UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Execution of Scientific Applications on Heterogeneous Architectures

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

Doctor of Philosophy

in

Computer Science

by

Mehmet Esat Belviranli

December 2016

Dissertation Committee:

Dr. Laxmi N. Bhyuan, Chairperson
Dr. Rajiv Gupta
Dr. Walid Najjar
Dr. Nael Abu-Ghazaleh

The Dissertation of Mehmet Esat Belviranli is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

It is my great pleasure to thank those who made this dissertation possible. I would never have been able to finish it without the guidance of my committee members, my wife, help from friends, and support from my family.

I would like to express my deepest gratitude to my advisor, Dr. Laxmi Bhuyan, for his excellent guidance, caring, patience, and providing me with a pleasant atmosphere for doing research. I am also very grateful to Dr. Rajiv Gupta for his previous contribution and endless help in my research. I would also like to thank Dr. Walid Najjar and Dr. Nael Abu-Ghazaleh for guiding my dissertation, giving precious advice and participating in my final defense committee.

Another person who made this study possible is my wife who always has been with me, giving moral support and taking care of everything on behalf of me when I am busy with my work.

I would also like to thank my dear parents, my brother and my sister. They were always supporting me and encouraging me with their best wishes. I also thank all members from my lab and my collaborators, who were always willing to help, discuss ideas, and give helpful suggestions.

Finally many thanks to all and friends for always being on my side throughout my Ph.D study. Without them, it would have been very difficult.

This dissertation includes content published in the following journals and proceedings:

- ACM Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers Volume 9 Issue 4 Article No. 57, January 2013.
- Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing, Pages 29-34.
- Proceedings of the 29th ACM on International Conference on Supercomputing, Pages 25-35.
- Proceedings of the 2016 International Conference on Supercomputing, Article No. 31.

I dedicate my work to the one who has planted

the seeds of faith, strength and courage into the deepest trenches of my heart

to guide me through the most challenging labyrinths of life...

ABSTRACT OF THE DISSERTATION

Efficient Execution of Scientific Applications on Heterogeneous Architectures

by

Mehmet Esat Belviranli

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2016
Dr. Laxmi N. Bhyuan, Chairperson

Today's heterogeneous architectures bring together multiple general purpose CPUs, domain specific GPUs and FPGAs to provide dramatic speedup for many applications. However, the challenge lies in utilizing these heterogeneous processors to optimize overall application performance so that workload completion time is minimized. Operating system and application development for these systems are in their infancy.

In this dissertation, we propose various techniques to improve overall system throughput on heterogeneous systems. We develop run-time and compile-time mechanisms to efficiently distribute the workload between various processors and accelerators, transfer the corresponding data to execute them. We explore various data partitioning, synchronization and scheduling schemes to improve load balance, maximize resource utilization and minimize the execution time. First, we propose a dynamic scheduling mechanism to incorporate all available processing units in the execution of a given parallel loop. Our scheme automatically detects the computation speed of each CPU and accelerator and distributes the workload accordingly during run-time. We, then, focus on improving data transfers over

PCI-e bus to further improve the system throughput in the existing of multiple applications sharing a single GPU. We present a framework to exploit automatic transfer/execution overlapping without requiring any modifications to source code.

We further improve heterogeneous system efficiency by optimizing in-GPU execution. We find that barrier synchronizations cause a bottleneck in performance. We develop a task based execution scheme that utilizes distributed queues to exploit better cache locality and inter-SM load balance. Finally, we introduce a tile-based wavefront execution technique by removing global barriers and employing a novel peer-SM synchronization mechanism. Through extensive experiments, we observe that our schemes significantly outperforms existing state-of-the-art approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Accelerators have played a remarkable role in high performance computing in the last decade. At least half of the top supercomputers today[1] employ GPUs, FPGAs or many-core architectures (e.g. Xeon-Phi) to obtain considerable speedup for scientific applications. The massive parallelism provided by such *heterogeneous architectures* may increase the computational throughput by hundreds to thousands of times compared to the serialized CPU execution [81].

Among the three of top-5 supercomputers in the world, as of early 2016, NUDT Tianhe-2 [95] and ORNL Titan [44], employ Xeon Phi many-core accelerators and NVIDIA Titan GPUs respectively, whereas K Computer [56] was powered entirely by SPARC64 RISC CPU cores. Table 1.1 gives a breakdown on the total CPU core and accelerator counts in these three systems along with their theoretical performances in PFLOPS. Both type of accelerators constitute the majority of total computational power in heterogeneous systems, with high power efficiency metrics around 2 Gflops/Watt. On the other hand, K Computer

[1]According to data in http://www.top500.org/list/2015/11/

	NUDT Tianhe-2	**ORNL Titan**	**K Computer**
CPU cores per node	24	16	8
Accelerators per node	3 (57 cores)	1 (14 SMs)	-
# of nodes	16,000	18,688	88,128
CPU PFLOPS	6.75	2.6	16.3
Accelerator PFLOPS	48.14	24.5	-
Total PFLOPS	54.9	27	16.3
Power (PFLOPS/Watt)	1.935	2.143	0.824

Table 1.1: Processor & node counts, performance and power breakdown of the three supercomputers in TOP-5 list (early 2016).

is significantly behind these two systems with higher power usage and lower processing capabilities. High processor utilization, minimized data transfer & synchronization cost and ease of programmability are among the most critical challenges in obtaining a performance close to the theoretical peak on these architectures.

Heterogeneous systems provide a hybrid execution environment that allows applications to run concurrently on accelerators and CPUs. Scientific applications benefit from these systems by *offloading* their data-parallel regions entirely to accelerators. The corresponding data as well as the output produced by the computation are transferred via the interconnect (i.e. pci-e bus) that links CPUs to such devices. Improving the system throughput in heterogeneous platforms relies mainly on *better processor utilization* and *efficient data transfers*[80, 26]. The heterogeneity across these architecturally diverse processing units introduces many challenges to application design. Programming these architectures requires careful consideration of computational capabilities of the host and accelerators. Moreover, since limited bandwidth of such interconnect prevents fine-granular interaction between CPUs and accelerators, any computation on the host side regarding the uploaded data is usually blocked until the offloaded execution finishes. Therefore the total speedup obtained by accelerating a parallel region is also affected by the data transfers.

Utilizing all processors in the system, including CPUs as well the SIMD cores inside accelerators, is key to achieving maximum performance in heterogeneous architectures. Improving utilization involves addressing the idleness in two different levels: To achieve *system-wide utilization* (1), all processing units attached to the server should be utilized in the execution of data-parallel regions. Even in the most state-of-art runtime systems, once the computation is offloaded to accelerators, CPUs are mostly left idle waiting for the output[26]. As the technology allows modern servers to employ CPUs with hundreds of cores, the possible performance benefits of using them for data-parallel computing along with the accelerators will be enormous. *Accelerator-level utilization* (2), on the other hand, targets to keep internal vector processing units (i.e. SMs in CUDA) occupied. These units usually become idle when their corresponding data is not ready for processing due unsatisfied dependencies[86] or there is insufficient amount of data transferred from host[87]. Applications should be designed to utilize all SIMD compute units along with their threads to exploit the massive parallelism offered by these devices.

Data transfer between CPUs and accelerators is also another major factor, and mostly a bottleneck [31], affecting the overall heterogeneous system throughput. Movements to/from the devices should consider the limited bandwidth as well the proper utilization of the underlying interconnect. For example, the PCI-e bus is often modeled as logP model [21] and sensitive to the amount of data being transmitted at once. This behavior requires the transfer size to be carefully determined so that neither the interconnect is under-utilized nor the computation is held back due to long-lasting data copy operations. Moreover, additional techniques like overlapping the transfers with execution are also helpful to further hide the slowdown caused by the data movement.

In this thesis we propose three ***solutions*** to the three problems given above to optimize scientific applications on heterogeneous architectures. We develop;

- A dynamic ***workload partitioning*** scheme to improve system-wide utilization,
- An automated ***transfer/execution overlapping*** framework to decrease data transfer overhead.
- ***Efficient inter-processor synchronization and scheduling*** techniques to increase accelerator-level utilization,

Workload partitioning is one of the most important issues that the heterogeneous programming research for data parallel applications targets to address. To achieve maximum gain and resource utilization on these architectures, the computation workload in an application should be distributed across accelerators and CPUs according to their computational capabilities. Workload partitioning often involves both input data distribution and assignment of different application tasks. The goal is *to find the optimal workload assignment for each heterogeneous computation unit.*

Heterogeneous data partitioning can be performed in two ways: *static* or *dynamic*. *Static partitioning* relies on estimating the run time of tasks and data transfer times between the CPUs and accelerators, before the execution begins. Estimations are based on theoretical accelerator capabilities[84], compile time parameters, or an offline training period over a fraction of the input data set[51]. However, the accuracy of such studies falls far behind representing the dynamic behaviors for these complex systems., on the other hand, relies on runtime to balance task and data workload among heterogeneous processors to better utilize them. It requires proper monitoring mechanisms for each processor type, identification of data as computational tasks, and accurate performance estimation throughout the execution. Dynamic partitioning involves various scheduling policies like work stealing and linear workload incrementing[36, 83]. However these studies fail to consider the non-linear relation between the workload size and processing time on accelerators.

We propose a dynamic heterogeneous scheduling scheme, *HDSS*, to partition the parallel loop iterations across CPUs, GPUs and FPGAs. Our scheme supports applications with or without iteration dependencies and characterizes device specific performance metrics on-the-fly for optimal load balancing. HDSS dynamically finds the minimum data size required for each type of computation unit to fully utilize that processor and the underlying interconnect. Our scheduler improves the execution by up to 219%. This research is presented in Chapter 3 of this dissertation.

Transfer/Execution overlapping is another common technique to increase overall system throughput in heterogeneous systems. Data transfer over the PCI-e bus is a major performance bottleneck for accelerated computing. Although general I/O improvement techniques like zero copy DMA transfer and WC buffers have been adopted by some GPUs [12, 28], architecture and application specific software and hardware techniques are also necessary to use them. Overlapping data copy operations with kernel execution has been one of the most popular software solutions to hide the transfer latency. There have been some application-specific studies [73, 40]; however, only a few researchers were able to generalize their techniques beyond a limited class of applications [52, 42] via HW modifications.

A notable study in recognizing the importance of data-transfer in GPU sharing is carried by Sengupta et al. [72]. Whenever applications request different type of GPU operations (i.e., D2H, KE, or H2D) around the same time, their scheduler tries to execute these calls concurrently on the corresponding resources, if they are idle. However, this work fails to exploit full benefits of transfer/execution overlapping due to their scheduler being unaware the predicted execution times of the transfers/computations currently being processed. A better resource utilization is possible by re-ordering the execution order of applications waiting to be served.

We introduce a fully automated data transfer and kernel execution overlapping framework, CuMAS, for multiple applications sharing the same heterogeneous system. CuMAS captures data copy and device execution calls on the host side and re-orders them to exploit maximum overlapping benefits. Our framework achieves near to optimal scheduling by estimating the durations of captured calls and delaying or prioritizing them across applications while maintaining the computation accuracy within an application. CuMAS improves total system throughput by up to 44% with minimal scheduling overhead. The design and evaluation of this study is presented in Chapter 4

Inter-processor synchronization between compute units of an accelerator in applications embedding data dependencies is commonly achieved via global barriers [37] to ensure the execution order between consecutive diagonals, including both element-level and tile-level. Similarly, in accelerators, more specifically in GPUs, diagonals are processed in different kernel launches and these calls are separated by cudaDeviceSync() operations which act as global barriers. However, they cause two main issues: loss of data locality and compute unit idleness.

First problem with global barriers is their degrading effect on preserving locality across L1 and L2 caches. For example, in most GPU wavefront implementations[93, 25, 24], each tile-diagonal is separated by different kernel launches and each tile in a diagonal is represented by a thread block (TB). Similarly GPU implementations of BFS algorithm process each *breadth* in lock-step fashion, where each kernel launch process the next breadth. TBs in a kernel are assigned to SMs in round robin fashion with no specific ordering or affinity during run-time. Since threads in TBs access elements/nodes assigned to neighboring TBs, across different kernel launches, relevant elements/node become evicted from caches due to loss of both temporal and spatial locality. To solve this problem, TB (i.e. task) scheduling should be made SM-locality aware by using the indirect locality information implied via dependencies.

We develop a task based GPU execution framewok to exploit better data locality across SMs while keeping the load balanced. Our framework employs an in-GPU scheduler and enables SMs to operate as independent processing units (i.e. workers). Our approach replaces global barriers with inter-task synchronization. Unlike existing central queue based task schedulers [3, 86], our scheduler operates on distributed queues to decrease global memory pressure due to queue access contention. The scheduler kernel continuously checks the tasks output by worker kernels and inserts new tasks as their dependencies are resolved. We compare our framework with central queue based task scheduling and global barriers approach and achieve speedups up to 2x and 1.2x, respectively. The design and analysis of our queue-based software scheduler is presented in Chapter 5.

Second problem with global barriers is compute unit (i.e. SM) idleness. Element and tile dependencies in original wavefront computation are local. However, global barriers force a diagonal-wise wait and they cause some processors to stay idle although they could have proceed with the tiles whose dependencies are already satisfied. This is particularly true for applications like wavefront, where the dependency is limited only to a few tiles.

The processor under-utilization problem with global barrier based wavefront implementations have previously been addressed by Manjikian et.al [54] for multi-CPU platforms. In this study, each row of tiles are statically assigned to separate processors, and the synchronization is necessary only between neighboring CPUs operating on tiles with direct dependencies. This local synchronization technique eliminates the need for global barriers and improves the utilization significantly. On the other hand in GPUs, since there are no efficient synchronization primitives between SMs, removing global barriers becomes non-trivial and no previous research has attempted to solve this problem.

We then introduce a novel peer-SM synchronization technique, Peer-Wave, to eliminate global barriers and improve intra-SM and inter-SM utilization. Our approach relies on one-way flags that provides direct communication between SMs corresponding to dependent tiles. Peer-Wave also minimizes intra-SM under-utilization by using flexible hyper-tiles, which proposes a hyperplane index transformation technique to utilize all threads in the TB for all diagonals in a tile. Our approach further decreases SM idle times by determining tile sizes using a performance estimation model designed uniquely for wavefront execution on GPUs. Our proposed technique achieves up to 2x speedup against global barrier based execution of wavefront computations on GPUs. The design and analysis of a peer-SM synchronization is presented in Chapter 6

In this dissertation, we introduce various techniques to improve overall system throughput on heterogeneous architectures while executing scientific applications. The dissertation is organized as follows:

- We develop a dynamic partitioning algorithm for applications with independent and dependent loop iterations across CPUs, FPGAs and GPUs in Chapter 3.
- Then, we introduce a generic multi application scheduling framework to improve pci-e based data transfers in Chapter 4.
- We design a task-based GPU execution framework to increase accelerator speedup via locality-aware distributed queues in Chapter 5.
- Finally, we propose a peer-SM synchronization mechanism to remove global barriers and improve accelerator utilization further in Chapter 6.

We will first give background information and related work on the topics covered in this paper and elaborate on the details the techniques listed above in four separate chapters. Then we conclude the thesis with the final chapter.

Chapter 2

Related Work

This chapter covers the literature related to the studies proposed in this thesis.

2.1 Heterogeneous Workload Partitioning

Heterogeneous multiprocessor systems featuring hardware accelerators have been used for a wide range of applications yielding significant speedups compared to CPU only execution. Each type of hardware accelerator, such as FPGA or GPU, provides substantial performance improvement for the applications within its target domain. Image processing [23] , data mining [6], and bioinformatics [33] are examples of applications with hardware implementations on FPGA and K-Means [14], AES encryption [96], and network packet processing[75] are examples of GPU accelerated applications. Heterogeneous architectures like Cray XD1[27] and SGI Altix 4700[74] employ FPGAs as accelerators, whereas nVidia Tesla [49] employs GPUs for acceleration.

2.1.1 Static partitioning

Static partitioning techniques are ones where the developers estimate running times of tasks and/or time to process a range of data so that accelerator and CPU utilization can be maximized. Many applications [29, 100, 71, 82, 85] have employed various compile time data and task partitioning schemes. However, many of them deal with either reconfigurable or GPU architectures only without considering simultaneous execution on the CPUs as well.

One of the most notable research work on heterogeneous systems, Axel[84], models communication and computation capabilities specific to each hardware configuration. For each processor, data transfer and execution times are estimated offline based on bandwidth speed as well as processor count and frequencies of the processor. The disadvantage is that the model must be updated for each application based on how it performs on that specific hardware. Also, theoretical capabilities of accelerators are very prone to misrepresenting actual runtime performance.

Qilin[51], another important study on static workload partitioning, builds performance models per accelerator and CPU by measuring transfer and execution times during predefined training runs. These models are used to estimate the workload balance across processing units for the actual run. The models are again application specific and only represent the actual performance for a limited subset of the data. Therefore, static partitioning techniques fail to address run time dynamics of the application and underlying architecture.

Some studies have applied graph algorithms to task graphs of heterogeneous applications. Kim et al. [46] have built an architecture-dependent mathematical model to partition the tasks in the DAG of tiled QR decomposition. Load balancing is achieved by distributing equal sized matrix tiles across heterogeneous processing units. Kedad-Sidhoum et al. [10], builds a heuristic based model to minimize the makespan of a given task graph. However, these techniques focused on functional partitioning of tasks to CPUs and accelerations rather than data based workload assignment.

2.1.2 Dynamic partitioning

Dynamic partitioning, on the other hand, can more accurately balance task and data workload among heterogeneous processors[3, 36, 83], and can better utilize all computational units during execution. Studies in [3, 36] use work stealing schemes which are not suitable for accelerator based systems due to possible redundant data transfers. The work to be stolen by CPU should have already been transferred to GPU, therefore the time spent on data transfer will be wasted.

The work in [83] uses two basic scheduling policies, exponential incremental and linear incremental. These schedulers increase task size exponentially (by a factor of m) or linearly (increasing by n) for each time a processor request a block. In both cases, faster processing units reach bigger task sizes more quickly as execution progresses. However, exponentially growing task sizes increases the likelihood of load imbalance due to assignment of large blocks to slow processors towards the end of the execution. On the other hand increasing linearly causes the use of excessive amount of blocks which decreases accelerator utilization.

Song, et. al. [76] re-design Cholesky and QR factorizations by using different sized sub-matrices to accommodate the processor heterogeneity. They introduce an autotuning method to determine the sub-matrix sizes to attain both high performance and load balancing. Small tiles are assigned to multi cores CPUs whereas big tiles are processed by GPUs. However, all CPUs are involved in processing a single tile, therefore introducing significant loss of locality.

2.1.3 Self Scheduling

A simple form of dynamic scheduling is self scheduling, where each processor decides how much load to grab from a shared pool of iterations. Guided self scheduling[65] is one of the earliest studies designed for homogeneous CPUs and suggests each processor to grab $1/n$ of the remaining workload, where n is the total number CPUs in the system. Self scheduling is effective in load balancing in homogeneous processors, however, fails to improve execution when the compute powers of processors are different.

To account for the heterogeneity in multiprocessor systems, a common solution is to assign iteration blocks according to a weight factor representing processing speed of each processor. Early approaches used fixed weight factors determined at compile time with limited success[39]. Variations in processor speed lead to adaptive schemes which measures weights and changes the workload during run-time. Adaptive weighted factoring[7] cumulatively updates computational weights per each processors after each block assignment. Similarly, existing self scheduling algorithms have also been modified to adjust the block sizes dynamically based on measured weight factors[98, 18, 19, 99]. However, since accelerator show a non-linear utilization curve, therefore cannot be represented with a single variable, none of these studies were efficient on keeping the heterogeneous load balanced across CPUs and devices.

2.2 Transfer and Execution Overlapping

2.2.1 Multi application scheduling

Ravi et al.[66] identified and proposed SW based concurrent kernel execution mechanisms using spatial and temporal sharing. They characterized several kernels and paired them using the two sharing methodologies. Adriaens et al.[2] have implemented HW based spatial sharing solutions to improve utilization in the existence of smaller kernels with less thread blocks. Elastic kernels proposed in [63] use different concurrency policies for various classes of kernels. The study in [79] developed two HW based preemptive execution policies, context switch and drain, to replace a running thread block with another. Chimera [64] has improved the pre-emption with an additional policy, flush, allowed dynamic selection of policies depending on the load characteristics. Jog et al.[43] have studied the on-GPU memory system to improve throughput in the existence of concurrent kernels.

The researches in [42] and [8] have proposed SW runtime environments to handle data allocation and transfers on-the-fly by keeping track of dependencies across kernel executions. Using a similar technique, Sajjapongse et al.[68] distributed kernels to multiple GPUs, to reduce the wait times on kernel dependencies. TimeGraph [45] is a driver-level GPU scheduler for graphics APIs and it supports various priority-aware scheduling policies.

2.2.2 Automated transfer / execution overlapping

Huynh et al.[40] proposed a transfer/execution overlapping enabled framework for streaming applications written using StreamIt language. Similarly, GStream[73] provides a graph processing methodology to handle overlapping. Among the few general purpose automatic overlapping work, Lustig et al.[52] proposed HW extensions to enable the detection of the transferred bits so that the kernel can start execution as soon as the required data arrives. PTask [67] is an OS task scheduler extension and uses existing OS scheduling policies and an API to exploit transfer/execution overlapping. However, this work does not employ task re-ordering. Helium [53] focuses on the task graph of a single application and performs a compiler-based DAG analysis to exploit any overlapping if possible. However the optimizations in this work are limited to a single application.

2.2.3 Kernel performance estimation

Hong et al.[38] have built an analytical model to estimate the overall execution time of a given kernel based on the number of parallel memory requests issued within a warp. A later study by Baghsorkhi et al.[5] has proposed an adaptive performance model to identify the bottlenecks in execution flow by taking bank conflicts, control divergence and uncoalesced memory accesses into account, in warp level. Both studies have used average warp execution times to find thread block execution times and scale them to the total number of TBs. A more recent study[92] have proposed machine learning techniques for scalable performance estimation on varying architectures and applications. They employ a runtime that collects performance counters and feeds them to a neural network that decides which scaling curve should be applied to properly estimate the execution time based on the application characteristics and number of SMs employed by the GPU.

2.3 Inter-SM Synchronization

2.3.1 Global Barriers

To ensure dependencies between elements and tiles, traditional wavefront parallelism employs global barriers in between the executions of diagonals at both intra-tile and inter-tile levels. The lack of efficient inter-SM communication mechanisms makes global synchronization the only practical solution for wavefront parallelism on GPUs. In CUDA, inter-tile synchronization may be obtained via CPU or GPU initiated *cudaDeviceSync()* calls placed between kernel launches each of which process a tile-diagonal. Similarly, intra-tile barriers can be implemented by the *syncThreads()* function to synchronize all the threads inside a single TB. Although intra-tile synchronization is fast and efficiently implemented by the SIMD hardware, inter-processor (i.e., inter-tile) barrier synchronization is known to have large overheads both in CPU and GPU implementations. Feng et al. [94] tried to improve the performance of barrier synchronization in GPUs by employing several techniques used for CPUs like the tree-based and lock-free synchronization. However, global synchronization is an overkill as dependencies exist only between neighboring tiles. Instead we require an inter-SM distributed synchronization scheme.

2.3.2 Inter-SM Communication

Global synchronization overhead of barriers is present as there is no HW support on GPUs to enable efficient inter-SM synchronization. A naive way in wavefront parallelism is to break execution into per-diagonal kernel launches where the barriers are provided via cudaDeviceSync() calls. In-kernel global inter-SM synchronization has been studied in [94] where mutex-tree and lock-free queue based methods are used to speed up synchronization. However all these approaches still require all SMs to communicate with each other and thus increases contention on device memory.

2.3.3 Task Based Execution

There have been several studies on high level task management schemes to distribute a workload across multiple heterogeneous processors including CPUs and GPUs[9, 3, 36, 83, 20]. They divide the workload into variable sized blocks and dynamically assign them to processors as they become idle. Although they target a balanced distribution, they stay at a coarse-granular level where whole GPU is considered as a single processor entity. They don't consider the distribution between the SMs inside the GPU.

Some recent studies proposed a finer granular solution by assigning tasks into multiple streams attached to available CPUs and GPUs in the system [88, 40]. Similarly Sanchez et.al. [69] proposes a pipeline based approach by dividing tasks into smaller blocks. Although all these solutions exploit concurrent kernel execution on GPUs via asynchronized streams and pipelines, they do not to schedule tasks at the SM level, hence fail to exploit a true SM-aware task-based execution scheme.

Only a few studies [62, 17, 86] have brought the paradigm into a finer granular level where each SM is treated as standalone processing unit. Okuyama et.al.[62] exploits inter-TB task-level parallelism for a specific problem by a CPU controlled execution scheme. They implement optimizations to decrease CPU-GPU memory transfer between CPU scheduler and GPU. Another study[17] proposes a similar CPU oriented model where a scheduler is running on the host side and continuously tracking the progress of workers via a centralized task queue located in the global memory. Although these two studies models task-management in a fine-granular level by considering SM loads inside GPU, their host controlled scheduling requires excessive memory transfers between CPU and GPU via PCI-express bus. Limited bandwidth and continuous contention are the two obstacles for the effectiveness of such a design.

The study by Tzeng. et. a.l[86] uses the idea of 'persistent worker threads' bound to SMs. They propose a centralized queue solution where each worker repeatedly tries to grab a task from this queue. Once a task is processed, the dependencies are again solved by the worker threads. A major problem with this approach is that the workers need to synchronize at the point where they access the queues and also while iterating through dependencies. This causes a considerable divergence in the execution path of each SM.

Chapter 3

Dynamic Scheduling across Heterogeneous Processing Units

3.1 Introduction

Heterogeneous systems provide a hybrid execution environment that allows applications to run concurrently on accelerators and CPUs. These systems take advantage of special purpose hardware by providing a hybrid execution environment for applications[4]. To fully utilize such systems, applications should be run on accelerators and CPUs concurrently. However, programming heterogeneous architectures requires careful consideration of underlying computational capabilities and memory bandwidth between the host and the accelerators.

To achieve maximum gain and resource utilization on these architectures, the computation workload in an application should be distributed across accelerators and CPUs according to their computational capabilities. Workload partitioning often involves both input data distribution and assignment of different application tasks.

An ideal dynamic load balancing scheme should be able to achieve minimum possible amount of execution time without any offline runs. For heterogeneous architectures, two factors are critical to achieve such an ideal balancing: 1) accurate measurement of computational weights per each processing unit and 2) the ideal size of computational task and data (i.e., *block size*) that is sent to an accelerator during each task assignment. If the block size is too small, data transfer to the device will take most of the time due to underutilization of DMA and, the device initialization overhead will be excessive. On the other hand, large blocks will lead to better performance in accelerators; however, taking this to an extreme will result in workload imbalance. Finding the right block size will minimize underutilization and load imbalance; thus, yielding the shortest execution time. To best of our knowledge, none of the existing dynamic heterogeneous scheduling algorithms has considered effects of block sizes on accelerator performance.

In this chapter, we present HDSS, a Heterogeneous Dynamic Self Scheduler, a new dynamic load balancing scheme for loop iterations on heterogeneous architectures. HDSS uses *weighted self scheduling* to fully utilize all available processing units in the system during the application lifetime. Our algorithm, dynamically resizes blocks to prevent underutilization and load imbalance of GPUs or FPGAs due to small or large block sizes. Unlike existing approaches, HDSS does not require an offline training period or device and application specific performance calculations. Moreover, HDSS is also able to schedule loops with dependent iterations as well via wavefront parallelization.

For CPU+FPGA experiments, we have developed FPGA versions of Blackscholes and Histogram applications. We have implemented and compared our load balancing scheme with the recent major studies on heterogeneous load balancing: Axel[84], Qilin[51], and Monte-Carlo[83]. The results show that HDSS provides performance improvements of up to 219% when compared to its closest load balancing scheme. Results also show that HDSS is able to minimize processor idle times throughout the execution. We present HDSS along with a generic C++ API for easy adaptation.

This study makes the following contributions:

- A new self scheduling algorithm, HDSS, optimized for heterogeneous architectures is presented. The algorithm is block size aware and achieves near optimal utilization for all processors without any need for offline training.
- The algorithm has been implemented and tested on CPU+GPU and CPU+FPGA platforms. Our algorithm is able to increase performance by up to 219% when compared to the best performing existing algorithm. Moreover, HDSS achieves running times that are very close to those achieved by ideal partitioning.
- An API has been developed for easy adaptation of the algorithm for new/existing parallel applications.

The remainder of this chapter is organized as follows: Section 2 presents motivations behind our scheme. Section 3 describes how HDSS works in detail. Section 4 presents our performance evaluation and comparison to previous approaches.

3.2 Motivation

3.2.1 Effect of Block Size

A common approach in loop scheduling is to divide the entire data range into small groups of iterations, called *blocks*. Throughout the rest of this chapter, we will use the term *block* to refer to a chunk of data sent to a computational unit for processing. The term *block size* will be used to refer to the quantity of data (e.g., number of array indicies, loop iterations, bytes etc.) in the block. Division of a large workload into smaller blocks reduces the idling of some processors due to waiting for other busy processors towards the end of the execution. For heterogeneous systems, the data is also sent in blocks to the accelerator memory before execution starts. We have observed that size of the blocks have much more impact on the performance of heterogeneous systems than on multi-core systems.

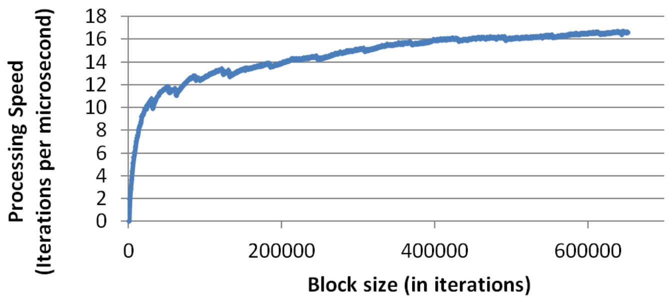


Figure 3.1: GPU Processing speed for varying block sizes.

To observe the effects of block sizes on accelerated computing, we conducted an experiment on our CPU + GPU system. In this experiment we ran the Blacksholes application on the nVidia C2050 GPU while linearly increasing the size of the blocks sent to GPU from 1024 to 67M. After execution of each block, the amount of time to process the block is divided by the size of the block and plotted as the processing rate. Figure 3.1 shows how accelerator performance, quantified in terms of 'iterations executed per microsecond', changes with the block size. Processing rate increases rapidly with the block size initially and then reaches stability. Several factors play a role in causing this type of behavior. A primary reason is due to the fact that accelerators usually overlap computation and communication so that input data is processed in a streaming fashion. When the amount of data to be sent to the GPU is small, it is underutilized and the optimization is less effective due to lack of data. Direct memory access (DMA) is also underutilized for small amount of data transfers. Another reason is the underutilization of possible pipelined or threaded parallelism implemented inside the hardware accelerator. If the size of the block is smaller than the pipeline steps, or the number of parallel execution units inside the accelerator, then the full computing power of the accelerators will not be exploited. The last reason is the constant initialization overhead associated with the use of accelerators. Small block sizes result in a greater number of blocks which increases the total time spent on initialization and hence an increases the execution time.

Thus, we can conclude that while small blocks provide better load balancing, large block sizes achieve higher utilization. An ideal scheduling algorithm should address this trade-off by using large enough blocks for better performance while keeping the workload balanced so that the threads finish their execution at nearly the same time.

3.2.2 Computational Rates (Weights)

Processors in a heterogeneous system have varying amount of computational powers. Hardware characteristics of the device and computational requirements of the application together determine the computational power of a processor. *Computational rate*, which represents the relative speed of each unit, is necessary for estimating correct fraction of workload to be assigned to the processing unit.

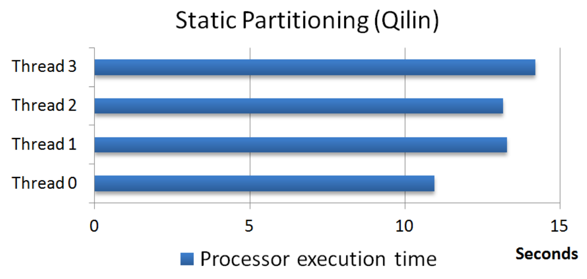


Figure 3.2: Qilin[51] workload distribution with 1 GPU (Thread 0) + 3 CPU threaded execution of Blackscholes application.

Wrongly estimated compute rates will overload slow processors with excessive workload starting from the first assignment. This would cause faster accelerators to wait for the slow processors when there are no more iterations remaining to execute. Figure 3.2 shows how our implementation of a previous static partitioning scheme, Qilin[51], on the CPU+GPU system fails to balance the workload for the Blackscholes application. Computation power of GPU, running as thread 0, has been underestimated during the offline training and thus resulted in idle periods for the GPU. Thread 0, the GPU thread, has been assigned less workload than it can actually handle and it stays idle towards end of the execution. Offline training was not successful at producing correct compute ratios for GPU and CPUs.

3.2.3 Online Training

In this study we use online training that allows us to capture the compute rates (weights) on the fly without wasting any time on a prior offline run. The length of the online training should be long enough, since an overly short training period may yield inaccurate compute weights. Online training should carefully measure the vastly different computational powers of the executional units by assigning blocks of loop iterations to processors and tracking the time required to process them.

The effects of block size on performance observed in Figure 3.1 implies that the training period should find the correct computational rates by testing with increasing number of block sizes until the weights do not change. However, once correct rates are found, large blocks should be avoided towards end of execution since that will increase the chance of load imbalance by keeping slow processors busy at the end.

To address these concerns, we developed a dynamic self scheduling scheme with a two-phase algorithm: the first phase starts with small blocks and increases them gradually until it determines accurate compute rates (weights). Then we switch to the second phase which uses large blocks initially, then decreases block sizes as it reaches completion to ensure that all units complete execution at nearly the same time.

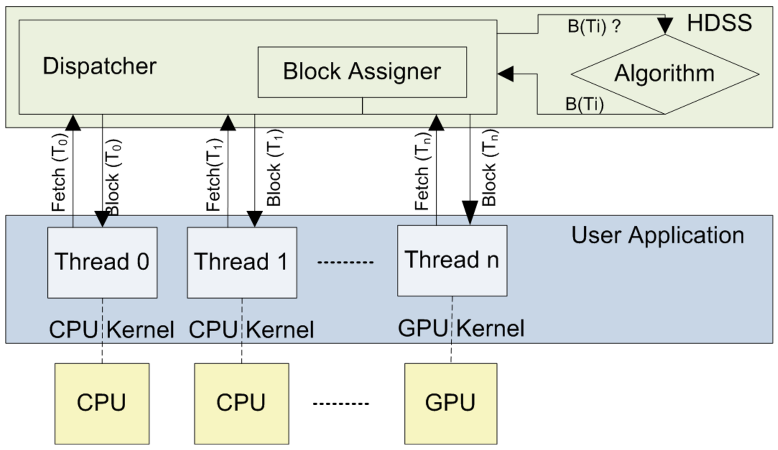


Figure 3.3: Overall Mechanism of HDSS

3.3 HDSS: Heterogeneous Dynamic Self Scheduler

3.3.1 Overview

In a typical data-parallel application, parallel threads simultaneously process a specific portion of the shared input data in chunks (blocks). When all data is processed, parallel threads merge and the application either terminates or proceeds to the next phase of computation. Determining the amount of data for each thread during this parallel processing phase is the responsibility of workload scheduler.

HDSS is a self scheduling scheme which assigns blocks of loop iterations (i.e., input data) to each processing unit (i.e., CPUs, GPUs, FPGAs) when they become idle. The overall mechanism of HDSS is given in Figure 3.3. In HDSS, each processing unit P_i is represented by a dedicated parallel *thread* T_i (i.e. *representative thread*) which is responsible for fetching next available block and executing device specific binary for the retrieved block of iteration. For CPUs, representative threads are basically main threads which handle the execution, whereas they are responsible for transferring input/output data and launching computation kernel for GPUs and FPGAs.

HDSS Dispatcher: Fetch request is satisfied by the *HDSSDispatcher* which internally calls HDSS algorithm to find the right block size $B(T_i)$ for the requesting thread T_i. Once a block size $B(T_i)$ is determined, next unprocessed iteration block, $Block(T_i)$, with size $B(T_i)$ is created by the *Block Assigner* and returned to the requesting thread.

HDSS is a non-centralized scheduler, where the scheduling algorithm is run by the thread requesting the block. Scheduling overhead is distributed across representative threads, therefore increasing the scalability of our scheme considerably. Parallel runs of HDSS algorithm need only to be synchronized when the global data pointer for the processed data needs to be advanced after a block assignment. However, no computation is needed during this synchronization and the overhead is very small and negligible.

API and Usage: HDSS provides a simple API, *HDSSDispatcher* class, to communicate with the parallel threads representing heterogeneous processors. As shown in Figure 3.4 (a), *HDSSDispatcher* has a constructor that takes total number of iterations, number of processors, initial block sizes for each processor, block size factors for each processor, and maximum length of adaptive phase as parameters. Initial block sizes and block size factors have a default value of 128 and maximum length of adaptive phase is taken as 20 percent of total number iterations by default. *fetch_block* method simply takes *thread id* as argument and returns *start* and *end* indicies of the iteration space that is assigned by HDSS to requesting thread.

```
// HDSS public API
class HDSSDispatcher{
    HDSSDispatcher(
        int n_processors,
        int n_iterations,
        int initial_block_sizes[],
        int block_factors[],
        float max_adaptive_percent);

    void fetch_block(
        int thread_id,
        int& start_index,
        int& block_size);
};
```

(a) The constructor and the *fetch_block* method

```
// Shared HDSS instance created by main
HDSSDispatcher hdss_dispatcher(
    n_iterations,
    n_processors);

// Executed by each representative thread
while (hdss_dispatcher.fetch_block(
        t_id, startIndex, blockSize)==1){
    if (t_id == GPU_THREAD_ID){
     runOnGPU(startIndex, blockSize);
    }
    else{
     runOnCPU(startIndex, blockSize);
    }
}
```

(b) Example usage of HDSS via representative threads

Figure 3.4: HDSS API and its usage

Figure 3.4 (b) shows the usage of HDSS. For each application, a shared instance of *HDSSDispatcher* object is created. Whenever a representative thread for a processor becomes idle, *fetch_block* is called to get new workload. This method is blocking when there are no available blocks whose dependency constraints are satisfied. Once a block size is retrieved, the programmer must call processor specific implementations (kernels) of their application on the determined data block.

3.3.2 The Algorithm

Overview: HDSS algorithm determines the ideal block size per processor using a two phased approach. The initial phase, called the *adaptive phase*, is responsible for accurately finding the computational weights which reflect the relative processor speeds. This phase processes a relatively small amount of data whose upper bound is set as a

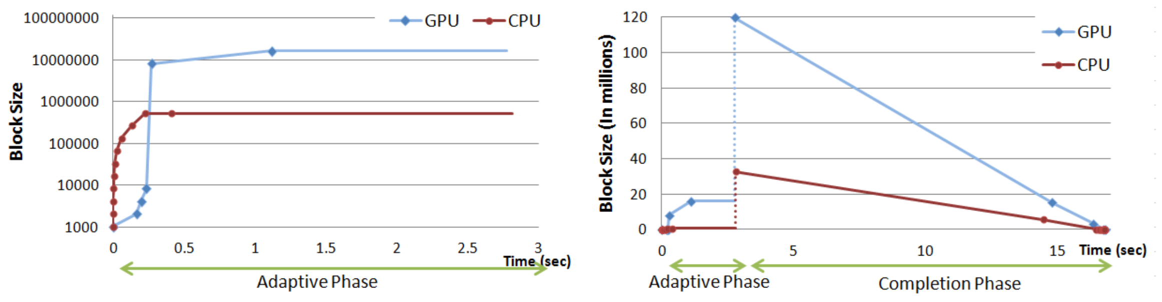


Figure 3.5: An example showing block assignments by time for a two threaded (1 GPU and 1 CPU thread) run of Histogram application on CPU+GPU system. Figure (a), on the left, illustrates the assignments dispatched during the adaptive phase. In this figure, block size axis is shown in logarithmic scale for a better separation of block assignments in the chart. Figure (b), on the right, shows the two phases of the algorithm separated with a clear jump in assigned block sizes just after computational weights are determined and stabilized.

fixed percentage of the total data. The final phase, called *completion phase*, processes the remainder of the data using the weights computed in the adaptive phase. This phase starts with the largest possible blocks to optimize the execution time by reducing the dispatching overhead. Block size decreases steadily as the computation works towards end so that all units complete execution at nearly the same time. As an example, the progression of block sizes during adaptive phase is illustrated in Figure 3.5.a. The experiment is for Histogram application on CPU+GPU system with 210M inputs. The overall process including completion phase is illustrated in Figure 3.5.b for the same experiment. The sudden increase in the curve is the moment where HDSS assigns largest possible block for the very first block request of the second phase.

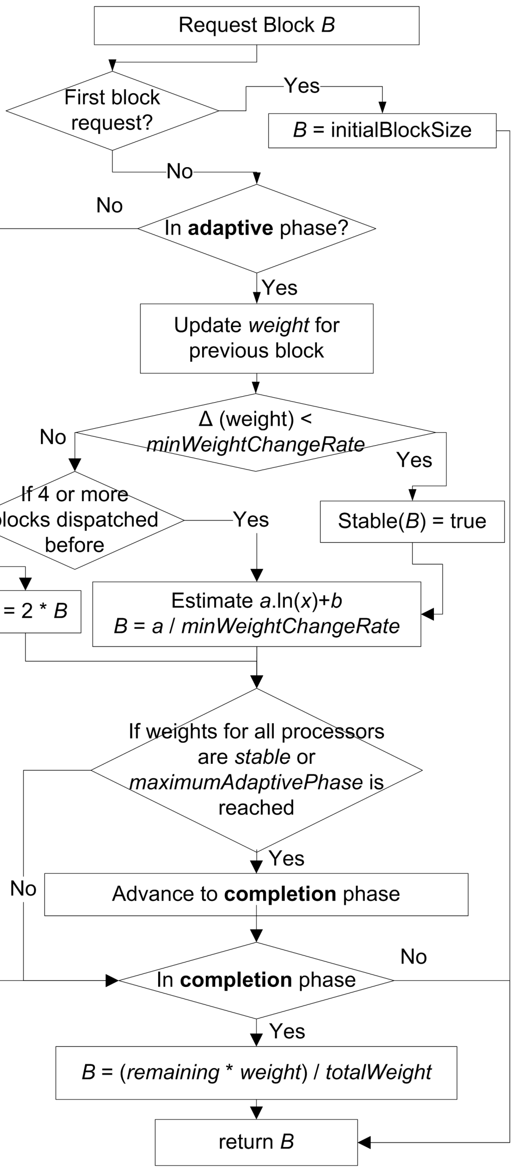

Request Block B
First block request?
Yes
B = initialBlockSize
No
In adaptive phase?
No
Yes
Update weight for previous block
Δ (weight) < minWeightChangeRate
No
Yes
If 4 or more blocks dispatched before
No
Yes
Stable(B) = true
B = 2 * B
Estimate a.ln(x)+b
B = a / minWeightChangeRate
If weights for all processors are stable or maximumAdaptivePhase is reached
Yes
No
Advance to completion phase
In completion phase
No
Yes
B = (remaining * weight) / totalWeight
return B

Figure 3.6: Flow chart for the HDSS algortihm

A flow chart for our proposed self scheduling algorithm, HDSS, is given in Figure 3.6. Our algorithm stores the remaining number of iterations and total computational weight as global variables. In addition, each processing element has the following attributes:

- The initial block size (user defined)

- Computational weight in iterations per micro second.
- Block size factor (user defined)
- Dynamic list of blocks that has been assigned to the processor before. Each entry in the list contains the size of the block as well as the execution start and end times of the block.

For the first block assignment, initial block size parameter for the requesting processor is set as the block size. During subsequent block requests, HDSS decides whether to continue with adaptive phase or move to the next phase based on dynamics of the execution. Once HDSS concludes that the adaptive phase should be terminated, all subsequent block assignments by all processors are calculated by the completion phase.

Computational rates (weights) are calculated based on the total time required for both data transfer (in/out) and execution time. This is essential for accurate load balancing as our results will demonstrate.

Adaptive phase: This phase finds the relative speed of processors resulting in weight factors to be used to assign workload in the completion phase. The main challenge is to keep the initial adaptive phase as short as possible so that the bulk of the data is left for the final completion phase.

To achieve this, we start with small block sizes initially and increase them gradually based on the observed performance until we are able to determine accurate computation weights for each device. For an accelerator, ideal starting block size would be a multiple of the number of threads inside that accelerator.

Adaptive phase first measures the number of iterations completed in the previous step per unit time. This gives us the computational weight, a relative processing speed of that processor for the last step. If the change in weights between previous step and current step is less than MIN_CHANGE_RATE (which is set as 0.01), current weight for that processor is considered as stable. Otherwise, we need to increase block size for the next step.

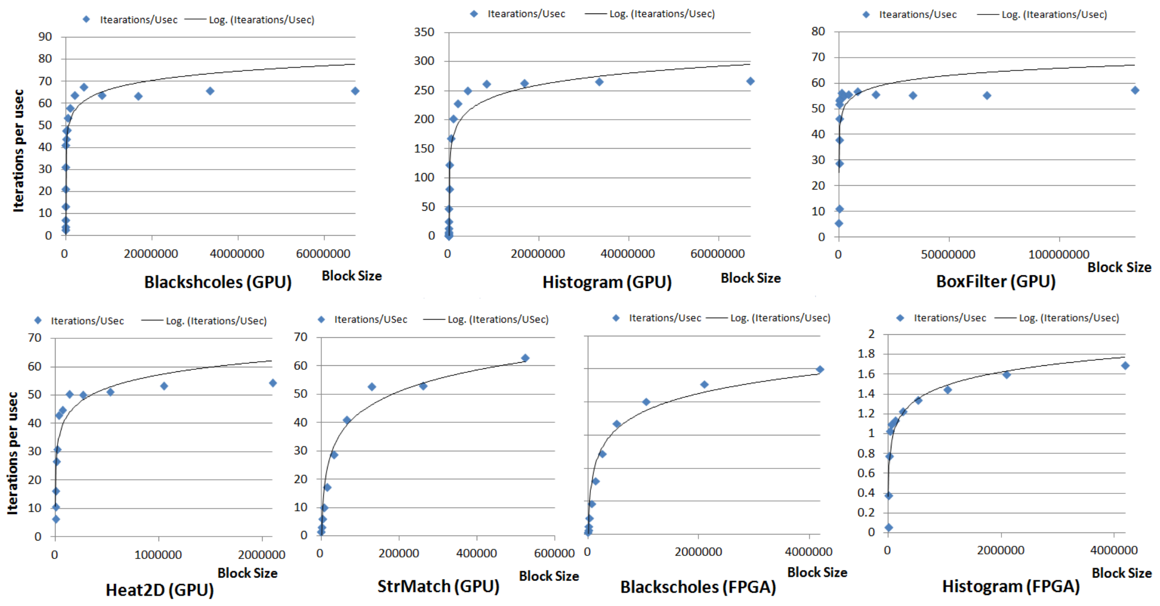


Figure 3.7: Change of compute rates (iterations/usec) for varying number of blocks. Solid black line shows logarithmic curve fitting for each evaulated benchmark.

In order to reach convergence in weights quickly, we estimate the block size using *least squares estimation* for logarithmic curves. Through our experiments, we have determined that logarithmic curve (Figure 4.11) is an appropriate fit for representing the change in computation weight for the applications we have used. After we have enough measurement points (which is set as 4), we estimate parameters a,b for the logarithmic equation $a\ ln(x)+b$. Once the parameters are obtained, we estimate a stable computational weight for that processor by finding the point where the slope of the fitted curve is equal to a / MIN_CHANGE_RATE.

Least squares estimation allows HDSS to dispatch fewer number of small blocks in the adaptive phase. Since small blocks under utilize accelerators, HDSS quickly skips them and proceeds to larger blocks which have higher utilization. After the estimated block size is dispatched to the processor; in the subsequent block request, HDSS checks whether the estimation is accurate by calculating the computational weight again. If the weight change is still not stable, the process of estimation continues.

Once a computational weight is found to be stable, HDSS keeps increasing block sizes for other unstable processors until either all compute weights becomes stable or a fixed percentage (20%) of all data is processed. The latter case is a forced upper bound to limit the length of adaptive phase.

For applications with dynamically changing weights, *MIN_CHANGE_RATE* constant can be set to a value very close to zero and *max_adaptive_percent* can be set to 1.0. This will enable HDSS to monitor the change until the execution ends. However, our experiments show that setting *max_adaptive_percent* to 0.2 is enough to achieve stable compute rates.

Completion phase: Once all weights are stable, the algorithm moves into the completion phase. Block sizes from the adaptive phase are no longer used. Instead, HDSS uses the Modified Guided Self Scheduling (MGSS) to find the correct block sizes. Completion phase integrates computational weights calculated in the previous step into MGSS. Equation 3.1 is the original GSS formula, which simply divides the remaining number of iterations R to the total number of processors P. MGSS uses Equation 3.2 which distributes R to processors proportional to their calculated weight w_i.

$$B = R/P \tag{3.1}$$

$$B = Rw_i / \sum_{j=1}^{P} w_j \tag{3.2}$$

Clearly, the number of iterations allocated to each processor reduces as processors finish execution of a blocks. MGSS is more suitable with heterogeneous computing because it starts with largest block size possible and accelerators are better utilized with larger block sizes (Figure 3.1). GSS was proved to finish executions of processors at the same time in a homogeneous SMP system. With accurately estimated computational weights, proof for the original GSS is valid for the MGSS. This is due to the fact that different block sizes calculated for different processors for the same remaining number of iterations will always take the same amount of time because computational weights cause proportional distribution of blocks. In other words, w_i / $\sum_{j=1}^{P} w_j$ will replace 1/P in the equation if the weights are accurately determined.

3.3.3 Dependency Resolution

Once a proper block size has been determined for the requesting processor, HDSS also determines the starting index (i.e., iteration number) for the iteration block. For loops with independent iterations (i.e., DOALL loops), the start index is set to the first position among the non-assigned iterations. This choice is made regardless of whether the computation for the previous indicies has completed or not.

On the other hand, if an iteration depends on a data value that is calculated in a previous iteration, HDSS automatically exploits wavefront parallelism by considering the dependency requirements between assigned blocks. To achieve parallelism in such loops, the 2D iteration space is divided into fixed width strides [20]. Elements in stride is further divided into blocks each of which assigned to different processors. Blocks that don't depend each other can be processed in parallel(inter-block parallelism). The computation of a block is carried in '*waves*' where elements lined over a diagonal of that block can be processed in parallel(intra-block parallelism). Whenever an idle processor requests a block for execution, HDSS first looks for a stride that contains iterations whose dependency constraints have been satisfied. Next HDSS determines a block size, as was the case for independent iterations, but limiting the remaining iteration space to those iterations that are ready for execution.

Dependencies are determined by dependency vectors per each application. HDSS runtime is responsible for tracking the dependency between blocks assigned to different processors. When there are no independent blocks to process, processors requesting for blocks are put on hold. Once a block is processed, HDSS lets the next idle processor know that there is a 'ready to execute' block available and performs necessary assignments and inter-processor synchronization. Dependency vectors can easily be changed by users by modifying HDSS source related to run-time.

Figure 3.8 shows an example partitioning of a 2D iteration space divided by strides with size 2. Once the top-left most block is processed serially by P1, P2 and P3 executes corresponding blocks in parallel. Once P2 and P3 finish processing, HDSS run-

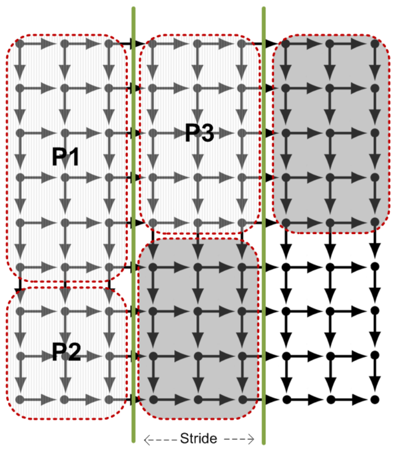


Figure 3.8: Wavefront parallelism employed by HDSS: Arrows represent dependencies (target depends on source). Strides are separated by vertical dashed lines. Rectangular areas with light shade represent the blocks that are executed by the labeled processors. Dark shaded blocks represent the iterations whose dependencies are satisfied and ready to execute.

time will pick proper block sizes based from the 'ready-to-execute' iterations marked with shaded rectangles in Figure 3.8. If the block size determined by HDSS is larger than the number of independently executable iterations (data), then the block size is reduced to available number iterations. As a side note, although HDSS currently supports two dimensional wavefront parallelism, it can easily be extended to support multiple dimensions of dependences.

3.4 Evaluation

3.4.1 System Configuration

In order to evaluate HDSS, we have used two different systems. The first system is a GPU+CPU architecture with 4 16-core AMD Opteron 6200 series CPUs and an nVidia Fermi C2050 GPU. The second system is an SGI Altix 4700 connecting 16 dual core Itanium Montecito CPUs to Xilinx Virtex 4 FPGA via fast interconnect named NUMALink4.

For the CPU+GPU platform, we have used CUDA SDK to run GP-GPU applications on the device. SGI Altix 4700 features RASC library to transfer data to/from FPGA and execute applications on the device.

3.4.2 Applications

For CPU+GPU platform evaluation, we have integrated HDSS into *Blackscholes*, *Histogram* and *BoxFilter* applications from nVidia CUDA Programming SDK. *Blackscholes* is a popular financial analysis algorithm for calculating prices for European style options. *Histogram* counts the the number of occurrences of 256 bit elements in a given input stream. The third application, *BoxFilter*, applies a filter of a predefined radius to a given image. Data sizes for all three benchmarks are chosen to be the maximum possible amount that a single GPU can store at once, which is around 750 megabytes. For a fair evaluation of our dynamic compute rate calculation, we have scaled the input with different data instead of simply replicating a sub-set of the input.

In addition to these three GPU applications, we have also implemented a CUDA version of 2D heat equation (*Heat2D*) and string matching (*StrMatch*) to provide examples for scheduling loops with dependent iterations. *Heat2D* simulates propagation of heat on a number of time step. In each time step, heat propagation of each element depends the left&up elements in current step and right&bottom elements in the previous step. Therefore dependency on top-left elements should be resolved before proceeding. *StrMatch* is based on Smith-Waterman algorithm and it matches two given vectors using an internal 2D array of similarity values. Calculation of each value depends on top&left values.

For CPU+FPGA platform, we have developed FPGA versions of *Blackscholes* and *Histogram* applications. The amount of data to be sent to FPGA at once is limited around 250 megabytes by the RASC library.

Since both of our CPU+GPU and CPU+FPGA platforms are shared memory systems, we have modified each application to execute CPU and device (GPU or FPGA) specific implementations concurrently on different parts of the same shared input data.

3.4.3 Algorithms Compared

We have compared HDSS with the most recent major studies on heterogeneous load balancing: Axel[84], Qilin[51] and Monte-Carlo[83]. Axel and Qilin uses static load balancing whereas Monte-Carlo method uses dynamic scheduling to balance the workload.

For Axel, we have developed communication and computation models for GPU and CPU. These models are determined based on the architectural specifications of processors. The inverse ratios of these values give relative computational weights for each processor in the system that we have used to distribute the workload.

Qilin requires an offline training period to develop a linear model of performance for increasing block sizes. We have performed two runs, the initial being 30% of the original, for each application. As suggested by the technique, during the first (training) run, we fitted a linear curve for the measured execution times for varying sizes of input. The slope of this curve gave the compute ratio for each processor. Later in the second (actual) run, the workload is partitioned statically based on these computed ratios. For a fair evaluation, we have excluded training run while calculating execution time for Qilin and comparing with our results.

The last study by Tse et.al [83] based on the Monte-Carlo method offers the use of two basic dynamic scheduling schemes: *exponential* and *linear incremental* self scheduling. *Exponential incremental* policy multiplies the block sizes by a factor of a constant m. Every time when a processor requests a new block, it gets m times more workload than previous request. Similarly, *linear incremental* scheduling increases the block sizes with a constant n. In both cases, the increase in block sizes are more rapid for faster processors, since they request more blocks than slow processor during the same amount of time. Although these two schemes are not unique to the study mentioned, authors claim that they offer good dynamic balancing for heterogeneous architectures. Hence, we also implemented them and obtained their results for comparison.

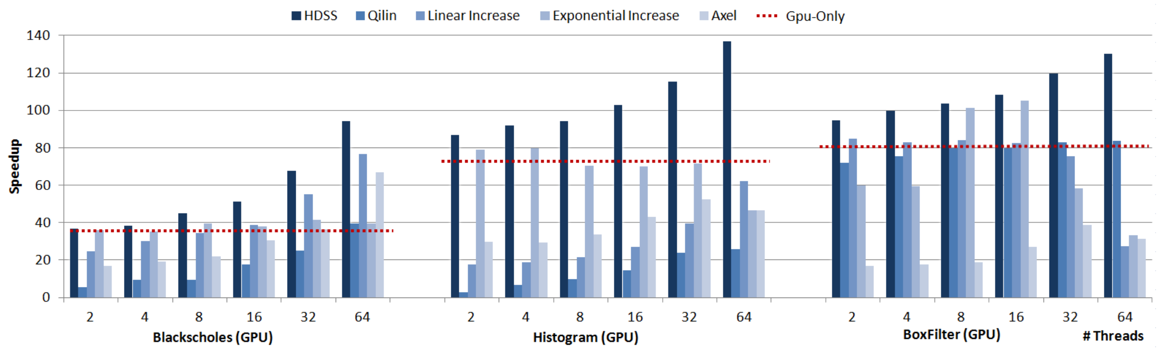
3.4.4 Results

Speedup on GPU: Table 3.1 gives reference execution times (in seconds) of single threaded runs of applications on a single CPU core only and also on the device only (FPGA or GPU). During these runs, we have not applied any load balancing techniques. All iterations are sent to device at once without any executions by multiple CPUs.

	1-CPU Only	Device Only
Blackscholes (GPU)	169.8	4.7
Histogram (GPU)	133.4	1.82
BoxFilter (GPU)	346.1	4.25
Heat2D (GPU)	592.7	124.6
StrMatch (GPU)	447.5	91.4
Blackscholes (FPGA)	246.8	15.0
Histogram (FPGA)	155.2	25.1

Table 3.1: Base Execution Times: Device and 1-CPU only execution times (in seconds) for each application.

Figure 3.9 shows speedups for CPU+GPU system with regards to single threaded CPU execution. Results are obtained by running each algorithm for varying number of representative threads: 1 thread for GPU plus 1,3,7,15,31, and 63 CPU representative threads for 2,4,8,16,32, and 64 threaded executions respectively. All CPU and device threads of each run is configured to share the same input but process different parts of it. Speedup for GPU only execution is also included for reference. Please note that, although we are using a single CPU-side representative thread to handle GPU execution operations, GPU still runs the application with all available hardware threads inside. In other words, by using more CPU threads in addition to the GPU representative thread, we are improving GPU only results by including additional CPUs in the computation as well.



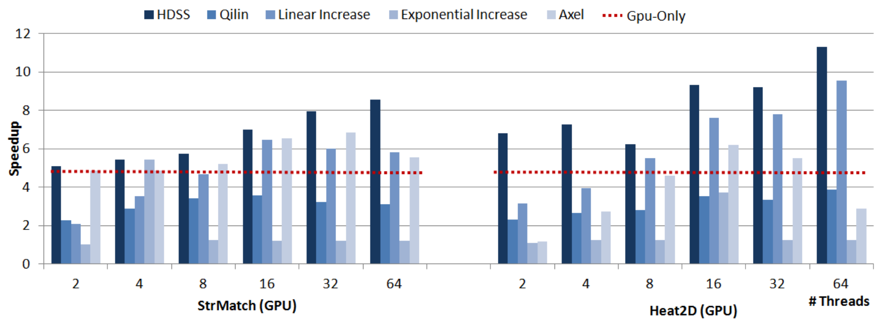


Figure 3.9: Speedups obtained in CPU+GPU system when compared to single threaded CPU only execution. Varying number of threads 2, 4, 8, 16, 31 and 64 refers to 1 GPU + 1 CPU, 1 GPU + 3 CPU, 1 GPU + 7 CPU, 1 GPU + 15 CPU, 1 GPU + 31 CPU and 1 GPU + 63 CPU representative threads respectively. GPU only execution is also shown separately for each application.

It may be observed that, HDSS is the only algorithm that provides more speedup than GPU only execution for all applications and thread counts. This is an essential goal of heterogeneous computing which tries to utilize all available processors in the system. Other algorithms fail to do better than even GPU only execution for most cases because of high GPU idle periods. HDSS results in performance improvements of up to 219% (for 64 threaded execution of Histgoram application).

Static scheduling schemes, Axel and Qilin, usually performs poor due to under (or over) estimation of weights. Qilin's linear adaptation results in inaccurate weights which fails to represent actual computation power of GPUs. Qilin starts from an initial block size and increase it gradually to build a linear time function based on varying block sizes. However, if the initial block size and subsequent measurements are small enough to under-utilize the accelerator (as described in Section 2.1), the linear adaptation will end up in a mis-balanced workload ratio. On the other hand, HDSS follows a logarithmic curve fitting approach, which characterizes the relationship between block size and execution time much better than a linear curve.

The other static scheduling scheme, Axel, achieves speedups close to GPU only execution, since theoretical calculations formulates GPU faster due to its massive number of cores (240 in our case). Modeling CPUs, on the other hand, are not straight-forward due to non-quantifiable capabilities (such as, instruction level parallelism, hardware threads, speculation etc.) of these architectures. As a result, CPUs are left idle most of the time because of the inaccuracy of static performance estimation. Moreover, this method requires manual adaptation of communication and computation models for each benchmark, which makes the technique very difficult to apply to a range of applications.

Linear and exponential incremental scheduling are the two dynamic schemes that we have compared with HDSS. Exponential-incremental is not providing more speedup than GPU only execution in most cases because growing blocks become too large towards end of the execution. The last large block assigned to CPUs takes more time to process than the one assigned to GPU, hence causing the accelerator to sit idle until CPU finished execution.

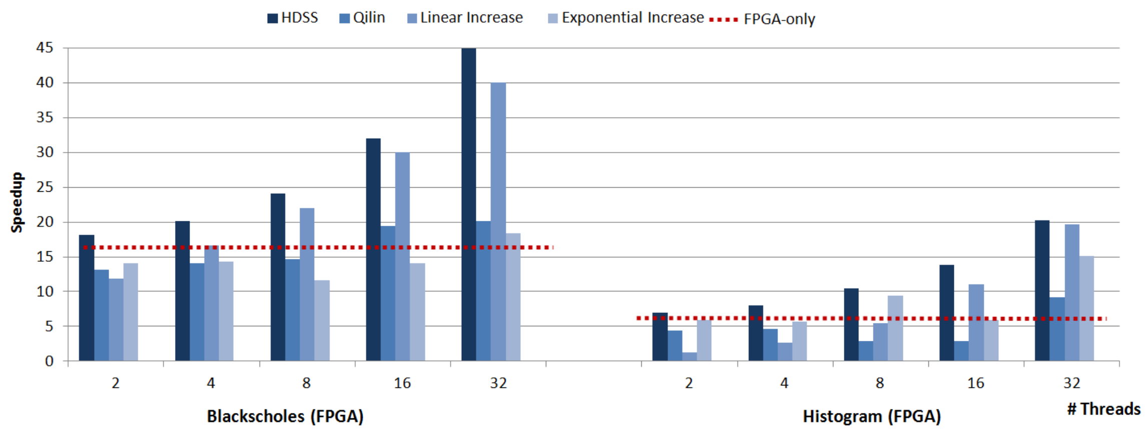


Figure 3.10: Speedups obtained in SGI Altix System when compared to single threaded CPU only execution. Varying number of threads 2, 4, 8, 16 and 32 refers to 1 FPGA + 1 CPU, 1 FPGA + 3 CPU, 1 FPGA + 7 CPU, 1 FPGA + 15 CPU and 1 FPGA + 31 CPU threads respectively. FPGA only execution is also shown separately for each application.

On the other hand, linear incremental scheduling seemed to perform surprisingly better than others. Since block sizes are linearly increased, they do not grow up very quickly. This behavior of linear incremental scheduling decreases the chance of accelerators being idle at the end, due to quick processing of small blocks by the CPU cores. However, since linear incremental scheduling slowly increases block sizes, it assigns too many small blocks during the initial phases of the execution. As described earlier, GPUs fail to unleash their full power for small block sizes, even though they are always kept busy during all the time. In our experiments, Heat2D and StrMatch are the algorithms employing dependent loop iterations and inter-block synchronization. As the results in Figure 3.9 shows, although HDSS shows better performance than others, it suffers to exploit major speed up for this application. This is due to the long idling periods of processors where they wait for other parts of the data to be computed. HDSS performance on dependent loop iterations can be improved by employing priority distribution queues where faster processors are preferred for larger chunks of 'ready-to-execute' iterations, however, this is left as a future work.

Speedup on FPGA: Figure 3.10 shows speedups for SGI Altix CPU+FPGA system when compared to single threaded CPU execution. Results are very similar to the ones obtained on the GPU system. The logarithmic behavior of block sizes shown in Figure 3.1 versus execution time is also observable for this platform as well. FPGA is also utilized better when the amount of task executed at once gets bigger. This is mainly due to overlapping of execution time with the communication time in the SGI Altix architecture.

Initial block assignments: For a better exploration of how HDSS is able to use less number of small sized blocks in the beginning, when compared to linear and exponential incremental schemes, Figure 3.11 is given to illustrate initial block assignments for the three dynamic scheduling algorithm. The experiments are derived from two-threaded execution of Histogram on CPU+GPU platforms given in Figure 3.9. The initial block sizes are set as 1024 and only the assignments during the first two seconds are shown. Stable compute ratios are achieved when the block size is around 2.7M, so blocks having sizes smaller than this number considered as 'small'.

Exponential-increase doubles the block sizes consecutively until end of the execution. Similarly linear-increase adds 1024 to the previous block size on every assignment. Both of these two algorithms uses a considerable amount of small blocks until they reach 16.7M. On the other hand, HDSS algorithm runs the least squares estimator after first four block assignments. Based on the resulting logarithmic curve, the block sizes are increased from 8192 to 8M and all potential small blocks that exponential and linear schemes have assigned are skipped. HDSS achieves the stable block size (16.7M) on the fifth block just after one second, whereas exponential increase reaches stability in 3 seconds (not shown) and linear increase never reaches that large block size. In other words, all of the blocks assigned by linear-increase method are considered as 'small blocks which cause under-utilization of the accelerators. On the other hand, HDSS is able to quickly converge to the stable large size without spending time on small sized blocks.

Load balancing: For a better understanding of the load balancing efficiency of the algorithms, we have also plotted the time difference between the earliest and latest finishing threads for each run in the CPU+GPU system (Figure 3.12). The results for HDSS and linear-incremental are indistinguishable because they line up in the x-axis along the zero difference line. HDSS and linear have the smallest amount of difference among all the algorithms implying perfect load balancing of processors. Other algorithms show considerable differences mostly because of the idle GPU at the end of execution. The load is not properly balanced to address the heavy computational weight of the GPU compared to

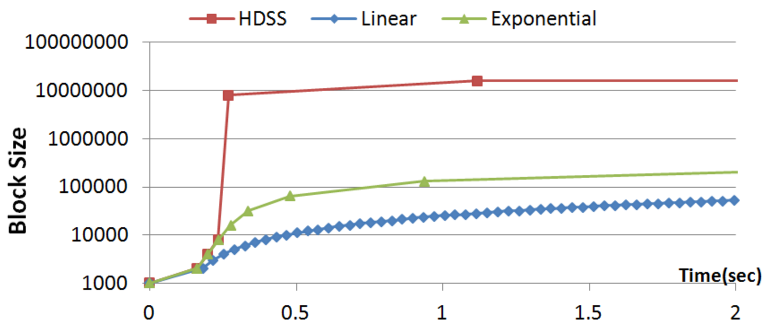


Figure 3.11: Illustration of initial block sizes for HDSS, Linear and Exponential increase algorithms for the 2 threaded execution of Histogram on CPU+GPU platform.

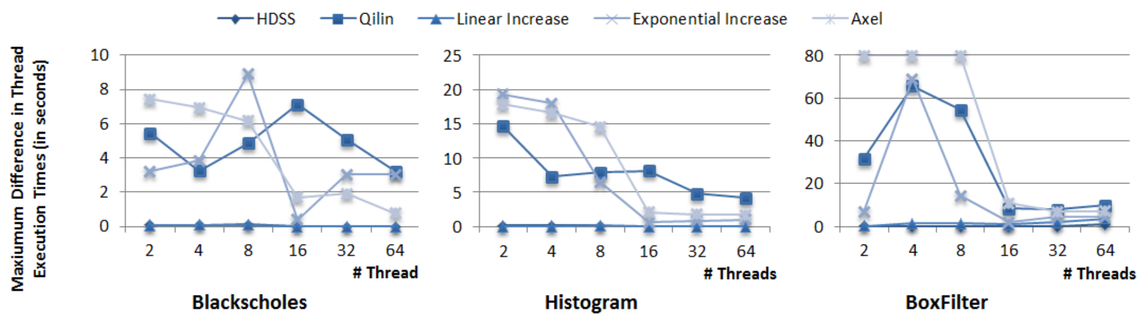


Figure 3.12: Time differences between the earliest and latest finishing threads for each run per application in CPU+GPU system.

a CPU. Even though linear incremental policy shows very minimal difference, accelerators are under-utilized due to excessive usage of small sized blocks, as previously shown in Figures 3.9 and 3.10.

Please also note that, Heat2D and StrMatch are excluded from this experiment since idle periods in the life time of a thread is distributed over the entire execution due to unsatisfied dependencies, therefore it is not possible to quantify load balancing for these two applications using this kind of experiment.

Workload partitioning: In order to further demonstrate how HDSS achieves close to ideal workload balancing, we have compared the percentage of the blocks assigned to the accelerator (GPU and FPGA) with the percentage that ideal partitioning would assign to that device. In this example, the ideal partitioning is determined in the following way. First the applications are executed with the same inputs separately on single core CPU and the accelerator. After obtaining the execution times for both cases, compute ratios (iterations per unit time) are calculated for the accelerator and CPU. The CPU

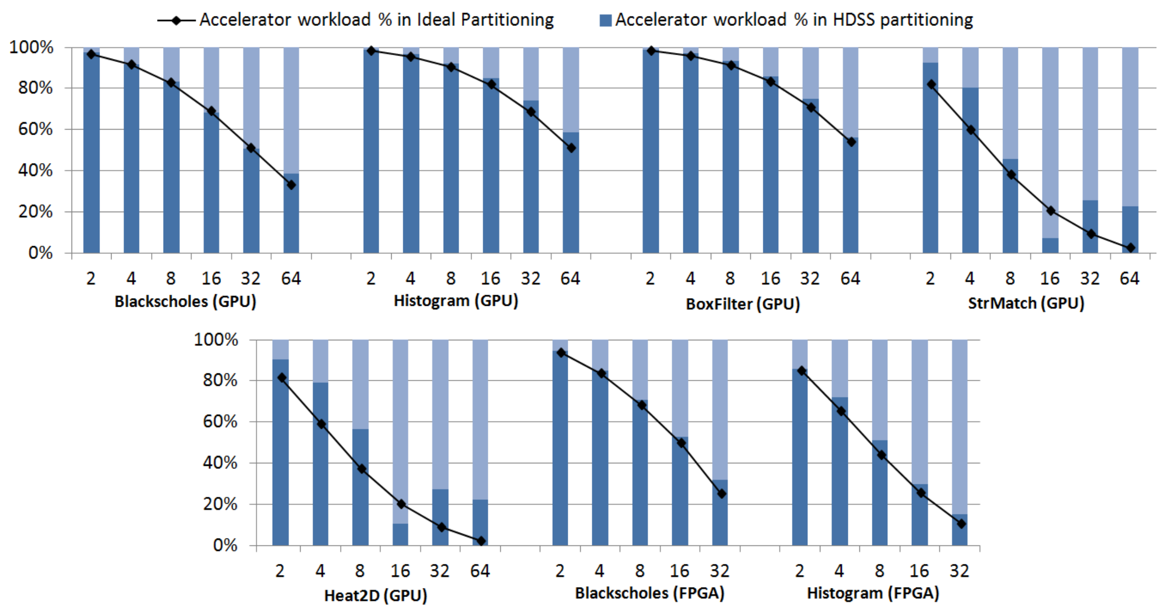


Figure 3.13: Comparison of accelerator workloads in ideal and HDSS partitioning. Ideal values are scaled from single core CPU and accelerator alone executions.

processing rate is scaled by the number of CPU cores (1 to 63) used in the system. Based on the ideal&scaled compute ratios, the ideal percentage of accelerator workload is calculated and indicated by connected points in the chart. Actual workload balancing between the accelerator and CPU produced by HDSS is drawn as bars in Figure 3.13, where the dark bar represents the accelerator percentage. The results clearly shows that HDSS is performing very close to the ideal partitioning, without requiring an offline run or configuration. A notable observation is that, since scalability of two dependent-loop applications, Heat2D and StrMatch, are lower compared to others, HDSS distributed more workload to GPU than the ideal partitioning curve. This is an expected results for dependent-loop applications.

Chapter 4

Data Transfer and Computation Overlapping

4.1 Introduction

General purpose GPU (GP-GPU) computing has had a remarkable impact on the evolution of scientific applications over the past decade. Various SW and HW improvements like unified memory access in CUDA 6 and dynamic parallelism in Kepler architecture [60] have enabled developers to utilize GPUs for a wide range of application classes with moderate programming effort. The introduction of concurrent kernel execution [89] and simultaneous work queues (i.e., hyper-Q) has enabled GPUs to be shared across multiple applications; hence acting more like general purpose processors rather than dedicated accelerators. Therefore, researchers have started focusing more on efficient sharing of GPUs across different applications.

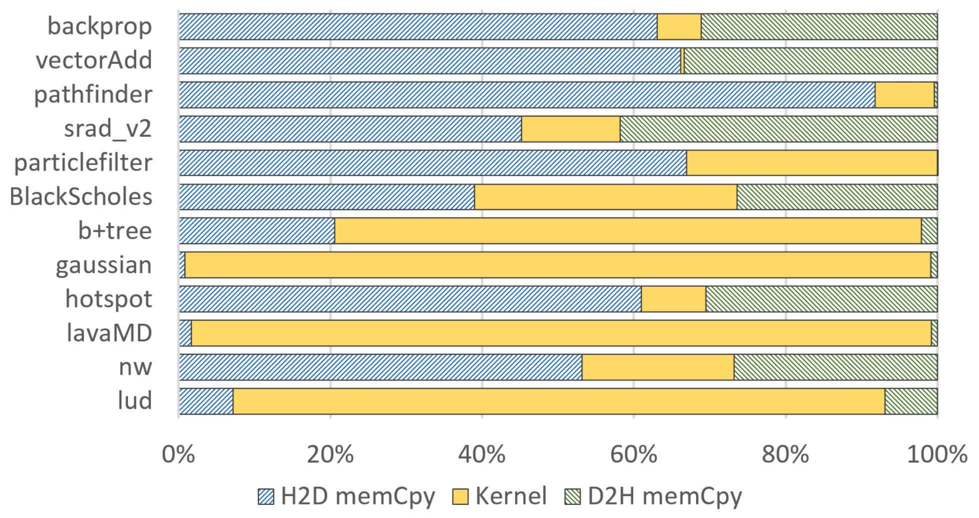


Figure 4.1: Execution time breakdown of 12 NVIDIA SDK and Rodinia applications.

While the attempts to improve concurrent kernel performance in various scenarios has been beneficial, they do not address execution phases other than kernel call(s); in particular, CPU↔GPU data transfer(s) phases also account for considerable portion of the execution time. Figure 4.1 shows the execution time breakdown of applications among three phases: device-to-host (D2H) memory transfers, host-to-device(H2D) memory transfers, and kernel execution (KE). This data corresponds to stand alone executions of 12 Rodinia benchmark suite [15] applications on NVIDIA K40C GPU. The breakdown shows that data transfers occupy a considerable portion of a CUDA application's lifetime.

Overlapping data copy operations with kernel execution has been one of the most popular software solutions to hide the transfer latency. There have been some application-specific studies [73, 40]; however, only a few researchers were able to generalize their techniques beyond a limited class of applications [52, 42]. These approaches either require significant programming effort involving adaptation of libraries and changes to the programs [8] or are capable of handling a very limited set of applications with accompanying HW modifications (e.g., [52] allows only regular and linear data accesses).

The open problem addressed in this work is '*how to schedule multiple applications sharing the same GPU to improve overall system throughput*' via data-transfer/kernel-execution overlapping while supporting a wide range of applications, without requiring any code modifications. A notable study in recognizing the importance of data-transfer in GPU sharing is carried by Sengupta et al. [72]. The authors propose a runtime framework, *Strings*, to auto-balance the load between multiple servers with GPUs and exploit automated transfer/execution overlapping for each server/GPU. Whenever applications request different type of GPU operations (i.e., D2H, KE, or H2D) around the same time, *Strings* scheduler tries to execute these calls concurrently on the corresponding resources, if they are idle. *Strings* was shown to improve the overall system throughput when compared to the case where no overlapping exists. However, the success of exploiting overlapping between three types of calls depends on:

- Arrival order of same type of operations belonging to different applications
- Idleness of the corresponding resource (PCI-e uplink & download and the GPU itself)

Currently, to the best of our knowledge, neither *Strings* [72] nor the latest CUDA runtime API take the above two factors into account while scheduling different type of GPU operations on a single shared device. To better understand the impact of these limitations, let us consider an experiment involving the shared execution of the subset of Rodinia benchmarks introduced in Figure 4.1. Figure 4.2(a) shows the execution where concurrent D2H, KE, and H2D calls are overlapped based upon a random call issue order of

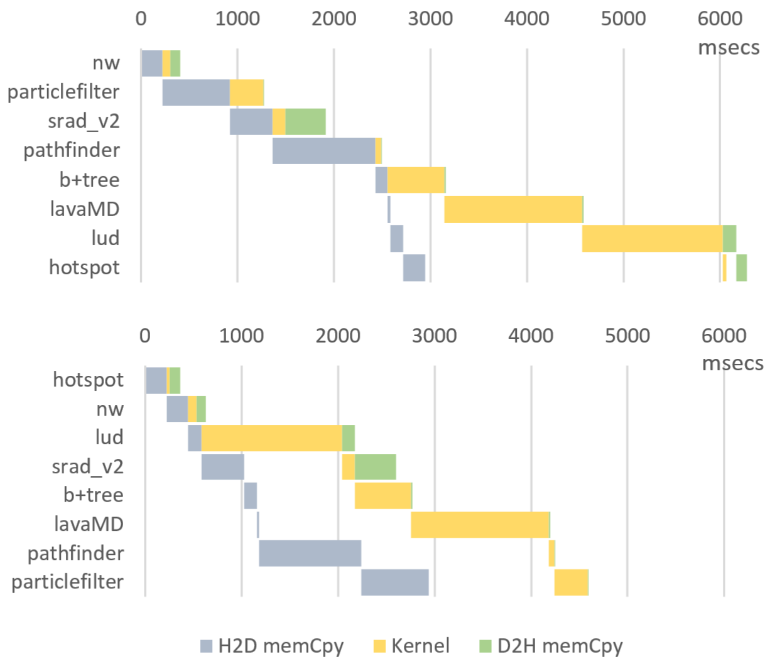


Figure 4.2: Effect of ordering on overlapping benefits: (a) On the top, an overlapping example exploited on a subset of Rodinia applications using a random arrival ordering; and (b) On the bottom, a data transfer-aware re-ordering of the same subset.

CUDA applications while Figure 4.2(b) shows how total execution time can be minimized by simply re-arranging the call execution order of the CUDA applications. Thus, we can see that by changing the execution order of different applications, much higher resource utilization and thus performance can be achieved. If the call scheduler is made aware of the utilization of each underlying resource as well as the execution time of the issued device calls, the re-ordering can be performed to increase overlapping for higher resource utilization.

In this study, we present CuMAS, a data-transfer aware CUDA call scheduling framework to efficiently overlap transfer/execution phases of multiple applications on a shared GPU. CuMAS captures CUDA API calls and re-schedules them to minimize the total execution time.

CuMAS run-time uses profiling and timing models to estimate durations for data transfers and kernel executions. Our run-time dynamically attaches to existing programs and it does not require any changes to the programs' source code. CuMAS operates on a queue of CUDA requests issued by multiple applications each having arbitrary number of data transfer (H2D and D2H) and kernel launch (KE) calls. We show that CuMAS framework improves the throughput of multiple kernel executions by up to 44% when run on NVIDIA K40c GPU using applications from CUDA SDK and Rodinia benchmark suite.

The remainder of this chapter is organized as follows. We first give a background on CUDA and shared application execution on GPUs in Section 2. Then we introduce the main idea (request re-ordering) of our proposed framework in Section 3. We focus on CuMAS scheduling algorithm in Section 4 and give details about our runtime in Section 5. We present evaluation of CuMAS in Section 6.

4.2 GPU Sharing in CUDA

In this section we first provide an overview of mechanisms used for sharing of NVIDIA GPUs across multiple CUDA applications including CUDA streams, and Hyper-Q. We then elaborate on the considerations to solve transfer/execution overlapping problem.

CUDA Streams and Hyper-Q: CUDA API provides streams to enable concurrent execution of multiple GPU applications. *Stream*s are analogous to threads in conventional CPU programming and they may represent different applications or different phases of the same CUDA application. A *command* is created for each CUDA call and queued into the *stream* specified in the call (the default 0 stream is used if none is specified). Commands queued in the same stream are serially executed whereas commands in different streams may be executed in parallel if corresponding resources (e.g., PCI-e bus, SMs on the GPU) are available. CUDA Kernel calls are asynchronous and memory transfers (i.e., *cudaMemcpy()* and *cudaMemcpyAsync()*) can be either synchronous or asynchronous. Synchronization between asynchronous CUDA calls is achieved via *events*.

Overlapping Kernel Execution and Data Transfers: Tesla and Quadro family of NVIDIA GPUs employ dual DMA copy engines [61] to enable simultaneous H2D and D2H data transfers. An ideal multi-kernel execution maximizes the use of both channels while keeping the GPU always busy. In a shared environment, with pending *transfer* and *execution* jobs, the scheduling order of jobs impacts system utilization. Currently, CUDA API schedules transfers in the order the calls are received from the applications (FIFO). In a shared environment, the applications may issue transfers at any time, hence appearing in a random order in the CUDA call queue. However, this may lead to performance degradation since the runtime is unaware of transfer times and execution times. If these times can be estimated prior to execution, a smarter scheduling decision can be made leading to improvements in utilization well beyond the default FIFO scheme.

Efficient sharing of GPUs across multiple applications requires each of the following issues to be handled carefully:

- *Maximization of overall resource utilization:* Individual and cumulative idleness of the resources –including the 2-way PCI-e links and the GPU itself– should be minimized.

- *User transparent execution:* The shared execution should seamlessly take place without requiring any code modifications or interventions from the users of the environment.
- *Support for various application classes:* Data access, data transfer, and execution patterns of an application should not limit the efficiency of sharing.

To address the issues listed above, we propose CuMAS (CUDA Multi-Application Scheduling), a host-side CUDA call scheduling framework, to enable efficient sharing of GPUs by multiple applications. CuMAS employs a scheduler and an accompanying runtime to enable on-the-fly integration to existing applications with no source modifications.

4.3 CuMAS: Tasks and Call Ordering

CuMAS is based on the idea of *re-ordering CUDA calls* to improve the overall system throughput. In this section we will explain this novel idea in detail.

In a typical shared execution environment (e.g., cloud computing), a high level load balancer will distribute incoming user requests across servers, as depicted in Figure 4.3. Each server queues incoming requests (e.g., CUDA calls) and executes them on their own GPU(s). In this paper we assume that the user applications are already assigned to a GPU and we focus on how requests from multiple applications are retrieved from the server queue and executed on the GPU.

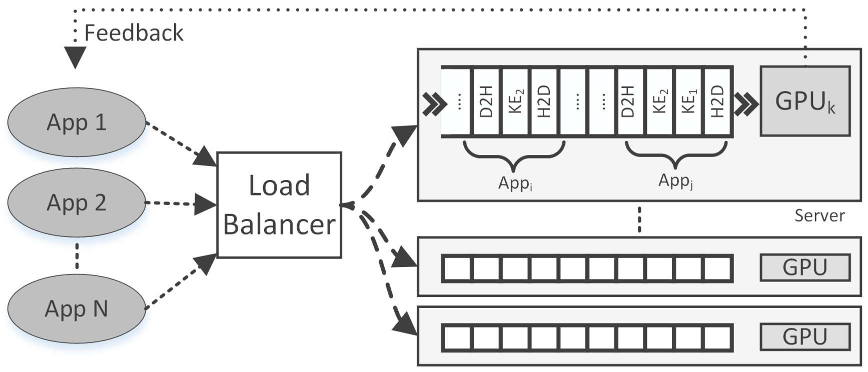


Figure 4.3: Multi server GPU Sharing and per-server CUDA call queues in a typical cloud computing environment.

4.3.1 CuMAS Tasks

We logically group the CUDA calls in the server queue into *CuMAS tasks*, which will be inputs to our scheduling algorithm in the later phase. *CuMAS tasks* are comprised of a series of H2D, KL and D2H calls all of which should belong to the same application. A typical CUDA application typically consists of multiple data transfer and kernel launch phases and thus we end up with multiple tasks being created for each application. A CuMAS task spans the CUDA calls which can be continuously executed without requiring any control on the execution flow.

From the application's perspective, CuMAS forces all CUDA API calls (except the last one of each CuMAS task) to act like *asynchronous call*s (i.e. they are immediately returned to the calling function). Whenever the last CUDA call is reached by the user program, all calls captured and grouped under the same task are issued on the real hardware. Since tasks are created dynamically as the application issues requests, at most one task per

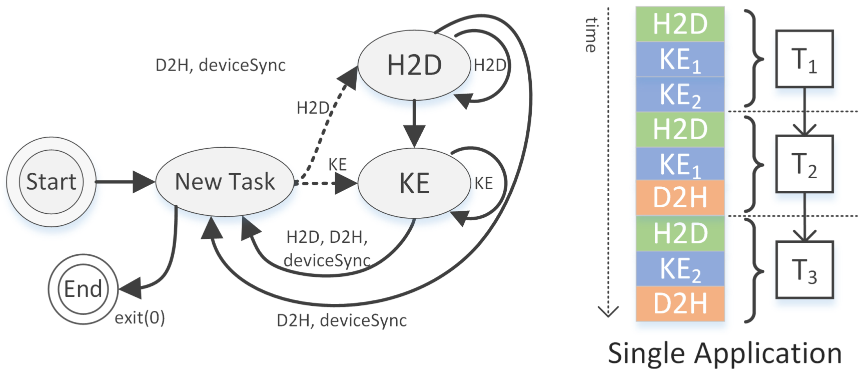


Figure 4.4: (a) On the left: The state diagram to decide whether create a new task based on the operation flow (b) On the right: An example stream showing the grouping of CUDA calls into tasks according to the state diagram given on the left.

application can exist at any given time. Once all CUDA calls belonging to a task are processed, the caller application is unblocked and allowed to resume issuing CUDA calls.

To build tasks from a series of CUDA calls , we use the state diagram given in Figure 4.4(a). We group the calls into CuMAS tasks such that KE and D2H call in a task come after all H2D calls and similarly, D2H calls come after all KE calls. In our state diagram, any sequence of CUDA calls that breaks the ordering of H2D $\rightarrow$ KE $\rightarrow$ D2H causes current task creation to be finalized and a new task to be formed. Also, any occurrence of D2H at any time finalizes the creation of current task, due to possible control flow changes on the host side based upon the data received from the GPU. In addition to data transfer and kernel execution calls, *cudaDeviceSync()* calls also cause a new task to be formed.

Existence of multiple tasks in the same stream implies a dependency between different phases of an application and such tasks must be executed in the order of the operations. Figure 4.4(b) depicts the break down of a sample CUDA application into multiple tasks (T_1 to T_3) based on the order and type of the operations that it contains. The dependencies between tasks are shown with arrows and they imply a serial execution for the CUDA calls from the same application. For most CUDA applications, the dependent tasks are not created until the previous task is finished and new CUDA calls are captured. On the other hand, since there are no dependencies between the tasks of different applications, CuMAS exploits overlapping across the calls belonging to such tasks.

4.3.2 Re-Ordering Window (ROW)

As CUDA calls are captured, tasks corresponding to the calls in each application's stream are created on-the-fly using the state diagram described in the previous sub-section. Similar to the queue of CUDA calls shown in Figure 4.3, tasks from all streams that are yet to be processed form a *task queue*.

We introduce *Re-Ordering Window* (ROW) to retrieve CuMAS tasks from the queue in batches and re-order them using our scheduling algorithm. Re-Ordering Window allows CuMAS to exploit better overlapping by allowing the tasks to be executed in different order than the arrival order while limiting the maximum distance that a task can be moved ahead or back in the queue.

Figure 4.5 shows a scenario where five artificial CUDA applications (A to E) are scheduled by CuMAS using ROW. Figure 4.5(a) depicts the initial state of the queue at time t_0, where applications have already issued CUDA calls around same time and corresponding CuMAS tasks (e.g. A_1 corresponds to the first task of application A) are created and placed in the queue in the original arrival order. In this example the size of ROW is set to three, which forces the last two ready to execute tasks, D_1 and E_1, to wait in the queue.

The tasks in a ROW are reordered using our scheduler (which is explained in the next section) and the CUDA calls in these tasks are issued in the order given by the scheduler. Figure 4.5(d) shows the HW pipeline for the execution of H2D, KE and D2H calls of the ordered tasks on the corresponding resources. *The execution order of the tasks are different than the initial placement of tasks in the queue due to the re-ordering applied by the CuMAS scheduler.* However, re-ordering remains only within the ROW, and the execution order of tasks across different ROWs is maintained. As CUDA calls grouped by the tasks in ROW are executed, corresponding applications are sent responses to continue their execution, which in turn generates new tasks, if any.

As soon as all the H2D calls of the tasks in a ROW are issued on the PCI-e device, ROW is shifted to cover the next available set of tasks, without waiting for the KE and D2H calls of the tasks in the prior window to finish. Figure 4.5(b) shows the time step t_1, where H2D calls of A_1, B_1 and C_1 are completed and up-link PCI-e bus has just become idle. At this moment, CuMAS shifts the ROW to next set of tasks in the queue. Since all CUDA calls in task A_1 are finished before t_1 and the application A has given chance to issue new calls, the ROW now covers a new task A_2 for these new calls in addition to the already existing tasks D_1 and E_1. On the other hand, the second task B_2 of application B is yet to be created due to unfinished calls remaining in B_1 at time t_1. Similarly, at time t_2 shown in Figure 4.5(c), only two tasks, B_2 and D_2 are included in ROW and A_3 is yet to be created after calls in A_2 finish execution.

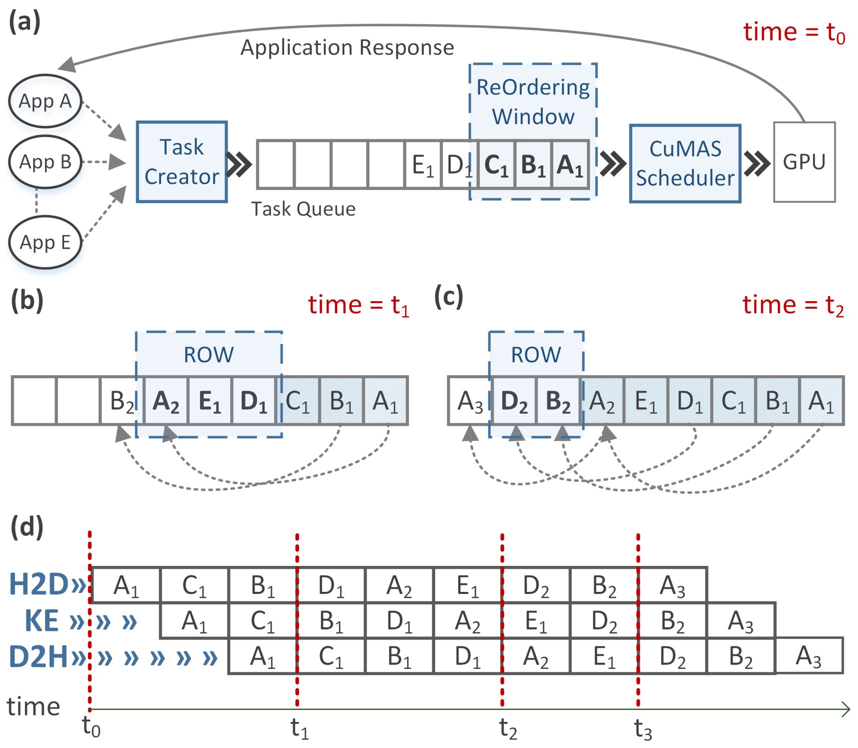


Figure 4.5: (a) CuMAS task creation, queuing and re-ordering for different applications at initial time t_0, (b) t_1, and (c) t_2. (d) On the bottom: View of the HW pipeline of tasks after re-ordering.

4.3.3 Scheduler Invocation

CuMAS is designed for shared GPU environment; therefore we assume there is a continuous flow of CUDA API call requests from existing or new applications sharing the system. The scheduler is invoked in two situations, whichever arises earlier:

- PCI-e uplink occupation of the H2D calls in a previously scheduled window is about to finish; and
- Total number of tasks ready to execute reaches the ROW size.

The first case is necessary to keep the resources busy as long as there are more tasks to process. In the second case, the window size w is to control the trade-off between the room to exploit better overlapping and per-task re-ordering distance. If w is kept large, the scheduler will have more opportunities for re-ordering, hence allowing a higher transfer-execution overlap. On the other hand, if w is smaller, then the wait time between initial call issue and actual execution will vary less due to smaller distance of task re-ordering. Also, larger w values will increase runtime overhead due to the complexity of the scheduling algorithm. We evaluate the effects of ROW size w in detail in Section 4.6.

4.4 CuMAS: Scheduler

Scheduler is the core component of CuMAS and it takes a set of ready-to-execute CUDA applications as its input. We assume that the underlying architecture has three concurrent resource channels: up-link PCI-e bus, GPU computation, and down-link PCI-e bus. The goal of the scheduler is to find an ordering that minimizes total execution time by maximizing the overlap between kernel executions and two-way data transfers of successive applications.

CuMAS scheduler takes the tasks in the re-ordering window (ROW) as input and produces a sequence between the tasks as output. The sequencing decision is based on the estimated duration of the tasks in current window and the tasks in the previous window which are still executing. The search space of the optimal ordering is limited to the possible permutation of the tasks in current ROW and such problems are proven to be NP-Hard [34].

Next, we discuss how CuMAS scheduler models and solves the problem of finding an optimal sequence for the tasks in a given ROW.

Our scheduler follows the '*flow shop*'[13] sequencing model, which is well studied in applied mathematics literature. In this model, there are n jobs and m machines with different processing times p_{ij}. Jobs consist of a sequence of operations o_k with precedence constraints and each operation must be processed on a single machine. The goal is minimize the maximum job completion time (C_{max}). The sequencing triple to identify the problem in the literature is represented with $F|prec|C_{max}$.

We establish the correspondence between our proposed scheduler and the *flow shop* problem as follows. Each CUDA application correspond to a series of *task*s (i.e., *job*s) where each CUDA call represents an *operation* of a task. We map machines m to resources S, P and R, which represents host-to-device (H2D) PCI-e bus (i.e., send), GPU itself (i.e., process), and device-to-host (D2H) PCI-e bus (i.e., receive), respectively. Each *operation* in a *task* is one-to-one mapped to the machines (i.e., S, P and R) and these operations have a precedence constraint $S \rightarrow P \rightarrow R$.

We define an *operation* as an atomic GPU call, a *stream* as an ordered series of *operations*, and *task* as a consecutive subset of operations belonging to the same stream. In CUDA terminology, *operation*s correspond to CUDA API calls such as *memcpy()* and *kernel* launches. A *stream* represents the entire CUDA application [1] and it is analogous to *CUDA streams* which guarantees the serial execution of the CUDA API calls issued by the owner application. A *task* represents a consecutive subset of the operations from a

[1]In this work, we assume that the CUDA application does not use multiple streams, which is the default case for the majority of the applications in the two popular benchmark suites, Rodinia and CUDA SDK, that we use in our evaluation.

stream. A *stream* is composed of one or more tasks. Tasks are the basic unit of scheduling in CuMAS, and only one task from a given stream can be scheduled and run at a given time. Each *operation* has an associated estimated duration of completion and each *task* maintains the cumulative sums of execution time for each operation type (i.e., S, P, and R) that it contains.

4.4.1 Scheduling Considerations

In a real-life scenario, the scheduling algorithm will be repeatedly run on a queue as new tasks arrive. We refer to each of these iterations as a '*a scheduling round*' and the goal of the scheduler is to minimize the total time for an execution round. Our scheduler treats the three resource channels (S, P and R) as pipeline stages for a task, but tasks may run in parallel as long as there is only one task executing on a channel at a given time.

The scheduler finds an ordering between ready to execute tasks so that overall system utilization is maximized. To achieve this, the scheduling decision must be based on: the durations of S, P, and R operations in each task; the delays occurring in S, P, and R channels due to resources being busy with operations from previous tasks; and the total memory usage requirement of the set of tasks that will be occupying at least one of the resource channels at any given time.

4.4.2 Formulating the Total Execution Time

The total execution time T_{total} for a given order of N tasks can be represented as a function of the time to send the data for all N tasks; the time to wait for each resource

S, P and R to become available (i.e., delays); and the time required to process and receive data for the last task.

Let a task i be defined as a triple of the execution times of send, process and receive operations: $\tau_i = [s_i, p_i, r_i]$. Send channel (S) can be assumed to have no delays, since S is always the first operation in each task and is not dependent on completion of prior operations in the same task. However, P and R channels may include some delays due to stalls in previous operations belonging to the same or previous tasks. The delays that need to be inserted right before executing p and r operations of i^{th} task are represented by *positive definite* functions $\delta^p_{\tau_i}$ and $\delta^r_{\tau_i}$, respectively, and they can be expressed with equations given in 4.1 and 4.2.

$$\delta^p_{\tau_i} = \delta^p_{\tau_{i-1}} + p_{i-1} - s_i \tag{4.1}$$

$$\delta^r_{\tau_i} = \delta^r_{\tau_{i-1}} + r_{i-1} - p_i - \delta^p_{\tau_{i-1}} \tag{4.2}$$

The processing of a task i cannot start until either of the following finishes: sending operation s_i of current task or processing operation p_{i-1} of previous task. However, if there are any delays $\delta^p_{\tau_{i-1}}$ before p_{i-1}, then P resource will not be available until the larger of the following two finishes: s_i or $p_{i-1} + \delta^p_{\tau_{i-1}}$. If s_i takes longer than $p_{i-1} + \delta^p_{\tau_{i-1}}$, then we should not insert any delays for p, therefore $\delta^p_{\tau_i}$ is a positive definite function.

The receive delay $\delta^r_{\tau_i}$ is similar to processing delay, but it also takes into account of the processing delay $\delta^p_{\tau_{i-1}}$ of the previous task due to propagated P resource idleness to the R channel.

Using the delay equations, total time can be expressed as the summation of total send time, $\delta^{P}_{\tau_i}$ and $\delta^{R}_{\tau_i}$ for the last task as well as the the duration for processing P and R operations of the last task, p_N and r_N respectively:

$$T_{total} = \sum_{i=1}^{N} s_i + \delta^{p}_{\tau_N} + p_N + \delta^{r}_{\tau_N} + r_N \tag{4.3}$$

4.4.3 Finding a Solution

Due to recursive conditionals in equations 4.1 and 4.2, a closed form solution could not be obtained for the total time given in Equation 4.3. Moreover, since our scheduling decision is indeed an ordering problem and we are not looking for values of variables, a closed form solution will not help in finding the optimal ordering. Therefore, the complexity of calculating total time of a given ordering is $O(N)$, due to a single iteration over N tasks.

A brute force search on the total execution times across all possible task permutations would give us the optimal execution time, hence the schedule. However, complexity of such a search is $O(NN!)$ which becomes impractical for large values of N.

An approximate solution to the problem can be obtained via *dynamic programming* (DP) in exponential time, which is much faster than the factorial time for larger values of N. The DP approach relies on the incremental representation of total execution time based on a subset of the solution space and can be described as follows.

Let $T = \{\tau_1, \tau_2, ..., \tau_N\}$ be the set of tasks to be scheduled and let U be a subset of T, $U \in T$. We define the 'near-optimal' completion time $C(U)$ of a given subset U as follows.

$$\begin{cases} C(\{\tau_i\}) = c(\tau_i, \{\emptyset\}), & \text{if } |U| = 1 \\ \\ C(U) = {}_{min_{\tau_i \in U}}[C(U - \{\tau_i\}) + c(\tau_i, U - \{\tau_i\})], & \text{if } |U| > 1 \end{cases} \tag{4.4}$$

where the incremental cost function $c(t_i, U)$ is defined as:

$$\begin{cases} c(\tau_i, \{\emptyset\}) = s_i + p_i + r_i, & \text{if} |U| = 0 \\ \\ c(\tau_i, U) = \delta^r_{\tau_i} + r_i, & \text{if } |U| > 0 \end{cases} \tag{4.5}$$

Equation 4.4 finds the task $\tau_i \in U$, which minimizes the total execution time when appended to the end of the near-optimal ordering of subset $U - \{\tau_i\}$. The completion time $C(U)$ of the set U relies on the minimum completion time of subset $U - \{\tau_i\}$ plus the cost $c(\tau_i, U - \{\tau_i\})$ of appending τ_i to the end of a the set $U - \{\tau_i\}$.

Equation 4.5 defines the cost function $c(\tau_i, U)$, which is basically the receive operation r_i plus the receive delay δ_{τ_i} required before the operation. As described previously, the calculation of δ_{τ_i} relies on the ordering of the elements in the set U. The base case for $c(\tau_i, U)$ is when U is empty and it equals to the summation of all operations $(s_i + p_i + r_i)$. Since overlapping is not possible for the first element, s_1 and p_1 operations of the first task directly contributes to the total execution time, whereas the execution time is incremented by only $r_i + \delta_{\tau_i}$ when there are other tasks in U which overlap with s_i and p_i.

The dynamic programming approach starts with the minimum possible subsets where $|U| = 1$ and grows these subsets by iteratively appending tasks τ_i. The ordering of the subsets giving the minimum cost is saved and used in future iterations. We employ this DP approach in our scheduling algorithm.

Algorithm 1 CuMAS Scheduling Algorithm

```
1: **Input:** Task set $T = \{\tau_1, \tau_2 ... \tau_N\}$
2: **Output:** Minimal ordering $\Omega_{min}(T) = (\tau_{\omega_1}, \tau_{\omega_2} ... \tau_{\omega_N})$
3: for $i = 1$ to $N$ do
4:     for each $U \in T$ where $|U| = i$ do
5:         $C_{min}(U)$ =FLOAT_MAX
6:         $\Omega_{min}(U) = \{\emptyset\}$
7:         if $|U| = 1$ then
8:             $C(U) = c(\tau_k, \{\emptyset\})$ where $U^i = \{\tau_k\}$
9:             Set $C_{min}(U) = C(U)$
10:            Set $\Omega(U_{min}) = \{\tau_k\}$
11:        else
12:            for each $\tau_k \in U$ do
13:                if MaxMem($\Omega_{min}$(U-$\{\tau_i\}$)$\cup$$\{\tau_i\}$) then
14:                    *continue*
15:                end if
16:                $C(U) = C_{min}(U - \{\tau_k\}) + c(\tau_k, U - \{\tau_k\})$
17:                if $C(U) < C_{min}(U)$ then
18:                    Set $C_{min}(U) = C(U)$
19:                    Set $\Omega_{min}(U) = \Omega_{min}(U - \{\tau_k\}) \cup \{\tau_k\}$
20:                end if
21:            end for
22:        end if
23:    end for
24: end for
```

4.4.4 Scheduling Algorithm

The CuMAS scheduling algorithm given in Algorithm 1 finds an ordering of tasks using the method described above. The algorithm is invoked by the run-time as the conditions given in the previous Section are satisfied. [Lines 1-2] The scheduling algorithm takes the set T of all tasks as input and outputs a near-optimal ordering $\Omega_{min}(T)$ of T. [Line 3] The algorithm iterates through the smallest to largest subset size i. [Line 5] For every subset size i, we look all subsets $U \in T$ that has size i. [Line 5-6] We want to find the minimum completion time $C_{min}(U)$ and an ordering $\Omega_{min}(U)$ which gives $C_{min}(U)$. [Line 7-10] If $|U| = 1$ then $C_{min}(U)$ is simply the cost $c(\tau_k, \{\emptyset\})$ of the only task τ_k. [Line 12] Otherwise, we iterate through all tasks $\tau_k \in U$. [Line 13] If any τ_k is exceeding the memory requirements when appended to the end of the minimum ordering of Ω_{min}(U-$\{\tau_i\}$), then we do not consider this ordering as minimum. [Line 16] We calculate the completion time $C(U)$ of subset U where τ_k is the last element. [Line 17-19] If the new $C(U)$ is less then the $C_{min}(U)$ found so far, then we save both the minimum completion time and ordering for the case where τ_k is the last element.

The last iteration of the outer loop has only one subset where $U = T$. In this iteration, the $\Omega(U_{min})$ found after the most inner loop will be the output $\Omega_{min}(T)$.

4.4.5 Complexity

The outer loop in Line 3 is iterated N times and the selection of subsets U with size i results in the loop at Line 5 and the innermost loop to be iterated $\binom{N}{i}$ and $\sum_{i=1}^{N} i \binom{N}{i}$

times, respectively, resulting in a complexity of $O(N^2 2^{N-1})$. Although this complexity is still exponential, it grows much slower than $O(NN!)$.

DP solution is faster than the brute force solution for any large N, however, it may not always yield to the optimal schedule. The recursive equation in (4.4) relies on the assumption that the minimum cost solution for a set U will always include the subset $U - \{\tau_j\}$. However, this may not always be true. We will evaluate the effectiveness of our proposed algorithm in the evaluation section.

4.5 CuMAS: Framework and Runtime

CuMAS framework is composed of several components as shown in Figure 5.4. *Call Interception* library dynamically captures CUDA calls issued by the applications assigned to the server. *Duration Estimation* component uses offline profiling results to estimate kernel execution times and a data-transfer model for estimating durations for D2H and H2D calls. *Task Generator* creates tasks from the captured calls and queues them. *The scheduler*, which is the core framework, re-orders the tasks in the ready-queue to improve overall utilization via transfer-execution overlapping. Once proper events and timers are inserted by *Stream&OPs Timer*, *CUDA Call Generator* re-issues CUDA calls in the given scheduling order. CuMAS also employs an offline profiler to measure standalone kernel execution times that are required by the scheduler. We elaborate on the details of these components in the rest of this section.

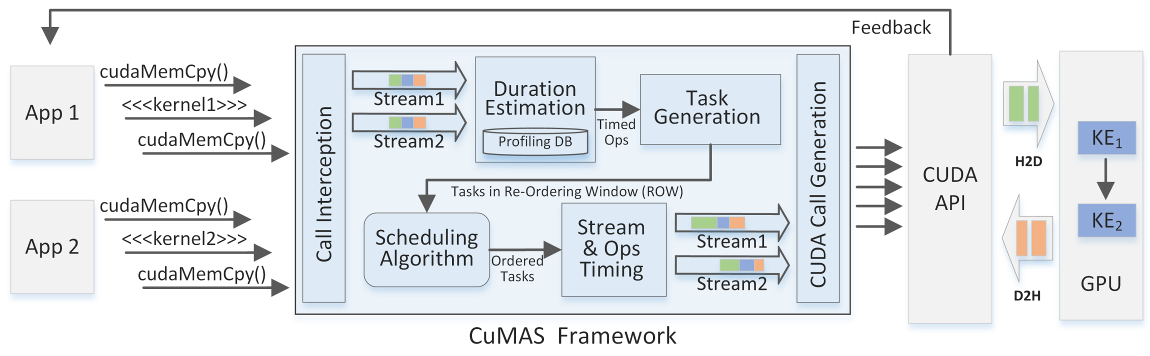


Figure 4.6: CuMAS framework, highlighted in the middle with shaded box.

4.5.1 Call Interception

The entry point of CuMAS framework is *Call Interception* which employs wrapper functions for a subset of the original CUDA API. CuMAS is attached to any existing CUDA C++ binary via the interception library through LD_ PRELOAD flag, hence requiring no modifications to the user code. Each application is allocated a dedicated CUDA stream and as the interceptor captures new calls a new operation is created for each call, along with all configuration parameters, and added to the application's stream.

Our library intercepts *kernel* launches, *memcpy()* and *cudaDeviceSync()* calls. H2D *cudaMemcpy* and *kernel* launches (KE) are immediately returned to the caller, regardless of whether they are synchronous or asynchronous. This is necessary to keep the applications continue issuing their CUDA API calls so that larger tasks are created with as many operations as possible to maximize overlapping in the generated schedule. On the other hand, synchronous D2H *cudaMemcpy* calls are not immediately returned to user and the calling procedure is blocked until all queued operations *plus* the last D2H call for the stream have been converted to a *task*, scheduled, and executed. D2H *cudaMemcpy* calls are always treated as blocking (i.e., synchronous) to prevent any miscalculation of conditional statements that depend on the retrieved GPU results.

4.5.2 CuMAS Runtime - HW Interaction

Using the task order returned by the scheduling algorithm, we issue CUDA API calls for each operation in the task list. Figure 4.7 illustrates the how CUDA calls are captured by CuMAS and scheduled for execution when the PCI-e bus and GPU becomes available.

CuMAS *runtime* is a daemon process which communicates with applications via CuMAS *interceptor*. Whenever the interceptor captures a CUDA call, it lets the *runtime* using a public POSIX message queue (*m_queue*). Upon the receipt of first message from each application, the *runtime* creates a private *m_queue* with the process id of the caller and tells the interceptor either *block* the call or *continue* collecting calls until a task is created for the caller. In the given example, both applications are instructed to *continue* for the H2D and kernel calls.

When the *runtime* receives D2H call for an application, it instructs *the interceptor* to *block* the caller and *wait* for the scheduling decision for that specific application. The *runtime* daemon constantly monitors the events corresponding to previously called CUDA calls so that it can call the scheduler whenever resources are idle. Once a schedule is determined, the *runtime* messages the caller applications to issue their collected calls. Here, it is important to note that, collected CUDA calls are called by the same process, to prevent any inaccessible host memory pointers due to inter-process security restrictions.

If the CUDA call does not specify any streams, the interceptor creates a CUDA stream and modifies *cudaMemcpy()* and kernel call arguments accordingly. Also, to enable overlapped execution, the host memory locations pointed by *cudaMemcpy()* calls are converted into pinned memory using *cudaHostRegister()* call. CuMAS assumes that the kernels do not use any mapped pointers to host memory.

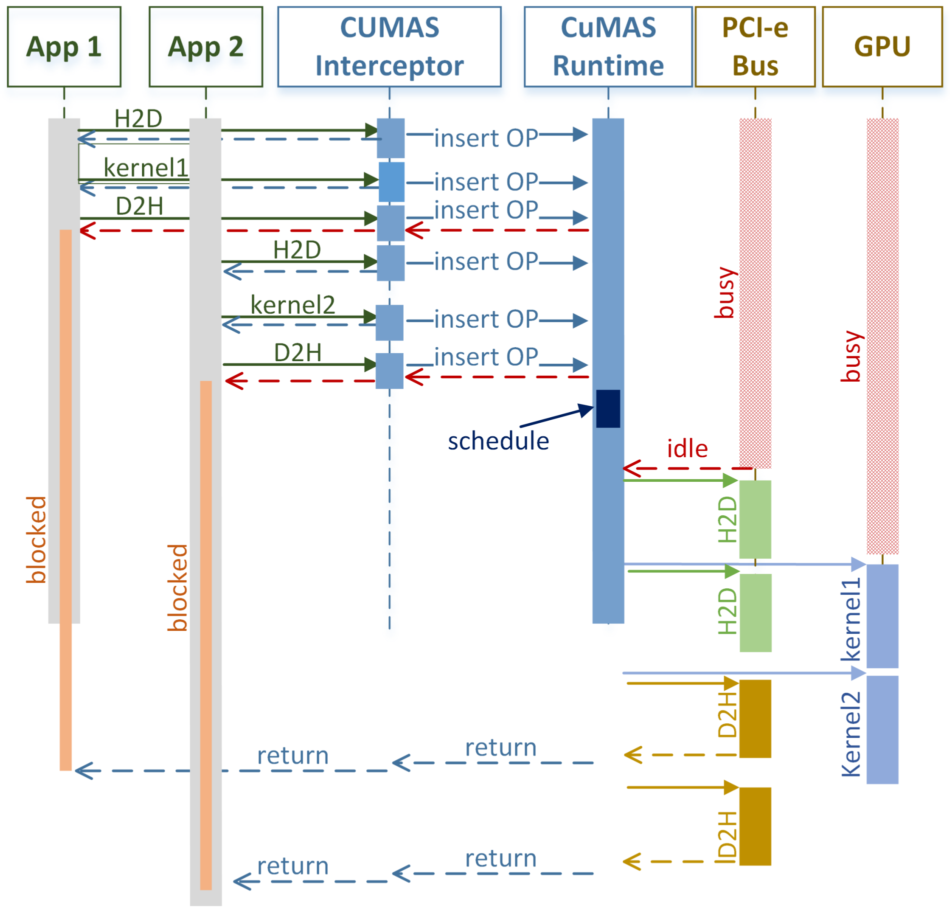


Figure 4.7: Activity diagram showing the interaction between CUDA application, CuMAS framework and the GPU. (The spacing and the length of rectangles do not represent actual timings.)

4.5.3 Duration Estimation & Profiling

Accurate estimation of transfer and kernel execution times is crucial for our scheduler to produce an optimal schedule. CuMAS uses both profiling data and mathematical model to estimate durations for S, P and R type of operations of the tasks in the ROW.

Data transfer time (s and r) is dependent on the size and direction of the data to be transferred. Data transfer time is a linear function of the total data size plus a constant, and the data transfer rate can be modeled with the following equation where x is the data size while a and b are device constants:

$$y = x/(ax + b) \tag{4.6}$$

The constants a and b are measured through experiments and linear curve fitting. The details of the measurement are given in the evaluation section.

Kernel execution time (p), on the other hand, is dependent on the kernel and the size of the input data, hence total thread block (TB) count, on which the kernel operates. Estimating p is harder since it varies across applications therefore CuMAS requires an offline run on a fraction of input data and obtain *msecs/TB* metric for each profiled kernel. We maintain a database of profiled kernels associated with the corresponding *msecs/TB* metric for future executions.

Later, during runtime, we use the dimension of the grid (i.e., TB count) specified during kernel launch and we linearly scale the metric up to estimate the execution time of the intercepted kernel. The accuracy of this simple estimation technique relies on the following factors:

- Same number of threads per TB is used for the profiling kernel and the actual execution.
- Total number of TBs used for the profiling kernel are large enough to utilize all SMs at a given time.
- There is a linear correlation between the grid size and kernel execution time.

If no profiling information is provided on the first execution of a specific kernel, CuMAS does not re-order the tasks containing such kernels and immediately issues them to the CUDA API call queue. If that kernel is encountered again in the future, initial execution time is pulled up from DB.

4.6 Evaluation

In this section we first describe the details of our experimental setup and then present our results.

Architecture: We evaluate CuMAS on NVIDIA's Tesla K40c series GPU attached to an AMD Opteron 6100 based system. The GPU supports PCI-e v3.0 and it has 15 SMX units each having 192 CUDA cores accessing 12GB of global DDR3 memory. K40c has a shared L2 cache size of 1.5MB. Host has 64 cores organized as 8 NUMA nodes connected with AMD's HyperTransport interconnect. For optimal PCI bandwidth we only use the cores in the NUMA node 5, which is directly connected to the GPU via the south-bridge.

Applications: We use a total of 12 applications from Rodinia Benchmark suite and CUDA SDK samples. Table 4.1 lists total number of H2D and D2D calls along with the transfer size, number of kernel calls and tasks created for each application. To evaluate a mixed load of long and short running CUDA applications, we have adjusted the input sizes so that total runtimes vary between 500 msecs and 2 seconds.

Application	**H2D#**	**D2H#**	**KE#**	**Task#**
b+tree	15 (402 MB)	1 (40 MB)	2	2
backprop	5 (1568 MB)	1 (830 MB)	2	2
BlackScholes	3 (1200 MB)	2 (800 MB)	256	1
gaussian	3 (19 MB)	3 (19 MB)	3070	1
hotspot	2 (1212 MB)	2 (606 MB)	1	1
lavaMD	4 (43 MB)	3 (18 MB)	1	1
lud	1 (205 MB)	1 (205 MB)	670	1
nw	2 (1072 MB)	1 (536 MB)	511	1
particlefilter	6 (1043 MB)	1 (1 MB)	77	2
pathfinder	2 (2139 MB)	3 (1 MB)	3	1
srad_v2	2 (1073 MB)	1 (1073 MB)	4	2
vectorAdd	2 (1608 MB)	3 (804 MB)	1	1

Table 4.1: Total data transfer sizes and operation/task counts for each application.

Although some of the applications (BlackScholes, lud, nw, gaussian and particlefilter) in our test-bed issues many number of kernel executions, they do not end up in creation of multiple CuMAS tasks. Our task scheduler merges consecutive kernel calls into fewer CuMAS tasks based on the state diagram given in Figure 4.4. This is mainly due to either redundant computations or statically scheduled kernel launches by the application programmer and the merge does not affect the accuracy of the computation.

Methodology: In our evaluation, we have assumed that first tasks of each application arrive the CuMAS task queue around same time. To achieve this behavior, we initially force our runtime to keep collecting CUDA calls until it captures at least one task from each CUDA application. In our experiments, we have measured data transfers and kernel executions only and excluded the time spent on host-side data and device initialization, cudaMalloc() and cudaFree() calls from our analysis.

We have made all 12 applications run to finish and used a round robin policy to insert the new tasks that are created dynamically as applications progress. Other common policies like fair scheduling were not used because current GPU architecture does not support preemptive kernel execution. Due to many possible permutations of 12 applications, the total executions times are largely affected by the order in which the initial tasks of each application are issued. Therefore, we have performed 25 runs and in each run we started with a randomly selected permutation of the given applications. For each run, we have varied Re-Ordering Window (ROW) size from 2 to 10.

4.6.1 Execution Time

In our experiments we compare CuMAS with the only (to the best of our knowledge) multi-application automatic transfer/execution overlapping technique, *Strings* [72]. As described in the introduction, *Strings* exploits overlapping only if two different type of CUDA calls are issued around same time. On the other hand, for CuMAS, we have used two scheduling algorithms; *CuMAS-BF*, which uses brute force search over all possible

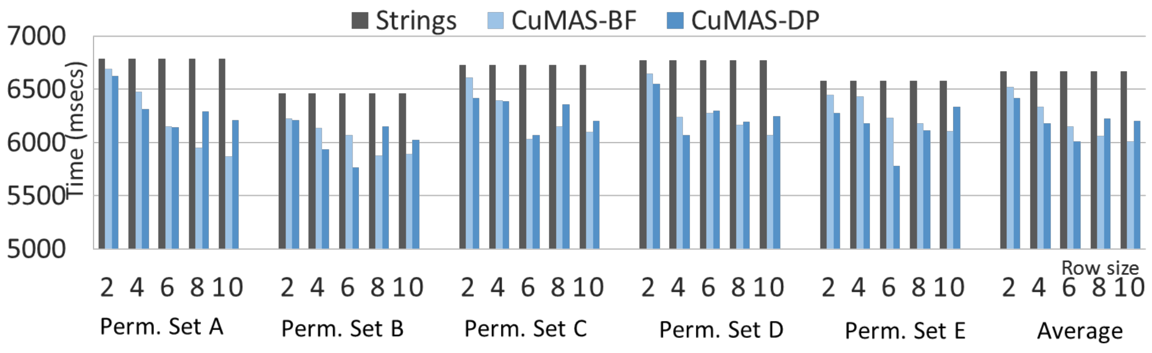


Figure 4.8: Total execution time with varying number of ROW sizes.

permutations of the tasks in the ROW and *CuMAS-DP*, which uses the faster dynamic programming based heuristic as described in subsection 4.4.

We have grouped 25 initial permutations of 12 application into 5 permutation sets, A to E, to better understand the execution discrepancies across different sets. The results given in Figure 6.9 show the average total execution time for each permutation set as well as the global average. x axis corresponds to different sizes of ROW and y axis denotes the total execution time in milliseconds. ROW size differences only affect CuMAS and *Strings* results remain same, since *Strings* relies solely on the incoming call order and does not involve re-ordering.

The results show that the benefits of re-sequencing CUDA calls is significant even with a window size of 2. In most of the permutation sets, increasing ROW sizes reduce the total execution time up to 14% on average and up to 44% percent when compared to the execution of *Strings* with the worst case initial permutation.

For small ROW sizes (2-6) CuMAS-DP performs better than CuMAS-BF due to the way we implement the two techniques. For DP, while calculating the schedule for the current ROW, cost calculation recursively depends on the tasks from previous ROWs, therefore the finish times of the CUDA commands in those tasks are taken into account as well. On the other hand, in BF, we find the optimal execution time of a given ROW by only looking at the tasks inside the current ROW regardless of the task execution times in the previous ROW. This causes possible resource idleness between the bordering tasks in consecutive ROWs.

For larger ROW sizes (8-10), we observe that CuMAS-BF performs better than CuMAS-DP since the target permutation set grows significantly larger than the heuristic algorithm can efficiently address. Despite the high overhead of BF technique for these ROW size, the resulting schedule is fast enough so that the total execution time is still less than the heuristic (DP). Moreover, since the scheduler is invoked right after the send operation of the last task in the ROW, algorithm execution overlaps with the remaining P and R operations of the last task, hence the overhead is partially hidden. However, for ROW sizes larger than 10, BF scheduling overhead is enormously high (as explained in sub-section 4.6.3) and is not compensated neither by the overlapping between pipeline stages nor the performance gain.

ROW size of 10 is evaluated for the sake of analysis only and it is not a practical value for real systems. Although CUDA supports up to 32 concurrent SW streams, latest generation of NVIDIA GPUs employ only 8 HW stream controllers. Any ROW size value, hence total stream count, above 8 is subject to serialization in HW.

It is also important to note that the execution times shown in Figure 8 are the combined outcome of the speedup provided by the scheduling techniques and their overhead.

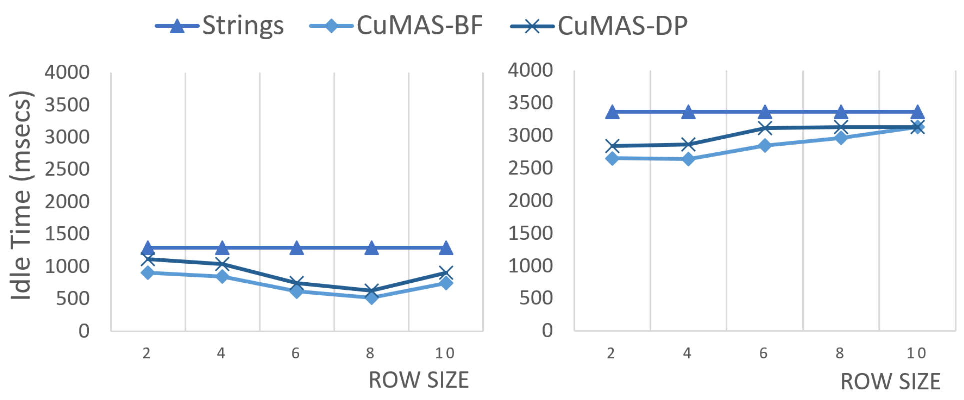


Figure 4.9: Idle time observed two resource channels: P[GPU] on the left (a) and R[D2H] on the right (b)

Since there is no straightforward way to clearly isolate the effects of possible overheads on the total time in pipelined execution schemes, we will evaluate overheads separately in the rest of this section.

4.6.2 Resource Idleness

To better understand how CuMAS improves the total execution time, we have measured the idle time spent by P(GPU) and R(D2H) resources. These times correspond to the sum of non-busy periods between the first and last KE call and D2H call, respectively, across the scheduled tasks in a ROW. We have excluded the idle time for S(H2D) channel, because we send the data for the next task as soon as the H2D operations in the previous window finish. There is no accumulated delay or idleness for S resource channel, provided that there is always a ready to execute task in the task queue.

The results given in Figure 4.9 show that both BF and DP approaches manage to keep the idleness lower than *Strings*. Another observation is that the idle times are

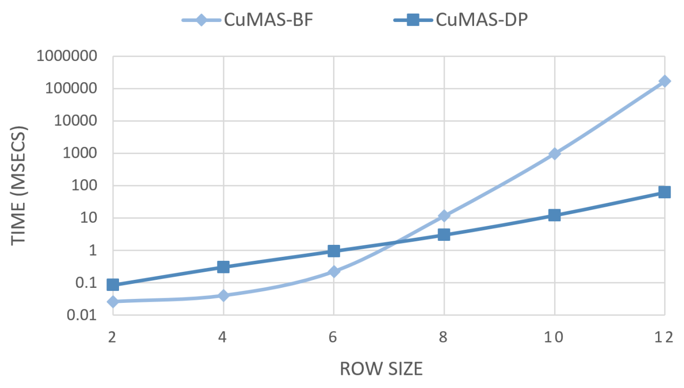


Figure 4.10: CuMAS scheduling overhead.

more significant in R channel, which are also affected by the accumulated delays in the P channel. Also, the idleness difference between the *Strings* and CuMAS scheduling becomes less significant as the ROW size increases. However, although resource idleness gives an overall idea on the reasons for speedup, call overlapping and cumulative resource utilization are hard to quantify and the overall speedup cannot be directly related with the idleness values given here.

4.6.3 Scheduling Overhead

The cost of scheduling for dynamic frameworks like CuMAS is crucial for higher system utilization and it corresponds to the majority of runtime overhead. We measure how much serial CPU time is taken to perform the scheduling for varying ROW sizes – the results are shown in Figure 4.10.

As stated previously in Section 3, there is a tradeoff between the increased overlapping benefits and the scheduling overhead as the ROW size is incremented. Although the brute force search approach (CuMAS-BF) provides the best speedup as shown in Figure 6.9, the overhead of enumerating all task permutations in the window becomes considerably

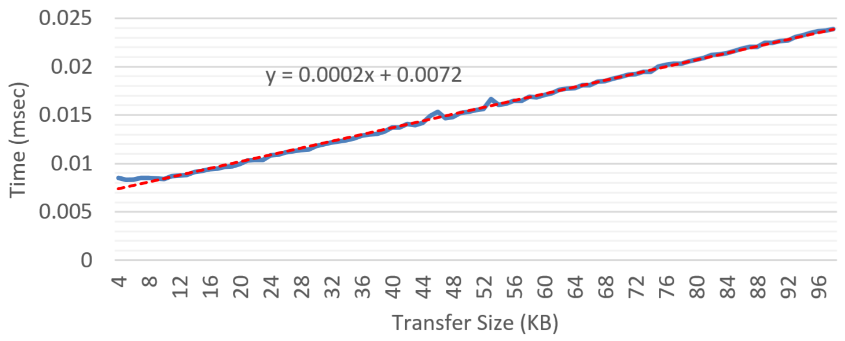


Figure 4.11: Data transfer times (solid) and corresponding curve fitting (dashed).

larger as ROW size exceeds 6. It takes 12 and 972 milliseconds in total for CuMAS-BF to search the best ordering for the tasks in a window of size 8 and 10, respectively. On the other hand, CuMAS-DP manages to keep the overhead at 2 and 12 milliseconds, respectively, for the same window sizes. For a ROW size of 12 tasks, CUMAS-BF takes considerably longer (168 seconds), which is not acceptable for a runtime solution. On the other hand CuMAS-DP overhead (61 msecs) still remains under practical limits.

A ROW size of 8 ends up in negligible overhead, which is under 0.5%, for both CuMAS scheduling approaches. Considering the HW stream controller limit, 8, we may conclude that even with the exponential and factorial scheduling complexities, both CuMAS call re-ordering policies (DP and BF) are practical enough to be deployed in real systems.

4.6.4 Data Transfer and Kernel Execution Time Estimation

Estimating transfer and kernel execution times accurately is essential for CuMAS to exploit maximum resource utilization. We build a linear model for data transfer times whereas we use a profiling based approach to estimate kernel times.

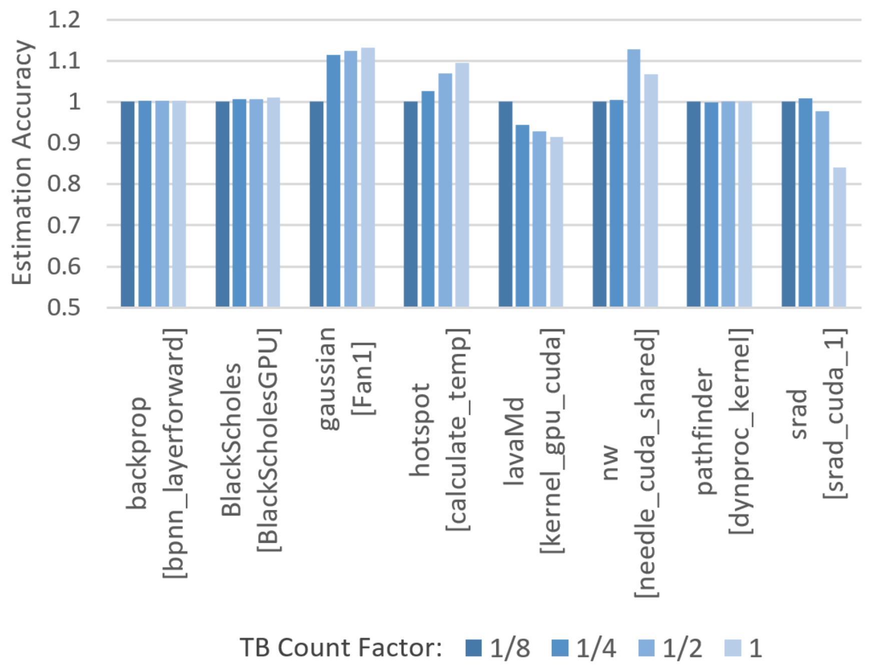


Figure 4.12: Kernel duration estimation accuracy for varying TB count factors.

For data transfer time estimation we have executed H2D calls with data sizes varying from 4KB to 100KB and plotted the execution time (solid line) in Figure 4.11. We have fitted (dashed line) our measurement to the linear model given in Equation 4.6 and obtained architecture specific parameters $a = 0.0002$ and $b = 0.0072$.

To estimate kernel execution times, we have profiled each application using kernels with thread block (TB) counts equaling to $1/8^{th}$ of the kernels, which are to be used in the experiment. We have calculated a $msecs/TB$ value for each profiled kernel and we have compared these estimated values with the real kernel runs. We have doubled the TB count at each run until we reach the TB count of the kernel in the actual experiment. Figure 4.12 plots estimation accuracies for 8 different kernels. The kernels used in the initial profiling with $1/8^{th}$ TB count are taken as baseline with estimation accuracy of 1.0.

The verification results showed that the *msecs/TB* metric we acquired by running the kernels with a fraction of the actual TB counts, estimated the execution times for larger kernels with a maximum error rate of 15% and an average rate of 4%. Such a high rate of estimation accuracy enabled CuMAS to exploit maximum overlapping between CUDA calls.

It is also important to note that, although simple interpolation works for most of the applications in Rodinia benchmark suite and CUDA SDK, the estimation will be incorrect if the correlation with the TB count is not linear (e.g. quadratic equations, kernel bodies with arbitrary loop iterations). In the future, CuMAS profiler can be extended to have more accurate estimation for such applications via compiler analysis or runtime performance profiling. Such techniques would require derivation of non-linear estimation curves by using multiple execution time-TB count data points obtained during training phase.

Chapter 5

Task Based Execution of GPU Applications

5.1 Introduction

Using Graphical Processing Units (GPUs) for general purpose computation have been increasingly popular over the last decade. Massive parallelism provided by hundreds of GPU cores offers considerable speedups for SPMD (single process multiple data) type of computation. Most of the top supercomputers today employ GPUs as primary co-processors in their configuration. Many scientific applications have been ported and optimized to run on GPUs to efficiently process massive amounts of data.

As all good things come with a price, designing algorithms for GPUs requires additional considerations due to the SPMD nature of GPU operations. Single processing cores (SPs) grouped as symmetric multiprocessor units (SMs) enable parallel execution with massive number of threads. However, lack of efficient hardware communication mechanisms between SMs forces parallel threads running on different SMs either being data independent from each other or synchronize via software barriers [77]. Moreover, since GPU hardware scheduler assigns thread blocks (TBs) to SMs in an indefinite order, any possible inter-block synchronization and dependencies need to be addressed statically by organizing threads into separate TBs and consecutive kernel launches. Although this approach is adaptable for applications having regular data dependencies, irregular dependencies will result in statically unpredictable workloads, which are possibly generated during run-time. Due to this dynamically changing behavior of such workloads, algorithms with complex dependencies have been classified as 'unsuitable' for '*data-parallel*' programming approach of GP-GPU computing[59].

We believe GPUs, with its hundreds of cores, can be used to solve problems with '*data-dependent*' characteristics as well. This needs development of a task-based execution model and efficient synchronization techniques somewhat similar to the MIMD operations. The conventional GP-GPU programming paradigms (e.g. OpenCL, CUDA) executes a TB on any SM in no specific order. We propose to treat each thread block as a 'generic task' and smartly assign them into SMs as and when their dependencies are resolved. This will greatly expand the pool of the applications which can be accelerated on GPU, however, implementing the support for this approach presents several challenges.

The key challenge in implementing such a task-based approach is employing an efficient 'synchronization' mechanism between SMs. Checking dependencies between tasks require communication and synchronization between SMs. With the existing CUDA execution model, inter-thread block(TB) communication is possible either via global memory using atomic and memory fence functions or CUDA provided device level synchronization (*cudaThreadSynchronize()* method). Implementing memory-based synchronization mechanisms requires application specific optimizations and tend to be a very slow with increasing number of thread blocks and data[77]. On the other hand, CUDA device level synchronization forces a global barrier across all SM's which will eventually cause some SM's waiting idle towards end of the synchronization periods. Although, NVIDIA's latest Kepler GK110 architecture [60] enables device-side finer granular synchronization using *cudaThreadSynchronize()*, using this mechanism as an inter-SM communication method presents several hardware limitations and programmability issues, which are to be discussed in the next section. Overall, minimizing inter-SM communication while resolving dependencies is a priority consideration in building a task-based execution environment.

Another challenge is to properly *'distribute tasks'* across SMs while preserving locality. Unresolved dependencies and dynamically created tasks cause load imbalance between SMs. Also, execution time of the same task on different SMs might considerably vary due to non-deterministic off-chip memory contention [78]. This causes further discrepancy in the execution times of tasks. In an attempt to solve the problem, Tzeng. et. a.l[86] proposed a centralized queue based scheme where the tasks inserted in a first come first served queue after the dependencies are resolved. Although this approach provides a good load balance, queue-locking overhead and task locality issues results in a considerable performance degradation. *Lock-free and distributed queuing schemes along with locality and load-aware insertion policies are necessary to obtain a better task distribution.*

In this chapter, we propose a new, dynamic task execution framework for executing both regular and irregular data-dependent applications on GPUs. Our run-time totally executes on GPU without relying on CPU and it supports both static and dynamically created tasks. Different from previous studies, our approach designs the scheduler inside the GPU as a concurrent kernel along with the worker threads. Persistent worker threads grab tasks from distributed lock-free queues. We employ a novel parallel-scheduling kernel which processes/populates the tasks in each queue in parallel. The scheduler is light-weight so that it can be placed on the same SM with other threads without the need for any dedicated resources. Our scheduler employs 'local-first, average load aware' queue placement policy for achieving maximum load balance across SMs while preserving task locality. We implement the proposed task-based execution model on a Tesla C2050 GPU and perform various measurements.

Our study makes the following contributions:

- We present a new, CUDA based dynamic task-based execution framework for applications having regular and irregular workload dependencies.
- We implement a novel GPU based light-weight parallel task scheduler, co-existentially running along with the workers, via concurrent kernel execution.
- We explore the performance our system using different queue-insertion policies and propose a new 'local-first, average load aware' policy especially suited for GPU based tasks.
- We compare our results with a centralized queue, all-worker approach based solution, and present the results.

The rest of this chapter is organized as follows: In Section 2 we motivate our work with discussions and experiments. We present our framework with full details in Section 3. Then, we show the results and evaluation of our framework in Section 4.

5.2 A paradigm shift: Task Based Execution on GP-GPUs

General purpose usage of GPU architectures have enabled considerably faster execution for applications which exploits data parallelism in an SIMD fashion. Hierarchical grouping of hundreds of scalar cores(SPs) into larger symmetric multi-processors (SMs) provides a fair trade-off between scalability and inter-SM communication. Threads belonging to same thread block (TB) run on the same SM with access to a fast, shared memory and can synchronize with each other very efficiently. On the other hand, they are totally isolated from the threads in other TBs so that applications can easily scale up to thousands of threads without any degradation in performance. Redundant number of thread blocks, which are provided by the programmer, are executed in no specific order by the thread block scheduler, during a specific kernel launch.

5.2.1 Dependency: An inherent problem of GP-GPU programming

Sacrificing efficient inter-SM synchronization capability in exchange for scalability was a necessary decision made by GP-GPU designers [30]. However, this sacrifice has resulted in severe restrictions on the domain of applications that can be accelerated by GPUs. An ideal GP-GPU application would require a data-parallel region in the program that requires no global synchronization points whereas data dependencies across different threads might require inter-TB communication for computational accuracy.

Except for a limited number of simple kernels, many applications in popular benchmark suites like Rodinia[15] and CUDA SDK Samples implement synchronization mechanisms by dividing parallel regions further into smaller data-independent sub-routines which are then executed on the GPU by consequent kernel launches. However, this approach is hard to design and implement, and also is inefficient for applications having dynamically generated task graphs due to redundant resource allocation required for unpredictable dependencies. For example, a common implementation of breadth first search (BFS) algorithm on GP-GPU requires all nodes to be mapped to all threads, although only a minor sub-set of them (i.e. nodes in a breadth) will be accessed during an iteration of a kernel launch. Most of the SMs are left idle due to redundant resource allocation. Our results from experiment is given later in this section to show the performance degradation due to under-utilization of SMs and kernel launch overhead.

5.2.2 Task-based Execution Model

Treating SMs like standalone processors, which are capable of running tasks independent of each other, will provide a more generalized approach to address the issues mentioned above. Once applications are represented as task graphs, similar to multi-CPU systems, SMs will be able to consume ready tasks as they become idle.

Figure 5.1 shows a sample task graph for Heat2D application, where each task corresponds to an SIMD parallel region of the application. Entire 2D surface is divided into

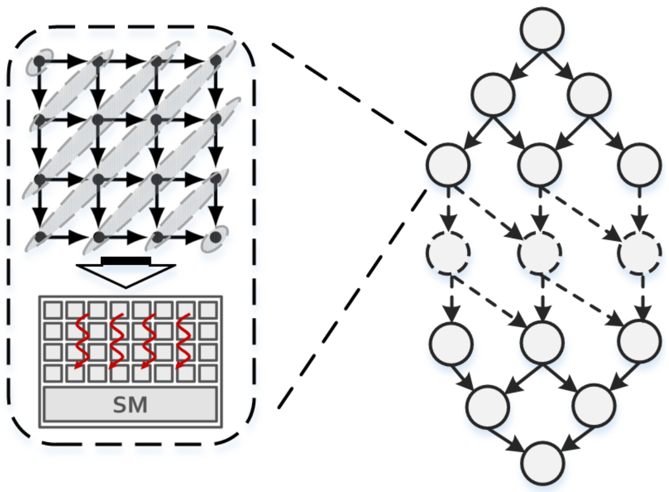


Figure 5.1: Sample task graph for Heat2D application

chunks where each of them is represented by a task. Each task is processed by symmetrical threads in an SM using the standard synchronization techniques. Dependencies between the tasks are represented by the task graph edges and tasks can be executed on different SMs in parallel, subject to the dependency resolution.

An execution model which will enable applications to be executed on GPUs based on their task-graphs will provide following benefits.

Better utilization and parallelism: Since applications are no longer coupled with the SMs via thread blocks, a 'ready-to-execute' task can execute on any SM. Moreover, there will be no need for unnecessary resource allocation due to statically unpredictable dependencies. Resources (i.e. SMs) will be assigned to a task only if the task is ready to run.

Support for scheduling schemes: Task-based execution model will enable further support for different task scheduling schemes and queuing mechanisms. Existing process/task schedulers designed for multi-core architectures can be inherited for better utilization and throughput.

Wider range of applications: A task-based model will enable efficient execution of applications having irregular and dynamic dependencies on GP-GPU systems. Graph algorithms and wavefront parallelizable applications are examples which will show utilization under the new execution model.

5.2.3 Possible Approaches

Designing task-based execution environment requires several considerations to be taken into account.

Persistent Worker Threads: In contrary to traditional approach with many thread blocks of short execution times, a task-based model will employ persistent threads to consistently fetch tasks and execute them. This approach is inline with per processor distributed scheduling function which is called periodically to consume 'ready' tasks as they become available.

Task queues: Similar to 'runqueues' on general purpose CPU systems, data structures to order tasks are necessary. A simple approach would be a centralized queue where each worker thread continuously retrieves tasks synchronously and executes them[86]. Worker threads check dependents of the tasks as they are processed. An alternative solution is to decentralize the queue access to improve scalability. We propose a decentralized approach in this paper.

Task scheduling: Decoupling dependency resolution logic and queue management from worker threads is an efficient technique to improve utilization in traditional

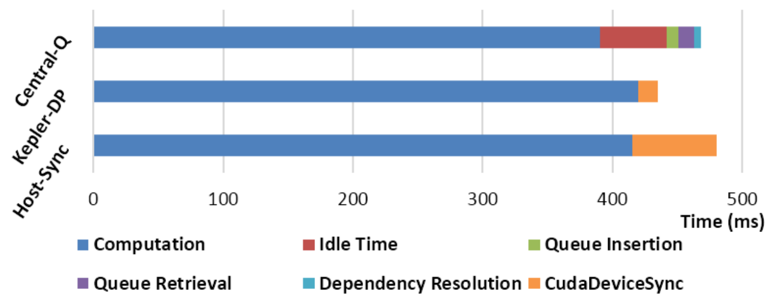


Figure 5.2: Execution breakdown of Heat2d on host&device base synchronization and dynamic scheduling with centralized queue.

task-based scheduling systems. Due to serialized nature of these two operations, CPUs can be utilized as schedulers on GP-GPU systems[17]. Queue(s) are placed on pinned host memory and maintain synchronization using *thread_fence()* calls. Main bottleneck in this approach is the data transfer and control latency between GPU and host.

5.2.4 A motivational experiment

In order to observe the performance effects of existing approaches, we have executed Heat2D, 2-dimensional heat simulation (refer to Section 4.2 for more details), on a surface of 32Kx32K which occupies 4GB of GPU memory. We have used a NVIDIA Titan GK110 GPU with all 14 SMs and excluded all host-side and memory transfer related timings in our results given in Figure 5.2. The application is run under three existing execution schemes:

- **Host-Sync:** Workload is statically partitioned as fixed size blocks, which are then grouped into diagonal 'waves'. Blocks in the same wave can be executed in parallel whereas execution of a wave depend on completion of the previous wave. Inter-wave synchronization are performed on the CPU using host initiated *cudaDeviceSynchronize()* function.

- **Kepler-DP:** Similar to the static partitioning above but synchronization is performed inside GPU without going back to host after each wave.

- **Central-Queue:** Tasks are dynamically inserted into a centralized queue as their dependencies solved. Whenever an SM is idle, it grabs the next task from the central queue in a locked fashion.

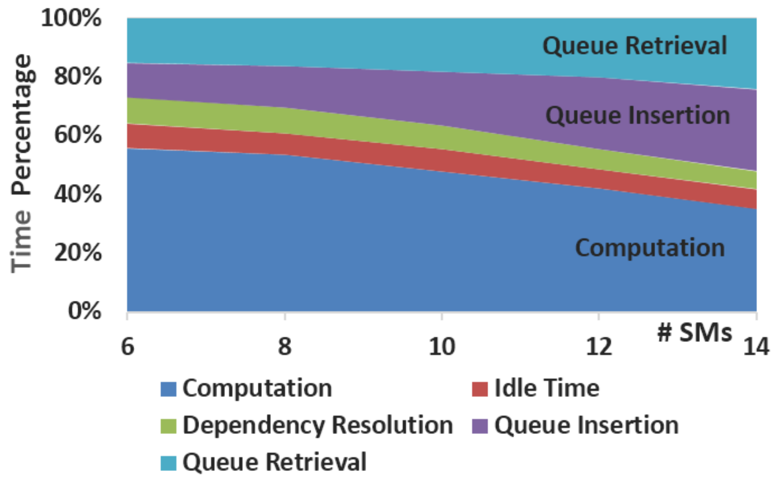


Figure 5.3: Percentage breakdown of timings for varying SM counts on centralized queuing scheme.

The results of the experiment showed that the overhead for the host initiated kernel launch might be considerable when the number of launches gets higher for a large amount of data. Device initiated synchronization supported by Kepler GK110 architecture significantly reduces this overhead at the small expense of hardware based dynamic kernel management overhead.

In contrast to the first two approaches, central queue approach [86] is able to decrease the total execution time. Total run-time as well the computation time of the task-based scheme decreases due to avoidance of multiple kernel launches and better utilized SMs. Dynamic scheduling of tasks has eliminated the time wasted by idling SMs just before global synchronization points. However, the overhead for queue access is still high, but this time, it is due to central queue access contention. Also the time spent by worker threads during dependency resolution contributes to the overhead.

To better understand the overhead of central queue based approach, We have conducted another experiment on the scalability of the mechanism. Figure 5.3 shows the percentage breakdown of execution and overhead for increasing number of SMs used in the system. As the SMs are increased, queue insertion and retrieval overhead significantly grew larger.

5.2.5 Proposed System

In this study, we introduce a new task-based dynamic task execution model. Based on our preliminary experiments, we propose a distributed queue based worker-scheduler mechanism running completely on GPU without relying the host side. The details are given in the next section.

5.3 A New All-In-GPU Task Based Scheduling Framework

In this section we first give an overview of our framework at a high level. Later we will focus on details of our run-time where we explain the scheduler and worker threads'

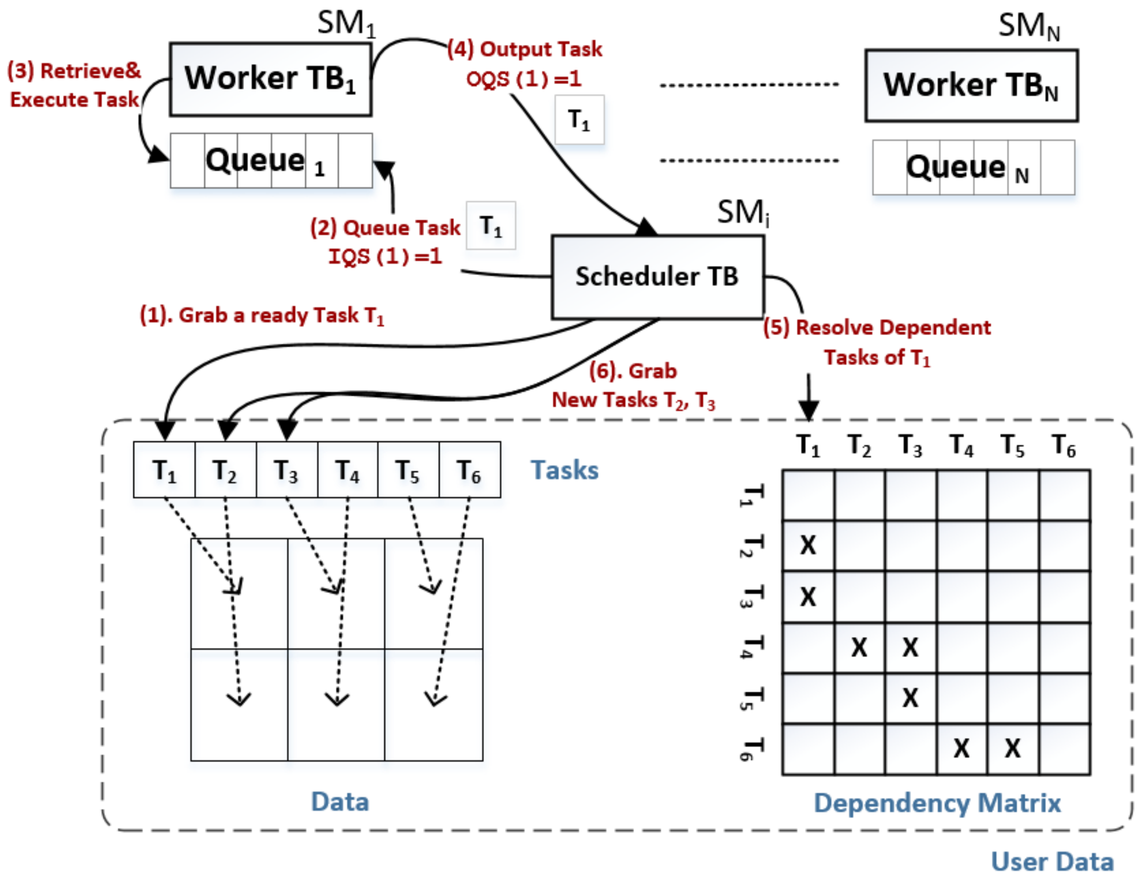


Figure 5.4: An overview of our proposed in-GPU task scheduling framework

operations. We will also elaborate on several queue insertion policies. Finally we will present our API and its use.

5.3.1 Overview of Task-based Execution

An overview of our framework is illustrated in Figure 5.4. The run-time is composed of N Worker thread blocks(TB) and a scheduler TB, where all TBs continuously run until the application ends. N is equal to the number of SMs. Each SM (interchangeably used with 'worker TB') is associated with a private task queue located in global memory.

Queue is decentralized (one per SM) so that each worker TB can read from/write to its own queue without synchronizing with other TBs.

Scheduler TB operates in a parallel manner where each thread accesses to a specific element in the task queues associated with each SM. Dependency checks are also done in parallel.

Initial tasks list, dependency graph and the actual data which tasks point to are to be provided by the user, as shown in the bottom part of Figure 5.4. Both static and dynamically populated task lists are supported. In a typical scenario, each task structure contains pointers to the actual data on the GPU global memory.

The proposed task execution follows a sequence of six operations as marked clearly in Figure 5.4.

1. Scheduler initially searches the tasks list and identifies the starting tasks whose *dependencyCounter* is zero (which means that all prior dependencies of the task is resolved and it is *ready* to execute).

2. Scheduler TB inserts the ready task into a proper worker queue based on the scheduling policy. Corresponding worker TB is notified by an increment of the Input Queue Size (IQS) counter, which is implemented as an atomic variable.

3. Worker TB grabs tasks, forwards it to the user application. The application can fully utilize the SM, since worker TB transfer the full control to the supplied user kernel.

4. When the user kernel finishes processing the task, control is transferred back to worker TB. To let the scheduler know that the task is finished, worker TB, increments Output

Queue Size (OQS) counter for the corresponding SM. Please note that this counter is for signaling purposes only and there is no separate output queue.

5. Scheduler TB continuously checks each worker TB for outputted tasks in parallel. Once a task is processed by an SM (i.e. OQS is greater than zero), the scheduler TB goes through its dependents via user supplied dependency graph and discovers whether new tasks become ready to be processed (i.e. independent).

6. The process repeats with the new ‘ready’ tasks. This step replaces step (1) for the rest of the execution.

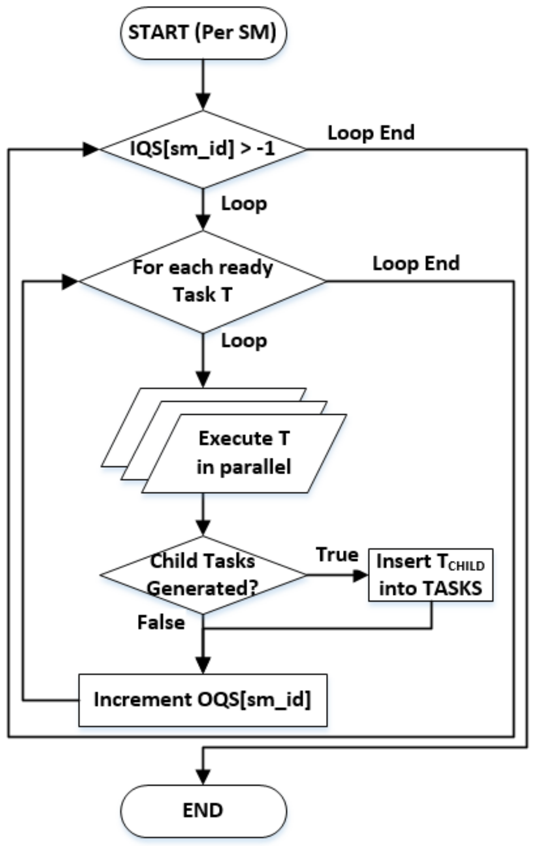
Both scheduler and worker TBs terminate when the scheduler decides that there are no tasks to process.

5.3.2 Run-time Components

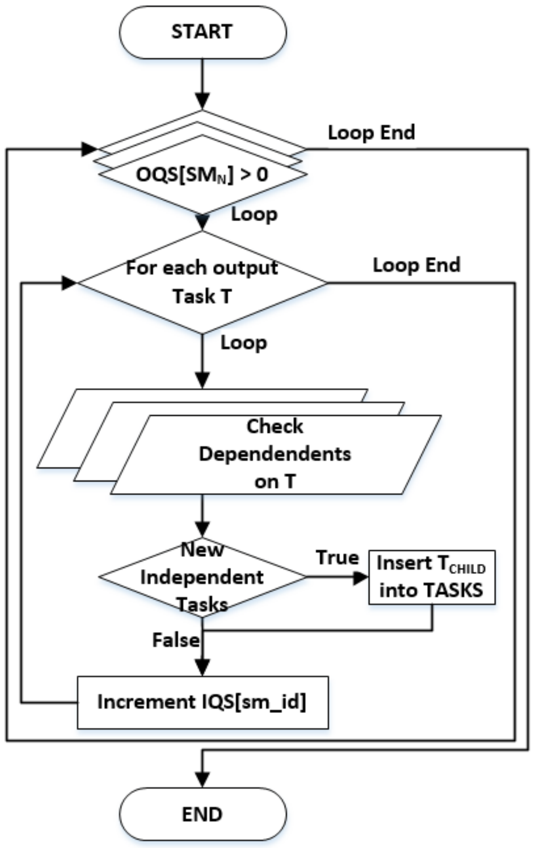
In this sub-section, we explain several components of our task-based execution model.

(a) Concurrent kernels:Worker&Scheduler

Our run-time is composed of two kernels: worker and scheduler. Worker kernels are wrappers for actual user functions. They also carry consumer operations like grabbing tasks from the distributed queues and writing processed tasks back to the queue. Scheduler thread, on the other hand, is solely transparent from the application code and runs as an on-GPU daemon.



(a) Flow chart for worker TB operation



(b) Flow chart for scheduler TB operation

Figure 5.5: Worker and Scheduler Operations

In order to have both scheduler and worker TBs to run simultaneously, our runtime uses concurrent kernel execution feature which was first introduced in Fermi architecture. We asynchronously launch two kernels via two different streams with scheduler and worker grid with dimensions of 1 and N, respectively. During kernel launch, GPU hardware scheduler assigns corresponding thread blocks to available SMs according to their resource availability. Since our scheduler is light-weight, we have observed that scheduler and worker TBs can be placed on the same SM without any noticable degradation on co-scheduled worker TB.

In a rare case, if a user kernel embeds thread blocks that fully utilize an SM, then, an SM needs to be reserved to the scheduler. In this case, our framework sets total number of worker TBs to N-1 instead of N. This would result in a performance degradation of 1/N at most.

(b) Inter-SM Signaling: *IQS* & *OQS*

Once the scheduler and worker TBs are assigned and start running on SMs, they need to check queues of each other for input and output tasks. A straightforward way would be to check the entire task queue for any modified or inserted tasks. However, such frequent access will result in excessive memory contention.

As an efficient way to check queues, we use two special arrays (i.e. counters), *input_queue_size[]* (*IQS*) and *output_queue_size[]* (*OQS*) to provide an atomic signaling between them. Each worker and scheduler TB continuously checks for *IQS* and *OQS*, respectively. Figures 5.5a and 5.5b show corresponding pseudo codes. When a worker TB encounters a greater-than-zero value in *IQS*,this implies that the scheduler has placed a task into that specific worker's queue. Similarly, a positive value encountered by the scheduler in the *OQS* of a worker implies that the worker has finished processing a task. To further decrease the contention, we have forced worker threads to briefly stay idle when *IQS* is zero.

(c) Lock-free Queues: Since global memory is the only way for SMs to communicate with each others, task queues are to be placed in GPU global memory. Although *IQS* and *OQS* signaling mechanism prevents any redundant global memory accesses (i.e. queue access) and decreases possible contention, queue operations still need to be optimized due to the bandwidth-sensitive cache and memory structure of GPUs.

As a part of this optimization, our run-time employs a distributed queue scheme in order to eliminate synchronization between workers on queue retrieval. Moreover, writes and reads between scheduler and worker are also lock-free. As also illustrated in figure 5.4, both worker and scheduler TBs hold separate pointers to the index of the task which has been inserted and processed, respectively. With lock-free queues, worker and scheduler TBs can execute simultaneously without waiting each other.

(d) Parallel Queue Access and Task Scheduling Another important part of our queue access optimization is exploiting parallel and aligned access. Worker and scheduler threads belonging to same wrap access consecutive queue locations in order to avoid 'costly' unaligned accesses.

In addition to parallel queue access, our scheduler thread block also exposes higher utilization of underlying parallel SPs by calculating dependency checks originated from each queue index in parallel. Scheduler TB is set to have *number of worker TBs (N)* times *queue length (M)* number of threads. Although not all of those threads are active all the time, any task subset of any queue are checked for dependency resolution in parallel. Not only does this approach prevent wasting GPU compute power, but also decreases the total time required by scheduler.

Once the scheduler detects processed tasks, resolves dependencies and identifies new tasks, the only remaining operation is to decide the queue in which the new task should be placed. Next subsection details how this decision is made.

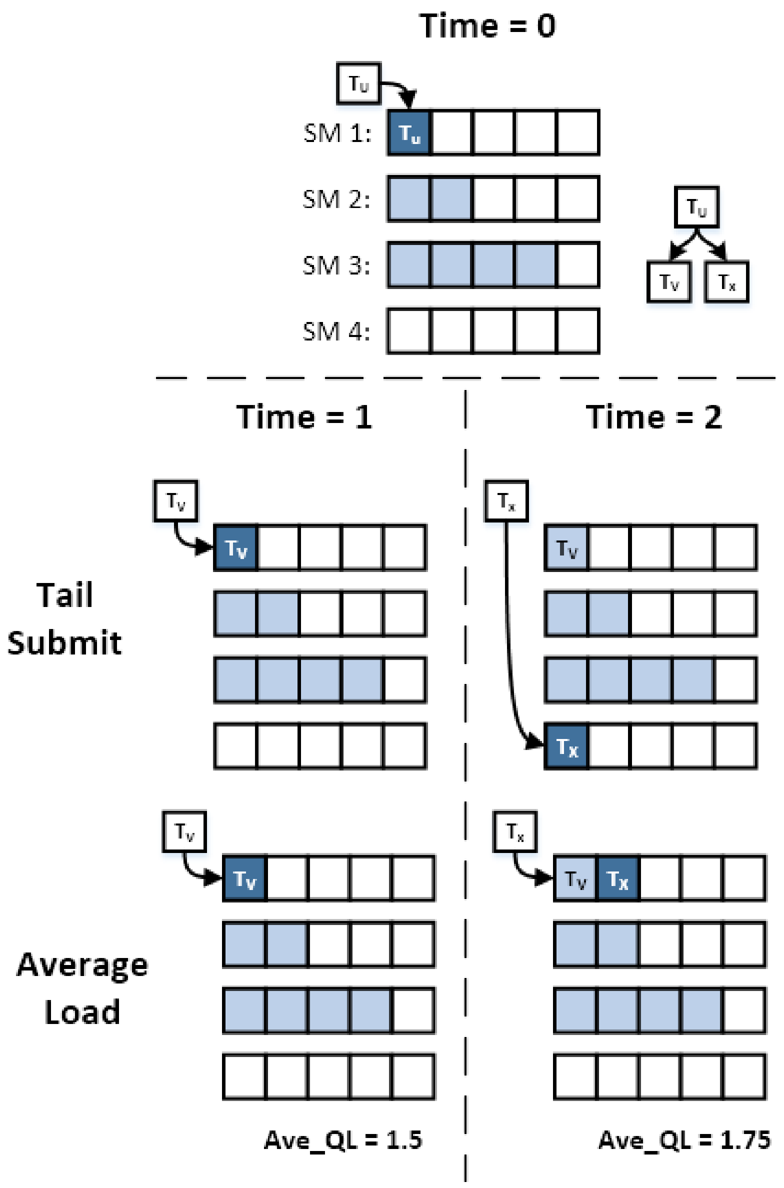


Figure 5.6: Illustrative step-by-step explanation of how Tail-Submit and Average Load works. At time 0, parent task t_u is inserted and processed by SM 1. Children tasks t_v and t_x are freed and become ready to execute. At time 1, tail submit and average load inserts first task (t_u) into the local queue, SM 1. At time 2, for the second task, TS chooses SM 4 and AL chooses again the local queue, SM 1.

5.3.3 Queue insertion policies

As in all other multi-processor decentralized task queue based systems, queue insertion policies play an important role on achieving load balance as while preserving data locality as much as possible. We have embedded three different policies and evaluated their effects through experiments.

Round Robin: Round robin is our most primitive policy which simply places a given task to next available SM provided that its task queue is not full. This approach is known with good load balancing for homogeneous processors and tasks. However this approach does not take locality and possible divergence between task execution times into account.

Tail-submit: Tail-submit executes the first generated on the local processor immediately, as shown in Figure 5.6. Later tasks are inserted to the queue that has minimum number of tasks. Tail-submit prefers locality and does not prioritize load balancing as an objective.

Average Load with local-first: We propose this policy as an alternative to previous approaches. In this policy, we keep average queue load and insert into a queue only if its load is under average load. To preserve locality we first try the local queue and then try other queues. This policy is also illustrated in figure 5.6.

Performance implications of these policies will be investigated in detail in the Evaluation section.

```
__global__ void scheduler_kernel(
    SchedContext);
__global__ void worker_kernel(
    WorkerContext, SchedContext);
__device__ void user_task(
    Task*, WorkerContext*, SchedContext*);
```

Figure 5.7: Worker and scheduler kernel releated API functions

5.3.4 API

Figure 5.7 shows CUDA kernel and device methods of our API. Scheduler and worker kernels are to be asynchronously launched using separate systems. For each task, worker kernel internally calls *user_task* device function which is to be implemented by the user. This function should include application specific computation that is to be carried for the task previously defined for the user. If a new task is generated by the *user_task*, it needs to be inserted into the Tasks array defined in the *WorkerContext*.

```
WorkerContext* initWorkerContext();
SchedContext* initSchedContext(WorkerContext*);

struct _WorkerContext {
    Task* tasks;
    int* dependencies;
    int* dependencyRowIndicies;
    int taskCount;
} WorkerContext;

struct _Task {
    int id; //required
    int dependencyCounter; //required
    int data_index;// App specific
} Task;
```

Figure 5.8: Worker and scheduler context related API functions

The two contexts, *WorkerContext* and *SchedulerContext* (depicted in Figure 5.8) hold user supplied data and internal run-time data, respectively. WorkerContext has some default fields (*tasks*, *dependencies* and *dependencyRowIndicies*) which are used to store app specific task graph. Users of the API can introduce any additional fields, (i.e. the actual data to be processed), inside this data structure. Two initialization functions *initWorkerContext* and *initSchedContext* are called to instantiate the fields inside each context on device memory and transfer related data to the device prior to the execution. Users are required to provide an implementation for *initWorkerContext*, where the app specific task graph is to be created and populated.

5.4 Evaluation

In this section we will present a detailed evaluation of our proposed framework along with platform and applications information.

5.4.1 Platform

We have carried our experiments on a 64-core AMD based system with a nVidia Tesla C2050 GPU running on 64GB of total system memory.The GPU is based on Fermi Architecture and it is attached to the system via PCI-Express slot. It has 3 GB of on-board memory along with 448 CUDA cores delivering up to 515 Gigaflops of peak performance. There are 14 SMs in total and each SM contains 32 SPs (CUDA cores).

In order to demonstrate the efficiency of our distributed queue (DQ) scheme, we have also implemented the centralized queue (CQ) approach described in [86]. For a fair comparison, we have done our best to embed all implementation-related optimizations while re-implementing CQ scheme.

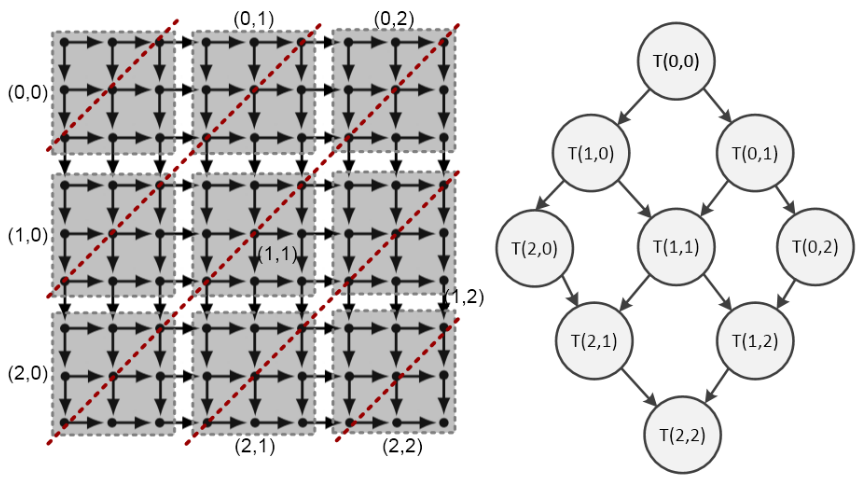


Figure 5.9: 2-level tile based wavefront execution for heat2D algorithm and corresponding task graph

5.4.2 Applications

In order to evaluate our task-scheduling scheme, we have selected two applications that represent the extreme ends of applications with dependencies. Heat Equation (Heat2D) has tasks with regular dependency patterns and breadth first search algorithm (BFS) has dynamically created tasks with irregular dependency patterns. We will now give more details about these two algorithms.

Heat2D: *Heat 2D* simulates dissipation of heat over a two dimensional surface during a specific time frame [1]. During each simulation step, the algorithm calculates lateral heat transfer of every point in the 2D surface by averaging the heats of its neighboring elements. The original method operates by executing diagonally aligned elements of a given matrix in parallel. Multi-processor enabled Wavefront approach uses a 2-level approach as depicted in Figure 5.9. The 2D data space is divided into smaller regions called 'tiles' as also proposed in [20]. Inter-tile wavefront execute tiles in the same wave in parallel. Intra-tile wave-front exploits finer grain parallelism inside a tile by processing elements of the same wave in parallel. In our experiments, we have set each tile as a separate task and our run-time assigns each of these tasks to SMs as their dependencies resolved.

Heat2D is an example of regular data dependencies where the dependency relation between tasks are fixed before execution. Figure 5.9 shows how the tiles are formed and its corresponding task graph. In the static approach, diagonal dashed lines cross over the tiles that are executed in parallel during the same kernel launch. In the task-based approach, a tile will be executed as soon as the dependencies of the associated task are satisfied.

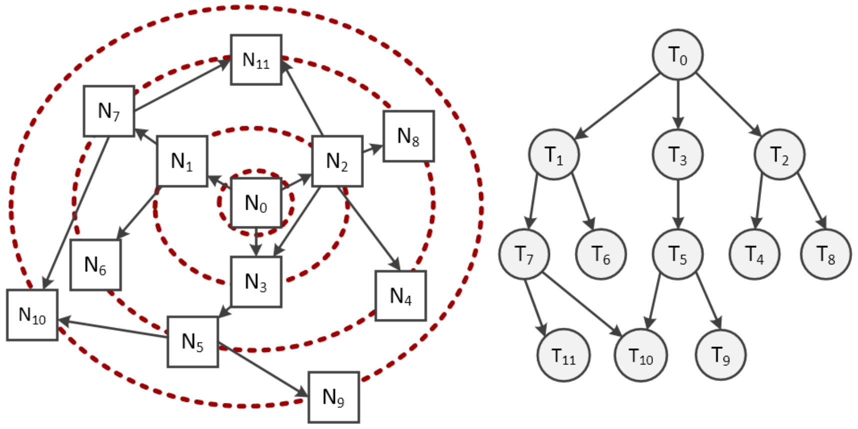


Figure 5.10: A sample graph for BFS algorithm and corresponding task representation

BFS: *Breadth First Search* (BFS) is a primitive operation on which many graph operations rely. Given a set of nodes, edges and cost values, BFS iterates over the graph in breadths and visits an unvisited neighbor of previously visited nodes. Traversal starts from a specific node denoted by the user and cost values are updated as the breadths proceed. For each node, minimum traversal cost from the root node is calculated.

A straight forward CUDA implementation involves assigning each thread to a node and calculate costs for the nodes in a breadth in parallel. Once processing of a breadth is completed, in order to provide synchronization between all thread blocks, another kernel is launched for the next breadth. However this approach leaves most of the threads that do not correspond to the nodes in the current breadth idle.

In our implementation we have created a task for each node and assign a group of tasks to each thread block. When a task is to be executed, neighbors of the corresponding node are traversed in parallel. This two level parallelism exploits a higher utilization than the traditional approach.

BFS is a good example of irregular and dynamic data dependencies which cannot be statically scheduled without excessive under-utilization. On the left side of Figure 5.10, dashed circles represent a sample iteration of 'breadth' over the nodes in the graph. In the static approach, kernel is launched to process each breadth. On the right hand side of the Figure 5.10 a task is created for each graph. The dependencies in this task graph are dynamically determined as the execution proceeds.

5.4.3 Results

In order to evaluate our proposed run-time we have performed a series of experiments. We have first measured execution time of our proposed distributed queue (DQ) with three different insertion policies (Round Robin, Tail Submit and Average Load) as well as the already existing Centralized Queue approach (CQ). Then we have investigated load balancing abilities of these policies. Finally we have evaluated scalability of our distributed queue scheme (DQ) versus CQ approach

Performance:

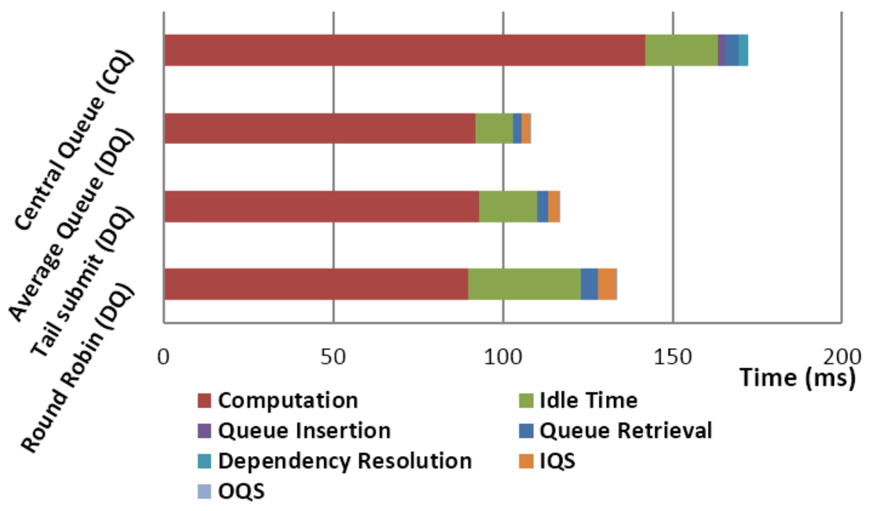


Figure 5.11: Heat2D: Average execution time breakdown for various scheduling and queuing policies.

As described in the previous section, we have implemented three different distributed queue insertion policies: Round-Robin (RR), Tail-Submit (TS), and our proposed Average Load with Local-First (AL).

We have run task based Heat2D and BFS applications using 14 SMs (Thread Blocks). The input size for Heat2D was a 2D array of 8K by 8K. Tile size is taken as 256x256 and total number of tasks was 4K. For BFS, We have used a graph containing 2K nodes, resulting in a task graph with the same number of nodes. The generated graph has 4 neighbors at max and the maximum breadth size was fixed at 128.

The results in Figure 5.11 and 5.12 depicts the breakdown of the average total execution time. The following timings are shown:

- **Computation:** The time spent for executing a task already fetched from queues

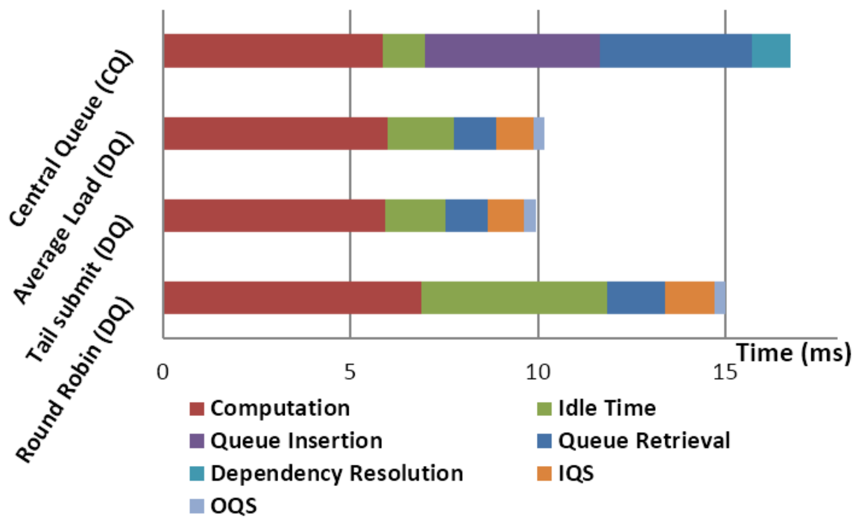


Figure 5.12: BFS: Average execution time breakdown for various scheduling and queueing policies.

- **Idle Time:** The total idle time spent by SM while waiting for a task to be assigned.
- **Queue Retrieval/Insertion:** (Central queue only) The time spent while acquiring central queue lock and retrieving/inserting a task.
- **DRT (Dependency Resolution Time):** (Central queue only) The time spent by workers while going through dependents of an executed task.
- ***IQS* & *OQS*:** (Distributed queue only) The time spent during atomic read of input and output queue signals between scheduler and workers.

Our newly introduced policy, Average Load (AL) has provided speedups up to 15% for Heat2d 62% for BFS when compared to centralized queue approach.

For both Heat2D and BFS, it can clearly be observed that centralized queue scheme shows poor performance in all cases. For Heat2D, computation time is significantly higher due to increased memory contention. The Heat2D algorithm is memory intensive, therefore memory accesses issued while accessing central queue are adversely affecting the execution. For BFS, quick execution time of the algorithm dramatically increases the retrieval/insertion rates of tasks, hence causing a significant jump on central queue access latencies.

Distributed queue schemes, on the other hand, all result in faster execution times against central queue. Signaling timings, IQS and OQS, replaces the delays for queue insertion/retrieval times by resulting in less overhead in total. Especially for BFS, the difference between these two set of timings shows the clear advantage of distributed queue approach over the centralized queue scheme.

When we compare distributed queue schemes with each other, it can easily be observed that round-robin (RR) causes SMs to stay idle more than others. This is primarily due to destroyed locality between tasks. When a task is processed, all of its children tasks whose dependencies are resolved are placed into other queues due to increasing round-robin counter. Although RR policy seems to be distributing equally across queues in total, misplacement might cause a delay in the execution of newly created tasks.

Tail submit (TS), is causing less idle time when compared to RRs. This is mainly due to partial locality preserved during the insertion of first task. Locality affects are more dominant with memory intensive Heat2D application. Possible load imbalances on local queues due to blind insertion causes slightly larger idle times for Heat2D when compared to AL.

Overall, AL is the winner due to increased load balance in addition to improved locality. Unlike TS, AL preserves load balance on the insertion of first task as well, hence

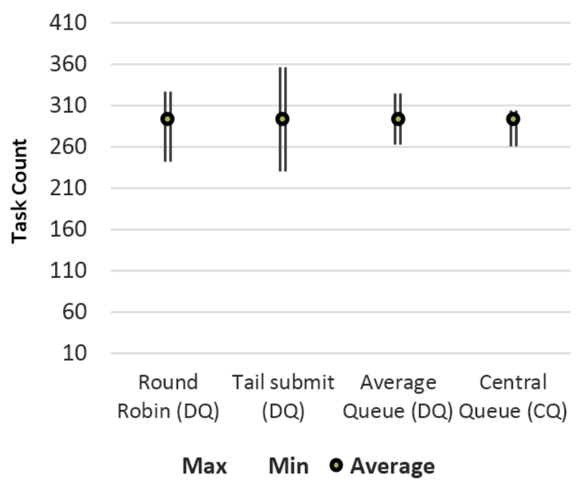


Figure 5.13: Heat2D: MIN/MAX/Average task distribution across SMs for all insertion policies.

decreasing the possibility of leaving other SMs idle. However, for BFS, there are almost no differences between TS and AL because locality is not a major concern due to lower access rates and scattered in-memory representation of node-edge graph.

In summary, Average Load (AL) policy has two important features which makes it a better approach than others.

- AL policy keeps locality for all newly created children tasks as long as local queue load is under the average load of all queues.
- Since locality will not help when there are idle SMs (as the case with TS policy), AL will place a child task to a remote under-loaded queue when local queue is overloaded.

On average, AL policy causes queues being idle only 10% percent of the time with a minor (5%) queuing overhead.

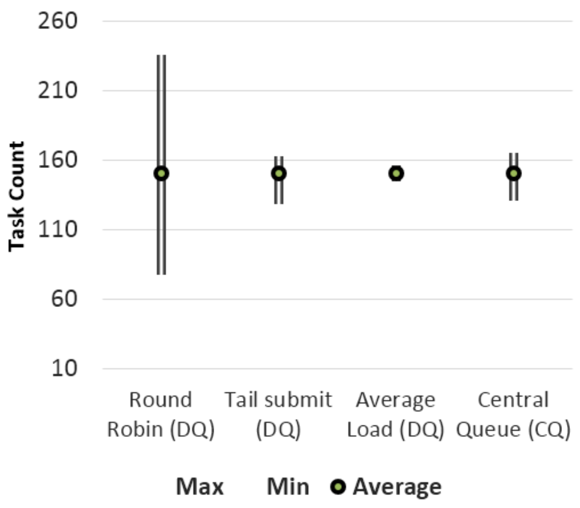


Figure 5.14: BFS: MIN/MAX/Average task distribution across SMs for all insertion policies.

In order to provide more insight to effects of load balance, we have also measured the discrepancy between total tasks assigned per each SM and showed the results in Figures 5.13 and 5.14. In parallel with the execution times, our newly introduced Average Load policy has resulted in the best load balance across SMs. Although CQ shows fair load balance in both cases, performance degradation is mainly caused by the loss of locality. TS policy has worse load balance than CQ because of tail submit on first task, however, still performing better than CQ in execution times, due to better locality . In both benchmarks RR shows poor performance due to random distribution.

Scalability:

We have also observed the effects of varying number of SMs on performance for the Distributed Approach with Average Load insertion policy (DQ) and centralized queue approach (CQ). In this experiment, while keeping other parameters same, we have changed

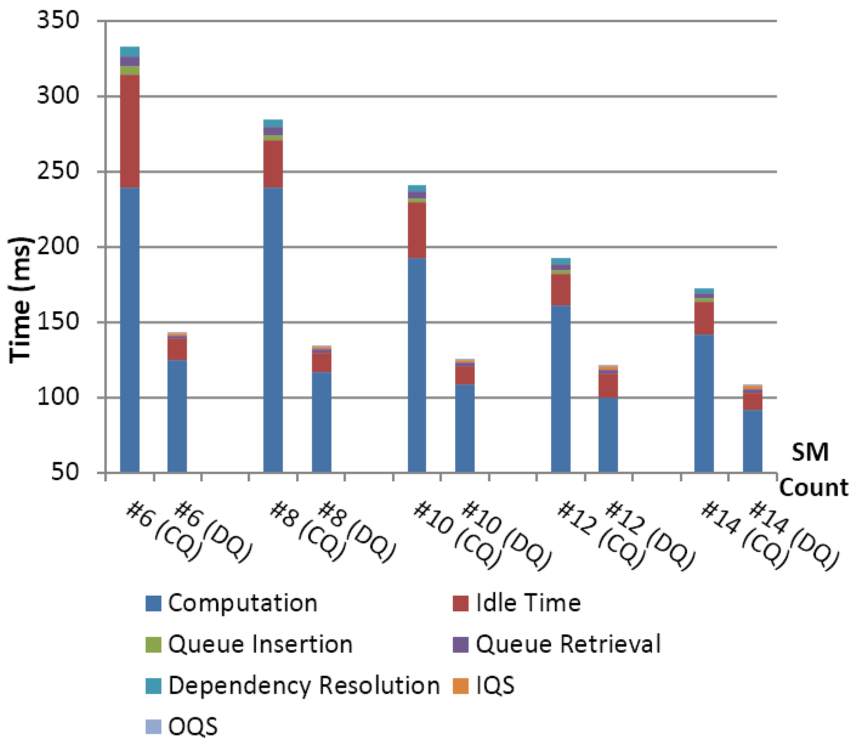


Figure 5.15: Heat2d: Performance of Distributed Queue with Average Load with varying SMs

the total number of SMs from 6 to 14 in order to observe the changes in idle time and queue access time. Time breakdown for each SM is shown in pairs in Figure 5.15 and 5.16.

An immediate observation is that the average idle time for CQ increases significantly for increasing number of SMs.This is mainly due to adversely affected task locality as more SMs insert un-related tasks simultaneously to the central queue. In other words, effects of locality become more prominent as more SMs are involved in the computation.

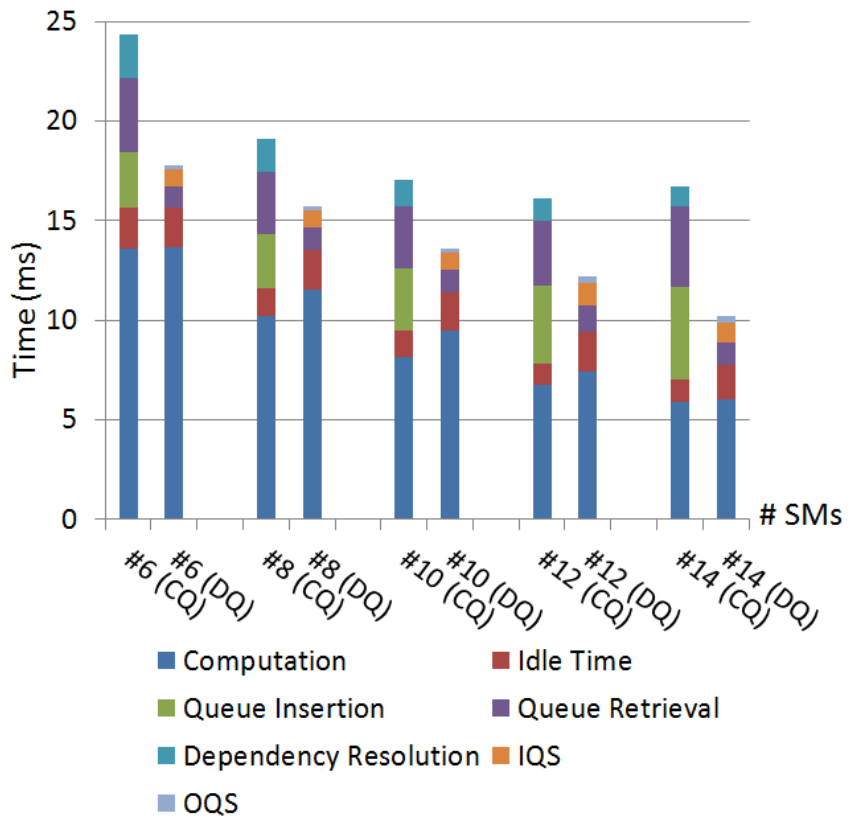


Figure 5.16: BFS: Performance of Distributed Queue with Average Load with varying SMs

Chapter 6

Peer-SM Synchronization in GPUs

6.1 Introduction

Since nested loops are a major source of parallelism, a great deal of research has focused on parallel execution of nested loops. Many classes of applications like time-based simulations, linear system solvers, and string matching algorithms employ a combination of DOACROSS and DOALL parallelism to process large volumes of data efficiently on massively parallel systems. Application and architecture specific challenges imposed by the data dependencies between loop iterations have lead to the development of various compile-time [25, 55], run-time[16, 57, 50], and algorithm-level [76, 22, 58] solutions.

GPUs have been used to accelerate wavefront based applications via use of their massive number of processing units. Many frequently used algorithms such as Smith-Waterman [70, 93], Cholesky Factorization [76], and SOR loop nests [24] have been adapted to run efficiently on GPUs. The two level parallelism via tile partitioning is also commonly exploited by GPU based techniques whose evaluation has been reported in several studies

[25, 24, 93]. In these approaches, a thread block (TB) is created for each tiles in a diagonal and these tiles are processed in parallel by the multiple streaming multiprocessors (SMs) of the GPU (*inter-tile* parallelism). Iterations along diagonals within a tile are also executed in parallel by the SIMD computation units (i.e. CUDA cores) inside an SM (*intra-tile* parallelism). Similar to CPU based approaches, global barriers are required for both tile-level (i.e. *inter-tile*) and element-level(i.e. *intra-tile*) diagonals in order to enforce data dependencies. However, on GPUs, using global barriers for 2-level tiled wavefront execution has two major drawbacks:

1. Low utilization due to load imbalance. The use of global barriers leads to low utilization of processing units during both intra- and inter-tile parallel execution. The intra-tile imbalance causes threads utilization to be lower towards the beginning and the end of diagonal iterations. The same thing happens in processing diagonals of tiles, where the number of tiles are less at the beginning and towards the end. Moreover, inter-tile processing is also imbalanced, when other processors wait for the longest running task (i.e., processing of a tile) in a diagonal. SMs are forced to stay idle, even though new tiles are ready to execute as their dependencies have been satisfied. Intra-tile load imbalance for GPUs is addressed in [25] where tiles are transformed into polyhedral planes [41] to maximize utilization of threads processing a tile. However, this approach still uses inter-tile global barriers causing some SMs to unnecessarily wait for the longest running task.

2. Loss of data locality across tiles is another cause of performance degradation in global barriers. In CUDA, thread blocks, hence tiles, are executed in no specific order; therefore there is no guarantee that neighboring tiles will be processed by the same SM. This

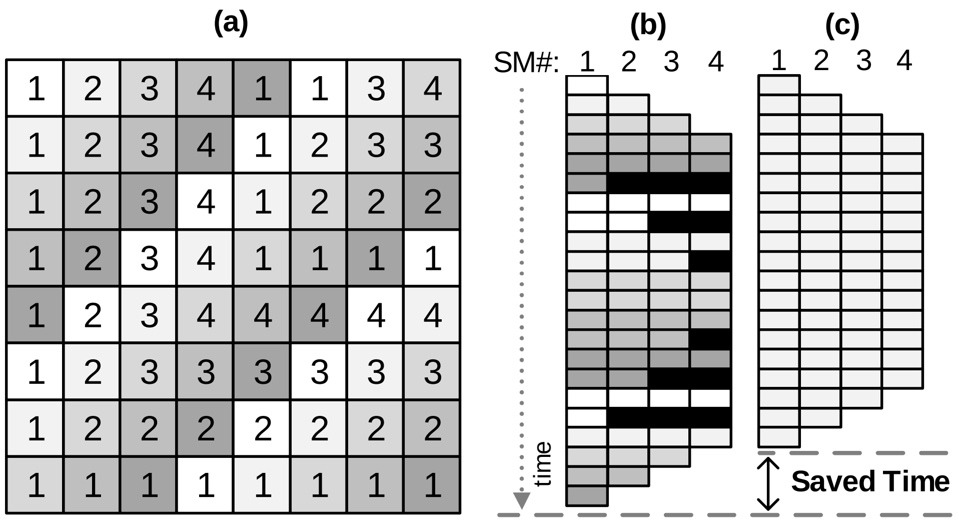


Figure 6.1: SM Under-utilization: On the left in (a) tiles are numbered with SMs executing them using global barriers; (b) shows the execution time-line of each SM using global barriers where dark-black slots indicate idle times; and (c) shows the ideal execution with the idle slots are eliminated.

behavior destroys the dependency-implied locality across the border elements of adjacent tiles. Efforts have been made to overcome this drawback of global barriers. To improve data locality, [54] statically assigns row of tiles to the same processor and for shared memory multiprocessors large CPU caches significantly improve data reuse between neighboring tiles. However, intra-tile operations are still carried out on a diagonal basis, which constitute most of the computations. In contrast to CPUs, both intra-tile and inter-tile locality is not exploited on GPUs since L1/L2 caches are much smaller. The effects of row or column assignments of tiles on GPU execution has not been tried before.

We present an example in Figure 6.1 that shows the execution time line of a wave-front parallelism to demonstrate the load imbalance caused by global barriers. Figure 6.1(a)

$$idle(d,p) = \begin{cases} p - |d| mod p, & \text{if } |d| > p \\ 0, & \text{otherwise} \end{cases}$$

$$n_{idle} = \sum_{d=1}^{t_x+t_y-1} idle(d,p)$$

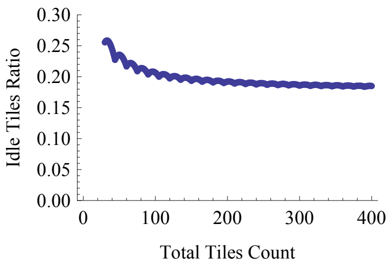


Figure 6.3: Number of idle tile execution slots on the left(a) and the ratio of idle slots to total tiles on the right(b).

shows a 2D data partitioned into 8 tiles in each dimension and executed on 4 SMs using global barriers. Tiles in a diagonal are shaded with same color and numbered with the SM ids that are assigned to execute them. The scheduling is assumed to be the default round-robin TB scheduler believed to be used by NVIDIA GPUs. Figure 6.1(b) shows the time line for each SM where shaded rectangles correspond to busy time and dark-black slots correspond to idleness. Due to varying length of tile diagonals, some SMs are left idle for nearly all the diagonals. Figure 6.1(c) shows the ideal execution if global barriers are removed and SMs synchronize directly with each other to enforce dependencies. As we can see, the idleness due to inter-tile load imbalance is completely removed. Theoretically the saved time can be as high as 25% based on the total tile and SM count.

The number of idle tile execution slots can be expressed with the equation given in Figure 6.3(a), where p is the number of processors and d is the diagonal iteration index. Using this equation, we have plotted the ratio of idle tile slots to the total tile count, which increases as smaller tiles are being used. As shown in Figure 6.3(b), even using smaller tiles to increase total tile count does not help this imbalance to be eliminated.

In this chapter we address the above issues by developing PeerWave, a new GPU parallelization scheme for nested loops with data dependencies. We develop a decentralized synchronization scheme called *PeerSM* which eliminates global barriers by utilizing efficient SM-to-SM communication. PeerSM considerably reduces load imbalance by decreasing redundant idle waits and letting SMs start executing tiles independently as soon as their dependencies are satisfied. Our approach also improves cache locality across consecutive tiles by *Row-to-SM* assignment. *Flexible Hyper-Tiles* further reduces SM under utilization by allowing fine-grained control over synchronization intervals while keeping intra-tile utilization at maximum.

Our study has the following contributions:

- We design a fast SM-to-SM peer synchronization mechanism (*PeerSM*) to eliminate load imbalance between SMs.
- We implement a row allocation scheme (*Row-to-SM*) which assigns consecutive tiles to same SM via programming of thread blocks (TBs).
- We extend hyperplane tile transformation (hyper-tile) (*Flexible Hyper-Tiles*) to further decrease idleness via adjustable synchronization intervals .
- We evaluate our scheme on the state of the art NVIDIA K40c GPU for 6 different applications and achieve speedup of up to 2X compared to approach that uses global barriers and hyper-tiles.

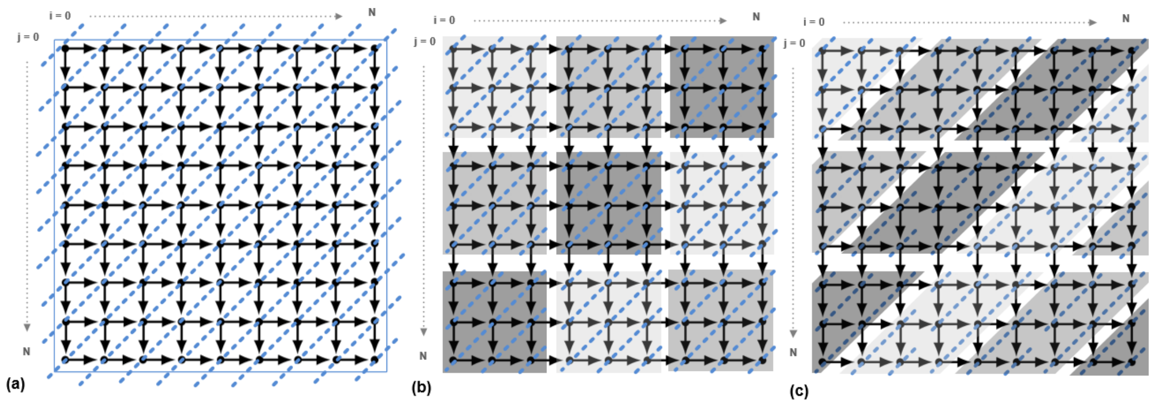


Figure 6.4: On the left (a), basic wavefront parallelism where dashed lines corresponds to 'waves'. On the center (b), tiling enables a second level of parallelism. On the right (c), hyper-plane transformation is shown.

- We develop a generic API for CUDA to support a wide class of applications with nested loops and regular dependencies.

The rest of this chapter is organized as follows0. Next Section gives background on wavefront parallelism. In Section 3 we describe our scheme in detail and in Sections 4&5 we elaborate on the details, optimizations, our run-time and API. In Section 6 we present the evaluation of our scheme.

6.2 Wavefront Parallelism

The wavefront technique exploits parallelism in nested loops that contain regular data dependencies across loop iterations. The main idea is to *skew* the iteration space and re-order loops to obtain a DOALL parallelism in one of the loops [90]. An example loop nest that processes a 2D array with cross iteration dependencies is given in Algorithm 2. The original loop nest is serial, and by skewing the inner j loop, DOALL parallelism is obtained across the iterations of the outer i loop. To utilize this parallelism, the two loops are swapped and their boundaries are adjusted accordingly. Figure 6.4(a) illustrates the iterations of this skewed&swapped loop nest. The dependencies between elements are shown via arrows and *diagonal*s of elements are marked with dashed lines. Each diagonal corresponds to an iteration of the outer j loop that can be executed in parallel. An important property of such parallelism is its non-uniformity, where the parallelism first increases and then decreases along the outer diagonal iteration space.

6.2.1 Tiling

In wavefront parallelism the control over granularity and locality is achieved by partitioning the iteration space into $t \times t$ square regions called *tiles* (see Figure 6.4(b)).

Typically, tiling decreases inter-processor communication and also increases cache locality by grouping elements. Similar to the element-wise diagonal pattern, streaming multiprocessors (SMs) can be used to process diagonals of tiles in parallel (*inter-tile* parallelism) while diagonal elements in each tile can also be processed in parallel (*intra-tile* parallelism) by SIMD compute units (CUDA cores).

6.2.2 Wavefront Execution on GP-GPU

Wavefront parallelism in the literature has drawn attention under various contexts like dynamic programming, stencil computations and time based iterations (i.e. Gauss-Seidel or Jacobi) and we discuss the GPU related sub-set of the literature in this section.

Algorithm 2 Wavefront execution via loop skewing

```
// Original loop nest
for $i$=2 to $n$ do
    for $j$=2 to $m$ do
        $A[i,j] = (A[i-1,j] + A[i,j-1])/2$
    end for
end for

// Skewed
for $i$=2 to $n$ do
    for $j$=$i+2$ to $i+m$ do
        $A[i,j-i] = (A[i-1,j-i] + A[i,j-i-1])/2$
    end for
end for

// Skewed & Swapped
for $j$=4 to $n+m$ do
    for $i$=MAX$(2, j-m+2)$ to MIN$(n, j-2)$ do
        $A[i,j-i] = (A[i-1,j-i] + A[i,j-i-1])/2$
    end for
end for
```

Datta et.al [22] are among the first who study the effectiveness of stencil computation on GPUs and compare the performance with multi-core architectures. They focus on data allocation and bandwidth optimizations and present a run-time that auto-tunes architecture specific parameters. [93] is another early study which employs tiling and successive placement of data on the memory for coalesced access on GPUS. They implement in-kernel global barriers using device memory, to prevent going back to CPU for ensuring diagonal dependencies and compare the performance with host-based *cudaDeviceSync()* function. Yan et al. [97] employs synchronization between different stages of parallel prefix sum to remove global barriers between reduce and scan phases.

6.2.3 Tiling on GPUs

The two level parallelism via tile partitioning is also commonly exploited by GPU based techniques whose evaluation has been reported in several studies [25, 24, 93]. In these approaches, a thread block (TB) is created for each tiles in a diagonal and these tiles are processed in parallel by the multiple streaming multiprocessors (SMs) of the GPU (*inter-tile* parallelism). Iterations along diagonals within a tile are also executed in parallel by the SIMD computation units (i.e. CUDA cores) inside an SM (*intra-tile* parallelism). Similar to CPU based approaches, global barriers are required for both tile-level (i.e. *inter-tile*) and element-level(i.e. *intra-tile*) diagonals in order to enforce data dependencies. However, the use of global barriers leads to low utilization of processing units during inter-tile parallel execution. Processors (i.e. SMs) wait for the longest running task (i.e., processing of a tile) in a diagonal and are forced to stay idle, even though new tiles are ready to execute as their dependencies have been satisfied.

6.2.4 Hyperplane Transformation

An inherent drawback of using square/rectangular tiles in wavefront parallelism is the non-uniform diagonal sizes encountered during the execution of a tile. Hyperplane partitioning [41] converts square tiles into polyhedral quadrilaterals via a series of affine transformations. As shown in Figure 6.4(c), resulting tiles are composed of equal-sized diagonals with a total count of n, for n x n tiles whereas the square tiles end up in $2n - 1$ diagonal iterations. Unlike square tiles, which limit maximum intra-tile parallelism to the minimum of tile width and height, hyper-plane partitioning allows arbitrary tile widths while keeping the diagonal size equal to the tile height. It may also be observed that the data is passed directly from the elements of a diagonal to be executed in the next iteration, hence increasing cache hits in the SM even for small cache sizes. If we allocate the next tile in the same row to the same SM, there will be cache hits for the bordering data. A technique similar to hyperplane partitioning for GPUs was proposed by [25] via compiler transformations to improve intra-tile parallelism.

6.3 PeerWave

We propose PeerWave to employ direct SM-to-SM synchronization instead of global synchronization for wavefront parallelism to address the above drawbacks. As shown in Figure 6.5, PeerWave assigns a row of tiles to an SM to improve inter-tile locality and it uses decentralized synchronization to avoid barriers and load imbalance, where synchronization points are appropriately placed to maximize performance. Light triangles on the bottom-right of each tile correspond to the points where an SM writes to neighboring SM's flags after it finishes processing the tile; dark triangles on top-left of the tiles are for the places where a dependent SM reads these flags to continue execution. The rest of this section describes PeerWave in detail.

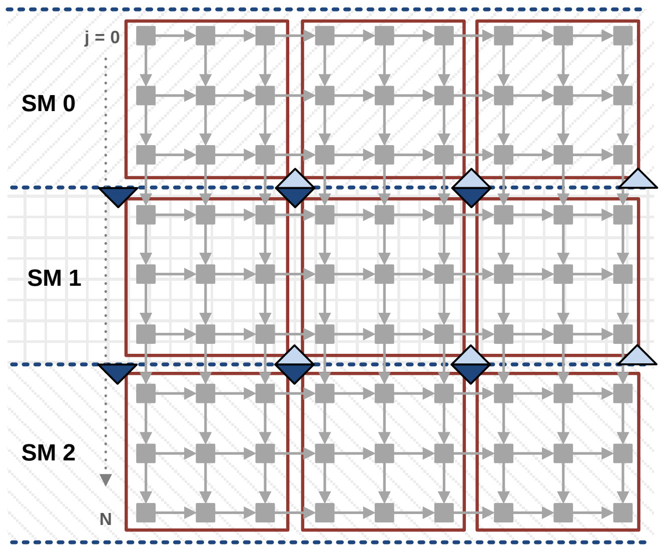


Figure 6.5: PeerWave: Partitioning of 2D iteration space into rows of tiles and assignment of rows to SMs. Triangles correspond to synchronization points, (light=write, dark=read).

6.3.1 Row-to-SM Assignment

PeerWave assigns a **tile-row**, a complete horizontal row of tiles in the 2D iteration space, to a single SM to exploit locality across tiles. In case there are more rows than SMs, the rows are distributed among the SMs in a round-robin fashion. Since CUDA does not provide any native mechanisms to manually assign thread blocks (TBs) to SMs, a TB is created for each row and *the TB-to-SM affinity* is achieved by limiting total number of TBs in a kernel launch to total number of SMs.

Row-to-SM assignment exploits locality by allowing elements close the horizontal border between tiles to remain in caches or local memory until they are accessed again. When combined with hyper-tiling, *tile-row*s further increase reuse across multiple columns with more hits on the last level cache. Moreover, this approach allows uninterrupted processing of elements across different tiles and enables two further optimizations, use of shared memory diagonal buffers and hybrid row-diagonal data storage (described in Subsection 6.3.4).

Algorithm 3 PeerWave main loop

```
1: for every row $r$ assigned to SM $i$ do
2:     for every tile $t$ in row $r$ do
3:         wait until $flag[t, i] = 1$
4:         $process\_tile(t, r, i)$                                ▷ SIMD
5:         $flag[t, i + 1] \leftarrow 1$
6:     end for
7: end for
```

Algorithm 3 shows how each SM processes its tile-rows. The outer loop iterates over the rows assigned to an SM in a round-robin fashion whereas the inner loop processes the tiles in a given row in sequence. Each SM processes its tiles one at a time while performing the major computation, i.e. processing of a tile, in parallel using SIMD computation units. In the inner loop, right before and after processing of a tile, the SM communicates with its neighboring SMs, called *Peer-SM*s, to enforce the dependencies between the tiles in adjacent rows.

6.3.2 Peer-SM Synchronization

Peer-SM synchronization is the key component of our method that replaces the global barrier synchronization between diagonal tiles with distributed synchronization among neighboring tiles. The decentralized nature of Peer-SM allows SMs within a GPU to work independently through synchronization points similar to CPU multithreading. Each SM processes the tiles in its row in dependence order and communicates with the SMs assigned to preceding and succeeding rows before and after processing each tile. By avoiding the redundant waits on global barriers and relying only on direct communication between neighboring SMs, Peer-SM improves load balance.

We implement inter-SM communication by using one-way *flags* located in the GPU global memory. Each SM owns a dedicated array of these flags corresponding to the tiles assigned to it. The one-way communication pattern is realized in PeerWave by having an SM read flags only from the prior row and write only to the flags for the succeeding row. Unlike the previous global-barrier based approaches, this one-way communication removes the need for atomic operations and locks, and scales well with increasing number of SMs.

When an SM is ready to process a tile, it continuously polls the corresponding flag, and begins processing once the flag is set to 1. After the SM has finished processing the tile, it sets the flag for the tile in the next SM's row that is waiting on the current tile, and then becomes ready to process the next tile in the row. When there are more rows than the number of SMs, and multiple rows are assigned to the SM, the SM resets all synchronization flags and starts processing next row immediately after the last tile of the current row has been completed. Per-SM flag arrays are created only once and then reused

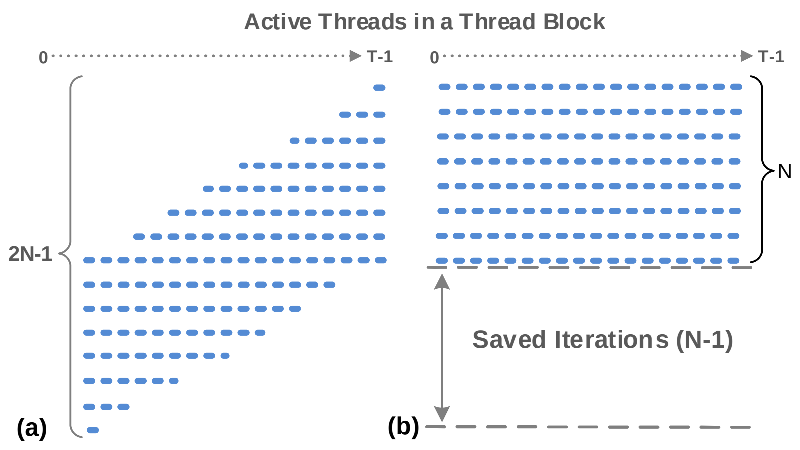


Figure 6.6: Intra-tile thread utilization is demonstrated for rectangular tiles on the left(a) and for hyperplane transformed tiles on the right(b)

as the SM moves from one row to its next assigned row. Implementation details of Peer-SM synchronization are given in Section 6.5.

6.3.3 Flexible Hyper-Tiles

While tile-row assignment and peer-SM synchronization improve inter-SM load balancing with better data locality, further speedup can be achieved during intra-SM computation. Rectangular tiles result in shorter diagonals at the beginning and ending iterations within a tile and this partitioning leaves many threads (hence CUDA cores) in a TB idle. Hyperplane tile transformation improves SM utilization by maintaining a uniform diagonal size across the entire tile and thus maximizing the throughput. The number of iterations needed for intra-tile computation for square and hyperplane tiles are shown in Figure 6.6(a) and (b), respectively. The hyperplane tile partitioning decreases total number iterations from $2n - 1$ to n in comparison to rectangular tiles. Thus, PeerWave uses hyperplane tiles (*hyper-tiles*) in addition to row-SM allocation and peer-SM synchronization. Each element in a diagonal is mapped to a thread in the TB and we iterate over diagonals until the end of the tile where inter-SM synchronization point is located. Figure 6.7(a) shows transformed tiles along with the synchronization points. Skewed indices change total number of tiles and shift the locations of inter-SM synchronization points. It should be noted that while [25] uses compiler level hyperplane loop transformations, unlike PeerWave, it still uses global barriers between diagonals.

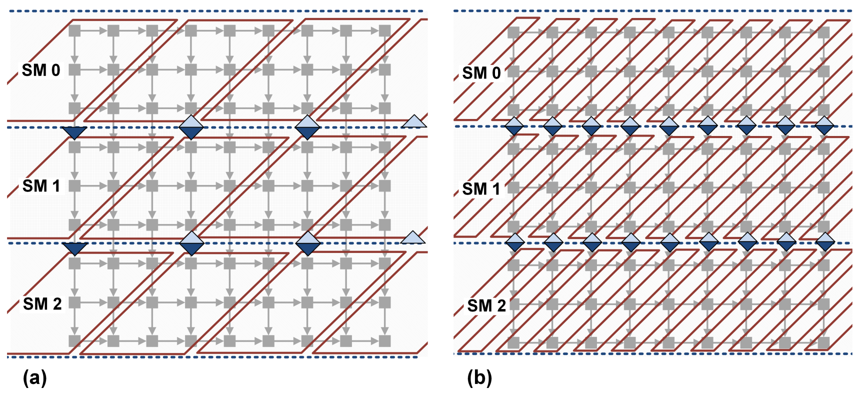


Figure 6.7: Hyper-Tiles on the left (a) and Flexible HT with variable synchronization intervals on the right (b)

We present *Flexible Hyper-Tiles* (FHT), an improved hyperplane transformation technique which allows us to *find* and *utilize* the optimum tile widths without changing the tile heights. Unlike the naive hyper-tiles approach with equal dimensions, FHT enables fine granular tile width adjustment for better control over the frequency of synchronization while maximizing intra-SM utilization without requiring tile heights to be changed. Smaller synchronization intervals decrease the time SMs must wait before processing their rows during the initial and final phases of execution when the parallelism is limited. Figure 6.7(b) shows an example of tile resizing where SM 1 can start processing tile (i=1,j=0) as soon as the diagonal corresponding to the element (i=0,j=2) has been processed by SM 0.

Different from prior studies, *Flexible Hyper-Tiles* uniquely models the optimal synchronization intervals (i.e. tile width) and finds the best trade-off between decreased load imbalance and increased synchronization cost. Optimal intervals are application and device specific and we develop an analytical model that outputs best tile width for given application/device characteristics. PeerWave runtime auto-adjusts the synchronization points based on the optimal tile width obtained from this model. Derivation of the model is presented in Section 6.4.

6.3.4 Optimizations

We further improve performance of PeerWave by employing the following optimizations.

Shared memory diagonal buffers: An important feature of PeerWave is its uninterrupted sequential processing of tiles within a row on the same SM. This enables efficient use of SM shared memory between consecutive iterations of tiles to dramatically reduce global memory accesses. Since processing a diagonal requires access to elements from preceding and succeeding diagonals, corresponding data can be stored in a sliding window of *diagonal buffers* that eliminates repetitive accesses to global memory as the diagonals are iterated upon by the same SM. When a diagonal is no longer to be accessed, it is written back to the device memory. The size of the sliding window is dependent on the length of the longest dependency vector in the transformed iteration space. FHT greatly increases the efficiency of diagonal buffers by combining accesses to partial diagonals, which are located towards the edges of neighboring tiles, into consecutively accessed ‘full’ diagonals.

Hybrid row/diagonal-major data layout: The diagonal access pattern during intra-tile processing requires CUDA cores inside SM to access non-consequent locations in the memory. This pattern results in highly un-coalesced memory accesses and causes significant slow down due to excessive number of memory transactions. Diagonal-major data representation [16] improves coalescing by storing elements in the same diagonal in consecutive memory locations. However, this approach does not preserve inter-tile locality, and causes the border elements to be redundantly being read and written as the processing of diagonals in one tile finishes and the next one begins. To prevent this behavior, we use a hybrid approach where the tiles in a row are continuously placed in the global memory in row-major order while the elements in the same diagonal are placed in diagonal-major order. The hybrid row-diagonal major data layout does not cause any memory space to be wasted since all diagonals in hyper-tiles have the same length.

The change in storage format requires the host data to be initialized properly. This can be done in two ways: (1) The data is converted to/from hybrid layout before and after GPU memory transfers. (2) When initializing the data from I/O resources or dynamically, the index transformation can be performed on the fly. The former method requires additional host-to-host data copy operations, therefore can be costly. The latter solution eliminates redundant copies and requires only a few additional instructions per index, whose cost are already hidden by much longer I/O instructions. We provide a simple

API, whose details are given in the next section, to translate given x, y coordinates to proper data indices.

6.4 Finding Optimal Tile Width

Next we develop an analytical model for finding the optimal tile width that will be utilized by Flexible Hyper-Tiles(FHT) to minimize the total execution time, which is a function of diagonal execution time and peer-SM synchronization cost. We compute the total execution time by identifying the critical path, i.e. the longest path through the execution. The output of our model is the optimal tile width for use in the runtime developed later.

Symbol	**Definition**
P	Number of SMs
W / H	2D input data width / height
w_t / h_t	Tile width / height
d	Diagonal execution time (measured)
τ_s	Peer synchronization cost (measured)
τ_e	Tile execution time

Figure 6.8: The notation used by our model

The notation used in the derivation of the model is given in Table 6.1. We assume that the input is a 2D data matrix with height H and width W. The matrix is partitioned into H/h_t tile rows and P SMs are assigned to these rows in a round-robin fashion. The constants τ_s and d represent peer synchronization cost and the processing time of each intra-tile diagonal, respectively. These parameters are obtained using the *findOptimumTileWidth()* function which is provided by our API and performs an offline run on a small fraction of the input data to measure these parameters. Tile execution time τ_e is derived as a function of d, where:

$$\tau_e = w_t d \tag{6.1}$$

We define B to represent the time during which the processing of a row assigned to the first SM is blocked by the execution of the previous row assigned to the same SM. This case happens when there are fewer SMs than the number of tile columns. B is defined as follows:

$$B = \frac{W + h_t}{w_t}(\tau_e + \tau_s) - P(\tau_e + \tau_s) \tag{6.2}$$

The critical path consists of processing the first two tiles of every row and all the tiles in the last row. The point where blocking (B) occurs, marked by a dashed arrow, is where the first SM finishes processing the last tile in a row and starts the first tile in its next assigned row. Hyperplane partitioning introduces an extra tile at the beginning of each row as there is no tile on which the first tile in a row is dependent.

Using equations 6.1 and 6.2, we can formulate the critical path, hence *the total execution time*, as follows:

$$\begin{aligned} T_{total} = &\frac{H}{h_t}(\frac{h_t}{w_t} + 1)(\tau_e + \tau_s) + B(\frac{H}{h_t P} - 1) \\ &+ (\frac{W}{w_t} - 1)(\tau_e + \tau_s) \end{aligned} \tag{6.3}$$

According to Equation 6.1, enlarging h_t is always beneficial as this will lead to better intra-tile utilization and tile execution time will not increase because τ_e is invariant w.r.t h_t. On the other hand, any values w_t, s.t. $h_t \geq w_t$, will not change tile execution time. However, by analyzing equation 6.3 (details omitted), it can be shown that total execution time decreases as h_t increases. Thus, using maximum h_t (i.e., maximum number of threads per intra-tile diagonal) for hyper-tiles will always give best performance. With fixed h_t, total execution time becomes a function of w_t, the synchronization interval.

Equation 6.3 can be transformed into the form aw_t+b/w_t+c, where the optimal w_t that minimizes the equation is given by $\sqrt{b/a}$. Hence, the optimal synchronization interval based on the model is given by:

$$w_t = \sqrt{\frac{\tau_s(HW + Hh_t + Hh_tP - h_t^2P)}{dh_tP(P-1)}} \tag{6.4}$$

We use equation 6.4 to find the optimal synchronization interval without the need for running the application on the entire input data. We validate the above model in the Evaluation Section.

6.5 Runtime & API

We develop a runtime to handle inter-SM synchronization and an easy to use API for users to implement wavefront applications.

PeerWave runtime employs a multi-core like execution scheme with each SM acting as an independent SIMD-capable processor. It uses persistent TBs each of which runs until all corresponding tiles are processed. The run-time kernel is launched with TB count set equal to the total number of physical SMs in the GPU. The default NVIDIA TB scheduler assigns each SM in a round robin fashion, and the number of TBs per SM is determined by the resource requirements of the kernel. For this reason, in our runtime, we use large enough TBs (hence tiles) so that HW scheduler only assigns one TB per SM. A major benefit of having persistent TBs is that SMs can preserve execution state in their local memory (e.g., shared memory in CUDA) and also retain application data across iterations.

```
struct user_data{
float* data;
long w, h;
int** dependencyVectors;
}
void transformIndex(int& x, int& y, int& index, int tile_w,
int tile_h, user_data udata);
long findOptimumTileWidth(user_data udata, int tile_h,
int nSM,float trainingRatio);
__global__ void PeerWaveKernel(user_data udata, int tile_w,
int tile_h, int nSM, int* flags);
```

Listing 6.1: PeerWave kernel header and other helper functions

PeerWave runtime consists of a user-modifiable data structure, preparation functions, a main runtime kernel, and peer-SM communication functions called by this kernel. The headers for these components are given in Listing 6.1 and the details are elaborated in the rest of this section.

User data: *user_data* struct contains device memory pointer to the user data, dimensions of the 2D data space and the *dependencyVectors* for the stencil computation. Users are responsible for initializing the device memory via *cudaMalloc()* calls and copy initial data from host memory to GPU via *cudaMemcpy()*. *user_data* struct can be modified to add application specific pointers and parameters.

Preparation functions: We present users two optional pre-runtime functions to increase the usability of our framework. The first one is the index transformation function, *transformIndex()*, to allow users a convenient two-way index translation from/to 2D x and y coordinates to/from a data *index*. The function uses the application parameters given in *user_data* and performs the transformation to support our row/diagonal major hybrid data layout. The second function is *findOptimumTileWidth()* and it is used to find optimum tile width via the provided model. The function finds applicationarchitecture specific parameters required by the analytical model by performing an offline training run on a fraction of the input data specified by the *trainingRatio* parameter.

***PeerWave* Kernel:** PeerWave runtime, hence the user algorithm, is initiated by launching this kernel with the following parameters:

- *TB Size & Grid Size*: These two parameters are supplied to CUDA runtime while launching the kernel. The grid size (i.e., the number of TBs) is set equal to the number of SMs (nSM) whereas TB Size is application and architecture specific and the programmer need to determine the optimal size according to resource requirements of the wavefront application.
- nSM: GPU specific value denoting the total number of SMs.
- *Flags*: The communication flags which need to be pre-allocated on GPU memory via provided *init()* function. The size of this array is dependent on the number of SMs and the number of tiles in a row.

- *tile_h*: Height of the tiles. Since PeerWave maps each element in the vertical axis of a tile to a thread in the TB, *tile height* should be equal to the TB Size (i.e. number of threads) specified during kernel launch.
- *tile_w*: Width of the tiles (i.e. synchronization interval). As discussed in Section 3 and 4, tile width determines the synchronization interval and it is an important parameter that directly affects the performance. Users can either manually derive the optimal tile width using Equation 6.4 derived in Section 4 or use *transformIndex()* as described above.

```
flagRead(){
if (threadIdx.x==0){ // Single thread only
while(flags[(nTiles_x)*smID+j]==0)  // check flag until set
cosineSleep(); // dummy computation
flags[(nTiles_x)*smID+j]=0; // reset for future use
}
syncthreads(); // sync with other threads
}

flagWrite(){
if (threadIdx.x==0){ // Single thread only
int nextSM = (smID+1)\%nSM;
flags[(nTiles_x)*nextSM+j]=1;// Set flags for next row.
}
syncthreads(); // sync with other threads
}
```

Listing 6.2: Implementation of flag read (a) and write (b) operations

Once the *PeerWaveKernel* is launched, PeerWave main loop given in 3 is executed by each SM until all the data is processed. The main loop involves read of synchronization *flags*, processing of the tile and writing back to the *flags*.

Flag Read: PeerWave relies on efficient synchronization of SMs via one-way communication flags. Listing 6.2(a) shows the code used to ensure that the corresponding flag is set (i.e., dependencies are met) before every SM starts processing a new tile. Since NVIDIA does not provide an efficient means of inter-SM communication, we keep polling the flag corresponding to the executing SM (smID) and the current column (j) in the tile row until its value is 1. We decrease potential memory contention by idling the SM for a small short period of time using *cosineSleep()* function derived from repetitive calls to cosine function. When the flag is detected as set, we reset it for future use and move on to tile processing.

Tile Processing: Each SM processes the elements in the same diagonal in parallel using the threads in the TB. Each row of elements is mapped to a single thread and the number of columns in the tile correspond to the number of diagonals. After processing of each diagonal, threads inside the SM synchronize using *cudaThreadSync()* method, which is an efficient intra-SM HW barrier implementation provided by CUDA.

Flag Write: Once a tile is processed, SMs need to set the flag for the next SM (smID+1) in the same column (j) as well as the tile in the next column(j+1) in the same row, as shown in Listing 6.2(b). Border conditions are omitted for simplicity. The overhead of both flag read/write operations are measured to be minimal and reported in the evaluation section.

Element Processing: PeerWave enables users to implement their core computation by overriding the *compute_element()* function as given in Listing 6.3. The Listing shows an example overriding of *compute_element()* function to implement the core computation of Smith-Waterman algorithm. During tile processing, runtime performs hyper-plane transformations on border elements of each tile and calls user function with the base tile index as well as the element's global x/y coordinates. *u_data.seq1* and *u_data.seq2* are the two input data sequences added to the *user_data* struct and the *u_data.data* array corresponds to the intermediate 2D computation matrix used while calculating the similarity between the two sequences. *compute_element()* function is run in SIMD fashion where all threads in the TB are called with corresponding x and y coordinates of the elements that they are mapped to.

```
__device__ void compute_element(
const struct user_data u_data,
int tileRow, int tileColumn,
int tile_w, int tile_h,
int x, int y, int index, int nTiles){
int upleft,left,up;
if (u_data.seq2[y] != u_data.seq1[x])
upleft = u_data.data[index-2*tile_height-1]-1;
else
upleft = u_data.data[index-2*tile_height-1]+2;
left = u_data.data[index+tile_height]-1;
up = u_data.data[index-tile_height-1]-1;

int max = MAX(0,MAX(upleft, MAX(left, up)));
udata.data[index] = max;}
```

Listing 6.3: An example override of *compute_element()* function.

6.6 Evaluation

In this section we first describe the details of our experimental setup and then present the experimental results.

Architecture We evaluate PeerWave on NVIDIA's Tesla K40c series GPU attached to an AMD Opteron 6100 based system. The GPU has 15 SMX units each having 192 CUDA cores accessing 12GB of global DDR3 memory. K40c has a shared L2 cache size of 1.5MB and shared memory of each SMX is configured to use 16KB/48KB L1 Cache/Shared with global memory requests are not set to be cached on L1 (Kepler's default setting).

Methodology In our evaluation, we only measure GPU kernel computation times only because our proposed technique focuses on kernel execution to improve overall execution. For all experiments, we use hybrid row/diagonal major data layout with on-the-fly index transformation as described in Section 6.3. Since data initialization and transfer times are observed to be similar w/ and w/o the hybrid layout, we exclude all host related operations (initialization and transfers) from our experiments to demonstrate the efficiency of our technique in more detail. Overlapping of data transfers and kernel computations for further speedup is orthogonal to our proposed method and not covered in this study.

In all of our experiments, we fix the data size to the highest amount that fits in GPU's global memory, and set the tile height and TB size to 1024 (unless specified otherwise in the experiment). For optimal performance, we implement global barriers by using NVIDIA Kepler's dynamic parallelism that allows in-GPU *cudaDeviceSync()* calls. We compare the following wavefront execution methods:

1. *Global barriers*: The baseline global synchronization approach using square tiles.
2. *PeerWave*: Basic implementation of our proposed peer-SM synchronization based technique using square tiles.
3. *Global barriers w/ HTiles*: Global synchronization approach using hyperplane transformed tiles with equal dimensions.
4. *PeerWave w/ HTiles*: Improved version of PeerWave using hyper-tiles with equal tile dimensions.
5. *PeerWave w/ Flexible HTiles*: Optimized version of PeerWave hyper-tiles with flexible non-equal tile dimensions.

6.6.1 Execution Time: Equal Tile Dimensions

We first evaluate the performance of the first four techniques all of which use equal tile dimensions. Figure 6.9 shows execution times of each application run for varying tile sizes. TB size is set as 1024, all 15 SMs in the GPU are used and tile width and height are varied together between 256 and 1024. The first two bars correspond to rectangular (square) tiles and the last two correspond to hyper-tiles. The second and fourth bars represent improvements due to PeerWave technique.

An immediate observation is the dramatic decrease in execution time with the use of hyper-tiles. This is due to the decrease in the number of intra-tile diagonal iterations from $2n-1$ to n. PeerWave further improves the performance in both hyper-tile and square tile cases due to increased inter-tile locality and decreased SM idle times. PeerWave shows

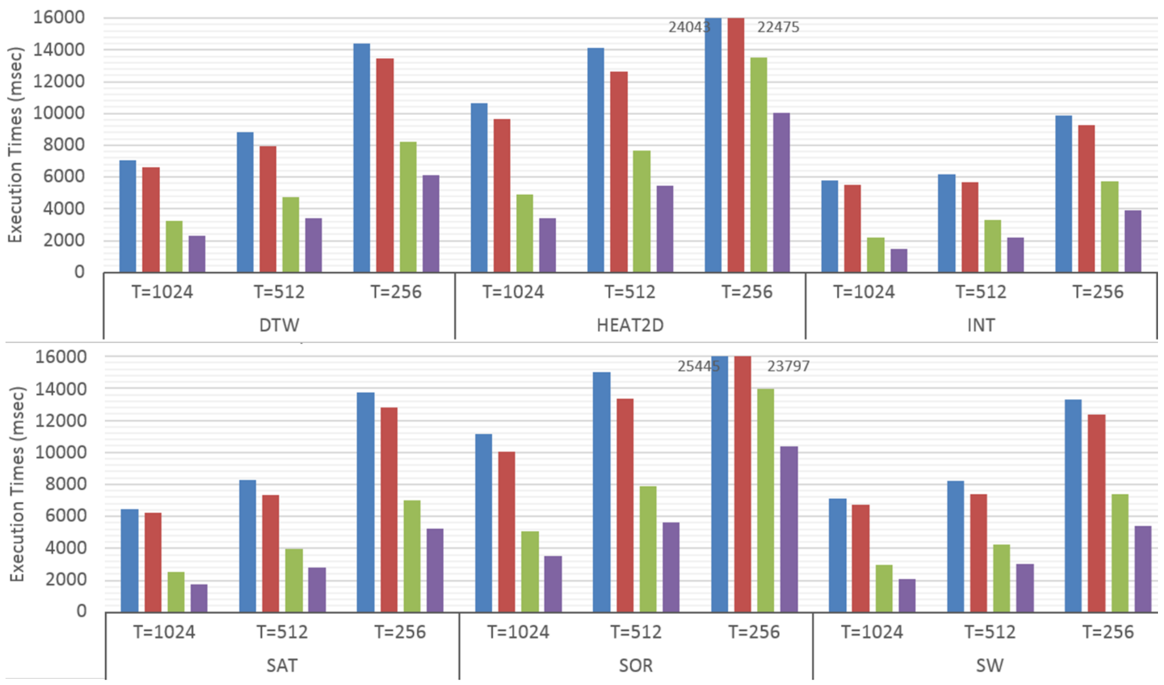


Figure 6.9: Kernel execution times for global barriers and PeerWave approaches with and without the use of hyper-tiles.

better performance when using hyper-tiles due to decreased memory contention. The details of the speedup will be analyzed throughout the remainder of this section.

The results also show that the decrease in tile sizes reduces the performance due to under-utilization of CUDA cores inside each SM. Since the maximum intra-tile parallelism is limited by the minimum dimension in the first four execution methods, tile size of 1024x1024 (hence TB of size 1024) gives maximum performance.

6.6.2 Execution Time: Flexible Hyper-Tiles

Flexible hyper-tiles (method 5) decreases total idle time by increasing number of synchronization points via smaller tile widths. Also, intra-tile utilization is kept maximum by using largest tile heights. The optimal tile width is determined by the provided model and then passed to PeerWave kernel as a parameter. In this subsection, we first evaluate the accuracy of our model in finding the best synchronization intervals, and then using the optimal values, we compare Flexible hyper-tiles (FHT) method with the best performing equal tile dimension methods 3 and 4.

Model Verification

To verify the accuracy of tile width optimization model presented in Section 6.4, we have compared the execution time estimated by Equation 6.1 with the measured execution time for the FHT method. The model parameters, d and τ_s are measured in an offline run using a 0.6% of the total data. This ratio corresponds to the initial corner region which spans 135 tiles out of 2025 in total. We fix the tile height to 1024 and vary the tile width from 2 to 1024. The comparison between the estimated and measured execution times is shown in Figure 6.10.

Overall, our model represents the execution pattern of PeerWave approach quite well. It can be seen that both experiment and simulation curves have a U shaped pattern which points to the trade-off between synchronization overhead and idleness reduction. The slight discrepancy between the simulation and experiment is due to variation of the values d and τ_s throughout the execution. Since we are using only an initial portion of the execution, the increased memory contention towards the middle region of the input matrix affects the blocking time B and hence the critical path. Although the execution times are slightly different, the synchronization interval estimation is accurate.

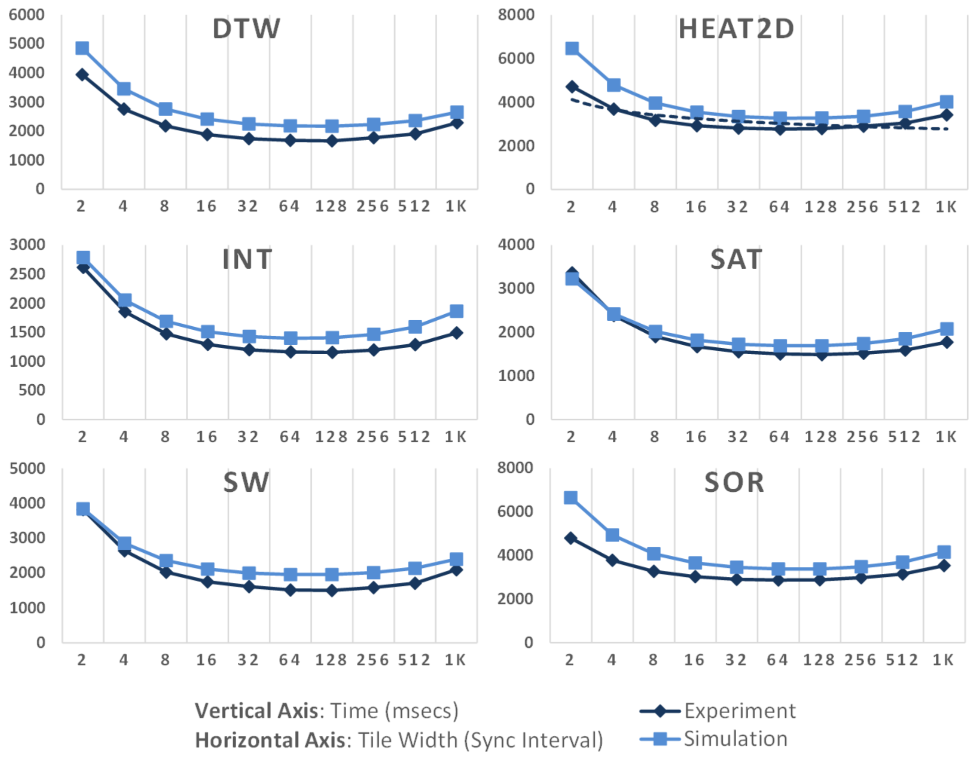


Figure 6.10: Comparison of numerical simulation and experimental results for Flexible hyper-tiles.

Flexible Hyper-Tiles (FHT)

To evaluate flexible hyper-tiles, we take method 3, global synchronization with hyper-tiles (GS+HT), as our baseline and compare its performance with methods 4 and 5, i.e. PeerWave w/ hyper-tiles (PW+HT) and PeerWave w/ Flexible hyper-tiles (PW+FHT). We fix both tile height and width to 1024 for GS+HT and PW+HT, as these give the best results. On the other hand, for PW+FHT, we have set tile height as 1024 and varied tile width between 64, 128, and 256, which are the most close-to-optimal values reported by our model as well as the experiments.

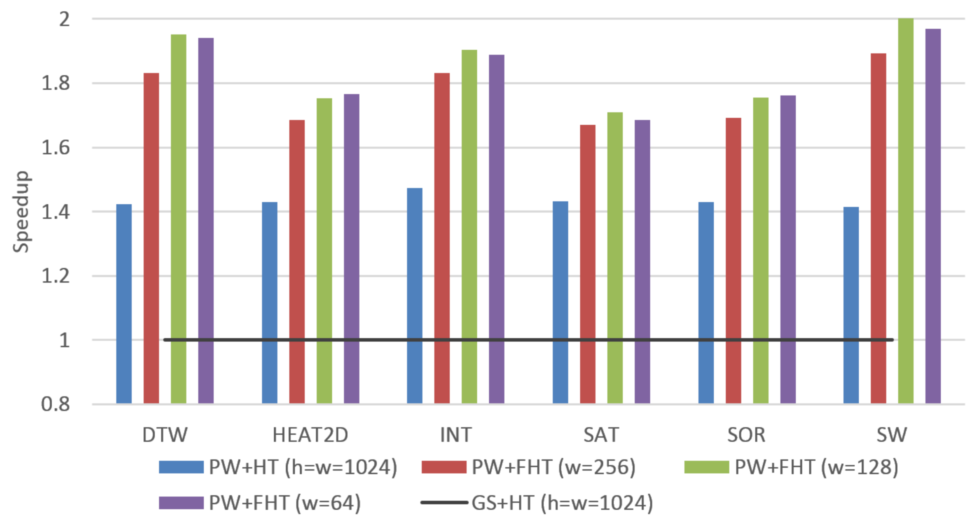


Figure 6.11: The speedup obtained by Flexible hyper-tiles method.

As the results given in Figure 6.11 show, flexible hyper-tiles approach improved upon our original PW+HT technique and increased the final speedup by up to 2x compared to the recent approach based on global barriers with hyper-tiles (GS+HT) [25]. When compared to the effects of using smaller tiles with equal dimensions shown in Figure 6.9, the results clearly show the benefits of changing synchronization interval without affecting utilization.

6.6.3 Idle Times

To further analyze the effects of peer-SM synchronization we measure the average time that an SM stays idle and compare it to the idle time spent on global barriers. As previously shown in the Figure 6.1 in Section 2, we classify the idle time into two categories: (a) *Corner*, the compulsory wait time spent during the beginning & end of the execution, where the parallelism is less than total number of SMs(15); and (b) *Intermediate*, where there are always more parallel tiles than SMs.

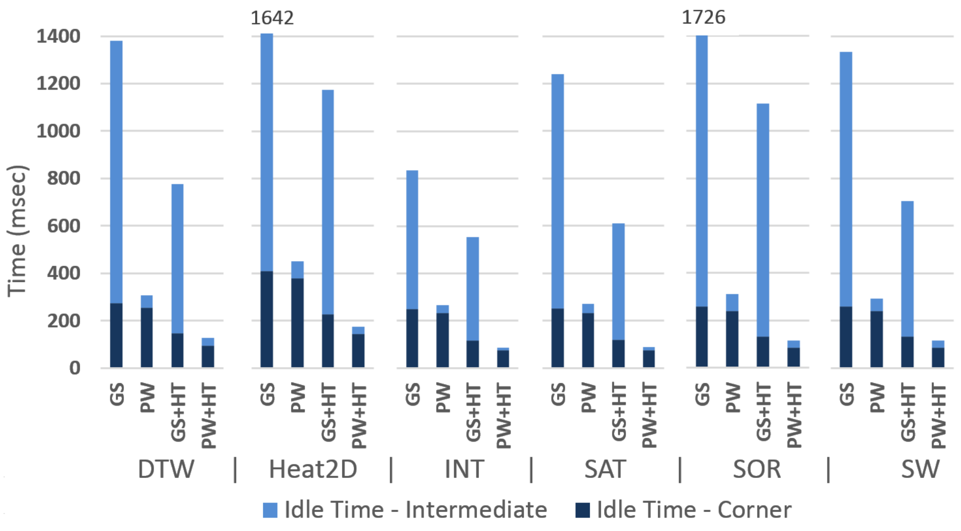


Figure 6.12: Breakdown of Per-SM average idle time between two phases of the execution.

The results in Figure 6.12 shows the break down between border (blue) and intermediate (orange) idle times for each application and the technique. PeerWave dramatically reduces the *intermediate* idle time both with and without hyper-tiles, whereas the *corner* idle times remain similar as expected, due to limited parallelism in those regions.

Reduced idle times cause more SMs to be active hence result in increased memory throughput as shown in Figure 6.14. PeerWave was able to improve memory bus utilization more in both cases with and without hyper-tiles. On the other hand, use of hyper-tiles dramatically reduces total memory transactions due to reduced number of intra-tile diagonal iterations and increased coalesced accesses, therefore less throughput is needed.

6.6.4 Tile Locality

Another important drawback of global barriers is overcome by PeerWave via increased inter-tile locality due to row-SM assignment. To observe this, we measure average

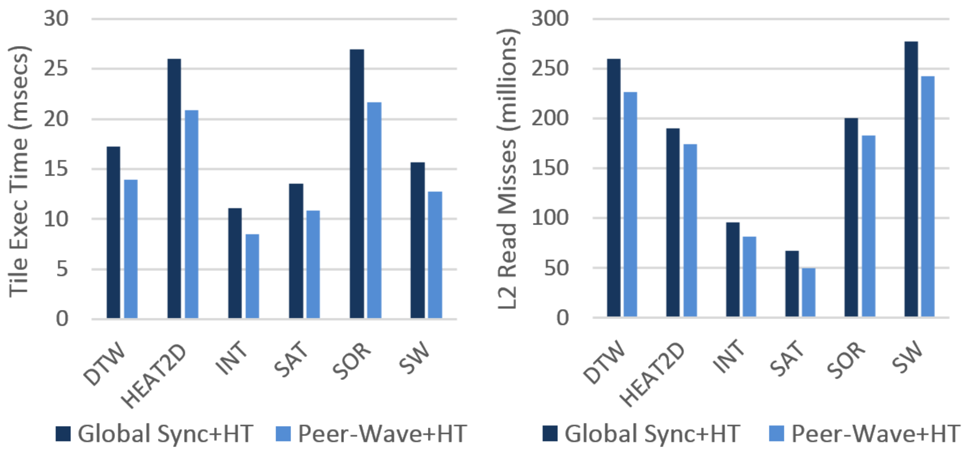


Figure 6.13: On the left (a) average per-tile execution times and on the right (b) total L2 read misses

time required to execute a single tile and give the results in Figure 6.13(a). Tile execution times are reduced between 15% and 25% compared to global barriers.

We have also profiled the total number of L2 read misses using NVIDIA's profiling utility, **nvprof**, and report the results in Figure 6.13(b) for 32-byte transactions. As expected, PeerWave reduces the misses in both cases. Although the improvement seems to be minor, the effect on execution time is considerable due to high cost of a global memory access.

6.6.5 Synchronization Cost

We measured the total time spent during *flagRead()*, *flagWrite()* and global barriers using the device-side clock64() CUDA function. The cost of updating global barrier mutexes in global-synchronization with HT (GS+HT) and the communication flags in PeerWave with HT (PW+HT) is given in milliseconds in Table 6.1. The values correspond to

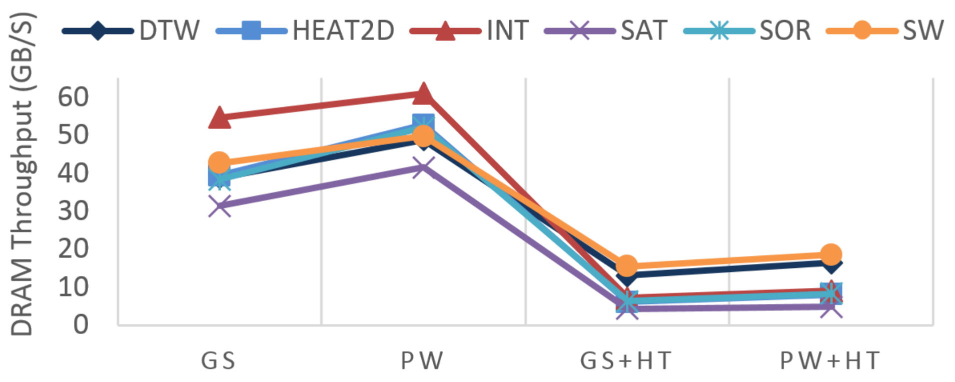


Figure 6.14: Total GPU global memory throughput.

the total time spent for synchronization . Although PeerWave improves the synchronization times by around 25%, the total sync cost in both cases is actually marginal and only corresponds to under 0.1% of the total execution times. Most of the speedup obtained by PeerWave is achieved via improved load balance and locality.

	DTW	HEAT2D	INT	SAT	SOR	SW
GS+HT	4.28	4.25	3.77	4.25	4.27	4.25
PW+HT	3.49	3.39	2.37	3.36	3.4	3.43

Table 6.1: SM synchronization times in milliseconds.

Chapter 7

Conclusions

Heterogeneous multiprocessor systems featuring hardware accelerators have been used for a wide range of applications yielding significant speedups compared to CPU only execution. They take advantage of special purpose hardware by providing a hybrid execution environment for application. Getting maximum benefit from such systems rely on high processor utilization and minimized data transfer & synchronization cost in both system and accelerator level. System-level utilization relies on distributing computation workload in an application across accelerators and CPUs and also overlapping computation and data-transfers. On the other hand accelerator-level utilization is achieved via efficient inter-SM synchronization and increased data locality.

In this thesis, our goal was to develop techniques to improve overall heterogeneous system throughput for scientific.

In Chapter 3, we have designed, implemented and evaluated a new scheduling and workload balancing algorithm, HDSS, for loops with independent or dependent iterations on heterogeneous multiprocessor systems. The algorithm runs in two phases. The first phase is dedicated for learning accurate computational weights for each processor and the second phase distributes remaining workload using computed weights. Our algorithm uniquely considers the impact of block sizes on performance on heterogeneous multiprocessor systems without an offline run or a prior knowledge of application or architecture specifications. Our experimental results showed that our algorithm gives best performance in all cases and provides improvements up to 219% when compared to its closest load balancing scheme on certain applications.

Then, we proposed CuMAS, a novel scheduling framework, to enable data-transfer aware execution of multiple CUDA applications in shared GPUs. CuMAS improves overall system utilization by capturing and re-organizing CUDA memory transfer and kernel execution calls, without requiring any changes to the application source code. CuMAS is evaluated on an NVIDIA K40c GPU, using a suite of 12 CUDA applications and it is shown to improve the total execution time by up to 44% when compared to best known automatic transfer/execution overlapping technique, In Chapter 4.

In Chapter 5, we developed a new, dynamic task-based execution scheme for GPGPU applications with regular and irregular data dependencies. Different from previous studies, our framework places both scheduler and worker threads inside the GPU in order to prevent the data transfer overhead between CPU and GPU and the centralized queue contention. We propose a new queue insertion policy, *Average Load with Local First*, optimized for SMs inside GPUs. Our experimental results on 64-core, NVIDIA Tesla 2050 GPU heterogeneous system showed that the proposed dynamic scheme clearly overcomes the bottleneck of centralized queue approaches. We have achieved speedups up to 15% for Heat2D 62% for BFS when compared to other approaches.

Lastly, we introduced PeerWave, a new parallelization scheme to efficiently run wavefront applications on GPUs, in Chapter 6. Our approach eliminates inter-tile global barriers and uses SM-to-SM direct communication instead. We improve inter-tile load balance and locality that greatly reduces SM idle times. We further improve PeerWave with Flexible Hyper-Tiles which allows fine-granular synchronization interval adjustment while keeping intra-tile utilization at its maximum. Our evaluations of PeerWave on NVIDIA K40c GPU showed speedup of up to 2x when compared to existing global barrier based approaches.

Some potential future research directions are as follows. First, this thesis only focused on single server platforms. It will be challenging to scale our workload partitioning solution to multiple levels of heterogeneity. Second, processor under-utilization problem due to global barriers exists also for applications other than wavefront. Generalization of peer-processor synchronization technique to remove fork/join type communication in data-parallel applications will be an important contribution. Third, our current automatic overlapping approach only works with data transfers across different applications. Implementing techniques to exploit transfer/execution overlapping within an application will enable support for larger applications that does not fit into the accelerator memory.

Bibliography

[1] A. Abrahamsen and D. Richards. The one dimensional heat equation. 2002.

[2] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for gpgpu spatial multitasking. In *HPCA'12*, pages 1–12. IEEE, 2012.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[5] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP '10*, pages 105–114. ACM, 2010.

[6] Z.K. Baker and V.K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. 2005.

[7] I. Banicescu and V. Velusamy. Performance of scheduling scientific applications with adaptive weighted factoring. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 791–801. IEEE, 2001.

[8] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with gpus. In *HPDC'12*, pages 97–108. ACM, 2012.

[9] Mehmet E Belviranli, Laxmi N Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):57, 2013.

[10] Raphael Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling independent tasks on multi-cores with gpu accelerators. *Concurrency and Computation: Practice and Experience*, 27(6):1625–1638, 2015.

[11] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT)*, pages 39–50. ACM, 2014.

[12] Pierre Boudier and Graham Sellers. Memory system on fusion apus - the benefits of zero copy. In *AMD fusion developer summit*. AMD, 2011.

[13] Peter Brucker. *Shop Scheduling Problems*, volume 3. Springer, 2007.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[15] 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 *IISWC'09*, pages 44–54. IEEE.

[16] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of the 25th ACM international conference on Supercomputing (ICS)*, page 13, 2011.

[17] 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.

[18] A.T. Chronopoulos, M. Benche, D. Grosu, and R. Andonie. A class of loop self-scheduling for heterogeneous clusters. In *cluster*, page 282. Published by the IEEE Computer Society, 2001.

[19] A.T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali. Distributed loop-scheduling schemes for heterogeneous computer systems. *Concurrency and Computation: Practice and Experience*, 18(7):771–785, 2006.

[20] F.M. Ciorba, T. Andronikos, I. Riakiotakis, A.T. Chronopoulos, and G. Papakonstantinou. Dynamic multi phase scheduling for heterogeneous clusters. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[21] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[22] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In

Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] B. De Ruijsscher, G.N. Gaydadjiev, J. Lichtenauer, and E. Hendriks. FPGA accelerator for real-time skin segmentation. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 93–97.

[24] 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.

[25] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP)*, pages 350–359. IEEE Computer Society, 2012.

[26] Toshio Endo and Satoshi Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[27] M.R. Fahey, S. Alam, T. Dunigan, J. Vetter, and P. Worley. Early evaluation of the Cray XD1. In *Proc. Cray User Group Meeting*, page 12, 2005.

[28] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data transfer matters for gpu computing. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ICPADS '13, pages 275–282, Washington, DC, USA, 2013. IEEE Computer Society.

[29] MD Galanis, A. Milidonis, G. Theodoridis, D. Soudris, and CE Goutis. A partitioning methodology for accelerating applications in hybrid reconfigurable platforms. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*, pages 247–252. IEEE Computer Society, 2005.

[30] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, 2008.

[31] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[32] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for gpus: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GP-GPU)*, pages 24–31. ACM, 2013.

[33] B. Harris, A.C. Jacob, J.M. Lancaster, J. Buhler, and R.D. Chamberlain. A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. In *Field Programmable*

Logic and Applications, 2007. FPL 2007. International Conference on, pages 765–769. IEEE, 2007.

[34] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial & Applied Mathematics*, 10(1):196–210, 1962.

[35] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J Ramanujam, and P Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS)*, pages 13–24, 2013.

[36] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. *Euro-Par 2010-Parallel Processing*, pages 235–246, 2010.

[37] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[38] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA'09*, pages 152–163. ACM, 2009.

[39] S.F. Hummel, J. Schmidt, RN Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328. ACM, 1996.

[40] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *PPoPP'12*, volume 47, pages 1–10. ACM, 2012.

[41] François Irigoin and Remi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[42] Thomas B Jablin, James A Jablin, Prakash Prabhu, Feng Liu, and David I August. Dynamically managed data for cpu-gpu architectures. In *CGO'12*, pages 165–174. ACM, 2012.

[43] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *GP-GPU'14*. ACM, 2014.

[44] Wayne Joubert, Rick Archibald, Mark Berrill, W Michael Brown, Markus Eisenbach, Ray Grout, Jeff Larkin, John Levesque, Bronson Messer, Matt Norman, et al. Accelerated application development: The ornl titan experience. *Computers & Electrical Engineering*, 46:123–138, 2015.

[45] Shinpei Kato, Karthik Lakshmanan, Ragunathan Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX ATC'11*, page 17, 2011.

[46] Dongjin Kim and Kyu-Ho Park. Tiled qr decomposition and its optimization on cpu and gpu computing system. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 744–753. IEEE, 2013.

[47] Eugene L Lawler, Jan Karel Lenstra, Alexander HG Rinnooy Kan, and David B Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993.

[48] Joseph YT Leung, Oliver Vornberger, and James D Witthoff. On some variants of the bandwidth minimization problem. *SIAM Journal on Computing*, 13(3):650–667, 1984.

[49] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[50] Lixia Liu and Zhiyuan Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 213–222. ACM, 2010.

[51] C.K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2010.

[52] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *HPCA'13*, pages 354–365. IEEE, 2013.

[53] Thibaut Lutz, Christian Fensch, and Murray Cole. Helium: a transparent inter-kernel optimizer for opencl. In *GP-GPU'15*, pages 70–80. ACM, 2015.

[54] N. Manjikian and T.S. Abdelrahman. Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 12(3):259–271, 2001.

[55] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing (ICS)*, pages 256–265. ACM, 2009.

[56] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, and Tadashi Watanabe. Overview of the k computer system. *Fujitsu Sci. Tech. J*, 48(3):302–309, 2012.

[57] Siddharth Mohanty and Murray Cole. Autotuning wavefront applications for multicore multi-gpu hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, page 1. ACM, 2014.

[58] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 24th ACM international conference on Supercomputing (ICS)*, pages 1–13, 2010.

[59] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[60] NVIDIA. Nvidia kepler gk110 architecture whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA Kepler-GK110-Architecture-Whitepaper.pdf.

[61] NVIDIA. Nvidia quadro dual copy engines. https://www.nvidia.com/docs/IO/40049/Dual_ copy_engines.pdf.

[62] Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, pages 284–291. IEEE, 2008.

[63] Sreepathi Pai, Matthew J Thazhuthaveetil, and R Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ASPLOS'13*, pages 407–418. ACM, 2013.

[64] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *ASPLOS'15*, pages 593–606. ACM, 2015.

[65] C.D. Polychronopoulos and D.J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Computers, IEEE Transactions on*, 100(12):1425–1439, 1987.

[66] Vignesh T Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC'11*, pages 217–228. ACM, 2011.

[67] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 233–248. ACM, 2011.

[68] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus. In *HPDC'13*, pages 179–190. ACM, 2013.

[69] Daniel Sanchez, David Lo, Richard M Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 22–32. IEEE, 2011.

[70] 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 (TPDS)*, 24(5):1009–1021, 2013.

[71] R. Scrofano, M.B. Gokhale, F. Trouw, and V.K. Prasanna. Accelerating molecular dynamics simulations with reconfigurable computers. *IEEE Transactions on Parallel and Distributed Systems*, pages 764–778, 2007.

[72] Dipanjan Sengupta, Anshuman Goswami, Karsten Schwan, and Krishna Pallavi. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *SC'14*, pages 513–524. IEEE, 2014.

[73] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. Gstream: a graph streaming processing method for large-scale graphs on gpus. In *PPoPP'15*, pages 253–254. ACM, 2015.

[74] SGI. Sgi altix 4700. *Delivering New Levels of Performance and Flexibility. Website: http://www. sgi. com/products/servers/altix/4000/index. html*, 2008.

[75] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 175–184. IEEE, 2009.

[76] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, pages 365–376. ACM, 2012.

[77] Jeff A Stuart and John D Owens. Efficient synchronization primitives for gpus. *arXiv preprint arXiv:1110.4623*, 2011.

[78] Jingweijia Tan and Xin Fu. Rise: improving the streaming processors reliability against soft errors in gpgpus. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 191–200. ACM, 2012.

[79] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *ISCA'14*, pages 193–204. IEEE, 2014.

[80] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 325–335, New York, NY, USA, 2006. ACM.

[81] J. L. Tripp, A. A. Hanson, M. Gokhale, and H. Mortveit. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 27–27, Nov 2005.

[82] J.L. Tripp, A.A. Hanson, M. Gokhale, and H. Mortveit. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 27. IEEE Computer Society, 2005.

[83] A.H.T. Tse, D.B. Thomas, KH Tsoi, and W. Luk. Dynamic scheduling monte-carlo framework for multi-accelerator heterogeneous clusters. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 233–240. IEEE.

[84] K.H. Tsoi and W. Luk. Axel: a heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124. ACM, 2010.

[85] K.H. Tsoi, A.H.T. Tse, P. Pietzuch, and W. Luk. Programming framework for clusters with heterogeneous accelerators. *ACM SIGARCH Computer Architecture News*, 38(4):53–59, 2011.

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

[87] B. van Werkhoven, J. Maassen, F.J. Seinstra, and H.E. Bal. Performance models for cpu-gpu data transfers. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, page 110. IEEE/ACM, 2014.

[88] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, pages 120–129. ACM, 2011.

[89] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, (2):50–59, 2011.

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

[91] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing (SC)*, pages 655–664. ACM, 1989.

[92] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. Gpgpu performance and power estimation using machine learning. In *HPCA'15*, pages 564–576. IEEE, 2015.

[93] Shucai Xiao, Ashwin M Aji, and Wu-chun Feng. On the robust mapping of dynamic programming onto a graphics processing unit. In *15th International Conference on Parallel and Distributed Systems (ICPADS), 2009*, pages 26–33. IEEE, 2009.

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

[95] Wei Xue, Chao Yang, Haohuan Fu, Xinliang Wang, Yangtong Xu, Lin Gan, Yutong Lu, and Xiaoqian Zhu. Enabling and scaling a global shallow-water atmospheric model on tianhe-2. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 745–754. IEEE, 2014.

[96] T. Yamanouchi. AES Encryption and Decryption on the GPU. *GPU Gems*, 3:785–803, 2007.

[97] 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)*, volume 48, pages 229–238. ACM, 2013.

[98] C.T. Yang, K.W. Cheng, and K.C. Li. An enhanced parallel loop self-scheduling scheme for cluster environments. *The Journal of Supercomputing*, 34(3):315–335, 2005.

[99] C.T. Yang, W.C. Shih, and S.S. Tseng. Dynamic partitioning of loop iterations on heterogeneous pc clusters. *The Journal of Supercomputing*, 44(1):1–23, 2008.

[100] J.H.C. Yeung, CC Tsang, KH Tsoi, B.S.H. Kwan, C.C.C. Cheung, A.P.C. Chan, and P.H.W. Leong. Map-reduce as a programming model for custom computing machines. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08.* IEEE, 2008.

[101] Yongpeng Zhang and Frank Mueller. Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters. *IEEE Transactions on Parallel and Distributed Systems (IPDPS)*, 24(3):417–427, 2013.