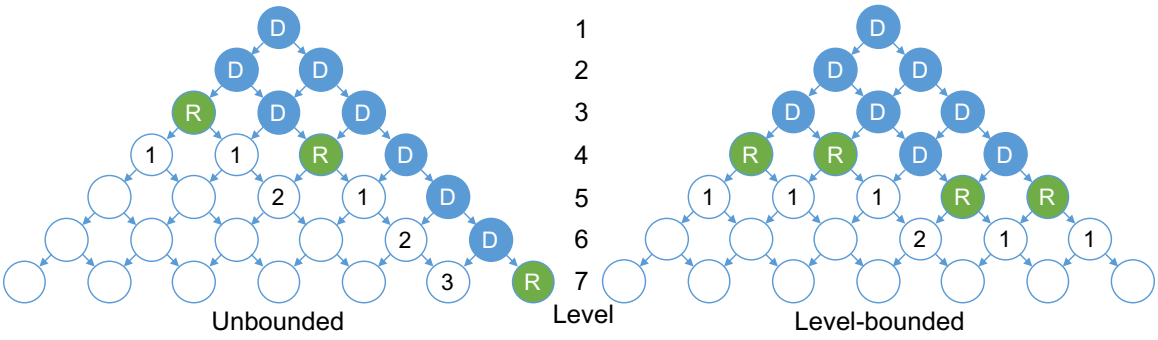


Figure 2.8: Effect of thread block scheduling on Dependency Graph node availability.

quicker. This results in greater ready node availability as shown in the figure. Under level bounding, we have 5 nodes with 1-level of dependency, and also 4 ready nodes. Intuitively this scheduler operates in the following manner: If a path runs ahead too far (reaches a level range limit), the Level-bounded scheduler will prevent that path from proceeding further and favor scheduling nodes from slower paths to allow the level range to narrow. Bounding the level range promotes completion of node dependencies, resulting in more ready nodes than the baseline unbounded scenario.

2.5 Evaluation

2.5.1 Methodology

We evaluate Wireframe on GPGPU-Sim v3.2 [16]. We use the default NVIDIA GTX480 configuration with 15 SMs, each having 8 CTAs, 128KB register file and 16KB L1 cache size. The shared L2 cache size is 786KB. The warp scheduling policy follows a greedy-then-oldest (GTO) policy [108]. Our thread block scheduling technique can be run with any warp scheduler, but we find GTO to provide the best performance. We modeled the

device-side kernel launch overheads by implementing the latency model proposed in [134]. We measured empirically and used the host-side kernel launch time of 30μs. The baseline machine runs at a core clock of 700MHz, where each SM consists of 2 shader processors (SP), each containing 32 CUDA cores, 16 LDST units and 4 SFUs.

We utilize a selection of data-dependent heavy workloads. For each workload, we implement four versions: Global Barriers (Global), CUDA Dynamic Parallelism (CDP), DepLinks synchronization primitives (DepLinks), and Wireframe with the LRR scheduler (LRR) and Level-bound scheduler (LVL). For the level-bound scheduler, we use a level bound of 3. Note that DepLinks enables barrier synchronization primitive support through task graph representation and does not change the way TBs are assigned to SMs. LRR and LVL, on the other hand, do not enforce any barrier behaviors, but rather control the TB assignments, allowing nodes with satisfied dependencies to execute, enabling them to run-ahead instead of waiting for other nodes at their level to finish first.

We verified the output of each workload implementation against the original to ensure output correctness and that dependencies are satisfied safely. Unless otherwise stated, we partition the workload with up to 4K nodes in the dependency graph. We will later explore the impact of the size of dependency graph on performance. In addition, we used a Local Node Array size of 128 entries and a Local Edge Array size of 512 entries for LVL scheduler, and 512 entries and 2K entries for LRR scheduler, respectively. We set the size of the Pending Update Buffer to 64 entries.

2.5.2 Benchmarks

The benchmarks used are DTW (Dynamic Time Warping) [86], HEAT2D [110], HIST (Histogram) [101], INT_IMG (Integral Image) [23], SOR (Successive Over-Relaxation) [38] and SW (Smith Waterman) [111]. DTW is a common algorithm in time series analysis for measuring similarity between two time series with varying speeds. DTW takes in two time series, one of size 12K and one of size 8K. HEAT2D is a common solver for heat equations in two dimensions. At every iteration, the temperature of each point is dependent on neighboring points. We use a 2D grid of size 12K x 12K. HIST calculates the integral histogram over a 13MP bitmap image. INT_IMG is an image processing technique that generates the sum of values in a rectangular subset of a grid. We similarly use INT_IMG with a 13MP bitmap image. SOR is a linear system solver which is implemented using a generic 5-way stencil pattern. We use a random 2D matrix with 144M entries as input. SW is a common local sequence alignment algorithm. We input two 8K strings. We verified that the size of the data is sufficient to utilize the entire GPU (maximize hardware CTAs, cache, etc.) with each workload's data set size in the order of hundreds of MBs.

2.5.3 Evaluation Results

Performance: Figure 2.9 illustrates the speedup for all implementations with respect to Global Barriers. Speedup is the ratio of the total execution time and kernel launch overhead for a given technique, with respect to the baseline global implementation. It shows how much every technique addition, up to LVL, is responsible in improving the performance. In all scenarios, CDP and our proposed techniques outperform global barriers

by removing costly host- and device-side kernel launches. CDP has an average speedup of 6.87%. DepLinks further remove device-side kernel launches and improve average speedup by 25.07%. Wireframe further enables task run-ahead. On average for Wireframe, LVL outperforms LRR (31.81% vs 29.81%). In certain scenarios, such as HIST, LVL performs slightly worse due to limited improvement to level range properties.

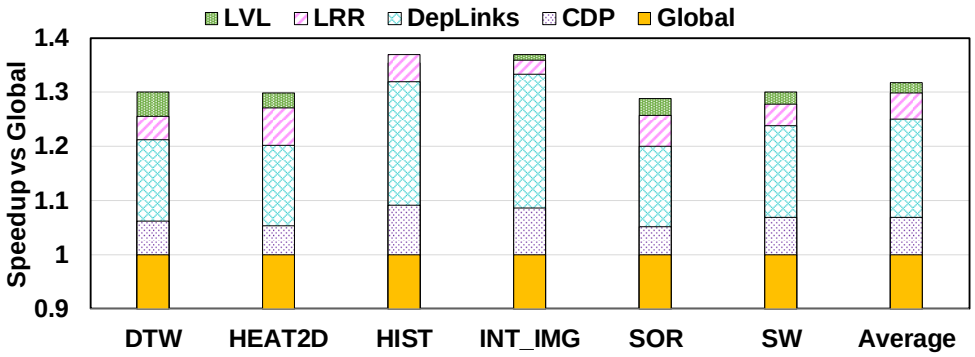


Figure 2.9: Normalized Speedup w.r.t. Global Barriers.

Memory Overhead: Figure 2.10 (left) shows the memory request overhead introduced by DATS. At most, DATS introduce 0.16% memory request overhead, with an average overhead of 0.12%. Despite making use of the global memory, Wireframe does not have a substantial negative impact to L2 cache performance as shown in Figure 2.11. In addition, the miss rate is consistent regardless of the programming model of Wireframe used (DepLinks vs LRR/LVL).

Level Range: Figure 2.10 (right) shows the impact of the LVL scheduler, with level bound of 3, on the maximum observed level range. We utilize a dependency graph of 9K as the benefits of level-bounding is more apparent with larger graphs. The observed

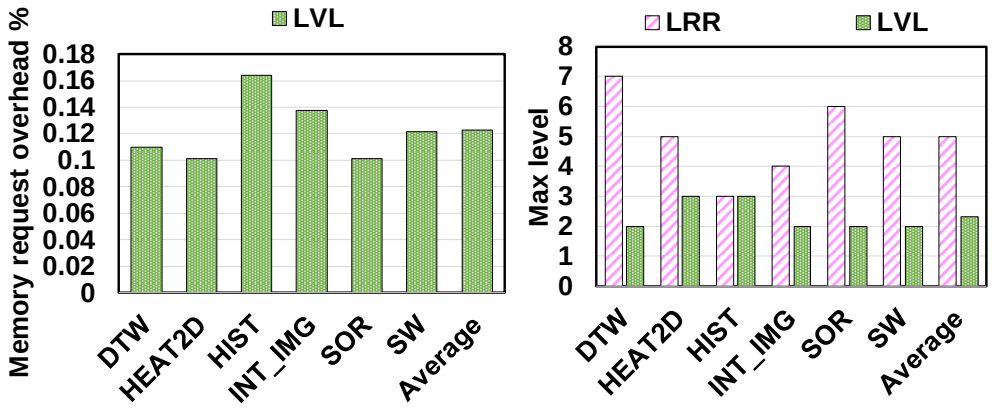


Figure 2.10: Memory request overhead (left) and maximum level range (right).

level range can actually be less than the bound. For example, GTO warp scheduler focus on the warps of thread blocks on the lower levels (older TBs) so they can finish faster, resulting in a lower range than anticipated. In certain scenarios, the level range reduced drastically from 7 to 2 (DTW) and 6 to 2 (SOR).

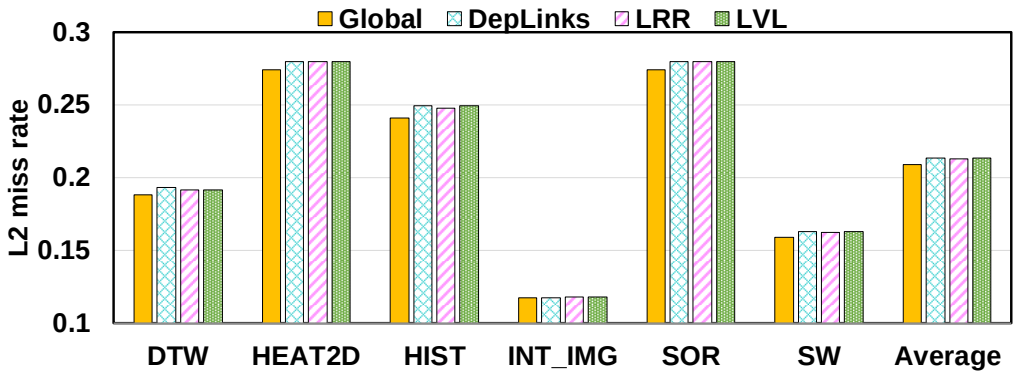


Figure 2.11: L2 miss rate in Wireframe.

Dependency Graph Size: Figure 2.12 (left) shows the effect of dependency graph size on overall speedup for the level-bound scheduler normalized to the global barriers implementation. As the size of the graph increases, there is generally more levels, and

greater opportunity for run-ahead. In addition, this is associated with removal of more global barriers. This can be observed as the average speedup increases as the graph size grows: 14.11% for 1K, 31.81% for 4K, and 45.07% for 9K dependency graph size, with a maximum speedup of 65.20%. Furthermore, Figure 2.12 (right) shows the computation time to kernel launch time ratio with a constant data size. Therefore, the ratio decreases as the graph grows. Notwithstanding, on average, we have significantly more computational time than kernel launch time, with an average of 8x, 5x, and 3x more compute with graph size 1K, 4K, and 9K, respectively. It can be seen that Wireframe does best with many smaller tasks rather than fewer larger ones.

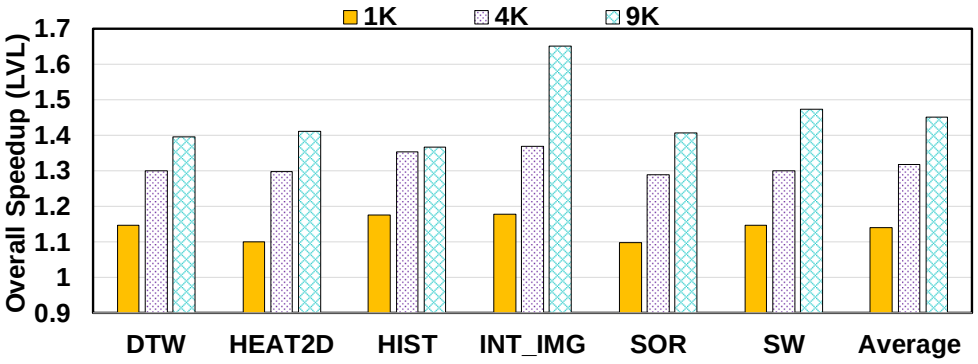


Figure 2.12: Overall speedup for same input size, but different graph sizes (left) and compute-to-kernel-launch ratio (right).

2.5.4 Overheads

Sensitivity to Local Node/Edge Array Size: Figure 2.13 (left) shows the maximum pending update buffer usage (solid line) and the IPC (dotted line) with respect to varying the Local Node Array size for LRR and LVL using the SOR benchmark. We

present SOR as it utilizes PUB the most. The LVL scheduler requires a notably smaller update buffer than the regular LRR scheduler. We use a node array size of 128 entries as it provides a high level of IPC for LVL, with a manageable PUB usage of almost 32 entries, which we pick as the best size for PUB. For a PUB usage of equal size, LRR scheduler requires Local Node Array size of 512.

Figure 2.13 (right) shows how the maximum pending update buffer usage changes as we decrease the Local Edge Array size. We set the Local Node Array size as before. For LRR, IPC falls as we use less than 512 entries. Thus, we select 512 as the best edge array size. LRR meanwhile requires over 1K entries. LVL scheduler can significantly reduce the size of the Local Node/Edge Array needed.

Dependency Graph Buffer Size: The DGB requires very little space. Each local node array entry needs 2 bits for state, 16 bits for the global node ID, 16 bits for the parent counter, 16 bits for the level and 9 bits to address the local edge array, i.e., 58 bits in total, which rounds up to 8 bytes for each node in the Node State Table and Local Node Array. As for the Local Edge Array, it only needs 16 bits per element to store the target's global node ID, for a total of 1KB. In addition, we have a Pending Update Buffer of 32 entries of 2 bytes each and a single Node Insertion Buffer of 128B, for a total of 256B. In total, the DGB has a size of 2304 bytes, which is negligible in comparison to the size of register file per SM (128KB).

DGB Access Overheads: We can access and update the DGB quickly due to the small size. Any timing overheads would occur due to fetching chunks from memory. However, this operation is off the critical path as fetching of memory chunks can occur as

TBs are executing on SMs. The only time there may be timing penalties due to our DGB mechanism is if there are no ready nodes due to nodes still being loaded from memory. Due to having a Local Node Array size of 128 entries, we observed that this scenario is rare as there are always plenty of other nodes to schedule. We observed the timing overheads of DGB to be less than 1%.

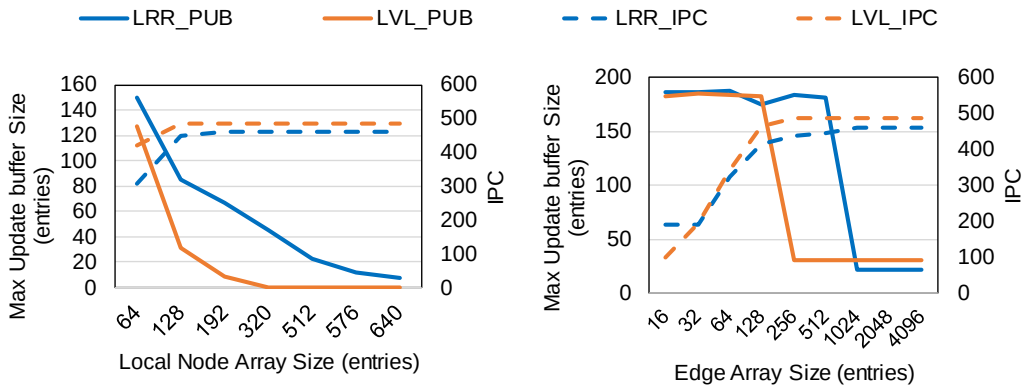


Figure 2.13: Effect of the local node size (left) and local edge array size (right) on the maximum update buffer size.

2.6 Related Work

Synchronization: Existing GPGPU programming models have been designed with support for coarse granular synchronization primitives (commands and streams in CUDA, events and pipes in OpenCL, and pipelines in OpenACC) to enable flow control across multiple kernel launches and data transfers. However, these primitives are at the device level and only able to provide coarse granular dependency management between host-initiated calls. Such constructs fail to address the data-dependency requirements across the threads within the execution of a kernel.

A finer granular in-GPU synchronization across the TBs of a kernel enables better utilization of SMs by allowing the dependencies to be resolved locally. One of the major issues which is often encountered in in-GPU synchronization is the deadlock problem [141]. Consider a case where there are many thread blocks with a global barrier, all parenting a single child kernel, as shown in Figure 2.2 (middle). If the number of thread blocks exceed the total number of CTAs that can run concurrently on all the SMs, at a point some thread blocks could be running on the SMs and hit the global barrier, whereas the others have never been dispatched, and, therefore, cannot context switch, causing the GPU to enter a deadlock state. In [141], the deadlock is handled by using atomic operations and memory flags. However, this situation will not occur in Wireframe due to the absence of global barriers. A similar method is to transform algorithms to remove global barriers, such as PeerWave [19]. However, this requires significant programming effort and is not general purpose. However, all of these techniques are software-based and result in significant run-time overhead.

Reducing Kernel Launch Overhead: In [135], authors proposed a locality-aware thread block scheduler to schedule child nodes to maximize cache locality within dynamic parallelism (CDP). However, the maximum recursion depth it can reach in any workload is limited to 24 [89]. Wireframe, however, support more complex parent-child relationships which are configurable by the user. This makes the whole execution flow more manageable and efficient. In [141], Xiao proposed an improvement of the subkernel launch using GPU lock-based and lock-free synchronizations. In [27], G. Chen et al. emphasize on re-using parent thread to operate on the data to be processed by the child kernel. In [123], Tang et al.

coordinates dynamically-generated child kernels to reduce launch overheads and schedules both parent and child kernels to improve launch overhead hiding. In our Wireframe work, we represent data dependencies through DepLinks rather than implicitly through kernels or barriers, and therefore, completely avoid kernel launch overheads.

Dataflow Scheduling: In addition to the GPU-related work mentioned above, the problem that Wireframe targets has generally been addressed in the architecture literature as "dataflow scheduling". Etsion et al. [42] have developed a superscalar, out-of-order task pipeline to execute dataflow programming models. Gupta et al. [50] utilized run-times to exploit parallel dataflow execution out of serial programs on multicores. Wang et al. [132] have implemented a task-level dataflow execution engine on FPGAs. More recently, Avron et al. [14] studied hardware task scheduling performance on Plural many-core-architecture. All these works present state-of-the-art examples for supporting dataflow based task execution on various platforms; however, none of them addresses the problem for GPUs.

Thread Block Scheduling: One of the works considering the CTA behavior for the scheduling decisions is OWL [60], in which the authors tackle the hardware under-utilization issue by prioritizing certain thread block groups and improving the cache hit rates. To the best of our knowledge, our scheduler is the first to target data-dependent parallelism.

2.7 Conclusion

In this chapter, we propose Wireframe, a general-purpose data-dependent parallelism paradigm which dramatically improves the performance on GPGPU by eliminating the need for global barriers and careful assignment of thread blocks as per the scheduling

policy. Wireframe has shown an average speedup of up to about 45% across multiple benchmarks, and, with the same input size, is seen to perform better if the dependency graph represents many smaller tasks rather than fewer larger ones.

However, Wireframe also has significant limitations. As mentioned before, it is upon the user to re-write the code to convert the application into a single-kernel format. That means ingraining an application's tasks into one kernel and providing the GPU with the dependency graph of the kernel's TBs, which requires significant effort for more complex and irregular applications, showcasing the low flexibility in Wireframe.

In the next chapter, we propose BlockMaestro, which mainly aims to minimize the user's burden through the use of code analysis, and allowing a fine-grained multi-kernel execution in the GPU.

Acknowledgments

The work in this chapter is partly supported by NSF Grants CCF-1423108, CCF-1513201. The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions.

Chapter 3

BlockMaestro: Enabling Programmer-Transparent Task-Based Execution in GPU Systems

3.1 Introduction

GPUs today are computationally powerful, power-hungry, and massively parallel devices, capable of processing applications using thousands of threads at once, taking advantage of its single-instruction, multiple-thread (SIMT) paradigm [5, 6, 13, 41, 59, 68, 69, 94, 127, 138, 146, 147]. As modern workloads grow in size and complexity, GPUs are stressed more than ever before [62,63,83]. For example, they are one of the main accelerators behind

modern machine learning frameworks [4, 32, 99] where typically every layer is encapsulated in a GPU kernel, and the main accelerator behind future exascale computers [35, 36, 120] where scientific computing applications make heavy use of iterative structured grid computations exhibiting wavefront parallelism [19, 20, 57], with multiple GPUs working together through specialized interconnects [76, 107].

These emerging workloads place significant burden on GPUs. By launching hundreds of kernels over the course of an application's execution, kernel launch overheads can become significant [28, 40, 55, 74]. These kernels also typically exhibit significant data dependencies between them [8, 20, 54, 58]. For example, layers in CNNs produce data that is consumed in the next layer. In stencil computations, which are common in scientific computing, operations performed on elements are dependent on the state of neighboring elements. These inter-kernel data dependencies are typically enforced in a coarse-grained manner through implicit barrier synchronizations in the form of kernel launches, which can result in stalling of computation that already have satisfied dependencies.

To circumvent these issues, many task-based execution models and runtimes have been proposed [8, 12, 20, 25, 39, 46, 54, 56, 65, 103, 135, 148]. These frameworks require programmers to decompose the application into *tasks*, and express task dependencies through proprietary programming models (such as AMD ATMI [12], CUDA Graphs [93], OpenMP Tasks [15], etc.), which will then be enforced by the runtime. The main benefit of task-based execution is that: (1) kernel launch overhead can be significantly reduced by collectively launching groups of kernels as a whole [12, 93] or by launching a persistent kernel which process tasks that enter its work queue [20]; and (2) dependent tasks can begin executing

as soon as their data dependencies are met. However, to gain these benefits, existing GPU applications must be refactored into these proprietary task-based programming models.

In this chapter, we propose *BlockMaestro*, which provide the benefits of task-based execution using existing SIMT programming models (such as CUDA or AMD HIP) and avoids the need for heavy code modification. The key insight behind BlockMaestro, is that *kernel pre-launching* and *fine-grained inter-kernel data dependency resolution* achieves the benefits of task-based execution models. By pre-launching dependent kernels, we are able to mask kernel launch overheads. To enforce correctness, thread blocks (TBs) of pre-launched dependent kernels are not executed until thread block-level data dependencies are resolved. Inter-kernel thread block data dependencies between neighboring kernels (which we denote as parent and child kernels, K_p and K_c, respectively) can be represented as bipartite graphs as illustrated in Figure 3.1. The entire GPU application can then be represented as a series of these bipartite graphs, collectively representing a task graph. We present a "Thread Blocks as Tasks" tasking paradigm that leverages the SIMT programming model's property where grids of thread blocks are inherently tasks with explicit input/output through global memory as defined in kernel launch parameters.

Therefore, the key to achieving the benefits of task-based execution is to automatically extract and enforce these bipartite dependency graphs while pre-launching dependent kernels. In short, this chapter makes the following contributions:

- We propose kernel pre-launching in order to mask kernel launch overheads of dependent kernels. In addition, we introduce command queue reordering to increase the opportunity for kernel pre-launching.

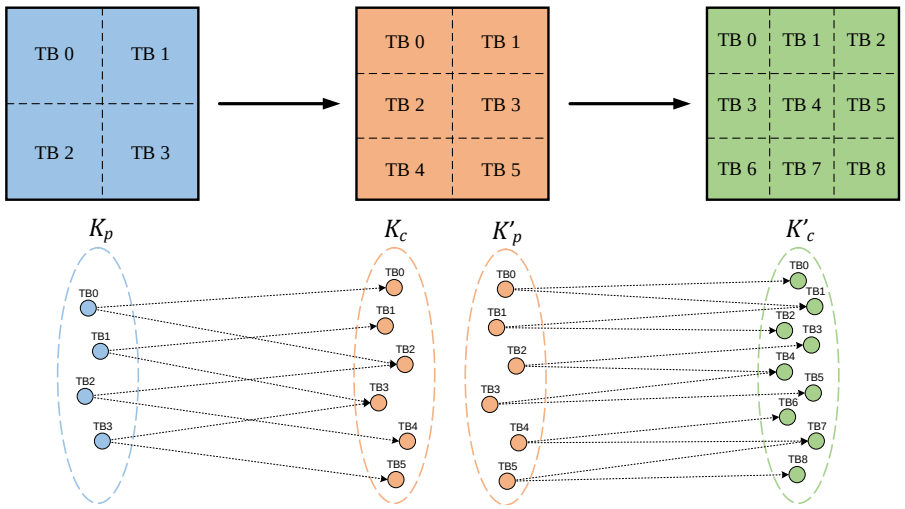


Figure 3.1: Data shared by the kernels constitutes dependencies among their TBs, shown as a series of bipartite graphs.

- We leverage compiler support to extract inter-kernel data dependency of existing GPU applications without the need for programmer intervention. The well-defined structure of GPU applications provided by the SIMT programming model allows us to extract data dependencies in the form of bipartite dependency graphs.

- We propose solutions to resolve fine-grained data dependencies between inter-kernel thread blocks. This ensures the correctness of pre-launched kernels and enables dependent thread blocks to start executing as soon as their data dependencies are satisfied.

In Section 3.2, we will provide background and motivate BlockMaestro. Next, we will explain the implementationd details of BlockMaestro in Section 3.3. We will then present the results of our evaluation in Section 3.4.

3.2 Background

3.2.1 GPU Execution

API calls and command queue: In GPU applications, the host calls a series of API functions to interact with the GPU. Common API calls include kernel launch, memory transfer to/from the host, synchronization, etc. All API calls are sent to a command queue (also known as *Stream* in CUDA and HIP terminology) for processing. The API calls (also known as *Events*) are serialized in the command queue with only one event being processed at a time. Therefore, only a single kernel may be executing from a single command queue. To support concurrent execution of *independent* kernels, kernels can be issued to multiple command queues and it is possible to synchronize kernels across command queues through complex synchronization events.

From the host's point-of-view, not all API calls are blocking. By default, memory operations, such as memory allocation and transfer to/from the GPU, are synchronous (blocking), and kernel launches are asynchronous (non-blocking). Therefore, when the host launches a GPU kernel, the host can continue to execute code, but it must explicitly synchronize and wait until the GPU kernel has completed before using the kernel's output. Therefore, the programmer should be aware of any dependencies between the host and the GPU, including read-after-write (RAW), and ensure proper synchronization.

Compiling GPU programs to assembly: GPU programs are typically written in high-level languages, such as CUDA, OpenCL, or HIP. During the compilation process, these programs are translated into assembly. For example, HIP is compiled into GCN assembly and CUDA is compiled into PTX, a form of intermediate language (IR), which is then just-

in-time (JIT) compiled at kernel-launch-time into SASS assembly. Depending on the target GPU and the language used, there is an offline compilation stage (HIP to GCN, CUDA to PTX) and potentially a second just-in-time compilation stage (PTX to SASS). The just-in-time compilation stage enables further optimization because additional parameters at kernel-launch-time, such as thread block size and grid size, are known and can be further optimized.

Kernel launch overheads: Due to the complexity in launching a computation kernel on the GPU, kernel launch overhead is not negligible. Prior works have found that each kernel launch can incur an overhead of $5-30\mu s$ [8,55]. To make matters worse, many GPU applications are also scaling in complexity and size. For example, modern machine learning frameworks that utilizes GPUs for compute-heavy operations (such as convolution) can incur hundreds of kernel calls as ML models grow. Many workloads also require significant synchronization which are implemented implicitly as kernel calls. Towards this end, many prior works have explored how to reduce kernel launch overheads [28,40,55,74]. (See Section 3.5 for details of prior works.)

Another common approach to reduce kernel launch overheads is to port programs in the SIMT programming model into a task-based programming model. Task-based runtimes can avoid kernel launch overheads and dynamically resolve data dependencies between tasks, for example, by using persistent kernels to process tasks in the work queue.

3.2.2 Task-based execution model paradigms

Many task-based programming models allow programmers to specify series of operations (tasks) and the dependencies between them. Existing task-based programming

models can be categorized broadly as following a *"Tasks as Kernels"* or *"Tasks as Thread Blocks"* paradigm. We will detail each paradigm and discuss the strength and weaknesses of each, and propose a new *"Thread Blocks as Tasks"* approach.

"Tasks as Kernels": In this paradigm, tasks in a task graph are mapped to kernels. For example, AMD ATMI [12] and CUDA Graphs [93] allow users to define kernels and the dependencies between them. To alleviate the effects of kernel launch overheads, these frameworks aim to identify common static operation graphs consisting of many kernels and consolidate the kernel launch into a single task graph launch. While these frameworks can lower the overhead from kernel launches, they fail to take advantage of fine-grained data dependencies that exist between kernels. For example, thread blocks in a dependent kernel may be ready to execute due to satisfied dependencies from thread blocks from the kernel before it, but cannot begin execution until their kernel is launched. Therefore, to handle these *dependency-stalled thread blocks*, finer-grained tasking paradigms are warranted.

"Tasks as Thread Blocks": A finer-grained approach to task-based execution is to map and execute tasks in a task graph as thread blocks. These task graphs can be defined by the programmer using a variety of task-based programming models [8, 20, 25, 46, 148] which defines tasks and their dependencies. At a high-level, these GPU task-based runtimes resolve dependencies between the tasks and then send ready tasks to a job queue, where they are processed by a persistent kernel. By using a persistent kernel approach, kernel launch overheads are avoided.

This fine-grained dynamic dependency resolution, along with persistent kernel, can reduce the amount of dependency-stalled tasks waiting for execution. However, there

are runtime overheads with task management and require significant programmer effort to map algorithms into new task-based programming models. Specifically, it requires the programmer to have domain-specific knowledge of the algorithm and be able to decompose the steps and express it in the form of a task graph.

"Thread Blocks as Tasks": The goal of BlockMaestro is to enable the benefits of task-based execution models with minimal programmer intervention. At the core of every task-based programming model is the ability to define a task graph for execution. Towards this end, we propose a "Thread Blocks as Tasks" paradigm where, instead of programmers defining tasks which are mapped and executed as thread blocks, we extract and derive tasks and task graphs from the existing thread blocks in the SIMT programming model. We take the view that *grids of thread blocks are essentially tasks with explicit input/output through global memory as specified in kernel launch parameters.* We leverage the properties of multi-kernel GPU applications in order to build a task graph from a series of bipartite dependency graphs (essentially, a decomposed task graph). To alleviate the overhead of kernel launch overheads, we propose pre-launching kernels before their dependencies are met and relying on dynamic data dependency resolution in hardware to enforce dependencies. This effectively removes the kernel launch overhead from the critical path by masking the kernel launch.

Towards this goal, our main challenges in achieving this new tasking paradigm are: (1) How to identify and extract inter-kernel thread block-level data dependency to derive task graphs? (2) How to reduce kernel launch overheads? and (3) How to dynamically resolve task data dependencies in a lightweight manner?

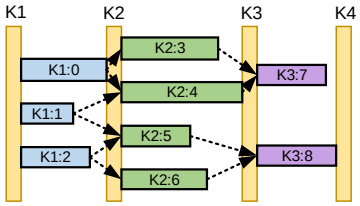
Task partitioning limitations: In order to re-map applications into task-based execution, the problem space must be partitioned into tasks. The tasks can be partitioned either statically based on the algorithmic properties of the workload (for example, pipeline stages in image rendering) or dynamically based on the input data. For example, CUDA Graph can record and capture a static task graph of kernels executing across streams. This operation is time-intensive and, in essence, profiles the workload to create a static graph which executes repeatedly. Static task graphs are not well-suited for workloads which are input-dependent. For example, a sparse solver might require an input sparse matrix that changes with each kernel call, rendering a static task graph from the previous run obsolete.

Typically, input-dependent task graphs require run-time information in order to partition tasks. Due to this, it is difficult to extract task graphs from existing applications in our proposed framework. Thus, the goal of BlockMaestro is to demonstrate the ability to extract static fine-grained task graphs, similar to CUDA Graph, in order to provide programmer-transparent task-based execution. We explore the ability to handle input-dependent task graphs in Chapter 4.

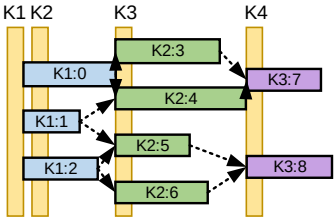
3.3 BlockMaestro

3.3.1 Overview

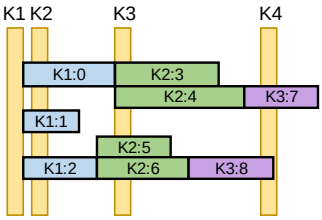
In this section, we present BlockMaestro, which enables programmer-transparent support for the "Thread blocks as Tasks" tasking paradigm on GPUs. BlockMaestro consists of three main components: (1) extracting fine-grained inter-kernel data dependencies from



(a) Baseline execution model. In BlockMaestro, inter-kernel thread block-level data dependencies are identified and extracted. (`K4` tasks are not shown.)



(b) Kernel pre-launching masks kernel launch overheads. Blocking of dependent thread blocks are enforced by hardware en masse.



(c) Dynamic inter-kernel thread block-level data dependency resolution enables overlapped execution of kernels, achieving benefits of task-based runtimes.

Figure 3.2: Baseline execution model suffers from high kernel launch overheads, dependency stalls and resource under-utilization. BlockMaestro's key insight is that kernel launch hiding and inter-kernel data dependency resolution can enable the benefits of task-based runtimes without the programmer burden. Kernel launch overhead is displayed as a vertical bar.

existing GPU applications; (2) kernel pre-launching to mask kernel launch overheads; and (3) dynamic inter-kernel data dependency resolution to ensure correctness of pre-launched kernels. Combined, these techniques allow GPU programs written in existing SIMT programming models to gain the benefits of task-based execution without the overhead of proprietary task-based programming models.

Figure 3.2 illustrates the operation of BlockMaestro. In this illustrative figure, we show the launching of four kernels, `K1` to `K4`, along with their corresponding thread blocks (labeled `0-2` for `K1`, `3-6` for `K2`, `7-8` for `K3`; blocks omitted for `K4`). The vertical bars

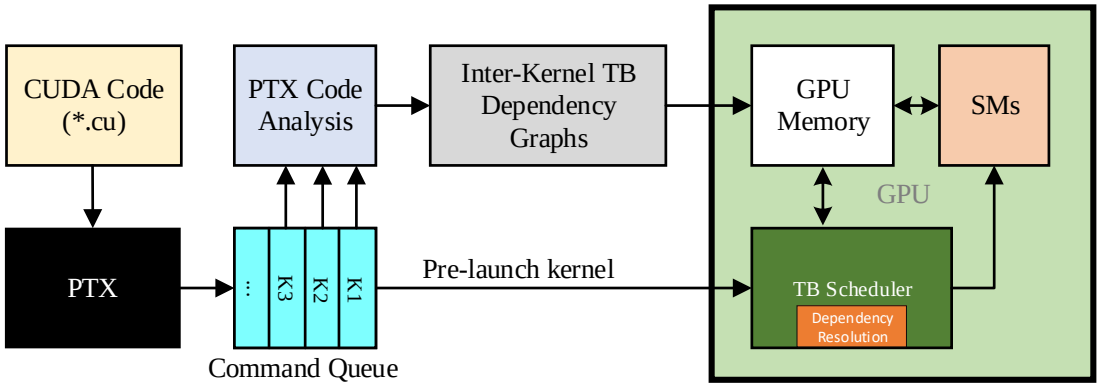


Figure 3.3: Overview of BlockMaestro.

represents each kernel's launch overhead. Figure 3.2a shows the execution timeline of the baseline GPU where kernel execution are serialized. In the baseline scenario, inefficiencies exist due to kernel launch overheads and dependency-stalled thread blocks resulting in GPU under-utilization. For example, even if `K2:5` and `K2:6` have already completed, `K3:8` cannot start until all of `K2` has completed.

To alleviate these issues, task-based runtimes allow programmers to express task execution by dynamically creating tasks and specifying their dependencies. This allows blocks to begin executing whenever dependencies are satisfied and can avoid kernel launch overheads with persistent kernels. In task-based runtimes, `K3:8` would be able to execute immediately once `K2:5` and `K2:6` have completed. To achieve the same goals, BlockMaestro introduces *kernel pre-launching* and *inter-kernel data dependency resolution* to eliminate kernel launch overheads and to enable overlapped execution of thread blocks from dependent kernels, respectively.

Figure 3.2b illustrates kernel pre-launching in order to hide the overhead of kernel launches. After kernel `K1` launches, BlockMaestro will pre-launch kernel `K2`. In order

to enforce correctness and resolve data dependencies between `K1` and `K2`, the thread block scheduler conservatively blocks `K2`'s execution until all blocks from `K1` has completed. While kernel launch overheads are eliminated, dependency stalls and under-utilization of resources can still exist.

To fully achieve the benefits of task-based execution, we further identify thread block-level data dependencies that exist between pairs of dependent kernel (annotated with arrows in the figure) at kernel-launch-time where just-in-time compilation occurs from PTX to SASS. Figure 3.2c illustrates inter-kernel data dependency resolution which utilizes the thread block-level data dependency information between dependent kernel pairs. These data dependencies are enforced by the thread block scheduler at run-time and are dynamically resolved. This enables any ready thread blocks to begin execution, regardless of which kernel they are running in.

Figure 3.3 shows the system overview of BlockMaestro. Data dependencies between inter-kernel thread blocks are acquired from just-in-time analysis at kernel launch time when PTX code is compiled to SASS assembly. These dependency graphs are then passed in the the hardware, where the dependencies are dynamically resolved. BlockMaestro further eliminates kernel launch overhead by enabling the application to pre-launch dependent kernels at the same time, without the need for synchronization. Once the data dependencies of any thread block from the dependent kernel are met, that TB will also be eligible to be issued to the execution units.

In the remainder of this section, we will discuss how BlockMaestro will enable kernel pre-launching and identify, represent, and enforce inter-kernel data dependencies.

3.3.2 Identifying Inter-kernel Dependencies

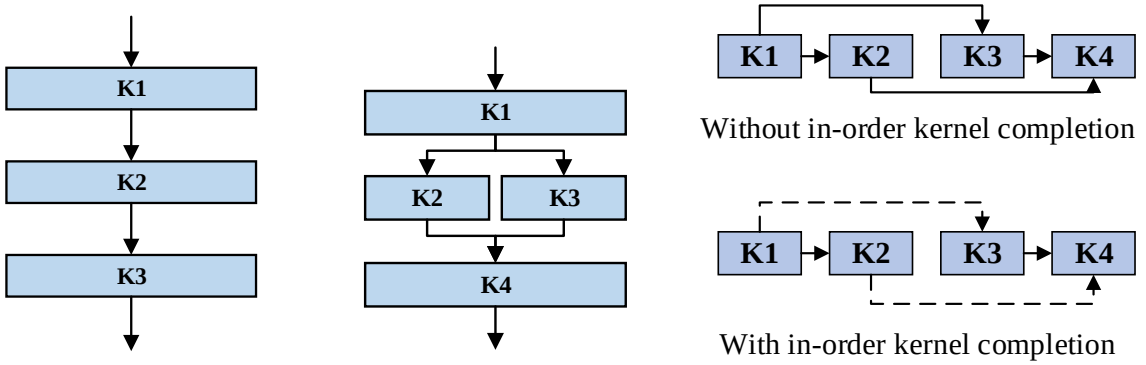
In many task-based runtimes, task dependencies are specified directly by the programmer. Dependencies can be conveyed at high-level task-based programming models such as CUDA Graph [93] or AMD ATMI [12], where programmers define a graph of operations. The key to BlockMaestro providing programmer-transparent support for the "Thread blocks as Task" paradigm is to identify inter-kernel thread block-level data dependencies.

1) Identifying kernel-kernel dependencies:

Data dependencies between kernels occur in data residing in global memory. Due to the SIMT programming model, the inputs and outputs of kernels are well-defined. Every region in global memory used by kernels are allocated with API calls, such as `cudaMalloc` in CUDA or `hipMalloc` in HIP. Load and store addresses can be identified through static analysis of the kernel's PTX or SASS code during the just-in-time compilation phase at kernel launch time.

If a kernel is to read from or write to a region of allocated global memory, the base pointer of the memory allocation must be passed to the kernel launch API. For memory APIs, base pointers are similarly passed. Writes are `cudaMemcpy` host-to-device operations and reads are device-to-host operations. Therefore, data dependencies between kernels and API calls can be identified within the command queue in a fairly straightforward manner.

Handling arbitrary inter-kernel dependencies: BlockMaestro can support both linear and non-linear patterns; examples of which are shown in Figure 3.4(a)-(b). When issued, these kernel launch commands would be serialized in the command queue. For example, for the application in Figure 3.4(b), the kernel launch order would be `K1` to `K4`. With multiple



(a) Linear dependency pattern

(b) Non-linear dependency pattern

(c) Dependency tracking required for (b).

Figure 3.4: Example types of inter-kernel dependencies and dependency tracking required for correctness. By enforcing in-order kernel completion we significantly reduce the amount of dependency tracking required (solid lines) due to implicit dependencies (dashed lines).

kernels being able to run at a time, it is possible that kernels can complete out of order. For example, `K2` and `K3` can execute concurrently but `K3` can be shorter and finish before `K2`. Therefore, to ensure correctness for `K4` we would need to keep track of dependency information for both `K2` and `K3` (Figure 3.4(c)(top)). This would require each dependent child kernel to keep track of dependencies for arbitrary number of parent kernels. *Clearly, this is not scalable to arbitrary inter-kernel dependency patterns.*

To simplify the amount of dependency tracking required, we enforce *in-order kernel completion.* As shown in Figure 3.4(c)(bottom), even if `K3` finishes, we do not mark it as complete yet or else `K4` will be incorrect. Instead, `K3` will only be marked complete if `K2` is complete. This way, any dependencies of `K4` on kernels prior to `K3` are implicit (dashed lines) and are guaranteed to be satisfied when `K3` is complete. This greatly reduces the amount of dependency tracking required (solid lines) and *limits dependency tracking to consecutive kernels.*

In addition, let us hypothetically assume `K2` completes before `K1`, in-order kernel completion would implicitly enforce the dependency between `K3` and `K1`. Note that if `K1` completes before `K2`, `K3` can begin execution since there's no explicit dependency between `K2` and `K3`, and `K1` is implicitly satisfied if we only allow 2 kernels to concurrently execute. Essentially, *BlockMaestro allows out-of-order execution of kernels, while enforcing in-order completion of kernels.* While we trade-off some potential kernel overlapping opportunities here, we gain the benefit of being able to scalably represent inter-kernel dependency using a series of bipartite graphs between all kernel pairs.

2) Identifying thread block-level dependencies:

While kernel-level data dependency can enable kernel pre-launching (as shown in Figure 3.2b), it does not realize the full potential of task-based runtimes (as shown in Figure 3.2c). In order to achieve task-based runtime benefits, BlockMaestro needs to avoid dependency-stalled thread blocks by enforcing thread block-level data dependency. By enforcing inter-kernel data dependency at the granularity of thread blocks, we can overlap the execution of thread blocks from dependent kernels.

BlockMaestro performs just-in-time compiler static analysis (at kernel-launch-time) to identify read-after-write (RAW) dependencies in global memory. These RAW dependencies are enforced at runtime by the thread block scheduler. The key to identifying RAW dependencies at thread block granularity is to identify the array indexing that each thread block touches.

In the CUDA programming model, programmers already specify the mapping of threads to data by calculating indices deriving from indexing variables such as `threadIdx`,

`blockIdx`, `blockDim`, etc. Using a simple vector add as an example, kernel maps threads to an index in the array using `int i = threadIdx.x + blockDim.x * blockIdx.x`. Then, each thread reads in an element in the arrays `A[i]` and `B[i]`, and store the sum into `C[i]`. Arrays `A[]`, `B[]`, and `C[]` are passed into the kernel function after being allocated with `cudaMalloc`. Based on the application's data-flow graph, we can identify all loads and stores in the program to identify the read and write sets, respectively.

To identify thread block-level read and write sets, we identify the indexing used to access the loads and stores by extracting the index representation as a function of parameters known at kernel-launch-time, e.g., `A + 4 * (threadIdx.x + blockDim.x * blockIdx.x)`. Each of these variables are known at kernel-launch-time, along with their possible values. Therefore, we can perform *value range analysis* to identify the range of array indices that each TB will access and create a read and write set per TB.

Value range analysis: We implement and perform this value range analysis for indices in load and store instructions per thread using the built-in PTX parser in GPGPU-Sim [16] as shown in Algorithm 1. Note that this algorithm is general enough that it can also be performed on any compiler frameworks that supports PTX, such as GPUOcelot [43].

We perform a backward pass on the CFG representation of the kernel and identify all global load and stores and track the origins of their source operands (lines 2-22). If we encounter a source operand that originates from the result of another load (an indirect memory access), we terminate and conservatively assume the entire kernel is dependent on the previous kernel (lines 7-9). Otherwise, we know that all load/store source operands are derived from known kernel-launch-time variables. Then given the kernel launch parameters,

Algorithm 1 Pseudo-code of PTX static analysis

```
1: Find all global loads/store instructions in kernel K
2: for all I ∈ instructions do
3:     S = {src(I)}
4:     while S is not empty do
5:         Go to the previous instruction j
6:         if dst(j) ∈ S then
7:             if j is a global load then
8:                 END (Possible non-static dependency)
9:             end if
10:            Remove dst(j) from S
11:            if src(j) is in local register (e.g. not immediate) then
12:                Add src(j) to S
13:            end if
14:        end if
15:        if j is first instruction then
16:            Break
17:        end if
18:    end while
       {Value range analysis}
19:    for all t ∈ Threads do
20:        Add address(I) to load and store sets, L_K and S_K
21:    end for
22: end for
23: Intersect L_K with S_{K-1} to find TB RAW dependencies
24: ...
```

such as block and grid sizes, we perform the value range analysis to identify the read and write sets of each thread block in the kernel (lines 19-21) and identify dependencies with the intersect of the read and write sets (line 23) of neighboring kernels.

To identify the inter-kernel thread block-level data dependency, every kernel call in the command queue is analyzed to obtain a read and write set. This kernel-launch time analysis overhead is performed off the critical path and is masked by the proposed kernel pre-launching technique. By comparing the read set of later kernels to the write set of earlier kernels, the intersection of the sets will determine where the RAW data dependencies exist. We create a dependency graph to represent data dependency where each node is a thread block and an edge represents a dependency. Since the nodes can be divided into two disjoint and independent sets (each belonging to a separate kernel), our dependency graph can be

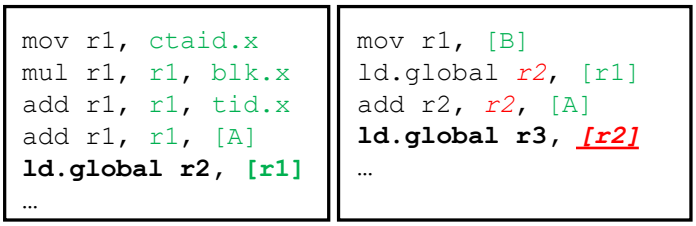


Figure 3.5: Value range analysis can only be performed on static memory indexing derived from variables known at kernel-launch-time (left). It is not possible to identify index ranges with non-static accesses before runtime (right).

classified as a bipartite graph. Therefore, the dependency graph for the whole application can be illustrated as a set of bipartite graphs, similar to Figure 3.1.

Why JIT analysis and not compile-time? This analysis can only be done at kernel-launch-time during just-in-time compilation from PTX to SASS as certain parameters are only known at kernel-launch time. For example, the grid size is dependent on the input data set. Similarly, `blockDim` (the number of threads in a thread block) and the range of `blockIdx` (range determined by the grid size) are also known at kernel-launch-time. The value range of these variables are unknown at compile-time (from CUDA to PTX) and, therefore, value range analysis cannot be performed to identify thread block-level data dependency. Furthermore, conducting this at kernel-launch-time from kernel API calls in the command queue allows us to dynamically create bipartite dependency graphs which can allow us to represent larger task graphs in a decomposed manner.

Limitations and other considerations: In this chapter, we focus on *static memory analysis*, i.e., analysis of memory locations that are known before runtime. They can include device variable addresses, immediate values, and kernel parameters. However, we cannot process global accesses that derive from another memory value (such as `A[B[i]]`), pointer chasing,

etc. Such instances are only known at runtime and would require runtime analysis, which is out of scope of this chapter. Figure 3.5 shows an example of static and non-static memory locations for the global load instruction.

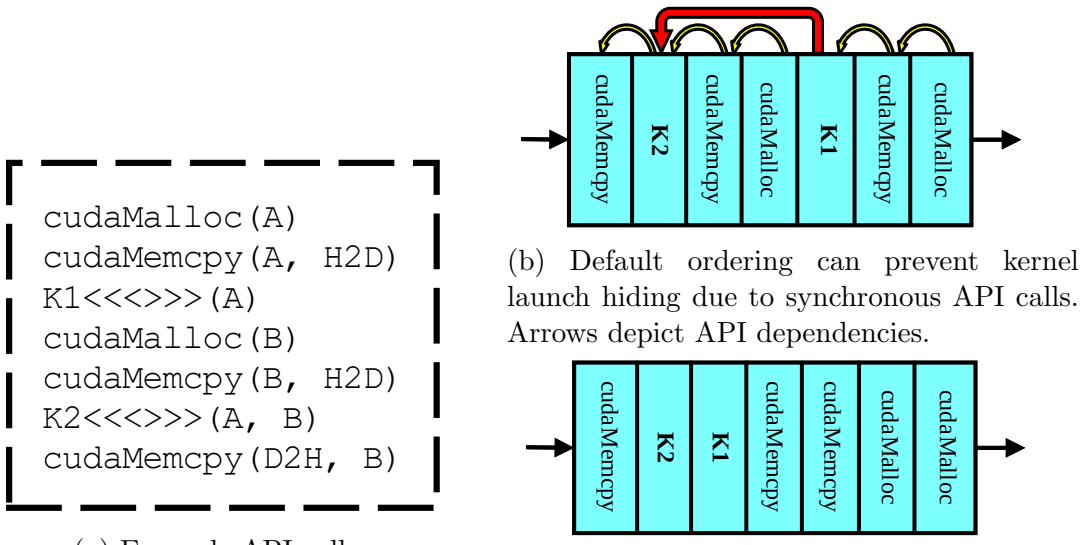
Note that even if the application makes use of Unified Memory, we can still identify read and write sets through value range analysis. Unified Memory are allocated with `cudaMallocManaged`, so we know which global memory address range needs to be monitored for RAW dependencies. Within the CUDA kernel, memory access occurs in the same manner as non-Unified Memory.

3.3.3 Enabling Kernel Pre-launching

In this section, we will discuss the changes necessary to enable kernel pre-launching. Overlapping is achieved by having future kernels launching before the completion of the previous kernel. In order to accomplish this, we need to: (1) enable multiple kernels to be simultaneously executed from a command queue; and (2) prevent certain CUDA API calls from blocking the command queue or from blocking the issue of future CUDA APIs in order to allow the command queue to fill.

To highlight the challenges in enabling kernel launch hiding, we will refer to Figure 3.6. In Figure 3.6a we show an example trace of CUDA API calls. When the host executes the code and reaches a CUDA API call, it sends the call along with the necessary data to a command queue (the default CUDA Stream[1]). Common CUDA API calls are to allocate memory, transfer data to/from GPU global memory and host memory, launch

[1]Note that we may use stream and command queue interchangeably. CUDA Streams is equivalent to OpenCL command queue and AMD HIP Streams.



(a) Example API calls

(b) Default ordering can prevent kernel launch hiding due to synchronous API calls. Arrows depict API dependencies.

(c) Re-ordered API commands to enable kernel pre-launching.

Figure 3.6: Effect of API ordering in command queue on kernel launch hiding.

kernels, and device synchronize. Kernel calls are asynchronous (non-blocking) by default. However, CUDA memory API calls are synchronous (i.e., host is blocked until the functions return) and can cause limited opportunities for kernel launch hiding.

Handling blocking APIs: In order to maximize opportunities for kernel pre-launching, we need to be able to have multiple kernel commands in the command queue. However, this scenario can be prevented due to the blocking behavior of certain API calls. In the example shown in Figure 3.6a, memory operations such as `cudaMalloc` and `cudaMemcpy` are blocking the host. So as kernel `K1` is executing, `cudaMalloc(B)` can be processed in parallel (CUDA commands which use different hardware engines can be executed in parallel), blocking the host until it completes. The host will have to wait until `cudaMalloc(B)` returns before being able to issue `cudaMemcpy(B)` to the command queue. Depending on the length of `K1`, this would prevent the opportunity for `K1` and `K2` to overlap, as `K2` may not be called

by the host until after `K1` completes. Therefore, we need to be able to fill multiple kernel commands in the command queues to maximize kernel pre-launching opportunities.

We can overcome this issue by treating certain blocking operations as non-blocking. Since BlockMaestro can resolve dependencies in the hardware, we can shift the burden of implicit synchronization to the hardware. The only API call requiring implicit synchronization to be enforced is when there is a RAW hazard with the host, e.g., a `cudaMemcpy` call from device to host. Explicit synchronization API calls, such as `cudaDeviceSynchronize`, can also be bypassed as long as no call after it incurs RAW hazard with the host. As long as data is not modified on the host, but only in the GPU, we can enforce correctness of implicit synchronization in the GPU.

Note that asynchronous memory APIs are used by programmers when programming with CUDA Streams. If the target application already utilizes CUDA Streams, then the command queue is already filled and is not an issue. BlockMaestro can also seamlessly support pre-launching in CUDA Stream-based application by overlapping kernel launches within the same stream. The only other consideration is to handle `cudaStreamSynchronize` in a similar manner to `cudaDeviceSynchronize` for API commands within the same stream. While BlockMaestro can generalize to support CUDA Streams, the remainder of the paper focuses on single default stream applications which experience worse kernel launch overheads and under-utilization issues.

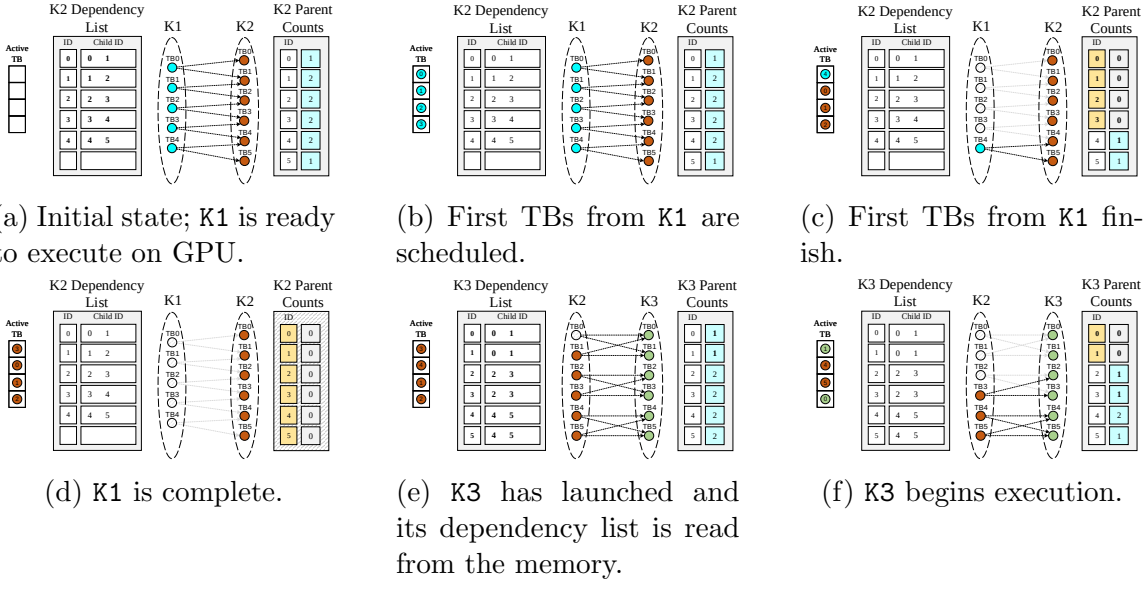
Programmer-transparent API command reordering: Figure 3.6b shows the state of the command queue after all the CUDA APIs are called. Commands in the queue are implicitly ordered which can cause orderings that limit the amount of kernel launch hiding. For

example, as `K1` is launched and executing, we cannot proactively pre-launch `K2` as the `cudaMalloc` and `cudaMemcpy` commands must complete first. One potential solution to maximize kernel launch hiding is to identify the true data dependencies between APIs in the command queue and reorder the commands to maximize kernel launch hiding. This is achieved by moving the kernel launches as close as possible. Figure 3.6c shows such an order that still satisfies the data dependencies between API calls. Kernels can then be launched if memory could be allocated to them. Otherwise, they will have to wait until resource becomes available.

Enabling multiple kernels to execute from the same command queue: In the baseline, kernel commands are blocking in the command queue. That is, only a single kernel from a command queue can be running at a time. Therefore, one modification that we require is to let the command queue process multiple kernel commands at once. This feature is already available in NVIDIA Hyper-Q which enables multiple kernel commands from different streams (with our modification, from the same stream). In our experience, we find that enabling the execution of only 2-3 kernels per command queue is sufficient to completely overlap kernel launches.

To enforce correctness and resolve data dependency between two running kernels in the same command queue, we rely on the thread block scheduler to enforce the second dependent kernel to only begin executing after it has detected that the first kernel has completed execution. This way, the second kernel's launch overhead overlaps with the first kernel's execution. If the kernels are independent (no data dependency exist between them), then the independent kernel can begin executing right away. Otherwise, data dependencies

are enforced by the hardware. Later in this section, we will discuss in detail how the thread block scheduler can enforce inter-kernel data dependencies. Note that all thread blocks in the current kernel can be executed in an out-of-order fashion. However, BlockMaestro enforces the completion of the parent TBs before starting their child TBs from the next kernel.



(a) Initial state; `K1` is ready to execute on GPU.

(b) First TBs from `K1` are scheduled.

(c) First TBs from `K1` finish.

(d) `K1` is complete.

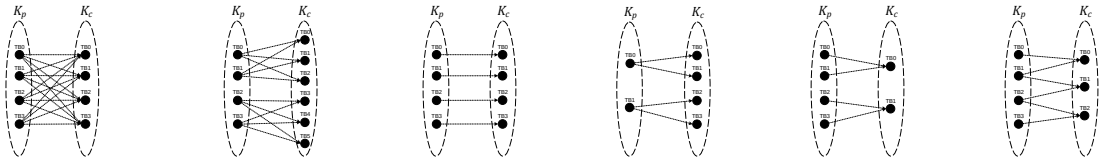
(e) `K3` has launched and its dependency list is read from the memory.

(f) `K3` begins execution.

Figure 3.7: TB scheduling example in BlockMaestro. Inter-kernel thread block-level dependencies are maintained using a dependency list and parent counter.

3.3.4 Enforcing Inter-kernel Dependencies

Inter-kernel dependency is enforced using a dependency list representing the bipartite dependency graph. In BlockMaestro, the dependent kernel owns the dependency list. We use this information in the hardware to enforce inter-kernel dependency through the use of a *Dependency List Buffer* and a *Parent Counter Buffer*. We will first illustrate in Figure 3.7 how BlockMaestro uses these structures to enforce dependencies and then detail



(a) Fully connected (b) N-group fully connected (c) 1-to-1 (d) 1-to-n (e) n-to-1 (f) Overlapped

Figure 3.8: Examples of common dependency patterns between TBs from adjacent kernels

architectural support. Recall that we only need to keep track of dependencies between consecutive launched kernels. Thus, we illustrate using two kernels and then will discuss how to generalize to support multiple pre-launched kernels.

Let us assume we have a GPU that can execute 4 TBs at once. The dependency list stores the bipartite dependency graph which is indexed by the thread block ID of a parent kernel (`K1`) and contains a list of dependent child kernel (`K2`). For example in **(a)**, TB0 of `K1` is a dependee of TB0 and TB1 of `K2`. The parent count table keeps track of how many pending dependencies are outstanding for the child kernel. For example, TB1 of `K2` is dependent on two thread blocks in the parent kernel.

The initial state of the example is shown in **(a)** with `K1` launched and `K2` pre-launched. `K1` is the first kernel, and so it has no dependencies. Thus, the device can start scheduling TBs 0-3 as shown in **(b)**. After TB0 finishes, the remaining TB from `K1` starts. At the same time, children of TB0 are read from the dependency list (TB0 and TB1 from `K2`), and their respective parent counter decrements, making TB0 from `K2` ready to execute once there are available resources in the SM. Soon after, TBs 1-3 from `K1` also finish, allowing TBs 1 and 2 from `K2` to be ready for scheduling **(c)**.

When TB 4 from `K1` finishes **(d)**, `K1` is marked as complete, `K2` is now the designated parent kernel, and we shift our attention to the next pre-launched kernel `K3`. In **(e)**, we now show the dependency list of `K3` which specifies the dependencies in `K2` and the parent counts of pending dependencies for `K3`. As the TBs of `K2` continue executing, we follow the same scheme as in parts **(b)**-**(d)** where `K3` will begin executing **(f)**.

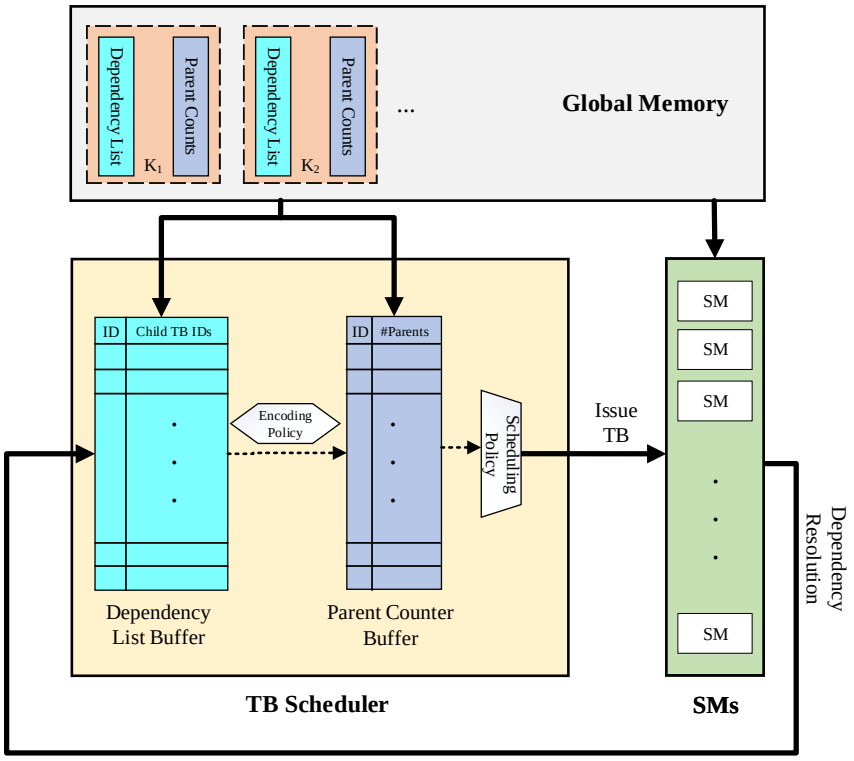


Figure 3.9: Supporting TB scheduler architecture.

1) Architectural Support:

Figure 3.9 depicts the proposed supporting hardware for the TB scheduler. When the device receives a kernel from the host, the dependency list and initial parent counters are stored in global memory. Thus, for every (pre-)launched kernel the GPU needs to keep track of the *dependency list base address* and *parent counters base address* in global memory. To

minimize the amount of global memory access, we include a *Dependency List Buffer* and *Parent Counter Buffer* in the thread block scheduler. The dependency list buffer keeps track of dependencies of actively executing thread blocks and the parent counter buffer keeps track of child thread blocks' pending unresolved dependencies.

When a thread block is scheduled for execution the thread block's entry in the dependency list is buffered in the dependency list buffer. Then the entry is read to identify the child thread blocks. If an entry does not already exist in the parent counter buffer, we allocate an entry and fetch the child thread block's parent counter value. Since the information in this dependency list and parent counter entry is not needed until the thread block finishes execution, this buffering process is off the critical path.

When a TB completes, we identify every child TB ID with the dependency list buffer and index into the parent counter buffer to decrement the parent counts. When a parent count hits 0, the corresponding child TB is now ready for execution. We deallocate an entry in the dependency list buffer when a parent TB completes and we delallocate an entry in the parent counter buffer when that child TB is selected for execution.

Resolving dependencies of multiple kernels: This design is easily scalable to support resolving dependencies of multiple running (pre-)launched kernel execution by simply appending bits to the thread block ID to represent the relative kernel IDs. For example, in order to resolve the dependency of 4 kernels, we can append 2 bits to the thread block ID as a kernel identifier. This kernel identifier is incremented whenever a new kernel is launched and wraps around to 0 when saturated. Since we only need to track dependencies between neighboring kernels, the kernel identifier is essentially the least significant 2 bits of the kernel ID.

Scheduling policies: BlockMaestro can support several scheduling policies across kernel TBs. By default, BlockMaestro gives more priority to the TBs in the producing kernel. TBs in the consuming kernel will not be scheduled until all producing kernel's TBs has been scheduled. It is also possible to give priority to the TBs from the consuming kernel. This will enable more opportunity to concurrently execute dependent kernels and improve utilization by essentially allowing more TBs to run ahead.

Note that these policies does not face any deadlock issues for any producer kernels under synchronization events. For example, the producing kernel can be deadlocked if some TBs are waiting on a barrier but other TBs cannot be scheduled since the consuming kernel is taking up resources and starving the producer kernel. This scenario does not occur since we will never fully starve a producer kernel to the point of deadlock. In the worse case scenario, eventually the consumer kernel TBs will face unmet dependencies which will allow the producer kernel to schedule, thus, avoiding any permanent deadlock.

3.3.5 Representing and Storing Inter-kernel Dependencies

We utilize a buffer in the TB scheduler to store the dependency list of the parent kernel. To reduce the amount of storage, we can take advantage of the dependency patterns among the kernels themselves to encode them. These patterns are rarely arbitrary, since the code is usually written to globally load and store the data using a large number of threads, e.g., each thread loading a block. Therefore, by analyzing the pattern, the graph can be stored on the device in an encoded fashion, which can greatly reduce the memory usage.

P#	Pattern	Overhead
(1)	Fully connected	$O(1)$ ($O(MN)$ without encoding)
(2)	n-group fully connected	$O(M+N)$
(3)	1-to-1 ($M=N$)	$O(N)$
(4)	1-to-n	$O(M+N)$
(5)	n-to-1	$O(N)$
(6)	Overlapped	$O(N+M.deg_{max})$
(7)	Independent	$O(1)$

Table 3.1: Hardware overhead w.r.t. dependency pattern between `K1` of size N and `K2` of size M thread blocks.

Table 3.1 displays the additional memory overhead that BlockMaestro would utilize for a graph with N parent TBs and M child TBs on the hardware. Even though it can be more difficult for a random dependency graph, the overhead can be drastically reduced by detecting specific patterns that can usually occur among the kernels (some shown in Figure 3.8) and using encoding to reduce the requirements.

For example, for a fully connected pattern, a single bit is enough to signal the GPU to simply prevent the consuming kernel from running until the producing kernel is finished. For the n-group case, TBs parenting the same child TBs could be encoded to be grouped together in the memory as well, all TBs in the same group referring to one location containing the child TB group, hence $O(M+N)$. For 1-to-n, every TB from `K2` (with M TBs) is mapped to a single parent TB, hence $O(M)$. In 1-to-n, each parent TB has exclusive child TBs, i.e., no child TB is shared between two parents. In the overlapped pattern, parent TBs can share multiple child TBs. Therefore, the overhead will be $O(N)$ plus $O(M)$ times the maximum degree of a child TB.

In addition, if the dependency resolution yields little benefits for the execution speedup, e.g., too large a dependency degree, the device can ignore the fine-grained depen-

dency resolution and treat the kernels as if they are fully connected, as it is shown in Figure 3.13 (it will be discussed further in Section 3.4).

Name	Description	# Kernels	P#
3MM [49]	3 Matrix Multiplications	3	(2,7)
AlexNet [64]	AlexNet network	22	(1,3,4)
BICG [49]	BiCG Sub Kernel of BiCGStab Linear Solver	2	(7)
FDTD-2D [49]	2D Finite Different Time Domain	24	(5,7)
FFT [37]	Fast Fourier Transform	60	(3,5,7)
GAUSSIAN [26]	Gaussian Elimination	510	(4,5)
GRAMSCHM [49]	Gram-Schmidt Decomposition	192	(1,4,5)
HS [26]	Hotspot	10	(6)
LUD [26]	LU Decomposition	46	(3,4,5)
MVT [49]	Matrix Vector Product and Transpose	2	(7)
NW [26]	Needleman-Wunsch	255	(4,5)
PATH [26]	Path Finder	5	(6)

Table 3.2: List of benchmarks used, number of kernels, and type of dependency pattern exhibited (See Table 3.1).

3.4 Evaluation

3.4.1 Methodology and benchmarks

We use a modified version of GPGPU-Sim v3.2.2 [16] as our baseline, with a Titan X Pascal-like configuration with 28 SMs, each able to run up to 32 TBs at once, though BlockMaestro should generalize to any SIMT architecture. Greedy-then-oldest (GTO) warp scheduling policy is used [108]. For kernel launch overhead calculations, we have used the average baseline launch overhead of $5\mu s$ from [55]. As shown in Table 3.2, we evaluate against various applications from Rodinia [26], PolyBench [49], SHOC [37], and Tango

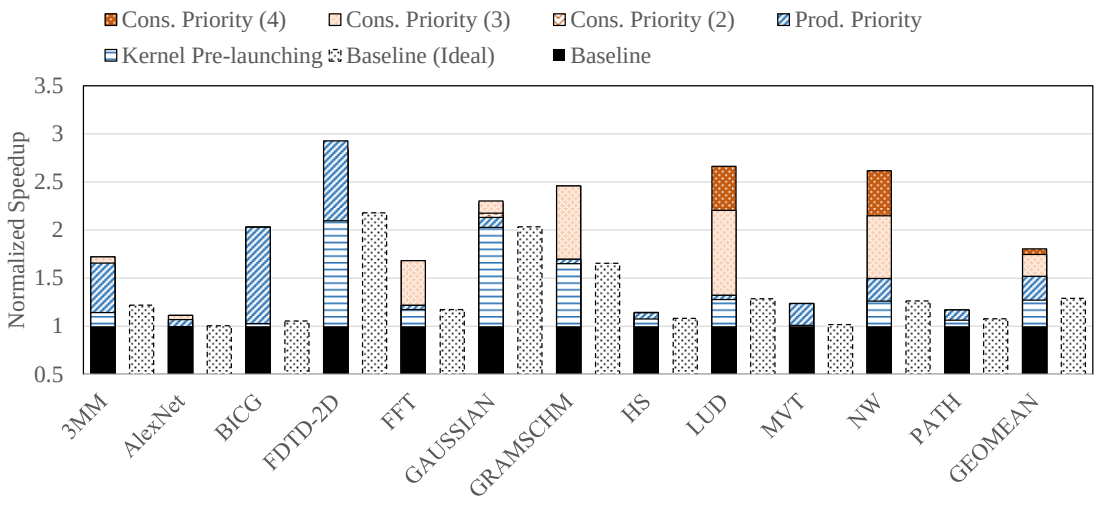


Figure 3.10: Normalized speedup w.r.t. baseline.

[64] benchmark suites, which cover a wide-range of multi-kernel applications from different domains.

3.4.2 Results

Speedup: Figure 3.10 shows the amount of speedup achieved by BlockMaestro with respect to the baseline. For reference, we have included the *ideal baseline* case with no kernel launch overheads (bar on right of each stacked bar). *Kernel Pre-launching* uses no synchronization APIs, but enforces the dependency by not allowing any consumer kernel TBs to schedule until all producer kernel TBs have completed. In addition, there is *Producer Priority*, which adds the fine-grained dependency resolution and gives scheduling priority to producer kernel's TBs. We have also included *Consumer Priority* which allows 2, 3, and 4 concurrently running kernels corresponding to 1, 2, and 3 pre-launched kernels, respectively. This consumer priority scheme prioritizes the consumer kernel's TB for scheduling.

We see an average speedup of 51.76% using producer priority. With consumer priority, it can be seen that increasing the number of pre-launched kernels can increase the mean speedup to 80.28%. However, we observe diminishing returns with more than 3 pre-launched kernels. This behavior can be best explained by the degree of TB data dependencies that exist between kernels. Workloads that most benefit from more pre-launched kernels require significant number of kernels in an application and with less connected data dependencies. For example, AlexNet has significant fully-connected dependencies while LUD has only 1-to-1/1-to-n/n-to-1 dependencies which are amenable to TBs running ahead.

Certain applications, such as GAUSSIAN and GRAMSCHM, experience significant speedup from just kernel pre-launching (and no thread block-level dependency resolution). These workloads tend to have large number of kernels, each of which finishes fast. Therefore, kernel launch overhead is the major bottleneck alleviated.

Other benchmarks, such as 3MM, BICG and FDTD, gained most of their benefit from fine-grained dependency resolution with simple producer priority scheduling. These workloads tend to have data dependencies that are easier to satisfy and captures more benefit with only two kernels active. Sometimes their kernels are independent and able to run in parallel. Thus, more TBs can be ready to run and advance the application's progress.

Utilization: This increase in utilization can also be seen in Figure 3.11 which displays the increase in normalized average TB concurrency with respect to the baseline. Workloads which are more compute-intensive with kernels consisting of hundreds or thousands of thread blocks, such as AlexNet, tend to suffer less from the overheads of kernel launches. Therefore, these workloads benefit less from kernel pre-launching alone but still sees an increase in TB

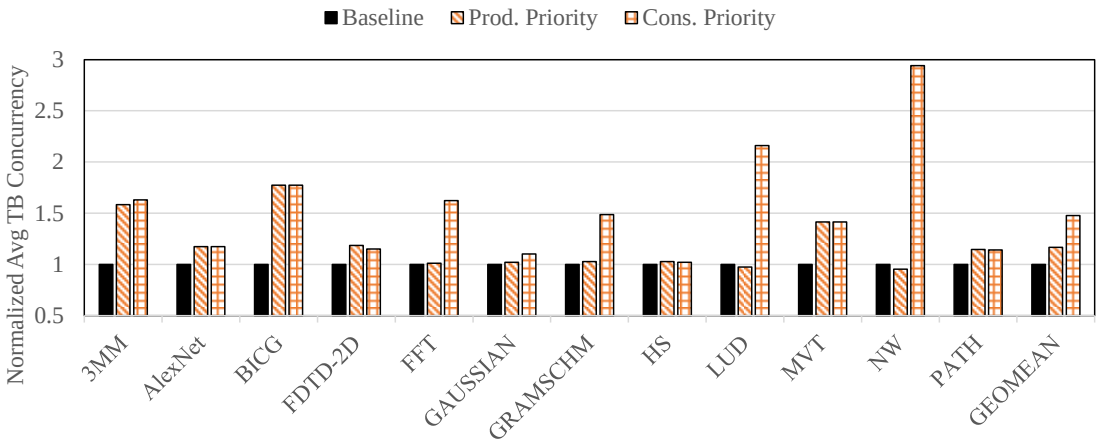


Figure 3.11: Normalized average TB concurrency w.r.t. baseline.

concurrency due to fine-grained TB dependency resolution. We still observe that AlexNet achieve speedup of 6.9%. It can be seen that more number of kernels along with simpler dependency patterns can produce more opportunities for overlapped kernel execution and utilizing the resources on the device more efficiently.

Figure 3.8 showcases various examples of basic patterns in data dependency of TBs from the child kernel K_c on those from its parent kernel K_p, which can be extracted from the PTX code. As the dependency pattern gets more complicated, it becomes more difficult to take advantage of. The fully connected pattern in Figure 3.8a is the worst-case scenario and is functionally the same as a synchronization barrier between the kernels. Therefore, the opportunity to speed up the application via execution overlapping ends with kernel launch overhead hiding. However, simpler patterns offer a greater opportunity, since after the execution of each TB in the producing kernel, TBs from the consuming kernel become ready for execution faster, which means more utilization of the resources on the GPU.

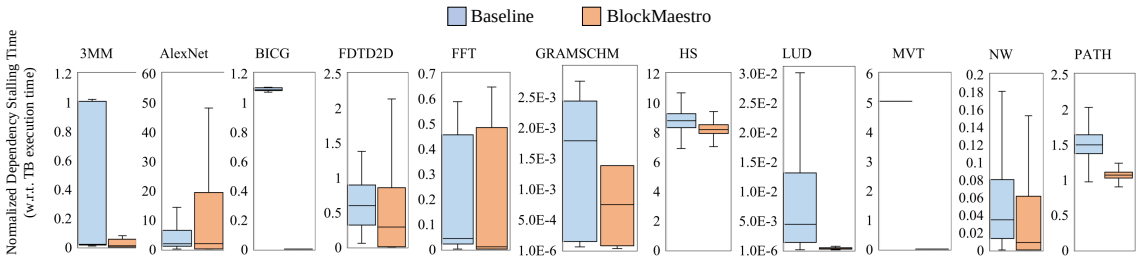


Figure 3.12: Dependency stall distribution normalized to TB execution time.

Dependency Stall Distribution: Figure 3.12 displays a distribution of the amount of dependency stalling each TB in an application is going through during the execution. Recall that dependency stalled thread blocks are dependent thread blocks that has dependencies that are satisfied but cannot execute yet due to its kernel not yet started. The box plot borders designate the first and third quartiles of the distribution, with the line in the box representing the median. In addition, the values are normalized to each TB's execution time. For example, a value of 2 for a TB means that that TB has waited for double the amount of time it would spend executing on the GPU.

As we can see, BlockMaestro can visibly decrease the amount of dependency stalling for most of the TBs in the applications. However, in some cases where the GPU capacity for TB execution is full, some of the remaining TBs in a dependent kernel will have to wait more than their peers to be run, increasing their stalling, as is shown in the case of AlexNet. Also, note that the two kernels in BICG and MVT can run in parallel in BlockMaestro, hence their dramatic stall reduction. These workloads are also reflective of CUDA Sterams benefits since independent kernels that can concurrently execute exists. However, CUDA Streams will not be able to be used with concurrently executing non-independent kernels. These results demonstrate that BlockMaestro can gain the benefit of executing

independent concurrent kernels across streams automatically, while also extracting benefits for more complex dependency patterns.

3.4.3 Overheads

Inter-connectivity Analysis: In Figure 3.13, we demonstrate the effect of the degree of dependency that exist between inter-kernel thread blocks and kernel size on the speedup of a microbenchmark based on *VectorAdd* with two equal-size kernels. In this application, there is a simple 1-to-1 dependency pattern between the two kernels by default. Each line represents the workload size (number of TBs in one kernel). During each workload, we increase each TB's dependency degree by artificially introducing dependencies between the kernels in the form of an n-group fully connected pattern. For example, a degree of 4 signifies that the first 4 TBs from `K1` are dependent on the first 4 TBs from `K2`, etc., resulting in a 4-to-1 dependency pattern.

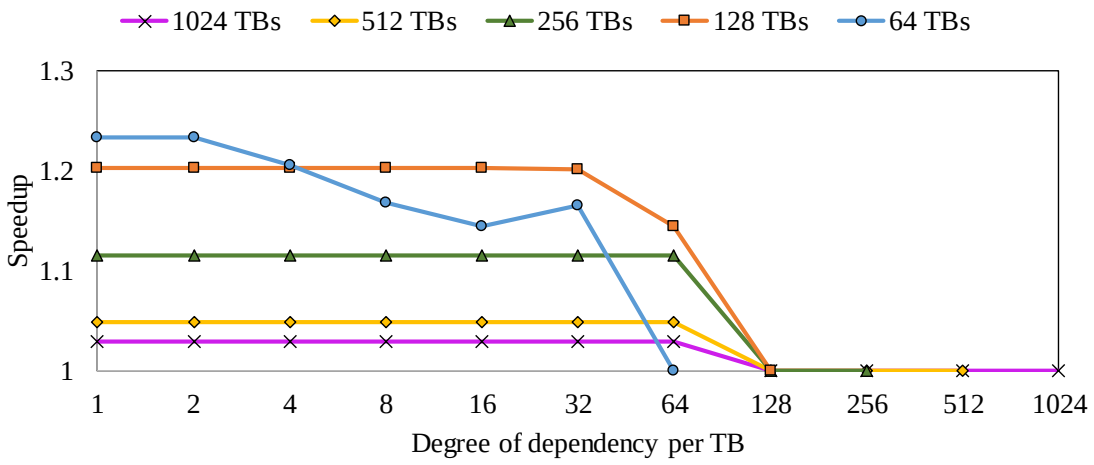


Figure 3.13: Interconnectivity analysis for BlockMaestro. The x-axis shows the size of each TB's dependency group.

It can be seen that the benefits we can get from dependency resolution begin to quickly deteriorate once the average dependency degree passes a certain threshold, in our case $deg = 32$. After this point, the speedup benefits reflect that of a fully-connected dependency graph.

In addition, the speedup we get even before this threshold decreases as the number of TBs in the kernels grow and ceases to exist by the time the workload size is 2048. With more TBs running in a kernel, the most resources a kernel require and limits the opportunity for pre-launched kernels to run-ahead. We leverage our insights on how the inter-connectivity of the dependency graph to minimize hardware overheads.

Area overhead: BlockMaestro mainly introduces a dependency list buffer and a parent counter buffer. Since the dependency list buffers actively running TBs, we require $28 \times 32 = 896$ entries. We similarly set the number of parent count buffer entries to the same. For each dependency list buffer entry, we choose to store 4 child TBs per entry. We aggressively choose a narrower entry since most workloads can be described by a dependency pattern. Thus, the encoding can derive child TB IDs. For the rarer scenario where we cannot encode, we will utilize the 4 child TBs per entry. If we require a wider entry (as it exist in global memory) we can simply split the wider entry across multiple entries in the dependency list buffer.

Each index into the dependency list buffer and parent counter buffer (representing the TB ID) is 32 bits + 2 bits for kernel identification. Each child TB ID in the dependency list buffer is 32 bits since kernel identification can be computed. Since we see diminishing return with greater levels of inter-connectivity (greater than 64-to-1), we use 6 bits for the

parent counter. Anything higher and we conservatively encode to fully connected without much loss to speedup. In total, we require a storage overhead of about 22KB, in addition to control logic.

Memory Request Overhead: Figure 3.14 shows the impact of BlockMaestro on the memory requests. Buffering the dependent list information from the memory can incur a request overhead in the order of $O(V)$. As it is seen, BlockMaestro's average memory request overhead is only about 1.36%.

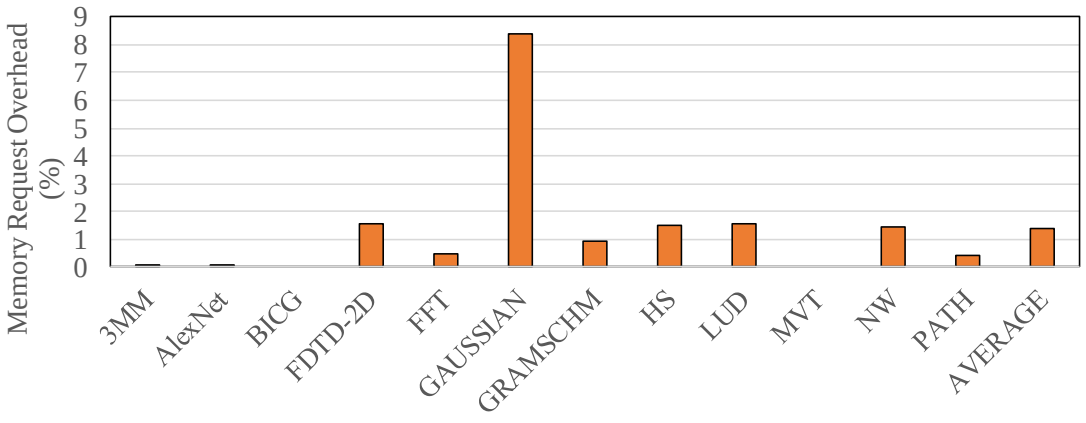


Figure 3.14: Memory request overhead for BlockMaestro.

Bipartite Dependency Graph Storage Overhead: Table 3.3 displays the amount of storage used for the entire run of each application normalized with respect to the case where no encoding is used, i.e., plain storage. As it is observed, the average storage is reduced by 34.7%. (Note that BICG and MVT are excluded here since their kernels are independent and, therefore, there is no memory storage used for them even without encoding.)

	Storage		**Storage**
3MM	0.210	AlexNet	0.012
BICG	-	FDTD-2D	1
FFT	1	GAUSSIAN	1.77E-04
GRAMSCHM	0.375	HS	1
LUD	0.938	MVT	-
NW	1	PATH	1
Average	**0.653**		

Table 3.3: Normalized total storage of bipartite dependency graphs for the entire application run w.r.t. plain storage.

3.4.4 Comparative Results

Comparison to Task-based Execution Models and Dynamic Parallelism: Figure 3.15 showcases a comparison with CUDA Dynamic Parallelism (CDP) [89], a "Tasks as Kernels" execution model, and Wireframe [8], a "Tasks as TBs" execution model. Wireframe requires the programmer to specify task dependencies using a proprietary API and relies on hardware dependency resolution. Essentially, Wireframe represents multi-kernel workloads into a single mega-kernel with tasks mapped to a TB. CDP represents each task as a device-side kernel launch, avoiding much of the overhead of host-side kernel launches. For a direct comparison with prior works, we have used the benchmarks in [8]; six applications with wavefront dependency pattern of 4K tasks. In other words, each kernel has an overlapped dependency pattern with its predecessor, and the number of TBs gradually grows until the middle of the dependency graph, where it starts to decline.

The widely used CDP kernel launch latency model [134] is based on Kepler and estimates a CDP kernel launch overhead of 20μs, significantly greater than modern host-side kernel launch times (5μs) [55]. Therefore, we model CDP's kernel launch latency as

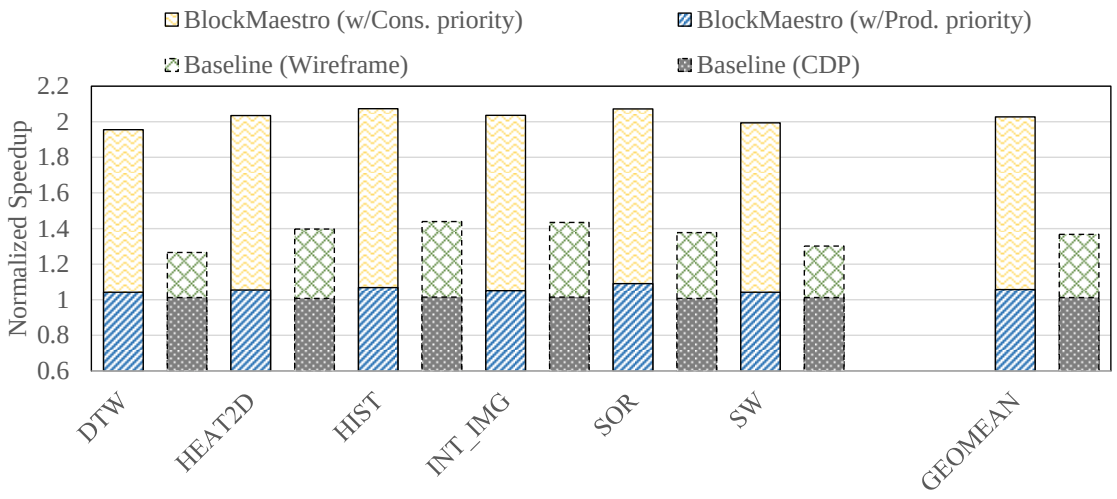


Figure 3.15: Comparison with existing "Task as Kernel" (CDP) and "Task as TBs" (Wireframe [8]) task-based execution models.

3μs by removing the kernel launch API call overhead (2μs) [55] from the host-side kernel launch time.

Figure 3.15 shows the normalized speedup of our comparison normalized to CDP. BlockMaestro with producer priority achieves only 5.8% speedup. Wireframe achieves a geomean speedup of 36.8% due to its ability for tasks to run-ahead up to three waves (levels of dependencies). This enables more tasks to run and utilize the GPU. To that end, we evaluate BlockMaestro with consumer priority to enable tasks to run-ahead. With this, we observe speedup of 2x.

We found that Wireframe's reliance on size-constrained hardware task management buffers (i.e., pending update buffers) can actually limit the amount of tasks that can run. Since BlockMaestro's dependency resolution can update task states stored in global memory, our execution is not constrained to increase GPU utilization, but at the cost of slightly higher memory traffic as shown in Figure 3.14. This successfully demonstrates the

benefits that BlockMaestro is able to achieve the benefits of task-based execution models without programmer intervention.

3.5 Related Work

Task Dependency: CUDA Dynamic Parallelism [89] enables device-side kernel launches to support dynamic kernel launches. This amortizes kernel launch overheads and allows tasks to dynamically spawn on the GPU. However, there are significant drawbacks such as limited levels of recursion [8]. Since CUDA 10, CUDA Graphs [93] allows the user to define a dependency graph between different kernels, perform optimizations on the whole graph during its instantiation, and execute it many times. It also lets the user capture a series of multi-stream operations and convert it into such a graph. CUDA Graphs can reduce the kernel launch overhead, since the kernel initializations are done together. However, it still does not address the GPU under-utilization during the execution of dependent kernels.

AMD has also added support the expression of dependencies among GPU "task groups" for years [12]. The authors in [103] use an asynchronous task-based paradigm to express three well-known applications as directed acyclic graphs (DAG) [34] on the Heterogeneous System Architecture (HSA) [45]. In [104], the same authors note the problem of queue oversubscription from parallel tasks, and propose a mechanism to prioritize the critical path in the task graph. In BlockMaestro, the priority is to finish the TBs from the producer kernel first in a step to potentially add more consumer kernel TBs to the ready TB pool. Kaushik et al [65] target hyperplane sweep kernels, which have a wavefront-like dependency pattern and are essential in solving some partial differential equations. To improve

the kernel's performance, it is fine-grained into smaller tasks. To improve the throughput, dependent tasks communicate through special packets, thereby resolving the dependencies within the GPU. Adaptive Task Aggregation (ATA) [54] has been proposed as a software solution to reduce the overhead of irregular applications, specifically sparse solvers, through fine-grained task scheduling. The tasks can be assigned to a compute unit even before their parent tasks are complete, hence avoiding the launch overhead.

Task Scheduling: There have been various works on GPU task scheduling. Juggler [20] employs a software-based runtime using persistent threads (PT) for single-kernel GPU workloads with data dependencies, trading synchronizations with scheduling through a DAG. The authors in [17] also target irregular algorithms and propose a framework to predict and set the TB resources dynamically through the use of a hardware-based table to track the used resources by all TBs. The authors in [70] propose overlapped kernel execution through a modified host code paradigm, obtaining the memory access information using compiler-generated profiler kernels and storing them on the GPU's reference count table a TB scheduler with the goal of maximizing parallelism. In [58], the authors seek to overcome the memory bottleneck in GPU applications by proposing a reuse-aware thread block scheduler to exploit data reuse between the kernels with producer-consumer data dependencies in mind, the majority being dependencies between TBs with the same ID from the two kernels ("self-dependencies"), as well as using work stealing to minimize load imbalance. PAVER [125] presents a hybrid TB scheduling method by measuring data locality among each kernel's TBs, scheduling them based on a heuristic method to reduce cache thrashing, and performing task stealing to reduce load imbalance towards the end of each kernel.

LocalityGuru [126] uses JIT analysis to improve the data locality during the execution of GPU kernels, and aims to add support for non-static memory loads as well. Note that BlockMaestro does not target load imbalance directly. However, by enabling and managing the scheduling of TBs from an additional kernel, the SMs have more TBs to run, reducing under-utilization.

There have also been works focused on optimizing the CPU-GPU pipeline. Versapipe [148] offers a flexible framework for the user to write GPU applications whose executions are more pipeline-oriented. Unlike in our proposal, here the programmer would write the operations for each stage, and Versapipe would assemble the stages together to improve the computation performance on the GPU. GOPipe [96] builds on VersaPipe and introduces a system to automatically determine the granularity of a task in a stencil program's pipeline without user intervention and dynamically schedule them on the device. HiWayLib [149] targets CPU-GPU pipeline communications for heterogeneous applications by identifying and remedying prominent problems in the communication, such as the slow data movement between the devices and contention among the GPU threads.

3.6 Conclusion

In many applications today, there are a large number of kernels, which can incur high launch overheads. In addition, a significant part of the GPU's resources could remain unused due to data dependencies between the kernels, which can also lead to under-utilization. Solutions have been proposed in prior works to solve such problems, but they also require significant programmer intervention.

In this chapter, we have proposed BlockMaestro, a software-hardware solution in order to hide the effect of kernel launch overheads as much as possible and manage the execution of thread blocks in a more fine-grained manner by tracking their data dependencies in hardware in order to enforce correctness. Our solution also increases GPU utilization during a GPU kernel execution while incurring a small memory overhead. By using this paradigm, we have observed an average speedup of 51.76% with producer priority scheduling policy (up to 2.92x) in various applications.

However, as it was seen, BlockMaestro is unable to process kernels with non-static accesses and when encountering them, reverts to the default coarse-grained synchronization to ensure output correctness. Such accesses in an application are not uncommon, prompting us to look for a more generalized alternative.

In the next chapter, we propose SEER, an ML-based framework in an effort to estimate a kernel's memory read/write addresses, so they can be used to create a dependency graph usable for fine-grained dependency management frameworks such as BlockMaestro.

Acknowledgments

The work in this chapter is partly supported by National Science Foundation under Grants CCF-1815643, CNS-1955650 and CNS-2047521. We would like to thank the anonymous reviewers for their invaluable comments and suggestions. We also extend our gratitude to Qiumin Xu for providing their implementation of Warped-Slicer [142] which was utilized as our baseline for concurrent kernel execution.

Chapter 4

SEER: Estimating Runtime Data Dependencies in GPU Applications

4.1 Introduction

Since their inception, GPGPUs have been the best choice in massively parallel computations. Therefore, the greatest focus in GPGPU design is usually on improving the computational power, and more recently, the energy consumption, especially for mobile devices. However, it is also imperative to pay attention to other aspects of the GPU workflow as well in order to maximize the benefits.

As mentioned before, GPU applications are getting more complex, requiring the user of a fine-grained task scheduling paradigm to have an increasing knowledge of the algorithms used in the kernels, and even the dependency patterns between them if necessary. However, it is possible to alleviate this burden through compiler and just-in-time (JIT)

analysis of the code used in the kernels. In the previous chapter, BlockMaestro was proposed to reduce programmer intervention via JIT analysis of static memory accesses in the code at kernel-launch-time, using read/write memory addresses accessed in adjacent kernels to derive inter-kernel dependency graphs for fine-grained thread block (TB) scheduling, leading to performance improvement while maintaining the correctness of the output.

However, it is impossible to merely use the kernel codes to derive the entire set of memory accesses when some of them depend on runtime values, i.e., non-static accesses. Sometimes, the kernel contains indirect memory accesses, which use indices related to another memory load, i.e., memory data which can only be resolved at runtime. In some cases, the load instruction could be on a branch with a condition derived from in-memory data. In some works, the user feeds the required data to the analyzer before running a kernel [102]. However, due to the way GPUs are structured and programmed, there is a possibility that some of these memory addresses could be estimated.

Inspired by prior works in load prediction value (LVP) techniques [78, 79, 100, 113, 117] and related machine learning (ML) works [21, 53], we have developed SEER, an ML framework seeking to determine the read and write sets of a specific GPU kernel by analyzing each global load and store instruction, the pattern of all the instructions leading to them, and other factors. To motivate the problem, we will first explore if a relationship could possibly exist between known and unknown values in kernel code. Then, we implement and train SEER on various kernels of different sizes in order to learn in-kernel instruction patterns and estimate the read and write sets related to each global memory operation. Its outputs could then be used to create dependency graphs of adjacent kernels which could be

fed to a fine-grained TB scheduling paradigm, such as BlockMaestro, in order to improve the GPU performance and utilization for various GPU applications, including those with non-static memory accesses.

4.2 Background

4.2.1 PTX Analysis

As mentioned before, in order for the GPU code to run on the device, the code has to be compiled into an intermediate representation (IR) format, which is independent of the GPU architecture, and then further compiled at kernel-launch-time into SASS, an assembly-like language, into an architecture-specific code which the GPU can execute. SEER uses PTX (parallel thread execution) [95] as the IR code.

If needed, TBs across kernels can only use global load and store instructions in their kernel to pass data to each other through the global memory. Therefore, to find the dependency graph between adjacent kernels, we need to find such instructions in both kernels and their access pattern according to each TB. Global load and stores usually use the generalized format below (data types are left out):

```
ld.global out_data_reg, [ld_addr+offset]
```

```
st.global [st_addr+offset], in_data_reg
```

where `offset` is a constant, `ld_addr` and `st_addr` are the load/store addresses, and `in_data_reg` and `out_data_reg` are the destination and source operand values for the global load and

store instructions respectively. Each address is made of a *base address*, which can be given to the GPU as a kernel argument, and an *index*, which could be a function of kernel parameters, such as thread ID, block ID, and/or memory data. Kernel parameters and arguments are transferred to the GPU at kernel-launch-time. Therefore, the base address can be determined before runtime. On the other hand, the index can potentially be unknown before runtime, since in-memory data cannot be known for certain until the kernel start executing the code. This is why BlockMaestro is unable to determine the read/write sets of the kernels using non-static memory accesses. However, it could be possible to estimate some of the accesses by observing the code structure and prior executions, and predicting the next likely addresses accordingly, especially if a correlation can be observed between a non-static memory access and other code elements.

4.2.2 Challenge

In order to estimate read and write sets for a kernel, we have to know if there could be a relationship between known variables (*prime variables*) and unknown variables (*indirect variables*) in the first place. To show this possibility, we will use one of the kernels from breadth-first search (BFS), one of the well-known applications from Rodinia [26] where the access patterns are irregular and indirect memory accesses exist in its kernels.

Figure 4.1 shows the code in the first of the two kernels used in BFS. In this application, two kernels are called in every iteration in order to traverse the input graph. Most of the load and store instructions used in this kernel are shown to the right of the figure. The dependencies among these instructions are displayed in Figure 4.2 in the form

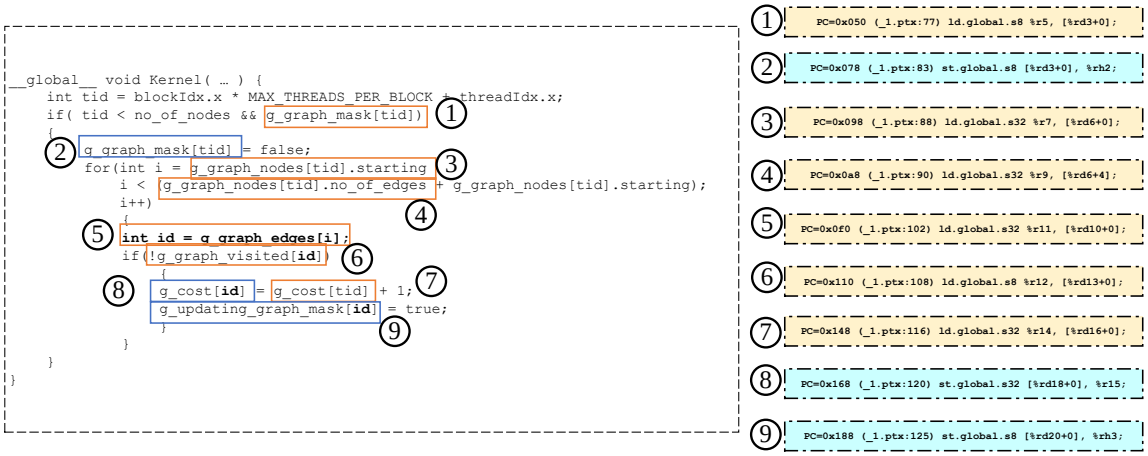


Figure 4.1: Code of the first kernel in BFS, which contains indirect memory accesses (left) and its most relevant load (orange) and store instructions (blue) in the PTX representation along with information such as PC (right). (Numbers used for easier referral.)

of a tree. It can be seen that `tid`, a prime variable, is used to determine array elements, which are then used as indices themselves, namely `i` and `id`, which are indirect values. The main challenge here is to see if there could be a relationship between them and the prime values so they could be speculated.

In order to see a correlation, we used GPGPU-Sim [16] to extract the addresses and values from each global load and store in the aforementioned BFS kernel during a run. The extracted data are then converted into a correlation matrix, which is comprised of values between -1 and 1. The further an element is from 0, the more correlation exists between the PCs in the row and column intersecting in that location.

Figure 4.3 indicates the correlation matrix of the addresses and values from different instructions in the kernel (indicated by their PC values). It can be seen that there might be a significant correlation between values and/or addresses that seemingly share little connection (marked on the figure). For example, it shows that there is a very strong correlation

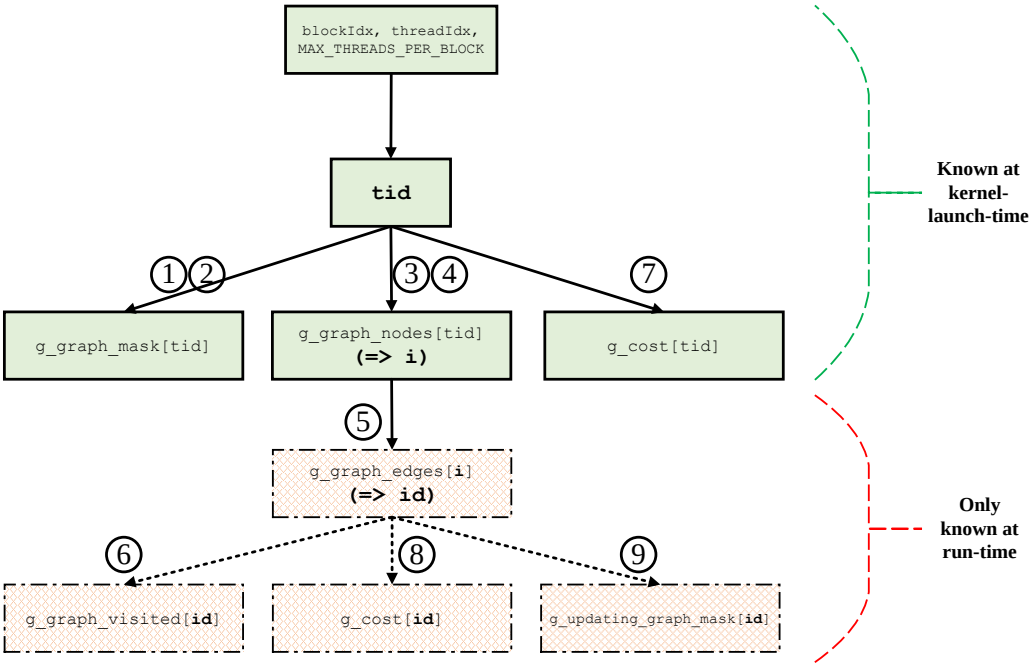
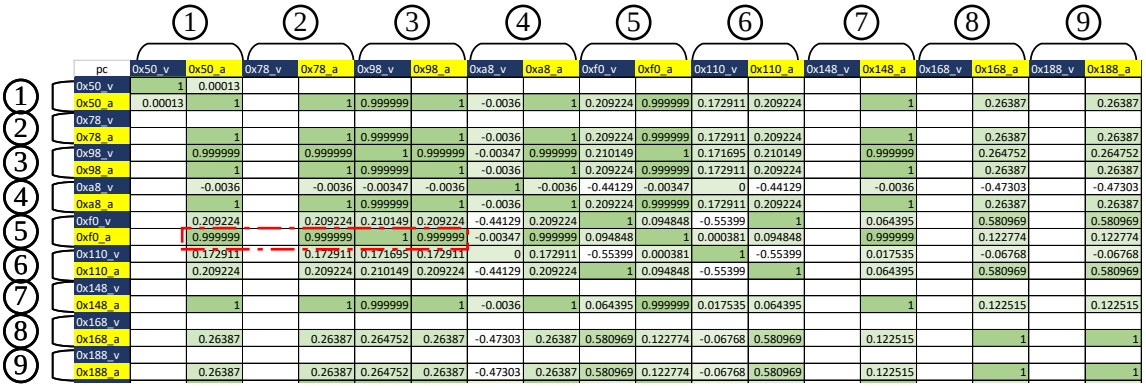


Figure 4.2: Dependency tree of load and store instructions in the BFS kernel.

between the addresses in `0x50` (Instruction `(1)`) and `0xF0` (Instruction `(5)`), which use `tid` and `i` respectively. Based on the code itself, `i` is derived from `g_graph_nodes[tid]`, which does not necessarily correlate with `tid` itself. However, based on the correlation matrix, such a connection could be assumed, and possibly, even be predicted to some extent.

Now that we see that such a relationship could be possible, we will now seek a more generalized method to speculate non-static, as well as static, memory accesses. In SEER, we use machine learning with the end goal of predicting read and write addresses per thread block in a kernel in order to estimate inter-kernel data dependencies among the application TBs. This way, we can additionally estimate runtime dependencies in order to improve the performance and better utilize GPU resources.



	pc	0x50_v	0x50_a	0x78_v	0x78_a	0x98_v	0x98_a	0xa8_v	0xa8_a	0xf0_v	0xf0_a	0x110_v	0x110_a	0x148_v	0x148_a	0x168_v	0x168_a	0x188_v	0x188_a
①	0x50_v	1	0.00013																
	0x50_a	0.00013	1		1	0.999999	1	-0.0036	1	0.209224	0.999999	0.172911	0.209224		1		0.26387		0.26387
②	0x78_v																		
	0x78_a		1		1	0.999999	1	-0.0036	1	0.209224	0.999999	0.172911	0.209224		1		0.26387		0.26387
③	0x98_v		0.999999		0.999999	1	0.999999	-0.00347	0.999999	0.210149	1	0.171695	0.210149		0.999999		0.264752		0.264752
	0x98_a		1		1	0.999999	1	-0.0036	1	0.209224	0.999999	0.172911	0.209224		1		0.26387		0.26387
④	0xa8_v		-0.0036		-0.0036	-0.00347	-0.0036	1	-0.0036	-0.44129	-0.00347	0	-0.44129		-0.0036		-0.47303		-0.47303
	0xa8_a		1		1	0.999999	1	-0.0036	1	0.209224	0.999999	0.172911	0.209224		1		0.26387		0.26387
⑤	0xf0_v		0.209224		0.209224	0.210149	0.209224	-0.44129	0.209224	1	0.094848	-0.55399	1		0.064395		0.580969		0.580969
	0xf0_a		0.999999		0.999999	1	0.999999	-0.00347	0.999999	0.094848	1	0.000381	0.094848		0.999999		0.122774		0.122774
⑥	0x110_v		0.172911		0.172911	0.171695	0.172911	0	0.172911	-0.55399	0.000381	1	-0.55399		0.017535		-0.06768		-0.06768
	0x110_a		0.209224		0.209224	0.210149	0.209224	-0.44129	0.209224	1	0.094848	-0.55399	1		0.064395		0.580969		0.580969
⑦	0x148_v																		
	0x148_a		1		1	0.999999	1	-0.0036	1	0.064395	0.999999	0.017535	0.064395		1		0.122515		0.122515
⑧	0x168_v																		
	0x168_a		0.26387		0.26387	0.264752	0.26387	-0.47303	0.26387	0.580969	0.122774	-0.06768	0.580969		0.122515		1		1
⑨	0x188_v																		
	0x188_a		0.26387		0.26387	0.264752	0.26387	-0.47303	0.26387	0.580969	0.122774	-0.06768	0.580969		0.122515		1		1

Figure 4.3: Correlation matrix of global load and store values and addresses in different PCs in the second BFS iteration. (The suffixes *_v* and *_a* after each PC represent the instruction's value and address respectively.)

4.3 SEER

One of the most important objectives in SEER is obtaining a good estimation of the read and write sets of each kernel by TB. This allows us to extract the dependency graph between adjacent kernels with more certainty. The base addresses in a GPU code are known at kernel-launch-time and can be extracted from the kernel code and its parameters at launch. Therefore, the focus of the address extraction will be the indices in the code.

4.3.1 Framework

Using Python, We have developed a framework in order to be able to analyze PTX code in order to extract the context for a certain operation and the necessary instruction-, basic block- and kernel-level information that can be used in training. Inspired by GPGPU-Sim [16], the framework is able to perform symbolic execution on various GPU kernels in order to extract their expected read and write sets on the TB level, which can be used to train the SEER model. The instruction lookup table can also be easily modified for future

opcodes both for symbolic execution and the ML model. For the latter, there is an entry reserved for unknown instructions so the model can continue the prediction until the opcode table is updated. As for the indirect memory accesses, input memory traces can be used.

4.3.2 Model archtecture

In order to extract the address offsets, we used PyTorch [99] to develop the ML network shown in Figure 4.4. As it can be seen, there are two main parts in our network: the embedding layer, and the fully-connected (FC) layers. In the beginning, the PTX code from the kernel is analyzed, and its global load (LD) and store (ST) operations are extracted. Provided to the network are also the kernel parameters, thread block ID, and other related information. Each global LD/ST is processed individually by SEER.

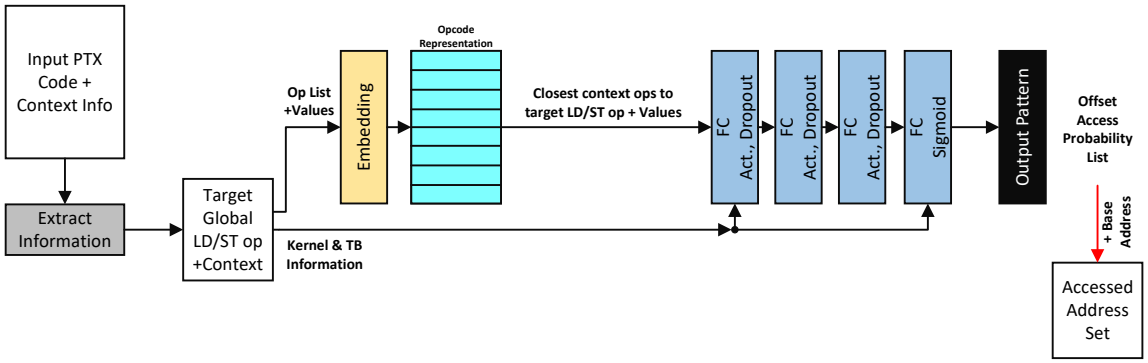


Figure 4.4: Network architecture for SEER.

4.3.3 Code representation

First, the target LD/ST is fed to the embedding layer, along with a list of opcodes which, along with the target instruction, constitute a chain of dependency (also *context instructions*). The embedding layer acts as a look-up table, and is responsible for converting

a set of IDs into a corresponding set of vectors. The aforementioned indices represent opcodes used in the PTX representation. An example of context instructions is shown in Figure 4.5. Here, the address for the global load instruction (`ld.global`) comes from an `add` instruction, which in turn has two arguments, each coming from a separate instruction, etc. This goes on until the entire chain of dependency is constructed for the target `ld.global`. Then the target instruction and all its context ops are put in a list and mapped to a unique ID, reserved for that opcode, which will be used by the embedding layer. For the uses of the SEER network, the list fed to the embedding layer is capped at 30 instructions, with `nop` (opcode for *no operation*) filling the remaining spots.

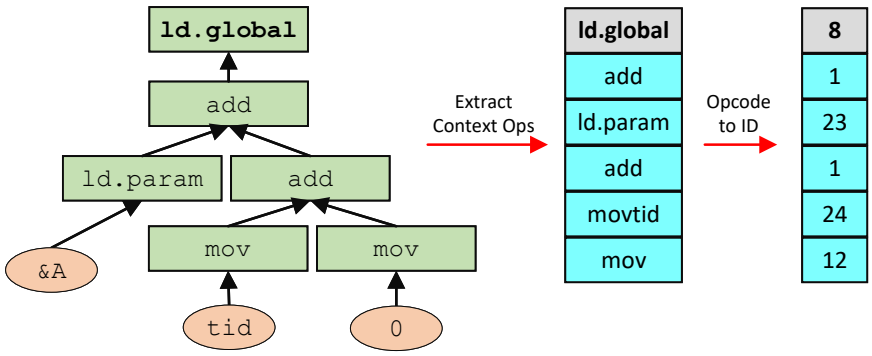


Figure 4.5: Example of context operations for a global load instruction.

Note that for special cases, a *dummy opcode* is used to distinguish it from the other use cases of the same opcode (the dummy is not part of the actual instruction set). For example, in the figure, there is a move instruction (`mov`) which uses thread ID (`tid`) as input. Since this operand might be consequential in determining TB read/write sets, the opcode is represented with a dummy opcode `movtid` in the embedding table to separate it from the other `mov` that uses an immediate value (`0`), which would not include as much useful information.

4.3.4 Fully-connected layers and output

The embedding layer outputs a matrix made of vector representations of all context opcodes (zero vector for `nop`). The matrix is then concatenated with each op's related information from the PTX, e.g., the immediate values they use, offsets, etc. After that, the output is flattened, concatenated with the kernel and TB information, normalized, and fed to a series of fully-connected layers. In order to mitigate the effect of overfitting (following the training data too much), a dropout feature is added to all FC layers except the output layer. An activation function has also been used by each layer to be able to learn more non-linear patterns; $Tanh$ for the first three layers, and sigmoid for the final layer. Currently, the output layer uses a width of $2^{16} = 65536$.

In order to obtain the set of accessed memory addresses, we use the network for a multi-label classification problem: classifying each of the address offsets as one of either 'accessed' and 'not accessed' group. The sigmoid function maps all its inputs to values between 0 and 1, which can show the probability of each offset being accessed. In addition, binary cross-entropy (BCE) loss has been used for the training process, which evaluates how close the predicted output values are to the expected output (in our case, an array of 0's and 1's). During the evaluation of the model, the peaks in the output are selected as 1's and the rest as 0's. The list would then be added to the base address associated with the target LD/ST instruction to produce the final read/write set for each TB in the kernel.

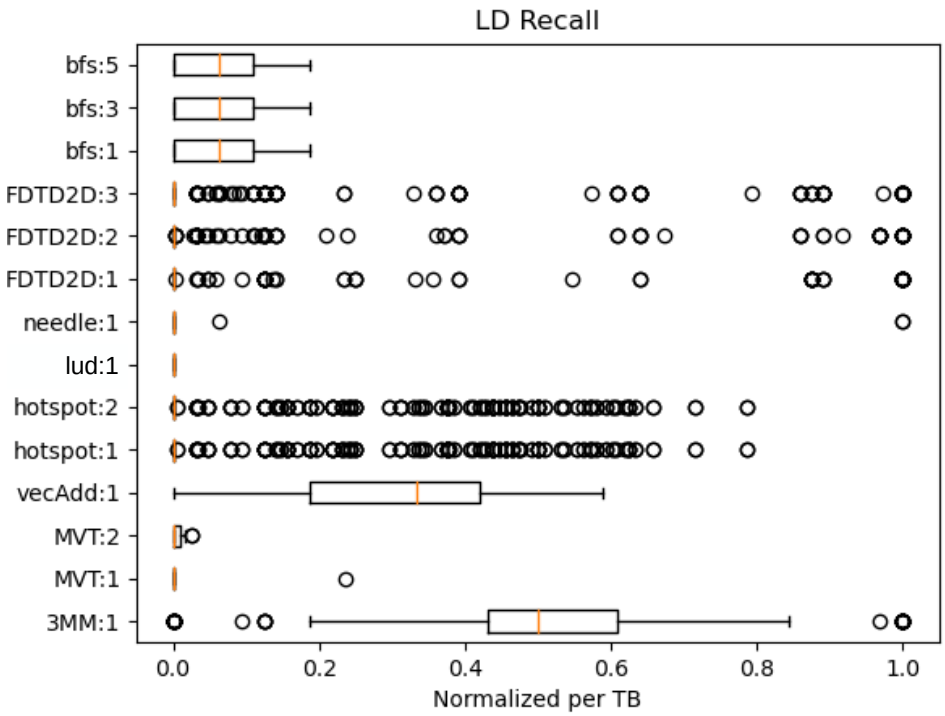


Figure 4.6: Recall of LD address prediction.

4.4 Evaluation

We used a modified GPGPU-Sim v3.2.2 [16] to extract the PTX of various kernels from the Rodinia [26] and PolyBench [49] benchmark suites in order to train our model. To evaluate our model's read/write set accuracy, we use *recall* and *precision* as metrics. Recall shows how much of the positives in the ground truth have been correctly identified (true positive vs. false negative), while precision shows the percentage of the correct identifications in the model's output itself (true positive vs. false positive). In our evaluation, being classified as 'accessed' is treated as a positive output. It is also noteworthy to discuss the impact of each metric in this evaluation. Having a low precision means that a large percentage of the output is not actually accessed by the TB, which, if used for fine-grained

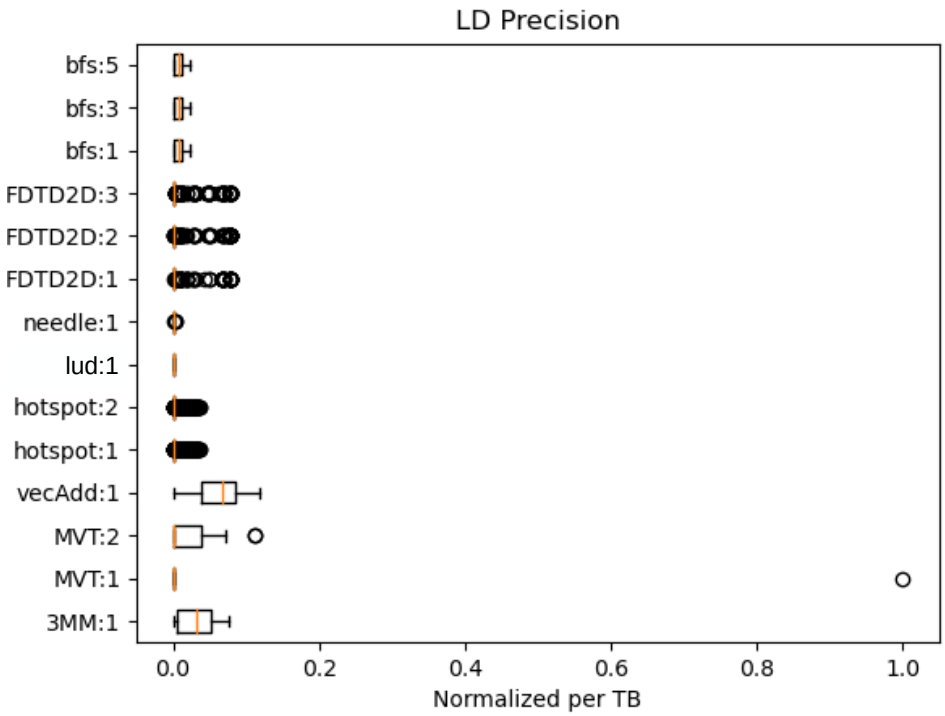


Figure 4.7: Precision of LD address prediction.

TB scheduling, can potentially lead to extra edges in the dependency graph and a lower speedup. On the other hand, having a low recall means that a great portion of the accessed addressed have not been detected, and this, in the dependency graph scenario, can result in some edges being removed erroneously, affecting the very correctness of the application's output. Therefore, for our purpose, even though precision is important, recall is considered a greater priority.

Figure 4.6 and Figure 4.7 display box chart distributions of the global load recall and precision of each TB from various kernels obtained from our trained model. It can be seen that the model is predicting the load addresses to some extent based on the PTX codes of each kernel, its parameters, and the global load's context. However, there are also

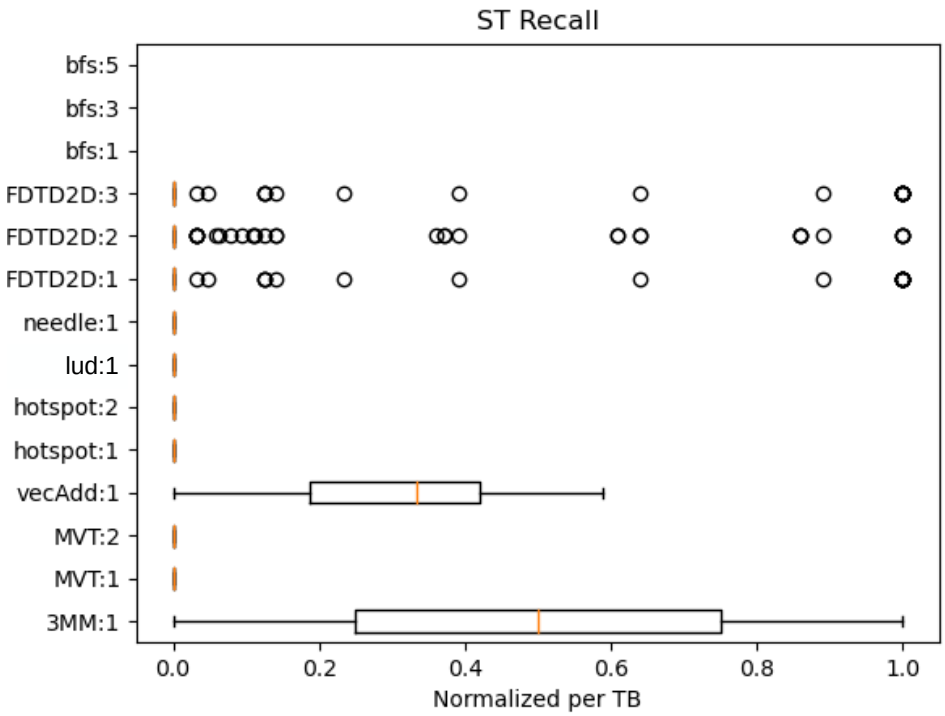


Figure 4.8: Recall of ST address prediction.

many mispredicted addresses, as signified by the much lower precision. Similarly, Figure 4.8 and Figure 4.9 show the same box chart distributions for global store instructions. It can be seen that the numbers are much lower compared to global loads, which could stem from less training due to less global store operations compared to global load ops. The wider variety we have in the kernels in our training set, the harder it becomes for our network to learn all the patterns for each of them. Better results could entail with more training data, hyperparameter fine-tuning, more training epochs, improvements to the network architecture and code representation, using LSTM or attention layers, etc.

We also used some of the read/write sets obtained by SEER to generate dependency graphs between kernel pairs, and compared them with the baseline, as shown in

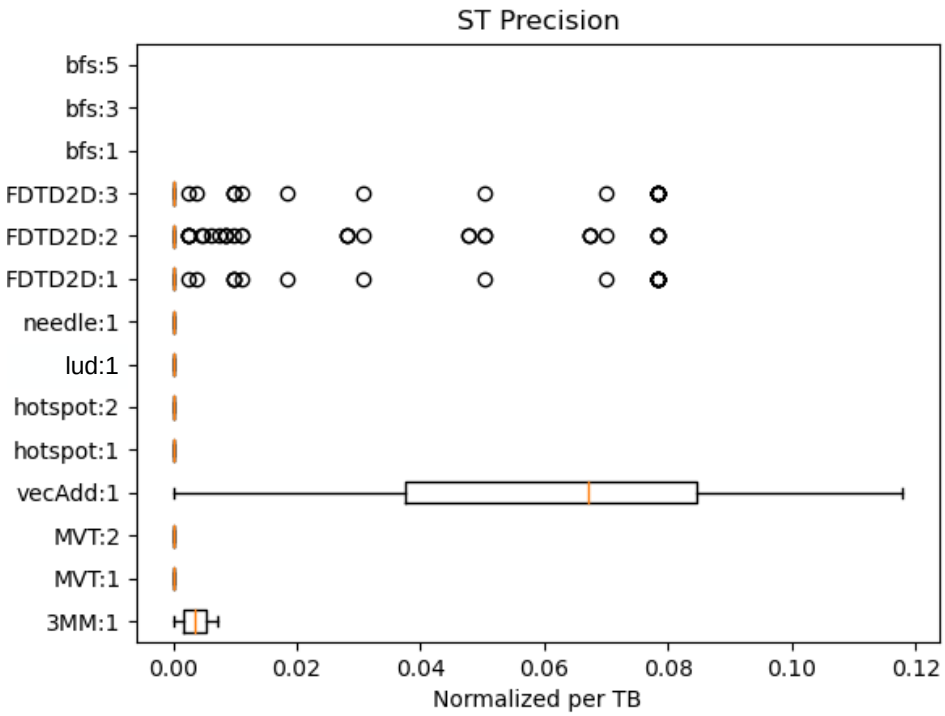


Figure 4.9: Precision of ST address prediction.

Table 4.1. It can be seen that, for the most part, the model is constructing fully connected dependency graphs.

4.5 Related Work

Load Value Prediction: Load value prediction has been used in CPU-related literature for decades. Before the rise of ML in modern computing, hardware tables were one of the methods used to predict values. In [79], the authors used an LVP table to keep track of load operations and update their status for each of them continuously depending on the prediction's success and based on previously seen values, with categories including unpredictable, predictable and constant. It also used a verification table to replace loads

Kernel Pair	Total Nominal Edges	Missed Edges	Redundant Edges
3MM, $(1 \leftrightarrow 2)$	0	0	23409
3MM, $(2 \leftrightarrow 3)$	2601	0	20808
MVT, $(1 \leftrightarrow 2)$	0	0	256
VectorAdd, $(1 \leftrightarrow 2)$	8	0	56
FDTD2D, $(1 \leftrightarrow 2)$	0	0	0
FDTD2D, $(2 \leftrightarrow 3)$	480	0	65056
Hotspot, $(1 \leftrightarrow 2)$	16512	16512	0

Table 4.1: SEER dependency graph statistics comparison vs. baseline

with constants if it met the requirement, invalidate the data in the table with a store operation, and demote a load value to predictable if its prediction failed. Its operation required the some bandwidth from the memory. In [78], they describe a method to run data-dependent serial programs in parallel through prediction of register values. Theoretically, it is possible to attempt load value prediction through machine learning, combined with symbolic execution, to estimate the read and write sets of a kernel. However, with GPUs consisting of numerous threads to keep track of, the amount of data used in comparison to CPUs, and the notion that we would be trying to learn the memory values rather than potential memory addresses, we decided to take the multi-label classification approach.

ML & Code Representation: In order to process code using machine learning for various tasks, our model must understand the code to some extent. Therefore, an embedding layer is used to differentiate between different syntactic structures, such as opcodes. There have been a variety of code-to-vector representations used in literature [10, 11, 33, 121, 129]. Sometimes the code is transformed into IR and then into a graph so as to retain control flow and/or data flow information. Some works also use the source code for extra benefit [52]. NCC [21] uses the code in the form of a contextual flow graph (XFG), a mix of control flow

graph (CFG) and data flow graph (DFG), for a number of classification tasks. Sometimes, the embedded information is fed into an RNN/LSTM network to learn sequences. This can be used to classify specific patterns in a code, such as malicious code detection [52, 80, 143].

Machine learning has also been used for memory pre-fetching [53, 116]. In [53], in order to address the memory bottleneck in program execution, the authors propose a model to predict what addresses to pre-fetch and load into the cache. They offer two models to this end: an embedding LSTM model to predict and pre-fetch the top 10 predictions; and an LSTM with clustering, where they create a vocabulary of common addresses during the training, model local context, cluster the address space and use k-means to determine the output. However, SEER targets GPUs and aims to work on the TB level rather than the kernel.

4.6 Conclusion

In this chapter, we propose SEER, an machine learning-based framework towards the end goal of predicting global load and store addresses accessed by a thread block in a GPU kernel using its PTX code representation and kernel parameters. The resulting read and write sets can then be used to detect a fine-grained dependency graph between two kernels in an effort to run them more efficiently using existing state-of-the-art fine-grained task scheduling paradigms, such as BlockMaestro.

Our results indicate that it is possible to detect the addresses to a certain extent. However, more work is required in order to improve the results and minimize false positives in our read and write sets, such as fine-tuning hyperparameters, more efficient network

architecture, better code representation and training on more data. The resulting framework could then be confidently used in order to minimize user intervention in determining fine-grained data dependencies among different kernels, even those containing non-static memory accesses.

Chapter 5

Conclusions

There is a growing trend for execution efficiency of data-dependent workloads on GPUs. Despite great support for massive data parallelism, traditional GPU architectures lack a decent support for handling data-dependent applications. Unnecessary coarse-grained inter-kernel synchronizations and numerous kernel launch overheads in GPU applications can further take away from a GPU's potential for a better performance. This dissertation offers several related methods in an effort to take advantage of an application's data-dependent design and, through a combination of fine-grained dependency support on the hardware and increased programmer transparency on the software, provide a greater speedup and more GPU utilization for a multitude of applications and dependency patterns.

In Chapter 2, we proposed Wireframe, a hardware-software solution which enables the expression of data dependencies in an application through conversion to a single-kernel format utilizing a dependency graph defined by the user, combined with a specialized level-bound TB scheduler, in order to increase the speedup of several GPU applications with a

wavefront dependency pattern. However, it had several disadvantages, such as a significant user burden to have knowledge of an application's algorithm, re-write the code in a special format, and provide it with a dependency graph, which can become tiresome with more complex, irregular, and larger workloads.

In Chapter 3, we proposed BlockMaestro, which utilizes static JIT analysis in order to extract data dependencies between adjacent kernels at kernel-launch-time, minimizing the user's intervention in the process. In addition, by pre-launching multiple adjacent kernels, it masks the kernel launch overhead within the execution of the previous kernels. Finally, using a fine-grained TB scheduler, it is able to keep track of the said kernels and execute TBs as soon as their dependency requirements are met. However, it was only able to process kernels with only static dependencies, i.e., dependencies that are known before runtime. Therefore, in the case of a kernel with non-static memory accesses, it would revert to the traditional coarse-grained dependencies for that kernel.

In Chapter 4, we proposed SEER, an ML framework trained in an effort to predicting a kernel's read and write address sets by the TB by analyzing a kernel's code structure, allowing it to estimate its access pattern even if it includes non-static memory accesses. It detects each global load and store instruction in a kernel, extracts their related information and context, including other instructions and parameters, and finally uses a deep neural network to estimate the kernel's address sets, which can then be used to construct the dependency graph between adjacent kernels.

Future work

There are numerous ways to improve the work done in this dissertation. For example, the SEER model can be improved both by fine-tuning hyperparameters and tuning the model's architecture. Furthermore, when the model is accurate enough for a significant number of applications, it can be included on the hardware to perform prediction on-chip, making it faster. However, such a change would also require a failsafe mechanism in the case a prediction was incorrect, leading to a TB being issued prematurely. The solution should be able to rollback a running TB's execution and re-run it once its requirement are found to be met. New challenges will be introduced with the passage of time. However, by continuously presenting novel and efficient upgrades and solutions on both the hardware side and the software side, it is possible to keep improving data-dependent parallelism in GPUs.

Bibliography

[1] Cacti 5.3 (rev 174). `http://quid.hpl.hp.com:9081/cacti/`.

[2] Cuda programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2016. Accessed: 09-27-2016.

[3] Cuda 9 features revealed: Volta, cooperative groups and more. `https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/`, 2017.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[5] Mohammad Abdel-Majeed, Daniel Wong, and Murali Annavaram. Warped gates: Gating aware scheduling and power gating for gpgpus. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 111–122, 2013.

[6] Mohammad Abdel-Majeed, Daniel Wong, Justin Kuang, and Murali Annavaram. Origami: Folding warps for energy efficient gpus. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Amirali Abdolrashidi, Hodjat Asghari Esfeden, Ali Jahanshahi, Kaustubh Singh, Nael Abu-Ghazaleh, and Daniel Wong. Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems. In *2021 48th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA). IEEE*, 2021.

[8] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet E Belviranli, Laxmi N Bhuyan, and Daniel Wong. Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 50)*, pages 600–611, 2017.

[9] Gargi Alavani, Kajal Varma, and Santonu Sarkar. Predicting execution time of cuda kernel using static analysis. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data &*

Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pages 948–955. IEEE, 2018.

[10] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[12] AMD. Atmi (asynchronous task and memory interface). `https://github.com/RadeonOpenCompute/atmi`, 2016. Accessed: 2020-06-11.

[13] Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael Abu-Ghazaleh. Corf: Coalescing operand register file for gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 701–714. ACM, 2019.

[14] Itai Avron and Ran Ginosar. Hardware scheduler performance on the plural many-core architecture. In *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*, MES '15, pages 48–51, New York, NY, USA. ACM.

[15] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[16] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.

[17] Jonathan Beaumont and Trevor Mudge. Fine-grained management of thread blocks for irregular applications. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 283–292. IEEE, 2019.

[18] Mehmet E. Belviranli, Chih-Hsun Chou, Laxmi N Bhuyan, and Rajiv Gupta. A paradigm shift in gp-gpu computing: task based execution of applications with dynamic data dependencies. In *Proceedings of the sixth international workshop on Data intensive distributed computing*, pages 29–34. ACM, 2014.

[19] Mehmet E Belviranli, Peng Deng, Laxmi N Bhuyan, Rajiv Gupta, and Qi Zhu. Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 25–35, 2015.

[20] Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Laxmi N Bhuyan. Juggler: a dependence-aware task-based execution framework for gpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–67. ACM, 2018.

[21] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336*, 2018.

[22] Lars Bergstrom and John Reppy. Nested data-parallelism on the gpu. In *ACM SIGPLAN Notices*, volume 47, pages 247–258. ACM, 2012.

[23] Berkin Bilgic, Berthold KP Horn, and Ichiro Masaki. Efficient integral image computation on the gpu. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 528–533. IEEE, 2010.

[24] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[25] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. Dynamic task parallelism with a gpu work-stealing runtime system. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 203–217. Springer, 2011.

[26] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[27] Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pages 407–419. IEEE, 2015.

[28] Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419, 2015.

[29] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[30] Li-Jhan Chen, Hsiang-Yun Cheng, Po-Han Wang, and Chia-Lin Yang. Improving gpgpu performance via cache locality aware thread block scheduling. *IEEE Computer Architecture Letters*, 16(2):127–131, 2017.

[31] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[32] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[33] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

[34] Nicos Christofides. *Graph theory: An algorithmic approach (Computer science and applied mathematics)*. Academic Press, Inc., 1975.

[35] Cray. El capitan. `https://www.cray.com/company/customers/lawrence-livermore-national-lab`, 2020. Accessed: 2020-04-14.

[36] Cray. Frontier. `https://www.cray.com/company/customers/oak-ridge-national-laboratory`, 2020. Accessed: 2020-04-14.

[37] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

[38] Peng Di, Hui Wu, Jingling Xue, Feng Wang, and Canqun Yang. Parallelizing sor for gpgpus using alternate loop tiling. *Parallel Computing*, 38(6):310–328, 2012.

[39] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[40] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[41] Hodjat Asghari Esfeden, Amirali Abdolrashidi, Shafiur Rahman, Daniel Wong, and Nael Abu-Ghazaleh. Bow: Breathing operand windows to exploit bypassing in gpus. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 996–1008. IEEE, 2020.

[42] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100. IEEE Computer Society, 2010.

[43] Naila Farooqui, Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–9, 2011.

[44] Alcides Fonseca, Bruno Cabral, João Rafael, and Ivo Correia. Automatic parallelization: Executing sequential programs on a task-based parallel runtime. *International Journal of Parallel Programming*, 44(6):1337–1358, 2016.

[45] HSA Foundation. Hsa platform system architecture specification 1.0 (jan 2015). `http://www.hsafoundation.com/standards/`, 2015. Accessed: 2020-08-20.

[46] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308. IEEE, 2013.

[47] Priyanka Ghosh, Yonghong Yan, and Barbara Chapman. *Support for dependency driven executions among openmp tasks*. IEEE, 2012.

[48] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. A prototype implementation of openmp task dependency support. In *International Workshop on OpenMP*, pages 128–140. Springer, 2013.

[49] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. Ieee, 2012.

[50] Gagan Gupta and Gurindar S Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 59–70. ACM, 2011.

[51] Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[52] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[53] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pages 1919–1928. PMLR, 2018.

[54] Ahmed E Helal, Ashwin M Aji, Michael L Chu, Bradford M Beckmann, and Wu-chun Feng. Adaptive task aggregation for high-performance sparse solvers on gpus. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 324–336. IEEE, 2019.

[55] Tayler Hicklin Hetherington, Maria Lubeznov, Deval Shah, and Tor M Aamodt. Edge: Event-driven gpu execution. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 337–353. IEEE, 2019.

[56] Richard D Hornung and Jeffrey A Keasler. The raja portability layer: overview and status. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[57] Kaixi Hou, Hao Wang, Wu-chun Feng, Jeffrey S Vetter, and Seyong Lee. Highly efficient compensation-based parallelism for wavefront loops on gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 276–285. IEEE, 2018.

[58] Muhammad Huzaifa, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D Sinclair, and Sarita V Adve. Inter-kernel reuse-aware thread block scheduling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(3):1–27, 2020.

[59] Hyeran Jeon, Hodjat Asghari Esfeden, Nael B Abu-Ghazaleh, Daniel Wong, and Sindhuja Elango. Locality-aware gpu register file. *IEEE Computer Architecture Letters*, 18(2):153–156, 2019.

[60] Adwait Jog et al. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406, New York, NY, USA, 2013. ACM.

[61] Pavel Karas. Efficient computation of morphological greyscale reconstruction. In *OASIcs-OpenAccess Series in Informatics*, volume 16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[62] Saeed Karimi-Bidhendi, Arghavan Arafati, Andrew L Cheng, Yilei Wu, Arash Kheradvar, and Hamid Jafarkhani. Fully-automated deep-learning segmentation of pediatric cardiovascular magnetic resonance of patients with complex congenital heart diseases. *Journal of Cardiovascular Magnetic Resonance*, 22(1):1–24, 2020.

[63] Saeed Karimi-Bidhendi, Faramarz Munshi, and Ashfaq Munshi. Scalable classification of univariate and multivariate time series. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1598–1605. IEEE, 2018.

[64] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Press, 2019.

[65] Anirudh Mohan Kaushik, Ashwin M Aji, Muhammad Amber Hassaan, Noel Chalmers, Noah Wolfe, Scott Moe, Sooraj Puthoor, and Bradford M Beckmann. Optimizing hyperplane sweep operations using asynchronous multi-grain gpu tasks. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 59–69. IEEE, 2019.

[66] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.

[67] Behram Khan, Daniel Goodman, Salman Khan, Will Toms, Paolo Faraboschi, Mikel Luján, and Ian Watson. Architectural support for task scheduling: hardware scheduling for dataflow on numa systems. *The Journal of Supercomputing*, 71(6):2309–2338, 2015.

[68] Farzad Khorasani, Hodjat Asghari Esfeden, Nael Abu-Ghazaleh, and Vivek Sarkar. In-register parameter caching for dynamic neural nets with virtual persistent processor specialization. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 377–389. IEEE, 2018.

[69] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. Regmutex: Inter-warp gpu register time-sharing. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 816–828. IEEE, 2018.

[70] Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, 2016.

[71] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[72] Scott J Krieder, Justin M Wozniak, Timothy Armstrong, Michael Wilde, Daniel S Katz, Benjamin Grimmer, Ian T Foster, and Ioan Raicu. Design and evaluation of the gemtc framework for gpu-enabled many-task computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 153–164. ACM, 2014.

[73] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974.

[74] Michael LeBeane, Brandon Potter, Abhisek Pan, Alexandru Dutu, Vinay Agarwala, Wonchan Lee, Deepak Majeti, Bibek Ghimire, Eric Van Tassell, Samuel Wasmundt, et al. Extended task queuing: Active messages for heterogeneous systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 933–944. IEEE, 2016.

[75] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271. IEEE, 2014.

[76] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020.

[77] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware cta clustering for modern gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–311. ACM, 2017.

[78] Mikko H Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237. IEEE, 1996.

[79] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147, 1996.

[80] Liu Liu, Bao-sheng Wang, Bo Yu, and Qiu-xi Zhong. Automatic malware classification and new malware detection using machine learning. *Frontiers of Information Technology & Electronic Engineering*, 18(9):1336–1347, 2017.

[81] Manju Mathews and Jisha P Abraham. Automatic code parallelization with openmp task constructs. In *Information Science (ICIS), International Conference on*, pages 233–238. IEEE, 2016.

[82] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[83] Shervin Minaee, Amirali Abdolrashidi, Hang Su, Mohammed Bennamoun, and David Zhang. Biometric recognition using deep learning: A survey. *arXiv preprint arXiv:1912.00271*, 2019.

[84] Alok Mishra, Martin Kong, and Barbara Chapman. Kernel fusion/decomposition for automatic gpu-offloading. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 283–284. IEEE, 2019.

[85] Sparsh Mittal. A survey of value prediction techniques for leveraging value locality. *Concurrency and computation: practice and experience*, 29(21):e4250, 2017.

[86] Meinard Müller. *Dynamic Time Warping*, pages 69–84. Springer Berlin Heidelberg, 2007.

[87] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[88] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 354–366. ACM, 2017.

[89] NVIDIA. Dynamic parallelism in cuda. `http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf`, 2012.

[90] NVIDIA. How to overlap data transfers in cuda c/c++. `https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/`, 2012. Accessed: 2019-11-17.

[91] NVIDIA. Cuda 9 features revealed: Volta, cooperative groups and more. `https://devblogs.nvidia.com/cuda-9-features-revealed/`, 2017. Accessed: 2019-11-17.

[92] NVIDIA. Cuda toolkit documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2018. Accessed: 2019-11-14.

[93] NVIDIA. Getting started with cuda graphs. `https://devblogs.nvidia.com/cuda-graphs/`, 2018. Accessed: 2019-11-17.

[94] NVIDIA. Nvidia turing gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, 2018. Accessed: 2019-11-13.

[95] NVIDIA. Parallel thread execution isa version 7.4. `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`, 2021. Accessed: 08-08-2021.

[96] Chanyoung Oh, Zhen Zheng, Xipeng Shen, Jidong Zhai, and Youngmin Yi. Gopipe: a granularity-oblivious programming framework for pipelined stencil executions on gpu. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 43–54, 2020.

[97] Marc S Orr, Bradford M Beckmann, Steven K Reinhardt, and David A Wood. Fine-grain task aggregation and coordination on gpus. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 181–192. IEEE Press, 2014.

[98] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 142–153. ACM, 2013.

[99] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch. *Computer software. Vers. 0.3*, 1, 2017.

[100] Arthur Perais and André Seznec. Practical data value speculation for future high-end processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 428–439. IEEE, 2014.

[101] Mahdieh Poostchi, Kannappan Palaniappan, Filiz Bunyak, Michela Becchi, and Guna Seetharaman. Efficient gpu implementation of the integral histogram. In *Asian Conference on Computer Vision*, pages 266–278. Springer, 2012.

[102] Kishore Punniyamurthy and Andreas Gerstlauer. Tafe: Thread address footprint estimation for capturing data/thread locality in gpu systems. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 17–29, 2020.

[103] Sooraj Puthoor, Ashwin M Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M Beckmann, and Gregory Rodgers. Implementing directed acyclic graphs with the heterogeneous system architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 53–62, 2016.

[104] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M Beckmann. Oversubscribed command queues in gpus. In *Proceedings of the 11th Workshop on General Purpose GPUs*, pages 50–60, 2018.

[105] Bo Qiao, Oliver Reiche, Frank Hannig, and Jïrgen Teich. From loop fusion to kernel fusion: a domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 242–253. IEEE, 2019.

[106] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. Automatic kernel fusion for image processing dsls. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 76–85. ACM, 2018.

[107] Kiran Ranganath, AmirAli Abdolrashidi, Shuaiwen Leon Song, and Daniel Wong. Speeding up collective communications through inter-gpu re-routing. *IEEE Computer Architecture Letters*, 18(2):128–131, 2019.

[108] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[109] Davide Rossetti, SC Team, et al. Gpudirect: Integrating the gpu with a network interface. In *GPU Technology Conference*, 2015.

[110] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[111] Edans Flavius de O Sandes and Alba Cristina MA de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013.

[112] Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. Timing characterization of openmp4 tasking model. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 157–166. IEEE Press, 2015.

[113] Rami Sheikh, Harold W Cain, and Raguram Damodaran. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–435, 2017.

[114] Rami Sheikh and Derek Hower. Efficient load value prediction using multiple predictors and filters. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 454–465. IEEE, 2019.

[115] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 214–227, 2018.

[116] Zhan Shi, Kevin Swersky, Daniel Tarlow, Parthasarathy Ranganathan, and Milad Hashemi. Learning execution through neural code fusion. *arXiv preprint arXiv:1906.07181*, 2019.

[117] Avinash Sodani and Gurindar S Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 205–215. IEEE, 1998.

[118] Michael Steffen and Joseph Zambreno. Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 237–248. IEEE, 2010.

[119] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. Flexible software profiling of gpu architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 185–197. IEEE, 2015.

[120] Rick Stevens, Jini Ramprakash, Paul Messina, Michael Papka, and Katherine Riley. Aurora: Argonne's next-generation exascale supercomputer. Technical report, ANL (Argonne National Laboratory (ANL), Argonne, IL (United States)), 2019.

[121] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[122] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. Power efficient sharing-aware gpu data management. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 698–707. IEEE, 2017.

[123] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T Kandemir, and Chita R Das. Controlled kernel launch for dynamic parallelism in gpus. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 649–660. IEEE, 2017.

[124] David Tarjan, Kevin Skadron, and Paulius Micikevicius. The art of performance tuning for cuda and manycore architectures.

[125] Devashree Tripathy, Amirali Abdolrashidi, Laxmi Narayan Bhuyan, Liang Zhou, and Daniel Wong. Paver: Locality graph-based thread block scheduling for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(3), 2021.

[126] Devashree Tripathy, Amirali Abdolrashidi, Quan Fan, Daniel Wong, and Manoranjan Satpathy. Localityguru: A ptx analyzer for extracting thread block-level locality in gpgpus. *Proceedings of the 15th IEEE/ACM International Conference on Networking, Architecture, and Storage*, 2021.

[127] Devashree Tripathy, Hadi Zamani, Debiprasanna Sahoo, Laxmi N Bhuyan, and Manoranjan Satpathy. Slumber: static-power management for gpgpu register files. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 109–114, 2020.

[128] Stanley Tzeng, Brandon Lloyd, and John D Owens. A gpu task-parallel model with dependency resolution. *Computer*, 45(8):0034–41, 2012.

[129] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–27, 2020.

[130] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.

[131] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of openmp dependent tasks with the kastors benchmark suite. In *International Workshop on OpenMP*, pages 16–29. Springer, 2014.

[132] Chao Wang, Junneng Zhang, Xi Li, Aili Wang, and Xuehai Zhou. Hardware implementation on fpga for task-level parallel dataflow execution engine. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2303–2315, 2016.

[133] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[134] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. *ACM SIGARCH Computer Architecture News*, 43(3S):528–540, 2015.

[135] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 583–595. IEEE, 2016.

[136] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 51–60. IEEE, 2014.

[137] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[138] Daniel Wong, Nam Sung Kim, and Murali Annavaram. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 176–187, 2016.

[139] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.

[140] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2433–2442. IEEE, 2012.

[141] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[142] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242. IEEE, 2016.

[143] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 52–63. IEEE, 2019.

[144] Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: Fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 229–238, New York, NY, USA, 2013. ACM.

[145] Yi Yang and Huiyang Zhou. Cuda-np: realizing nested thread-level parallelism in gpgpu applications. In *ACM SIGPLAN Notices*, volume 49, pages 93–106. ACM, 2014.

[146] Hadi Zamani, Yuanlai Liu, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. Greenmm: energy efficient gpu matrix multiplication through undervolting. In *Proceedings of the ACM International Conference on Supercomputing*, pages 308–318, 2019.

[147] Hadi Zamani, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. Saou: safe adaptive overclocking and undervolting for energy-efficient gpu computing. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 205–210, 2020.

[148] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599. IEEE, 2017.

[149] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Hiwaylib: A software framework for enabling high performance communications for heterogeneous pipeline computations. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 153–166, 2019.