ACCELERATION AND OPTIMIZATION OF DYNAMIC PARALLELISM FOR IRREGULAR APPLICATIONS ON GPUS

A Dissertation
Presented to
The Academic Faculty

by

Jin Wang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2016

ACCELERATION AND OPTIMIZATION OF DYNAMIC PARALLELISM FOR IRREGULAR APPLICATIONS ON GPUS

To my husband Haicheng and my parents.

ACKNOWLEDGEMENTS

First of all, my deepest gratitude goes to my advisor, Prof. Sudhakar Yalamanchili, whose full support, incredible patience as well as deep understanding and perspective in research have always been great guidance during both of my PhD study and personal life.

Prof. Hyesoon Kim, Prof. Richard Vuduc, Prof. Tushar Krishna and Prof. Santosh Pande, thank you for serving on my committee and taking time to provide insightful feedbacks and comments on my research work.

Dr. Norm Rubin, thank you for being a great mentor during my internships as well as for my NVIDIA fellowship. Your experience in various research aspects has generated substantial inspiration for my PhD dissertation.

I would like to express my sincere thankfulness to my husband Haicheng, who is my best friend, life partner and soul mate. As a top researcher himself, Haicheng is also a great professional colleague who has helped me throughout the entire PhD.

I would also like to thank my parents, who have provided nothing but full support, financially and emotionally, for my 30 years of life. Even after my mother passed away, I still learn from her wisdom, passion and positive attitude towards life.

Last but not least, I would like to thank my fellow lab mates and graduate students, faculty members and staff from School of Electrical and Computer Engineering and College of Computing, external researchers from both academia and industry that provide me with great advices during multiple technical events. Their cooperation and assistance were essential for the completion of my PhD study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AMR Adaptive Mesh Refinement

BFS Breath-First Search

BSP Bulk Synchronous Parallel

CDP CUDA Dynamic Parallelism

CTA Cooperative Thread Array

CUDA Compute Unified Device Architecture

DFP **D**ynamically **F**ormed Pockets of Structured **P**arallelism

DTBL Dynamic Thread Block Launch

DVFS Dynamic Voltage and Frequency Scaling

GPU Graphics Processing Unit

HPC High Performance Computing

HWQ Hardware Work Queues

IPC Instruction-Per-Cycle

KD Kernel Distributor

KMU Kernel Management Unit

PT Persistent Thread

SIMD Single-Instruction-Multiple-Data

SMX	Stream Multiprocessor
TB	Thread Block

SUMMARY

The objective of this thesis is the development, implementation and optimization of a GPU execution model extension that efficiently supports time-varying, nested, fine-grained dynamic parallelism occurring in the irregular data intensive applications. These dynamically formed pockets of structured parallelism can utilize the recently introduced device-side nested kernel launch capabilities on GPUs. However, the low utilization of GPU resources and the high cost of the device kernel launch make it still difficult to harness dynamic parallelism on GPUs. This thesis then presents an extension to the common Bulk Synchronous Parallel (BSP) GPU execution model – Dynamic Thread Block Launch (DTBL), which provides the capability of spawning light-weight thread blocks from GPU threads on demand and coalescing them to existing native executing kernels. The finer granularity of a thread block provides effective and efficient control of smaller-scale, dynamically occurring nested pockets of structured parallelism during the computation. Evaluations of DTBL show an average of 1.21x speedup over the baseline implementations. The thesis proposes two classes of optimizations of this model. The first is a thread block scheduling strategy that exploits spatial and temporal reference locality between parent kernels and dynamically launched child kernels. The locality-aware thread block scheduler is able to achieve another 27% increase in the overall performance. The second is an energy efficiency optimization which utilizes the SMX occupancy bubbles during the execution of a DTBL application and converts them to SMX idle period where a flexible DVFS technique can be applied to reduce the dynamic and leakage power to achieve better energy efficiency. By presenting the implementations, measurements and key insights, this thesis takes a step in addressing the challenges and issues in emerging irregular applications.

CHAPTER I

INTRODUCTION

There has been considerable success in harnessing the superior compute and memory bandwidth of Graphics Processing Units (GPUs) to accelerate traditional scientific and engineering computations [7][48][61][60] dominated by structured control and data flows across large data sets using Bulk Synchronous Parallel (BSP) execution models represented by Compute Unified Device Architecture (CUDA) [57] and OpenCL [34]. These applications can be effectively mapped to the rigid 1D-3D massively parallel grid structures underlying modern BSP programming languages for GPUs. However, emerging High Performance Computing (HPC) and enterprise data intensive applications represented by data analytics, graph processing, machine learning and similar applications are dominated by sophisticated algorithms characterized by irregular control, data, and memory access flows challenging the effective harnessing of GPU accelerators.

Despite the above observation, there still exist segments of the computation within many irregular applications that locally exhibit structured control and memory access behaviors. These **D**ynamically **F**ormed Pockets of Structured **P**arallelism (DFP) occur in a data dependent, nested, time-varying manner and their most straightforward implementations usually lead to poor workload balance, control flow divergence, and memory irregularity resulting in poor utilizations and hence lower performance. State-of-the-practice GPU execution models have introduced the device-side kernel launching functionality represented by NVIDIA's CUDA Dynamic Parallelism (CDP) model [58] and OpenCL's device-side kernel enqueue [34] which can be utilized to implement DFP. However, the heavyweight nature of a GPU kernel, especially the non-trivial *kernel launching overhead*, mitigates the efficiency and effectiveness of such models. This has led to growing demands for a better execution model with lightweight mechanism to support DFP abstractions that more efficiently targets fine-grained dynamic parallelism in irregular applications.

This research seeks to propose such a lightweight execution model based on the investigation and characterization of several irregular applications. The first part of this research characterizes DFP by implementing and evaluating these GPU benchmarks with device-launched kernels in CDP [74]. In comparison with the original flat implementations where DFP are processed by individual threads, it is demonstrated that while CDP does address the productivity and algorithmic issues and exhibits potential performance benefit in terms of better control and memory behavior, the ability to harness dynamic parallelism on GPUs is still difficult in most cases due to 1) the low utilization of GPU resources and poor memory latency hiding ability caused by the discrepancy between the fine-grained DFP kernels and limited kernel level concurrency that results in low GPU occupancy and 2) the non-trivial overhead introduced by hardware and software stacks that are associated with the device-side kernel launch functionality and are aggravated by the substantially large number of DFP kernels.

To address the above challenges, the second part of this research proposes an extension to the traditional BSP execution model - Dynamic Thread Block Launch (DTBL) [72], which provides a lightweight mechanism for harnessing dynamic parallelism by spawning Thread Blocks (TBs) from GPU threads on demand and coalescing them with existing native executing kernels. The finer granularity of a TB provides effective and efficient control of smaller-scale, dynamically occurring pockets of structured parallelism during the computation. The TB coalescing process enlarges the pool of TBs that belong to an existing kernel so that they can be scheduled together to the GPU computation units referred as Stream Multiprocessors (SMXs) (using NVIDIA terminology [57]). This effectively increases TB-level concurrency which leads to higher GPU occupancy and utilization. DTBL also introduces substantially more lightweight microarchitecture support, driver and runtime implementations such that the overhead of launching a TB is considerably smaller than launching a kernel. With DTBL, the implementations of irregular CUDA applications launch one or multiple TBs for identified DFP with sufficient parallelism. Benchmark applications implemented with DTBL are evaluated on a cycle-level simulator to demonstrate the improved execution performance compared with the implementations using the flat GPU programming methodology or CDP.

To further explore and develop appropriate optimizations to avail of the potential of the DTBL execution extensions, the third part of this research focuses on the TB scheduling strategies that can effectively exploit spatial and temporal memory reference locality between parent kernels and dynamically launched child kernels, or between child kernels launched from the same parent kernel thread (sibling kernels). Modern GPU microarchitecture schedulers are designed for non-dynamic parallelism settings and are unaware of this new type of locality relationships. Towards this end this research first provides an analysis of parent-child and child-sibling reference locality in a set of benchmark applications. This analysis motivates a locality-aware scheduler referred as *LaPerm* [73] for dynamic TBs across the SMXs. *LaPerm* provides multiple levels of prioritization schemes that seek to ensure that child kernels can exploit temporal locality with parent kernels, and spatial and temporal locality with sibling kernels. It also balances overall SMX utilization with effective utilization of the local SMX L1 caches resulting in overall improved system performance. Experimental evaluation on a cycle-level simulator demonstrates that by increasing both the L1 and L2 cache performance, *LaPerm* is able to achieve Instruction-Per-Cycle (IPC) improvement over the original baseline TB scheduler.

The fourth part of this research examines the DTBL from the energy efficiency perspective and proposes an energy saving optimization based on the SMX occupancy phase behavior of DTBL. A comparison is performed between the execution phase behavior of DTBL and the regular GPU programming methodology to handle dynamic workloads such as the Persistent Thread (PT) model where the number of TBs executed on the SMXs are fixed in advance and a global work queue is employed to assign new dynamic work to those TBs. The purpose of such a comparison is to demonstrate the existence of SMX utilization variations when dynamic parallelism evolves and therefore the potential energy saving opportunities brought by the DTBL execution model. Specifically, the SMX occupancy bubbles where the SMX occupancy falls below a threshold are utilized for better energy efficiency, which are performed by 1) designing a new TB diversion and scheduling strategy to convert the SMX occupancy bubbles to SMX idle periods and ii) employing a flexible GPU Dynamic Voltage and Frequency Scaling (DVFS) scheme to reduce energy consumption.

This thesis argues that *a GPU execution model with lightweight dynamic spawning of workload and associated scheduling and energy consumption optimizations can efficiently target the fine-grained dynamic parallelism in irregular applications*. Specifically, this thesis makes the following contributions.

1. Establishing the concept of **D**ynamically **F**ormed Pockets of Structured **P**arallelism (DFP) via illustrations of their forms and behavior in emergent irregular GPU applications.

2. Characterizing DFP and elaborating insights from implementing multiple data intensive irregular CUDA benchmarks with child kernel launching functionality introduced by the CUDA CDP execution model, specifically in terms of control flow and memory behavior, GPU utilization and hardware/software overhead.

3. Designing the Dynamic Thread Block Launch (DTBL) as an effective lightweight execution mechanism for spawning dynamically created parallel work that is able to effectively harness the compute and memory bandwidth of GPUs. This is achieved from two perspectives.

    - Fine-grained TB launching and aggregation that increase GPU occupancy and utilization.

    - Lightweight microarchitecture and runtime design to alleviate the hardware and software stack overhead.

4. Optimizing the scheduling strategy for DTBL to further improve the execution performance by exploring the memory reference locality between the parent TBs and child TBs.

5. Exploring energy saving opportunities in DTBL by demonstrating the existence of SMX occupancy bubbles which can be leveraged for better energy efficiency.

The remainder of this thesis is organized as follows. Chapter 2 provides the background and related work of this thesis. Chapter 3 introduces the concept of DFP and its features by characterizing multiple irregular applications with CDP implementations. Chapter 4 proposes the DTBL extension to the current GPU execution model with detailed semantics definition, microarchitecture design, benefits and overhead analysis, and performance evaluation. Chapter 5 describes a memory locality-aware optimization for the DTBL model that is motivated by the memory reference locality relationships between the dynamic TBs in DTBL implementations and efficiently utilizes the GPU memory hierarchy for such locality relationship to achieve overall performance improvement. Chapter 6 examines the DTBL model from the energy and power dissipation perspective and proposes to utilize the SMX occupancy phase behavior for a new energy saving optimization that increases the energy efficiency of DTBL. Chapter 7 is the conclusion of this thesis.

CHAPTER II

BACKGROUND AND RELATED WORKS

This chapter provides the background for this thesis, including an introduction to the GPU execution model and the baseline GPU architecture, the scheduling process on the GPUs, the dynamic parallelism execution model that is supported by the current GPUs and the basic GPU power model adopted by this thesis. The NVIDIA terminology is used throughout the thesis, including its CUDA programming models and the Kepler GK110 architecture as it is the first architecture to support CUDA Dynamic Parallelism (CDP). However, the methodology, analysis and conclusions in this thesis also apply to new architectures and programming models such as AMD's GPUs and the OpenCL [34] programming models. This chapter also reviews the research works from various aspects that are related to this thesis.

2.1 Baseline GPU Architecture and CUDA Programming Model

The baseline GPU architecture adopted by this thesis is shown in Figure 1. It is composed of several major functional units: the Kernel Management Unit (KMU), the Kernel Distributor (KD), the computation units referred as Stream Multiprocessor (SMX) and the SMX scheduler which dispatches workload to the SMXs. An NVIDIA Kepler GK110 architecture [54] comprises of multiple SMXs, each of which features 192 single-precision CUDA cores, 64 double-precision units, 32 special function units and 32 load/store units. It also includes 64K, 32-bit registers and 64KB scratch-pad memory that can be used either as a L1 cache or shared memory. An L2 cache is shared across SMXs and connects through one or more memory controllers to the off-chip DRAM which is used as the GPU device memory. The GPU is connected to the host CPU by the interconnection bus and accepts operation commands such as memory copy and kernel launching from the CPU.

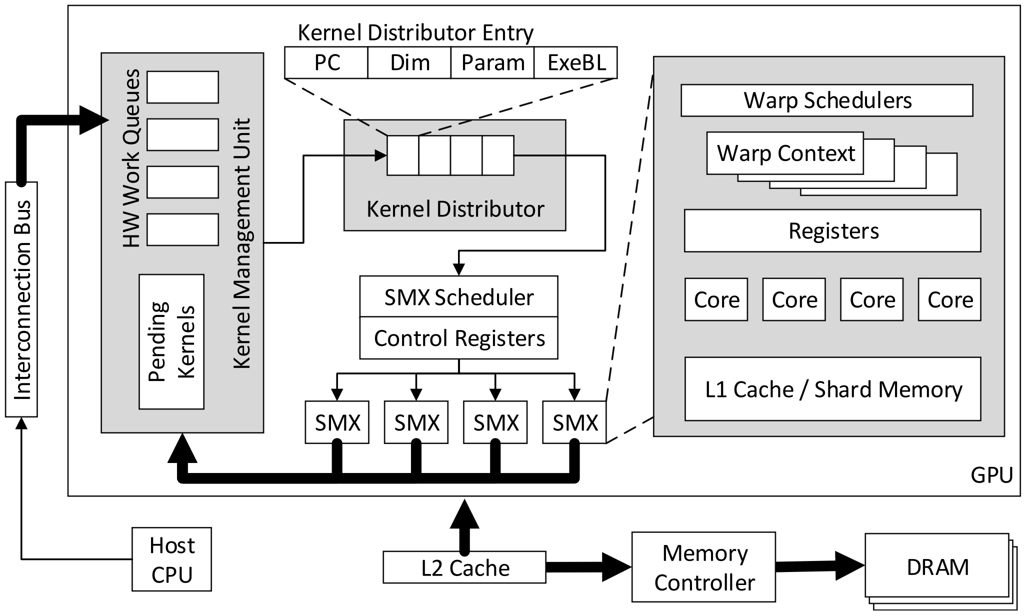


Figure 1: Baseline GPU architecture

Compute Unified Device Architecture (CUDA) [57] is the programming model introduced by NVIDIA for its GPUs. As shown in Figure 2(a), a CUDA program is expressed as a set of parallel kernels in which threads are grouped together into Thread Block (TB) or Cooperative Thread Array (CTA) and then into 1D-3D grids, all threads executing the same kernel code subject to user-defined synchronization barriers. Multiple memory spaces can be accessed by the GPU threads during their execution, including the global memory which resides in the GPU device memory and is visible to all the threads of a kernel, the shared memory which is visible only to the threads of a TB and holds the data that has the lifetime of a TB, and the local memory that is private to each individual thread. The constant memory and texture memory are two additional memory spaces that are visible to all the threads in the kernel. They reside in the GPU device memory but have different accessing patterns or addressing modes than the global memory.

2.2 Kernel, Thread Block and Warp Scheduling on GPUs

The CPU launches GPU kernels by dispatching kernel launching commands. Kernel parameters are passed from the CPU to the GPU at the kernel launching time and stored in

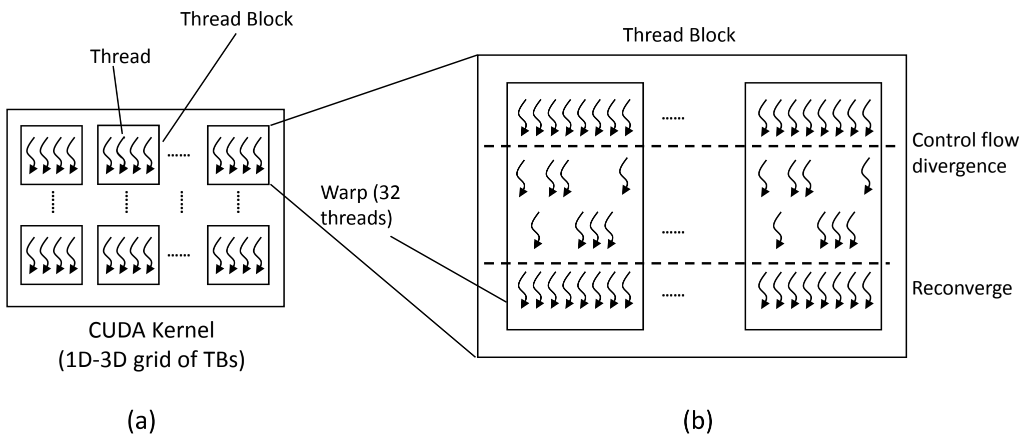


Figure 2: GPU programming model, including (a) CUDA thread hierarchy and (b) warps and control diveregence

the GPU global memory with necessary alignment requirements. The parameter addresses are part of the launching command along with other kernel information such as grid/TB dimension configuration and entry PC address. All the launching commands are passed to the GPU through software stream queues (e.g. CUDA stream). Kernels from different streams are independent from each other and may be executed concurrently while kernels from the same stream should be executed in the order that they are launched. The streams are mapped to Hardware Work Queues (HWQ) in the GPU that create hardware-managed connections between the CPU and the GPU. Earlier NVIDIA GPUs combine all streams and map them to only one HWQ, resulting in serialization of kernel launches from different streams. The current generation of NVIDIA GPU introduce Hyper-Q[53] - a technique which constructs multiple HWQs and maps individual streams to each HWQ to realize concurrency. However, if the number of software streams exceeds the number of HWQs, some of them will be combined and serialized. In the baseline GPU architectures, the Kernel Management Unit (KMU) manages multiple HWQs by inspecting and dispatching kernels at the head of the queue to the Kernel Distributor. Once the head kernel is dispatched, the corresponding HWQ stops being inspected by the KMU until the head kernel completes. The KMU also manages all the kernels dynamically launched or suspended by an SMX (e.g. through use of the CUDA Dynamic Parallelism feature) as discussed in section 2.4.

The Kernel Distributor (KD) holds all the active kernels ready for execution. The number of entries in the KD is the same as that of HWQs (32 in the GK110 architecture) as this is the maximum number of independent kernels that can be dispatched by the KMU. Each entry manages a set of registers that record the kernel status including kernel entry PC, grid and TB dimension information, parameter addresses and the number of TBs to complete. The SMX scheduler takes one entry from the KD in first-come-first-serve (FCFS) order and sets up the SMX control registers according to the kernel status. It then distributes the TBs of the kernel to each SMX limited by the maximum number of resident TBs, threads, number of registers, and shared memory space per SMX.

In today's GPU, the SMX scheduler dispatches the TBs of a kernel to the SMX in a round-robin fashion. Each cycle it picks one TB using the increasing order of the TB ID and dispatches it to the next SMX that has enough available resources to execute this specific TB. When a kernel is launched, all SMXs are unoccupied and therefore can accommodate one TB at each cycle, resulting in TBs being evenly distributed across the SMXs. For example, scheduling 100 TBs on a 13-SMX K20 GPU would result in SMX0 being assigned TBs (0, 13, 26, . . .), SMX1 being assigned TBs (1, 14, 27, . . .) and so on. When all the SMXs are fully occupied by the TBs, the SMX scheduler is not able to dispatch new TBs to the SMXs until one of the older TBs finishes execution. To illustrate it with the same example, suppose each SMX will be fully occupied by 3 TBs. TB 39 cannot be dispatched to SMX0 immediately after TB 38 as there are not enough resources available. At some point, TB 17 on SMX4 becomes the first TB to finish execution so the SMX scheduler can schedule TB 39 to the SMX4 instead of SMX0. This TB scheduling strategy in the baseline GPU architecture is designed to ensure the fairness of occupancy across all the SMXs and thereby execution efficiency. The SMX scheduler keeps updating the control register and the register in each KD entry to reflect the number of TBs that remain to be scheduled as well as those still being executed. When all the TBs of a kernel finish, the KD will release the corresponding kernel entry to accept the next kernel from the KMU.

During execution, a TB is partitioned into groups of 32 threads called a *warp* as the basic thread group executed on a 32-lane Single-Instruction-Multiple-Data (SIMD) unit made of CUDA cores as shown in Figure 2(b). SMXs maintain the warp contexts for the lifetime of the TB. At each cycle, the warp scheduler selects a warp from all the resident warps on the SMX that have no unresolved dependency according to a scheduling policy (e.g. round-robin) and then issues its next instruction to the cores. Each SMX in the GK110 has four warp schedulers and eight instruction dispatch units, allowing four warps to be issued simultaneously. By interleaving the execution of all the issued warps, an SMX is able to achieve hardware multithreading and hide memory latency. All the threads in a warp execute the same instruction in a lock-step fashion. When there is a branch and threads in a warp take different paths, the execution of threads on different paths will be serialized. This is referred to as control flow divergence and results in low SIMD lane utilization. The baseline architecture uses an immediate post-dominator reconvergence technique (PDOM) reconvergence stack [24] to track and reconverge the threads that take different branches. Memory accesses generated by 32 threads in a warp for aligned consecutive word addresses are coalesced into one memory transaction. Otherwise multiple memory transactions are generated to retrieve the data which may increase the memory access latency for the entire warp. This pattern of irregular memory accesses is referred as memory divergence [46].

2.3 Concurrent Kernel Execution

Concurrent kernel execution is realized by distributing TBs from different kernels across one or more SMXs. If one kernel does not occupy all the SMXs, the SMX scheduler takes the next kernel and distributes its TBs to the remaining SMXs. When a TB finishes, the corresponding SMX notifies the SMX scheduler to distribute a new block either from the same kernel or from the next kernel entry in the KD if the current kernel does not have any remaining TBs to distribute and the SMX has enough resources available to execute a TB of the next kernel. Therefore, multiple TBs from different kernels can execute on the same SMX [20]. In today's GPU, kernels can be executed concurrently to the limit of 32 (number of entries in KD). Large kernels which either have many TBs or use a large amount of resources are not likely to be executed concurrently. On the other hand, if the kernels in the KD only use a very small amount of SMX resources, the SMX may not be fully occupied even after all the kernels in the KD are distributed, which results in under-utilization of the SMX.

2.4 CUDA Dynamic Parallelism and Device-side Kernel Launch

Recent advances in the GPU programming model and architecture support device-side kernel launches - *CUDA Dynamic Parallelism* [58] - which provides the capability of launching kernels dynamically from the GPU without going back to the host CPU. This new functionality has been provided starting with the Kepler GK110 architecture. The kernel, TB or thread that initiates the device launch is the parent and the kernel that is launched by the parent is the child. Several device-side API calls can be invoked to specify the child kernel configuration, setup parameters, and dispatch kernels through device-side software streams to express dependencies.

When a child kernel is launched, the parameter buffer pointer of the kernel is retrieved through the device runtime API `cudaGetParameterBuffer`. Then the argument values are stored in the parameter buffer and the kernel is launched by calling `cudaLaunchDevice`. CDP allows explicit synchronization between the parent and the child through a device runtime API `cudaDeviceSynchronize`. Launches can be nested from parent to child, then child to grandchild and so on. The deepest nesting level that requires explicit synchronization is referred as the synchronization depth. The maximum synchronization depth supported on GK110 is 24. Parents will be suspended and yield to child kernels if explicit synchronization is required. If no explicit synchronization is specified, there is no guarantee of the execution order between the child and parent. Concurrent execution of the child kernels is possible but not guaranteed, depending on the availability of the GPU resources. The 32 concurrent kernels supported on the GK110 architecture include both the host-launched and device-launched kernels. Parent and child kernels have coherent access to global memory with full consistency only at the point when parent kernels launch child kernels or the explicit synchronization is requested. Shared memory and local memory are exclusive for parent and child kernels and are invisible to each other.

In the baseline GPU architecture, there is a path from each SMX to the KMU so that all the SMXs are able to issue new kernel launching commands to the KMU. Similar to host-side launched kernels, parameters are stored in the global memory and the address is passed to the KMU with all other configurations. When a parent decides to yield to a child kernel, the SMX suspends the parent kernel and notifies the KMU to hold the suspended kernel information. The KMU dispatches device-launched or suspended kernels to the KD along with other host-launched kernels in the same manner. Therefore device-launched kernels also take advantage of concurrent kernel execution capability.

Current architecture support of device-side kernel launching comes with non-trivial overhead. The total kernel launching time scales with the number of child kernels, which is composed of the time spent in allocating the parameters, issuing a new launching command from the SMX to the KMU, and dispatching a kernel from the KMU to the KD. Device-side kernel launches also require a substantial global memory footprint. A parent may generate many child kernels which can be pending for a long time before being executed, thus requiring the GPU to reserve a fair amount of memory for storing the associated information of the pending kernels. On the other hand, the device runtime has to save the states of the parent kernel when they are suspended to yield to the child kernels at the explicit synchronization points.

2.5 GPU Power Modeling

This thesis relies on a high-level GPU power model proposed and used in GPUWattch [41]. The model is implemented and integrated with the cycle-level GPU simulator GPGPU-Sim [9] based on the McPAT power model [43].

The GPU power is composed of the leakage power, the idle SMX power, as well as the dynamic power of all the GPU components, including registers, shared memory, execution units, caches, main memory, etc. The dynamic power of the GPU components are computed by collecting the microarchitectural parameters of each component through GPGPU-Sim and then feeding them into the McPAT model with several adapated blocks specifically designed for the GPU microarchitecture. The idle SMX power is determined by executing several microbenchmarks on a real GPU and controlling the number of active SMXs by using different number of TBs and configurations. The leakage power is modeled by measuring the GPU constant power as described in [41].

The GPU has multiple execution performance capability and power consumption states referred as the P-state, each of which has the corresponding voltage and frequency settings [52]. The P-state with the highest performance will also have the highest power consumption, and vice versa. The DVFS technique has been made available on the current NVIDIA GPU to adjust the P-state for the entire GPU. However, P-state of each SMX cannot be controlled separately. The voltages scaling follows the model used in [2] for different frequencies under different technology.

2.6 Related Works

There has been considerable effort in developing new GPU algorithms, frameworks and methodologies for the irregular application domain. This section briefly reviews the research works that are related to this thesis.

2.6.1 Characterization of GPU Workloads

Characterization and analysis of GPU applications can be traced to a very early time, mainly focusing on regular applications which have rigid 1D-3D data structures that can be directly mapped to GPU architectures. Kerr et al. [33] characterizes GPU kernels using different metrics and proposes methodologies to write GPU programs with better performance. The benchmark suites Rodinia [17] proposed by Che et al., Parboil [68] proposed by Stratton et al., and SHOC [21] proposed by Danalis are some representative benchmarks used in GPU studies. Research with these benchmarks have focused on approaches to utilize the structured BSP model efficiently.

Recently people have been investigating the performance of new irregular applications on GPUs which exhibit more unstructured control flow and memory behavior. These applications are represented by graph processing, machine learning, relational computing and etc. Examples of these applications implemented by GPUs include 1) Adaptive Mesh Refinement (AMR) used for combustion simulations [36] that operates on grid-like structure and refines each cell in the grid according to certain temperature conditions, 2) graph coloring problem whose goal is to assign a color to each vertex in the graph such that no neighboring vertices have the same color [19], 3) product recommendation systems that use item-based collaborative filtering algorithm to construct a similarity matrix containing the customer purchasing information, and 4) the relational JOIN operator where two input relation arrays are examined to generate a new relation array consisting of the key-value pairs where the keys are present in both of the input arrays [22]. To facilitate the development of new GPU programming methodologies or execution models, many researchers have performed systematic characterization and analysis of these irregular applications. Burtscher et al. [14] study the behavior of irregular applications on GPUs with quantitative metrics for both control flow and memory access irregularity. Che et al. [16] use the Pannotia benchmark suite to illustrate the characteristics of irregular graph applications on GPUs. These works show that implementations of irregular GPUs applications mainly suffer from workload imbalance and scattered memory accesses that cause control flow and memory irregularity.

2.6.2 Research on Irregular Applications on GPUs

Researchers have been investigating and seeking more efficient solutions for the irregular applications, e.g., implementing them using the BSP model but with new programming methodologies. Gupta et al. [27] introduce the Persistent Thread (PT) programming style on GPUs where a number of TBs that occupy all the SMXs are initially launched and stay on the GPU for the life time of the kernel. These TBs dynamically generate tasks that are appended to a globally visible software queue while persistently consuming tasks. The goal of the PT model is to achieve overall GPU workload balance especially across the SMXs while processing evolving irregularity in the programs. Merrill [47] implement Breath-First Search (BFS) on GPUs using fine-grained TBs to adaptively explore the neighbors of vertices in parallel which can utilize the SMXs more efficiently and achieve better fine-grained load balance. However, such effort has not been applied to other irregular applications in a more general form. Other research works on GPU irregular applications include redesigning data structures and re-organizing memory accesses through algorithm, compiler and runtime optimizations to harness the GPU capability. Solomon et al. [66] investigate and change the synchronization behavior of the BFS algorithm as well as that of a Matrix Parenthesization algorithm to improve performance. Zhang et al. [80] present a systematic transformation framework to remove dynamic irregularities in both control flows and memory references by data relocation and memory reference redirection. Wu et al. [77] employ the kernel fusion technique for the implementations of relational algebra operators on the GPU to reduce memory irregularity in all levels of memory hierarchies. The goal of these studies is to decrease the irregularity of the control flow and memory behavior or adapt the algorithm to the GPU architectures.

2.6.3 Research on Nested Dynamic Parallelism

It has been observed that many of the irregularities can be converted or implemented as fine-grained dynamically formed structured parallelism. Improving the performance of irregular applications by handling dynamic parallelism is an important and challenging question in the GPU programming community. Lars et al. [11] implement the nested parallel programming language NESL on the GPU by flattening the nested parallelism semantics using compiler and runtime techniques. Lee et al. [38] propose an auto-tuning framework that efficiently maps the nested patterns in GPU applications using logical multidimensional domain with pruning constraints and adaptive shared memory management. Steffen et al. [67] propose a dynamic micro-kernel architecture for global rendering algorithm which supports dynamically spawning threads as a new warp to execute a subsection of the parent threads code. Orr et al. [59] design a task aggregation framework on GPU based on the channel abstraction proposed by Gaster et al [25]. Each channel is defined as a finite queue in virtual memory (global memory space that is visible to both the CPU and the GPU) whose elements are dynamically generated tasks that execute the same kernel function. Kim et al. [35] implement a hardware work list and investigate several different work distribution schemes to process dynamically generated parallel work elements. All these works have been taking steps toward more effective and efficient solutions for processing dynamic parallelism in irregular applications.

The prevalence of dynamic parallelism execution models on GPUs such as CDP and OpenCL device-side enqueue enables a more general, flexible and productive implementation strategy for the irregular applications by using device-launched kernels for dynamically generated workload. As a newly introduced technology, the utilizations of CDP have been reported by a few research works. Wang et al. [71] propose the CDP implementation of graph-based substructure pattern mining by using device-launched kernels to expand the depth-first search tree in parallel. DiMarco et al. [23] analyze the clustering algorithm including the K-means clustering and the hierarchical clustering with CDP implementation which invokes child kernels directly from GPU for data updating to avoid CPU-GPU interaction. Zhang et al. [81] apply CDP to a set of graph algorithms on the GPU by adapting to their data-driven nature. The idea is to launch a kernel from the CPU to process the outer loop of the graph problems followed by another kernel launched from the GPU to process the inner loop for vertex expansion. Li et al. [42] uses device-side kernel launches for irregular loops and recursive computations. Improved overall performance are reported in these studies when the implementations are carefully tuned to avoid excessive CPU-GPU communications and device kernel launching overhead while taking advantage of the flexibility of the CDP model.

However, the device-launched kernels in the above implementations are applied only for the coarse-grained dynamic parallelism and are able to avoid the CPU-GPU communication overhead in the original non-CDP implementations. In most cases, the fine-grained dynamic parallelism is the major source of irregularity in these applications and can dominate the performance consequences. Device-side kernel launching functionality is still difficult to be fully utilized in its current form for the fine-grained dynamic parallelism due to its non-trivial overhead especially in the scenario where hundreds or thousands of kernels have to be launched. Therefore, researchers have been proposing different extensions and optimizations. Yang et al. [79] analyze the nested parallelism in several benchmarks and observe that CDP implementations dramatically reduce memory bandwidth. They then propose a compiler technique that can dynamically activate or deactivate the GPU threads to adapt to the evolving parallelism in the applications. Chen et al. [18], on the other hand, propose a compiler technique "Free Launch" that reuses the parent threads to process the child kernel tasks and dynamically converts programs written with CDP into the ones that only employ parent threads. This is accomplished by a set of transformations to deal with thread mapping, shared memory usage and synchronizations. All these techniques can apply to the regular BSP execution model and employ compiler and runtime optimizations to avoid the extra overhead introduced by device-side kernel launching.

2.6.4 Research on GPU Scheduling

The SMX scheduling strategy has an essential impact on the overall performance of GPUs, especially when they are incorporated with the cache and memory system performance. Research works in this area include new warp scheduler, TB schedulers and new memory system designs. Rogers et al. [63] employ multiple warp schedulers to achieve optimal cache performance in different scenarios. The schedulers are able to adaptively choose which warp to be dispatched according to the heuristics that are generated from cache behavior. Narasiman et al. [50] propose a two-level warp scheduler to minimize the memory access latency. The basic idea is to divide pending warps into multiple groups so that warps in each group can be scheduled together. Group switching only happens when warps from one group are all stalled because of the long-latency memory operations. Jog et al. [31] advance the two-level warp scheduling technique to make it TB aware so that the memory locality existing in the warps that belong to the same TB can be better accommodated when warp group switching happens. Kayıran et al. [32] demonstrate that the memory contention could be caused by scheduling maximum possible number of TBs as this limit is only determined based on SMX occupancy but does not take into account the memory system performance. They propose to use a dynamic TB scheduling mechanism to minimize such contention. Lee et al. [39] make the argument that consecutive TBs may have memory reference locality which can be better utilized to improve cache and memory system performance when scheduled on the same SMX instead of neighboring SMXs. Rhu et al. [62] design a locality-aware memory hierarchy that is both able to accept coarse-grained memory accesses that are common in regular applications as well as adapt to the fine-grained memory access patterns in irregular applications on GPU. In both cases, the newly designed memory system is able to achieve high bandwidth utilization. While the above works have demonstrated considerable effort in memory system aware scheduling, none of them are directly applicable to the domain of dynamic parallelism in GPUs where new types of locality behaviors are introduced, e.g., spatial and temporal reference locality between parent kernels and child kernels.

2.6.5 Research on GPU Energy Efficiency

Many studies have explored and analyzed the energy efficiency of GPUs. Different power models have been proposed [44][29] to estimate the GPU power consumption more accurately, which could in turn facilitate more research on power-based GPU optimizations. DVFS and power gating technology have been proposed for GPUs on multiple levels. Jiao et al. [30] study the benefit of concurrent kernel execution from the power consumption perspective and combine it with DVFS to improve energy efficiency. Ge et al. [26] investigate the impacts of DVFS on applications executed on the Tesla K20 GPU. Abdel-Majeed et al. [3] propose a tri-modal register access control unit as well as an active mask aware activity gain unit to reduce both the leakage power and the dynamic power of GPU register files. Lee et al. [40] present a warp-compression scheme that reduces the data redundancy within a warp by compressing the register values so that power gating can be applied to unused register banks to save register file power consumption. Abdel-Majeed et al. [4] propose to schedule instructions of the same type together so that the execution units can have a long window of idlenss to be turned off with power gating. They [5] also discover the existence of significant fine-grained pipeline bubbles in warp execution and convert these bubble to potential energy saving opportunities using a specific scheduling startegy so that power-gating can be applied on the idle warp lanes to reduce leakage energy. Xu et al. [78] study the behavior of branch divergence and propose a warp scheduler to schedule warps with similar branch divergence patterns together to create long warp lanes idleness for applying power gating. Wang et al. [76] proposes to power gate GPU caches when there are no cache requests. While the energy saving optimizations proposed by the above works are based on the GPU applications implemented with regular BSP model and may also show benefits in the dynamic parallelism from a general perspective, there is still a lack of new energy efficiency studies that are applicable to the dynamic parallelism settings on GPU.

2.6.6 Summary

In summary, while the earlier studies and research works discuss the irregular applications on GPUs from various aspects, this thesis differs with them in addressing the problem of fine-grained dynamic parallelism in irregular applications and seeks a more efficient solution with new GPU execution model extensions and optimizations.

CHAPTER III

DYNAMIC PARALLELISM IN IRREGULAR APPLICATIONS

As the first part of this thesis, this chapter explores multiple data intensive irregular CUDA applications on GPUs with CUDA Dynamic Parallelism (CDP) support and establishes the concept of **D**ynamically **F**ormed Pockets of Structured **P**arallelism (DFP) through detailed analysis of the benchmark applications in terms of control flow and memory behavior.

3.1 Impact of Irregular Applications

The CUDA and OpenCL programming model is structured around massively parallel threads grouped into TBs organized into 1D to 3D grids. Data parallel computations over multidimensional arrays of data fit well within this model where each thread can be mapped into a logically contiguous partition of the data set.

Emerging data intensive applications are increasingly irregular by operating on unstructured data such as trees, graphs, relational data and adaptive meshes. These applications have inherent time-varying, workload-dependent and unpredictable memory and control flow behavior that may cause severe workload imbalance, poor memory system performance and eventually low GPU utilization. For example, typical GPU implementations for vertex expansion operations that are commonly used in graph problems assign one thread to expand each vertex in the vertex frontier with a loop that iterates over all the neighbors. Since the number of neighbors for each vertex can vary, the implementation may suffer from poor workload balance across threads. Further, vertex expansion can generate (depending on the choice of data structure) non-coalesced memory accesses due to the lack of spatial locality across adjacent vertices leading to multiple memory transactions and increasing memory divergence, e.g., threads finish memory instructions at different times.

Another common strategy for handling unstructured data is to use loop iterations within each BSP thread to access non-contiguous data elements. This too leads to increased memory divergence and load imbalance. As data structures become more diverse, the mapping of data to threads becomes more complex and variance in memory access patterns and control flow grow accordingly.

3.2 Dynamically Formed Pockets of Structured Parallelism

Structured memory accesses and uniform control flow make the best use of the computational and memory bandwidth of GPUs. In spite of the observations in Section 3.1, one can observe **D**ynamically **F**ormed Pockets of Structured **P**arallelism (DFP) in these applications that can locally effectively exploit the GPU compute and memory bandwidth. For example, in vertex expansion common data structures used in graph problems (e.g. Compressed Sparse Row or CSR [65]) store neighbors of one vertex in consecutive addresses and the memory access can be coalesced when neighbors are explored in parallel. In adaptive mesh refinement used in combustion simulations, certain parts of the mesh will be refined in parallel into a finer grained mesh creating hierarchical nested grid structures each of which may be of different dimensions. In general, DFP commonly occurs in one of the following two patterns:

Static Data Structure Traversal. Applications have irregular but statically defined data structures while the algorithms that traverse them encounter varying degrees of parallelism. Graph and tree traversal algorithms such as breadth first search (BFS) are examples in this category.

Dynamic Data Generation. The application data structures themselves are generated during execution and their form and extent are themselves data dependent. For example, combustion simulation (adaptive mesh refinement), tree generation (indexing) and the relational JOIN operator all start from an initial data set and dynamically generate new irregularly structured data sets in parallel.

Given the preceding view of the behaviors of irregular applications the remainder of this chapter addresses the characterization and analysis.

3.3 Implementation Using CUDA Dynamic Parallelism

The introduction of device-side kernel launching in GPUs enables an implementation scheme that new child kernels are dynamically invoked for any identified DFP in irregular applications. In the vertex expansion example, the original neighbor exploration loop can be replaced by a dynamically launched kernel that employs uniform control flow. The approach can potentially increase the performance by reducing control flow divergence. For some common data structures used in this problem such as Compressed Sparse Row (CSR) where neighbor IDs of each vertex are stored in consecutive addresses, parallel neighbor exploration may also generate coalesced memory accesses.

A common code structure for implementing DFP using device-launched kernels in CDP is shown in Listing 3.1, where a parent thread checks some conditions and determine whether new parallel workload should be launched through `childKernel` either to traverse a new portion of the data structure or to generate new data sets. A typical condition is that whether the dynamic workload has sufficient parallelism, which is designed to avoid the low utilization of the SIMD lanes in each SMX. A straightforward understanding of CDP implementation is that it replaces the parallel loops performed by threads in the non-CDP implementation by child kernels. Specifically, such implementations consider the following aspects.

```
//executed by each thread in parent kernel
threadData = getData(threadId);
if(condition(threadData))
    childKernel<<<TBS, THREADS>>>
        (threadData, ...);
```

Listing 3.1: Common code structure that handles DFP with CDP

Parallel computation workload. As defined in DFP, the newly launched kernels handle the parallel computation workload that is discovered dynamically by a parent thread at runtime. This is in comparison to the unbalanced GPU implementations where each thread could use a loop with different iteration count to deal with the computation. Since parent threads only have to issue a child kernel and child kernels handle only parallel computation with little or no control divergence, the CDP implementation can achieve higher GPU utilization. However, it should be noted that sometimes there is not enough parallelism to launch a child kernel. For example, if the neighbor degree in vertex expansion problem is less than the warp size, SIMD lanes cannot be fully utilized if a new kernel is launched for expanding that vertex. In this case, the computation will still be left to the parent kernel.

Memory access patterns. The memory access pattern in a CDP implementation can be different from a non-CDP implementation. The memory addresses that are accessed by one thread in different loop iterations in a non-CDP implementation are now accessed by contiguous threads in a child kernel using one memory instruction. This could effectively change the number of coalesced memory accesses as well as cache hit rate.

Recursion. Recursive kernel launch is possible with the CDP support on GPUs. For some of the unstructured applications, it is necessary to recursively launch new computation dynamically. Non-CDP implementation tends to either convert the recursive algorithm to loop iterations or manage stack-based data structure at both the host and the device. The CDP implementation simply calls the same kernel recursively.

Concurrent kernel execution. Child kernels are launched independently from each other and can be executed concurrently. The implementation uses one stream for each child kernel launch. Although no concurrency can be guaranteed from the perspective of GPU architectural support of CDP [58], the use of CUDA streams can increase the possibility to the most extent.

Child kernel configuration. A common practice for GPU programming is to partition workload between TBs and then between threads. The same argument holds true for child kernels in the CDP implementations. The implementations experiment with different TB sizes and grid sizes to generate optimal performance. TB sizes should be multiple of 32 to eliminate any intra-warp thread divergence for child kernels. When the block size is not a multiple of 32, the remaining threads are executed by the parent kernel. This is analogous to the loop transformation that unrolls a loop *k* times by creating two loops - one that is unrolled *k* times and one that has loop bounds of ($N \; mod \; k$) where *N* is the loop bound.

Shared memory. The current form of CDP does not allow the child kernel to access the shared memory declared by the parent kernel. Therefore, if the dynamically launched child kernel needs to access the data stored in the shared memory, CDP implementations either pass the data value directly as the kernel argument or dump them into global memory. The former solution can only deal with small number of arguments and the latter solution could introduce large memory and runtime overhead.

Synchronization. CDP supports explicit synchronization between parent and child kernels at the substantial cost of both execution time and memory footprint. Therefore, the implementation would avoid using synchronization as much as possible. However, there are still a few cases that synchronization is necessary to conserve either temporal or spatial ordering consistency.

3.4 Characterization of CDP Implementations of Irregular Applications

In order to identify the major characteristics of DFP in the irregular applications, experiments are performed on multiple GPUs with Kepler GK110 architectures, including NVIDIA Tesla K20c, Geforce Titan and Tesla K40. Table 1 shows the features of these GPUs. Both non-CDP implementations and CDP implementations of the unstructured applications are examined. The CUDA 5.5 toolkit is used including the nvcc compiler and the runtime library. For CDP implementations, the compiler also links against CUDA device runtime library, i.e. `-lcudadevrt`. The CUDA Profiler NVProf 5.5 [55] is used to measure the metrics and the overall execution time of the kernels. Benchmark performance is

Table 1: Configurations of the GPUs used for CDP characterization experiments

	Tesla K20c	Geforce Titan	Tesla K40
SMX	13	14	15
Cores	2496	2688	2880
Clock Frequency (MHz)	706	837	745
Global Memory Capacity (GB)	5	6	12
Memory Bandwidth (GB/s)	208	288	288

evaluated on K20c and CDP overheads are compared across all the three GPUs.

The non-CDP implementation are compared with the CDP implementation of the unstructured applications. In both cases, inputs are evenly partitioned among threads and TBs. In the non-CDP case, DFP is handled by individual threads respectively, generally through loops. The CDP implementation uses dynamically launched kernels for parallel computations detected through DFP.

To evaluate the impact of CDP implementations on both the control flow and memory access, the following hardware metrics are used.

warp_execution_efficiency (WEE). This metric measures the ratio of active threads within a warp to the total number of threads in a warp (32) for all executed instructions. It is an indication of the control divergence or workload unbalance in the unstructured applications. Note that NVProf does not allow separate metrics measurement for parent kernels and child kernels in CDP, so the metric measured by NVProf are affected by the parent kernel execution, the child kernel execution and the child *kernel launching overhead* (recall that the overhead includes child kernel parameter passing and device runtime management). The following approach is uesd to measure and compute the warp execution efficiency excluding CDP kernel launching overhead and referred to as the ideal WEE (**WEEI**) since it represents the ideal efficiency that can be achieved:

$$\text{WEEI} = \frac{\text{WEE\_parent} * \text{inst\_parent} + \text{WEE\_children} * \text{inst\_children}}{\text{inst\_parent} + \text{inst\_children}}$$

In the equation, inst_parent and inst_children are the effective executed instruction by the parent and children respectively. When measuring WEE_parent and inst_parent, the child kernel launch code are removed from the parent kernel. A warm up kernel is executed before the parent kernel to make sure the parent kernel execution path does not change when child kernels are removed. When measuring WEE_children and inst_children, the child kernels are extracted and launched from the hosts using the same configuration and input data as the device launch. By doing this the CDP *kernel launching overhead* is excluded in the measurement. The argument is that this approach is accurate enough to generate WEEI as it only depends on execution path but not any hardware-dependent factors such as warp scheduling and child kernel scheduling policy. The purpose of WEEI is to demonstrate the potential benefit of CDP implementation of DFP by setting up the possible upper bound.

ldst_replay_overhead (LSRP). This metric measures the average number of replays for each load/store instruction executed. Instructions are replayed when there is bank conflict or non-coalesced memory access. Therefore, LSRP is able to capture the memory irregularity [14]. Since LSRP may be dependent on the execution history (e.g. cache and RAM access history), it is not reasonable to separate the parent kernel and child kernels to measure ideal LSRP for CDP implementations as for WEE. However, measuring the number of load/store instructions separately leads to the conclusion that the load/store instructions from *kernel launching overhead* only comprise a very small percentage of all the load/store instructions, so the directly measured LSRP can still be a good indication for memory divergence affected by CDP.

l2_cache_hit_rate (L2HIT). This metric measures the L2 cache hit rate and captures the memory locality in the program either for a thread or for threads from interleaved warps. Again, L2HIT is measured directly without excluding the child *kernel launching overhead*.

3.4.1 Benchmarks

Eight irregular benchmark applications are selected with different input data sets as shown in Table 2. The source code of these applications are from the latest benchmark suites or implemented as described in recently published papers. These applications are re-implemented with CDP. The following is a brief description of these irregular applications.

Adaptive Mesh Refinement (AMR): AMR operates on grid-like structure and refine each cell in the grid according to certain conditions. AMR is used to represent the combustion simulation problem [36] where each cell in the grid is given an average temperature. Cells are refined to smaller cells according to the energy computed out of the average temperature. The process stopped until the energy of each cell in the grid is below a threshold. Non-CDP implementation uses one kernel for each refine level and cell refinement is performed by each individual thread with loop iterations. CDP implementation launches cell refinement kernel recursively when energy threshold condition is satisfied.

Barnes Hut Tree (BHT): The BHT problem is part of the Barnes-Hut NBody Simulation [15] which computes the forces between the points in the space. A tree is built where each leaf node only contains at most one data point. Each thread takes one data point and compute the force between that point and any other point if they are close or the center of mass of any other cell if they are far away. The algorithm needs each thread to traverse the tree depending on the point-to-point distance. CDP implementation launches a new kernel if one thread needs another level of tree traversal. The input to BHT are randomly generated data points.

Breadth-First Search (BFS): BFS algorithm searches and visits all vertices in a graph using breath-first patterns. Graph is stored in the CSR format where a column buffer stores all the edges that are connected to each source vertex and a row buffer stores the starting edge index of each vertex in the column buffer. The vertex frontier is maintained for each search iteration. Each thread takes one vertex in the frontier, expands the edges, marks visit information and puts unvisited vertices to the new frontier. CDP implementation launches child kernels dynamically to expand the vertices in parallel according to the vertex degree (number of edges connected to the vertex). Three different graphs are used as the input to BFS [8]: citation network (citation), USA road network (usa_road) and a sparse matrix from Florida Sparse Matrix Collection (cage15). Note that while the implementation of BFS algorithm used in this thesis focuses on the data-driven methodology, CDP implementations apply to other implementation schemes such as the topology-driven methodology [51] as DFP still exists in the vertex expansion operation.

Graph Coloring (CLR): Graph Coloring problem is widely used in lots of research domains, e.g. compiler register allocations. The goal is to assign a color to each vertex in the graph such that no neighboring vertices have the same color. The algorithm [19] starts by assigning each vertex with a random integer and then in each iteration, each thread takes a vertex and marks itself with the iteration color if its value is larger than any adjacent vertex. The vertices with color assigned are ignored in subsequent iterations. Similar as BFS, CDP implementation launches new child kernels to examine the neighbors for each vertex. The input to CLR are the three graphs used in BFS.

Regular Expression Match (REGX): Regular expression match is the centric for many search and pattern match problems, e.g. network packet routing. The regular expression pattern is represented by finite automata (FA) and stored in the memory as a graph where each vertex represents a state and the edges represent transitions between states [75]. Each thread takes an input stream and traverse the FA. If a match is found, the corresponding state in the FA is returned. CDP implementation recursively traverses the FA and examine the transition edges in parallel. The input to REGX are the DARPA network packets collection (regx_darpa) [45] and random string collection (regx_string).

Product Recommendation (PRE): Product recommendation systems are widely used in industry especially on e-commerce website. These systems predict the customer purchase behavior according to past purchase records. The focus is on the item-based collaborative filtering algorithm for recommendation systems. One important part of item-based collaborative filtering is to construct an $M \times M$ similarity matrix of the items by examining the M items purchased by N customers. Each thread examines one customer and records item-item pairs into the similarity matrix [49]. For P items purchased by one customer, there are $P \times (P - 1)$ pairs to be recorded. CDP implementation launches child kernels to record these pairs in parallel. The input to PRE are data from MovieLens [1].

Relational Join (JOIN): The JOIN operator is relational algebra operator that is commonly used in relational database computation. The inner JOIN algorithm is evaluated where two input relation arrays are examined to generate a new relation array consisting of the key-value pairs where the keys are present in both of the input arrays. Each thread takes one element from one of the input arrays and uses binary search to find matching keys from the other array [22]. Workload imbalance can happen when matched element count varies for threads. CDP implementation resolves the problem by launching a new kernel to gather the result elements in parallel for each thread. The input data to JOIN are synthetic data arrays that have uniform distribution (join_uniform) and gaussian distribution (join_gaussian).

Single-Source Shortest Path (SSSP): SSSP is a classic graph problem which finds the paths with the minimal cost (sum of weights) from a given source vertex to all the vertices in the graph. Except the source vertex, all vertices are starting from infinite cost. It is then updated by examining the neighbors in each iteration to find the one with minimum cost after adding the weight from that neighbor to the vertex. CDP implementation launches a new kernel to examine the neighbors and uses reduction to find the minimum. Input to SSSP are the three graphs used in BFS.

3.4.2 Evaluation and Analysis

This section reports a comprehensive evaluation and analysis of the benchmark performance. First the CDP and non-CDP implementations of the benchmarks are compared in control flow behavior, memory behavior and overall execution time to illustrate the potential impact of handling DFP with CDP on the unstructured applications. Different inputs to the benchmarks are used to to capture behavior and performance on various characteristics. Then CDP overhead is evaluated from several aspects including kernel launch, memory

Table 2: Benchmarks used in the CDP characterization experimental evaluation

Application	**Input Data Set**
Adaptive Mesh Refinement (*amr*)	Combustion Simulation[36]
Barnes Hut Tree (*bht*) [15]	Random Data Points
Breadth-First Search (*bfs*) [47]	Citation Network[8] USA Road Network[8] Cage15 Sparse Matrix [8]
Graph Coloring (*clr*) [19]	Citation Network[8] USA Road Network[8] Graph 500 Logn20[8] Cage15 Sparser Matrix [8]
Regular Expression Match (*regx*) [75]	DARPA Network Packets [45] Random String Collection
Product Recommendation (*pre*) [49]	Movie Lens [28]
Relational Join (*join*) [22]	Uniform Distributed Data Gaussian Distributed Data
Single Source Shortest Path (*sssp*) [37]	Citation Network[8] USA Road Network[8] Fight Network [70] Cage15 Sparser Matrix[8]

footprint and algorithm overhead. This section also analyzes the child kernel workload intensity and scheduling policy.

3.4.2.1 Control Flow Behavior

The WEE of non-CDP implementations and both WEE and WEEI of CDP implementations are shown in Figure 3. Recall that WEE-CDP includes the child *kernel launching overhead*.

WEE of the non-CDP implementations ranges from 21.9% to 98.8%. Low WEE indicates lower SIMD lane utilization or more workload imbalance in unstructured applications. By using CDP implementation for DFP, the workload imbalance or control divergence can be reduced and WEE can be effectively increased. WEEI is shown here which is the ideal WEE that can be achieved by applying CDP implementation excluding the *kernel launching overhead*. For most applications, WEEI increases 2.2% to 65.3% from WEE-nonCDP. Examples of such applications include AMR, BFS_citation, CLR_citation, SSSP_citation, PR, REGX and JOIN, all operating on highly irregular data structure or

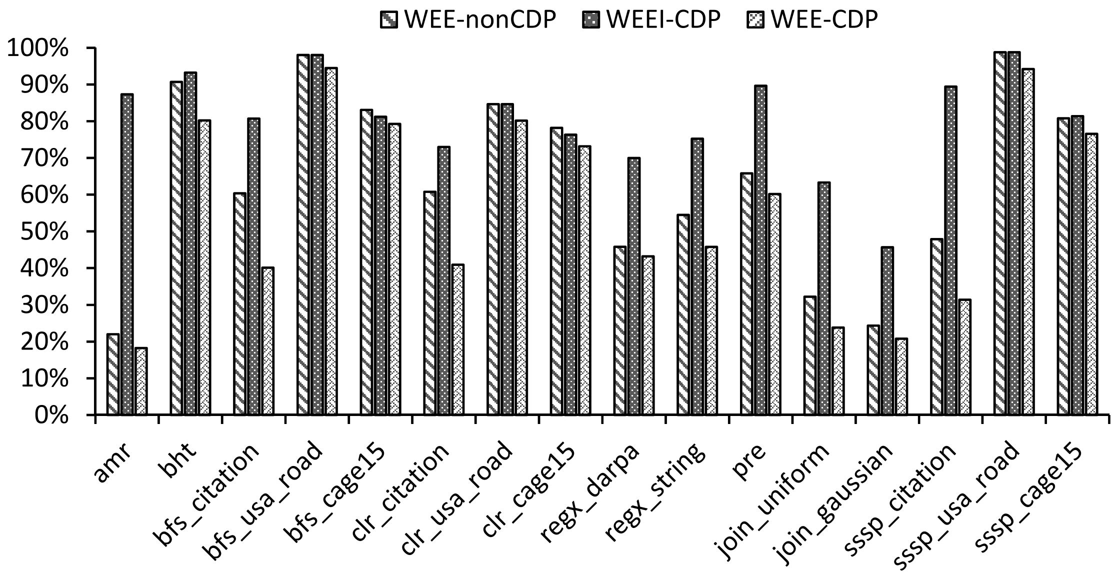


Figure 3: Warp Execution Efficiency for non-CDP and CDP implementations.

generating highly irregular computation. The refinement of AMR is completely dependent on the data point values and varies to a large degree from one thread to another. For the citation network graph, vertex degree which represents the number of cited authors varies largely from each other. The potential benefit is substantial when using CDP implementations for these benchmarks.

On the other hand, some applications are showing no or negative potential improvement. For example, BFS, CLR and SSSP with USA road network do not achieve any WEEI increase at all. The reason is that the degree of vertices in USA road network graph generally ranges from one to four, which does not trigger the condition to launch a new child kernel (recall that at least 32 threads are needed in a child kernel). These benchmarks already have high WEE because of the relatively balanced workload among threads and CDP implementations would not be necessary. It is even more interesting to notice that BFS and CLR for graph cage15 have WEEI decreased from WEE-nonCDP. The reason is that cage15 have relatively small variance in vertex degree. Launching a child kernel for some vertices but not for others actually intensifies the workload imbalance problem,

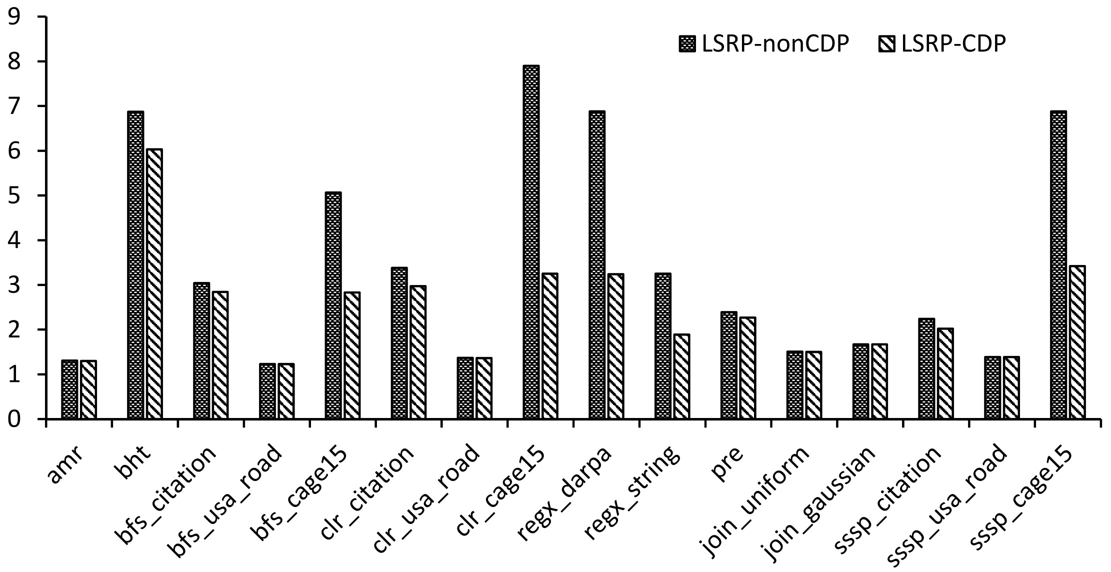


Figure 4: Average number of load/store instructions replay.

which results in decrease of WEE.

WEE-CDP shows the real measurement of WEE for CDP implementation. When including the *kernel launching overhead*, the SIMD lane efficiency decreases dramatically. Compared to WEE-nonCDP, WEE-CDP decreases from 3.5% to 20.2%. One hypothesis is *kernel launching overhead* introduces a large number of instructions with very low SIMD lane utilization and bring down the overall WEE (see kernel launching time analysis in section 3.4.2.4). The more child kernels are launched to increase WEEI, the more overhead is introduced and the larger drop-down can be observed from WEEI to WEE-CDP.

Insight. For unstructured applications that exhibit severe workload imbalance and relatively high dynamic parallelism, CDP can potentially reduce control flow divergence. However, for applications like BFS_cage15 and CLR_cage15 that do not have high thread-level workload variance, CDP does not show performance advantages. A strategy can be envisioned for invoking CDP based on the degree of workload variance.

3.4.2.2 Memory Behavior

Figure 4 shows LSRP for non-CDP implementation and CDP implementation to interpret the memory access irregularity. For all the benchmarks, CDP implementations reduce LSRP up to 58.8%. BFS, CLR and SSSP for cage15 and REGX have the most significant LSPR decrease among all benchmarks. These benchmarks have more scattered memory access by each thread within a warp in the non-CDP implementations. For example, the graph cage15 have a distributed neighbor list so the vertex expansion from different threads access vertices far away from each other, generating many memory transactions. By using CDP to handle DFP in unstructured applications, threads in the child kernel executing the same memory instruction are more likely to access contiguous addresses. Memory divergence can be greatly reduced for these benchmarks since more coalesced memory accesses are generated.

The graph citation network, road network and PRE, on the other hand, do not show much change in LSRP. They have the characteristic that neighbor vertices are stored close to each other in the memory (Citations tends to be from the same list of authors for a research area, nearby cities are connected together in the road network and for the PRE system, people are more likely to choose similar items), so even the original non-CDP implementation does not exhibit much memory access irregularity. CDP implementations do not show much benefit in these cases.

The LSRP behaviors for AMR and JOIN have different explanations. Unlike other benchmarks that traverse some irregular data structures which may generate non-coalesced memory accesses, AMR and JOIN have irregular data computation procedures by following dynamic and data-dependent execution paths rather than irregular memory access patterns. They would also exhibit low memory divergence and not take advantage from CDP implementations in terms of LSRP.

The L2 cache hit rate is also measured with the metric L2HIT as shown in Figure 5. Most of the benchmarks show unchanged or decreased L2 cache hit rate due to the fact that

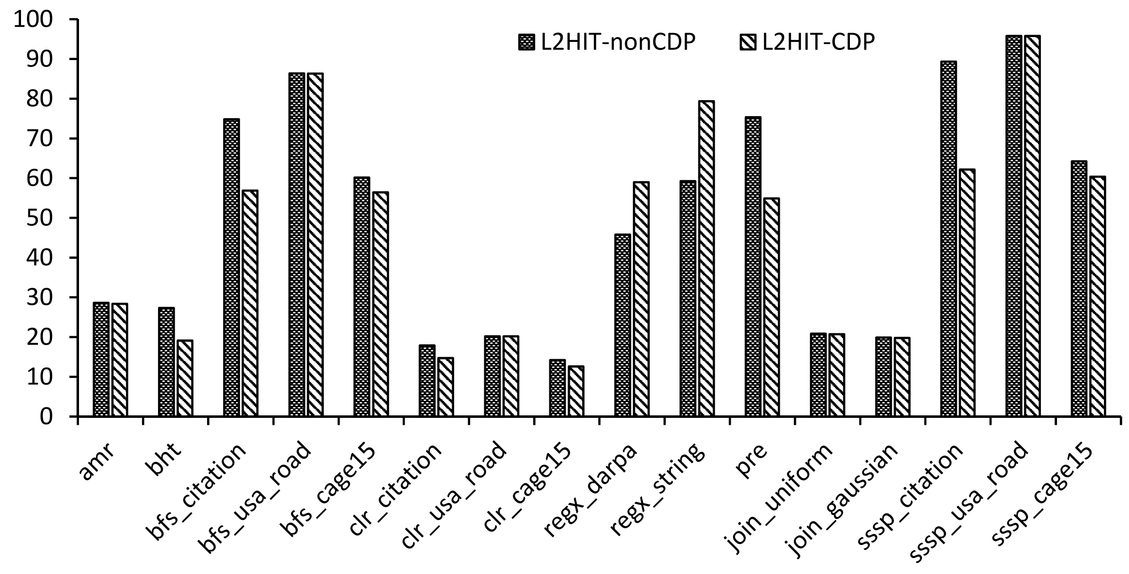


Figure 5: L2 cache hit rate.

CDP implementations break the spatial locality found in the non-CDP implementations where each thread may access contiguous addresses in different loop iterations.

The exceptions are two REGX benchmarks which show 13.2% and 20.1% cache hit rate increase respectively. Considering LSRP is also increased, their behaviors demonstrate the CDP implementations reserve both spatial locality within a thread and across the intra-warp threads. In these cases the child kernels with close memory address accesses are scheduled together, thereby increasing the cache hit rate.

Insight. Depending on the data arrangement and access patterns, CDP may reduce memory divergence by generating more coalesced memory accesses. However, it could reduce cache hit rate since accesses that were serialized in time in a non-CDP implementation now are redistributed across child kernels that execute concurrently. This can be mitigated by sophisticated child kernel scheduling policies much for the same reasons interleaved warp scheduling is effective at hiding memory latency.

3.4.2.3 Overall Performance

To evaluate the overall performance of the benchmarks using CDP implementation, the execution time of the computation kernels of the applications are measured. Note that the data transfer time between CPU and GPU is excluded. The following is the approach used to measure the ideal CDP implementation time excluding the *kernel launching overhead*. First, each child kernel is replaced with a dummy kernel that has an empty function body and the overall execution time is measured as $t1$. Then all the child kernel launches are removed to measure the execution time $t2$. In both cases, a warm up kernel is executed before the parent kernel to fill in the result data so that the execution paths of the parent kernel do not change. The time $t1-t2$ is used as the ideal child kernel launching time and is excluded from the actual CDP implementation execution time to generate the ideal execution time as a lower bound. Note the $t1 - t2$ depends on the number of child kernels which is determined by the patterns of dynamic parallelism in each application. While the most straightforward CDP implementation is chosen without explicitly controlling the number of child kernels, more sophisticated implementations are possible and could potentially reduce the overhead.

The speedup of both CDP ideal and actual execution time over non-CDP implementations are shown in Figure 6. BFS, CLR and SSSP with USA road network input shows no speedup or slow down for both scenarios because the child kernels launching threshold is never satisfied and no child kernel is launched. Other benchmarks show 1.13x-2.73x speedup for CDP-ideal. REGX_darpa and REGX_string have highest ideal speedup 1.96x and 2.73x respectively, which can be explained through the observation that CDP implementations have both positive impact on WEEI and LSRP. However, when including the *kernel launching overhead*, no benchmark can perform better than the non-CDP implementation with an average of 1.21x slow down. An interesting fact is that the higher speedup of CDP-ideal over non-CDP, the more slowdown of CDP-actual over non-CDP, since applications that can take more advantage of CDP implementations have more child kernel

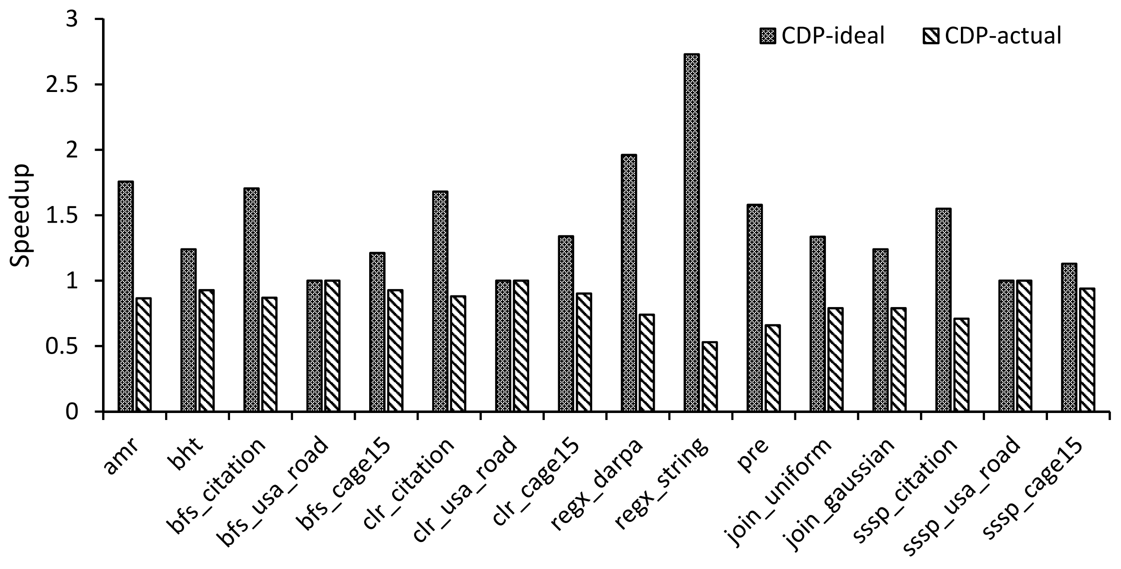


Figure 6: Speedup of CDP implementations (ideal and measured) of unstructured applications over non-CDP implementations.

launches and incur more overhead.

Insight. As the CDP implementations manage to reduce both control flow and memory access irregularity which are two essential metrics that affect the performance on the GPU, execution speed up is expected. However, with CDP support on GPUs in its current form, the overhead of device-side kernel launches have a substantially negative influence on the overall performance negating those gains.

3.4.2.4 CDP Overhead

As discussed in the previous sections, CDP implementations introduce substantial overhead which may negate potential performance benefit brought by handling DFP using device-side kernel launches. Such overhead is characterized in different aspects to get a comprehensive understanding of CDP.

CDP Launching Time The *CDP launching time* is measured using $t1 - t2$ with different thread numbers in the parent kernel to control the total number of child kernel launches.

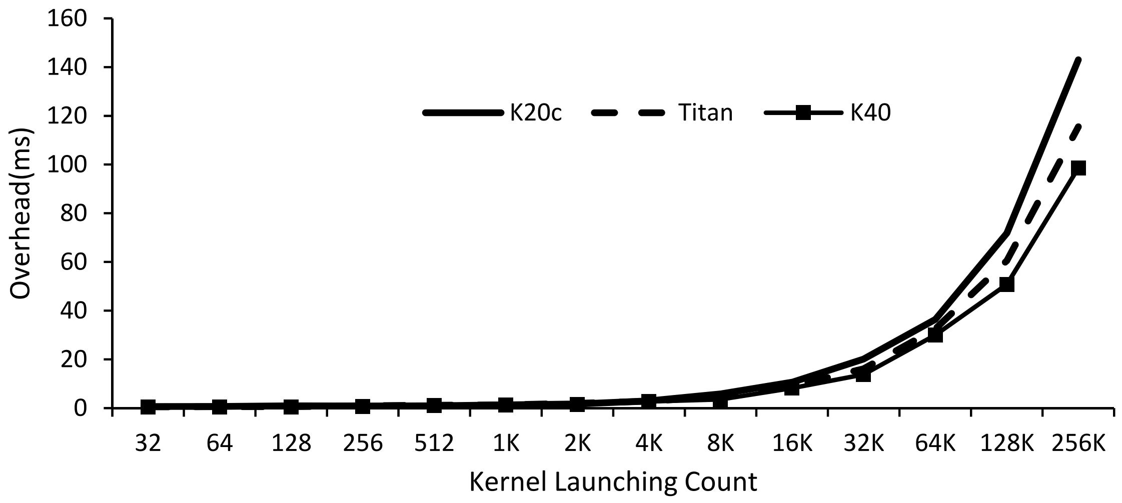


Figure 7: CDP launching time.

The output PTX code (assembly generated by compiling CUDA code with the nvcc compiler) is examined to make sure the dummy child kernels are not eliminated by the compiler optimizations. Recall that the *CDP launching time* includes time spent on kernel parameter parsing, calling `cudaGetParameterBuffer` and `cudaLaunchDevice`, as well as the time for device runtime to setup, enqueue and dispatch the child kernels. The time spent on data dumping by the parent kernel to pass data to the child kernels is excluded from *CDP launching time*.

Figure 7 shows the result for different child kernel count across three different GPU platforms. For all three GPUs, the *CDP launching time* stays around 1ms for kernel launching count from 32 to 512. Then it scales with the kernel launch count and reaches 143.3ms, 115.5ms and 98.57ms for 256K child kernel launches on K20c, Titan and K40 respectively (in comparison, the execution time of a typical kernel in the non-CDP implementation of BFS_citation is 3.27ms). The same method to measure the *kernel launching time* for each benchmark is used and the ratio over the overall execution time is computed and shown in Figure 6 which has average value of 36.1% and max value of 80.6%. The common problem for CDP implementation of the unstructured applications is that they require a large number of child kernel launches but the computation workload in each child kernel is very

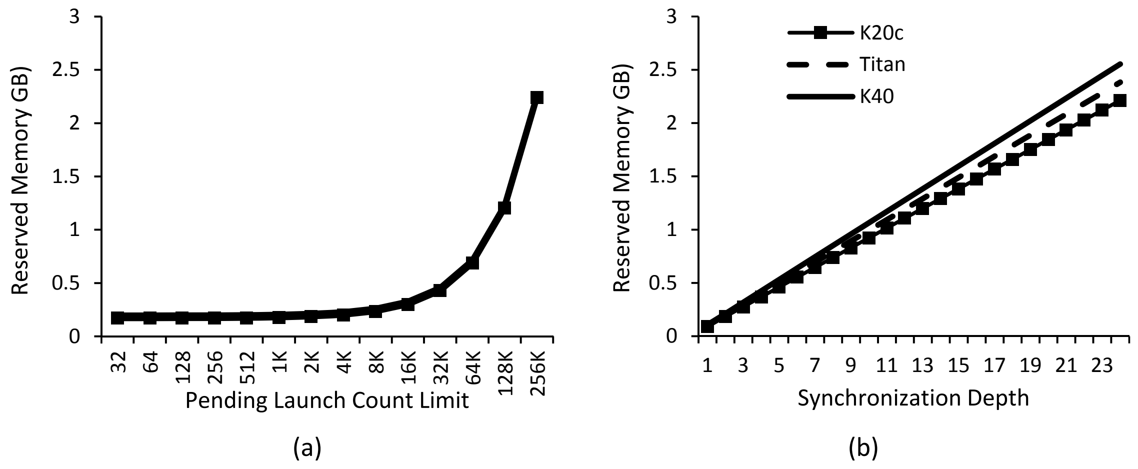


Figure 8: Reserved global memory for CDP kernel launch and synchronization.

light. As the launching time scales with the number of child kernels, the performance can dramatically degrade.

Memory footprint When using CDP, global memory in GPUs may be reserved by device runtime for child kernel launch. Device runtime maintains a kernel launching pool for all the launched but pending execution kernels due to unresovled dependency or lack of resources. The size of this pool is referred as *pending launch count limit* and can be specified using `cudaDeviceSetLimit` with `cudaLimitDevRuntimePendingLaunchCount` as its option. CDP execution reports a runtime error if the number of kernels pending execution on the fly exceeds this limit. For every pending launched child kernel, the device runtime uses reserved memory to store the launching information such as the parameters and configurations. On the other hand, CDP allows parent kernels and child kernels to explicitly synchronize with each other by calling `cudaDeviceSyncrhonize`. The device runtime has to save the states of parent kernels when they are suspended and yield to the child kernels at the synchronization point. The reserved memory size depends on the synchronization depth which can be specified using `cudaDeviceSetLimit` with the option `cudaLimitDevRuntimeSyncDepth`. Again, CDP execution reports an runtime error if the actual synchronization depth exceeds the limit.

The reserved memory size is measured by calling the runtime API `cudaMemGetInfo` before and after `cudaDeviceSetLimit` and compute the free memory size difference. Figure 8 shows the memory footprint for both scenarios.

As shown in Figure 8(a), the memory size reserved stays the same for *pending launch count limit* less than 32, which are 172MB, 186MB and 202MB for K20c, Titan and K40 respectively. If the *pending launch count limit* does not exceed 32K, the memory reserved is less than 10% of the total global memory on K20c. The average minimum *pending launch count limit* required to execute each benchmark and the reserved memory size are shown in Figure 9. Again, the benchmarks show diverse behaviors. REGX_string requires 127K *pending launch count limit* and 1.2GB reserved memory. As discussed before, CDP implementation of REGX can greatly increase WEEI and decrease LSRP by launching many child kernels. As a tradeoff, it requires much more memory space reserved. To the extreme opposite, the graph USA road network requires zero *pending launch count limit* since the parallelism degree is very low in DFP and CDP is not activated. However, there are still 172MB reserved memory which is the minimum cost to pay to link against device runtime library with CDP functionality enabled.

Figure 8(b) shows that the memory reserved for synchronization scales linearly with the synchronization depth. The highest synchronization depth 24 requires 2.2GB global memory reservation on K20c which is 44% of the total available GPU memory. A close analysis at the measurement shows that for each increase in synchronization depth, the memory sizes reserved are 95MB, 102MB and 109MB for K20c, Titan and K40 respectively, which scale with the number of SMXs in each GPU. This is because when the parent kernels are suspended, all the data (including local, shared memory data and etc.) currently occupying each SMX should be saved. As the three GPUs have the same SMX architecture, the total amount of reserved memory should be that of each SMX multiplied by the SMX count.

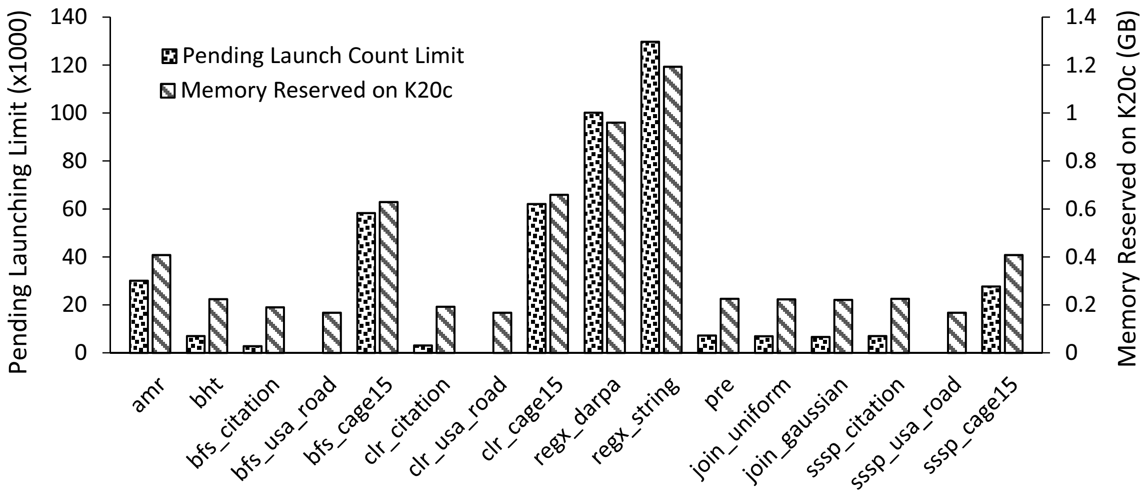


Figure 9: Pending launch count limit and reserved memory size.

Algorithm overhead Besides the overhead caused by the device runtime, sometimes the algorithm itself has to be changed for CDP implementation and may introduce overhead. Shared memory usage in a parent kernel can be tricky as the only ways child kernels can access the data is either through child kernel parameters or expensive global memory bypass. Therefore, algorithm has to be adapted to reduce shared memory passing between the parent and the children.

Spatial ordering requirement is another source that may introduce overhead. For example, in the JOIN benchmark, each block uses prefix-sum to compute the output offset for the result data since JOIN requires them to be strictly ordered. While in the CDP implementation, two prefix-sums are required instead of only one in the non-CDP implementation. One is used before the child kernel launch to compute the offset for child kernel output data, and the other one is required after the child kernel launch to compute the offset for remaining data that are not generated by the child kernel.

Insight. CDP introduces multiple sources of overhead from algorithm to device runtime management. The memory footprint reduces available global memory which can be a critical problem for large HPC applications. Both the *kernel launching overhead* and memory

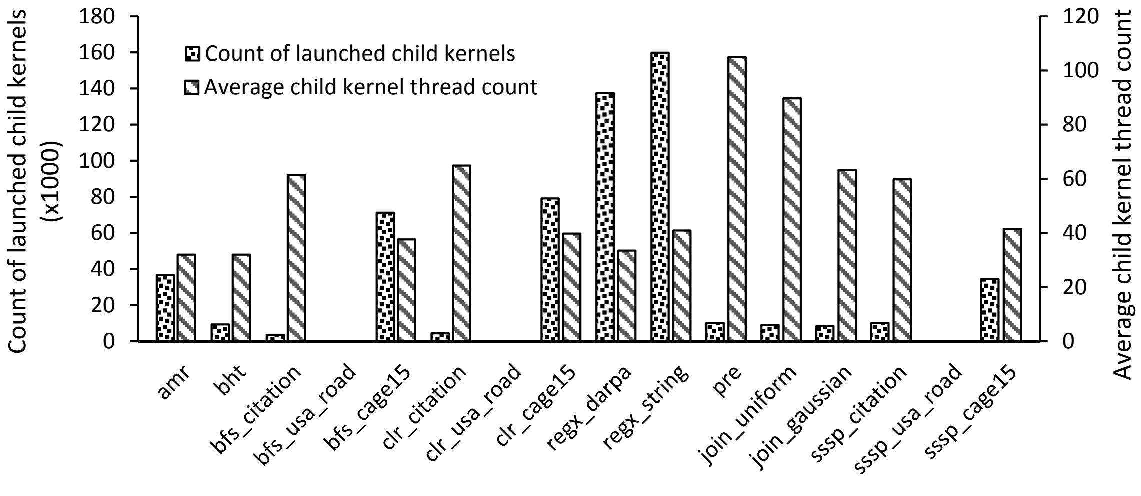


Figure 10: Total child kernel launching count and their average thread count.

footprint scale with the number of launched child kernels. To reduce the overhead requires either the programmers to decrease child kernel launch count by developing more performance-aware algorithms for CDP implementation, or the GPU architecture and software stack to advance the technology for reducing the time and space overhead of device-side kernel launching.

3.4.2.5 Child Kernel Workload Intensity and Scheduling

The child kernel launching traces generated by NVProf is investigated to understand the child kernel workload intensity and scheduling efforts. First the maximum number of child kernels launched by a parent kernel is counted, together with the average thread number in these child kernels in each benchmark, is shown in Figure 10. It can be noted that while a very large amount of child kernels are launched (up to 156K as in REGX_string), they are generally fine-grained kernels that perform very light workload (average kernel thread count is 44). Also note that the number of child kernels launched is only slightly larger (average 1.3x) than *pending launch count limit* shown in Figure 9, which implies that most child kernels are launched together in a short period of time to quickly fill the launching pool.

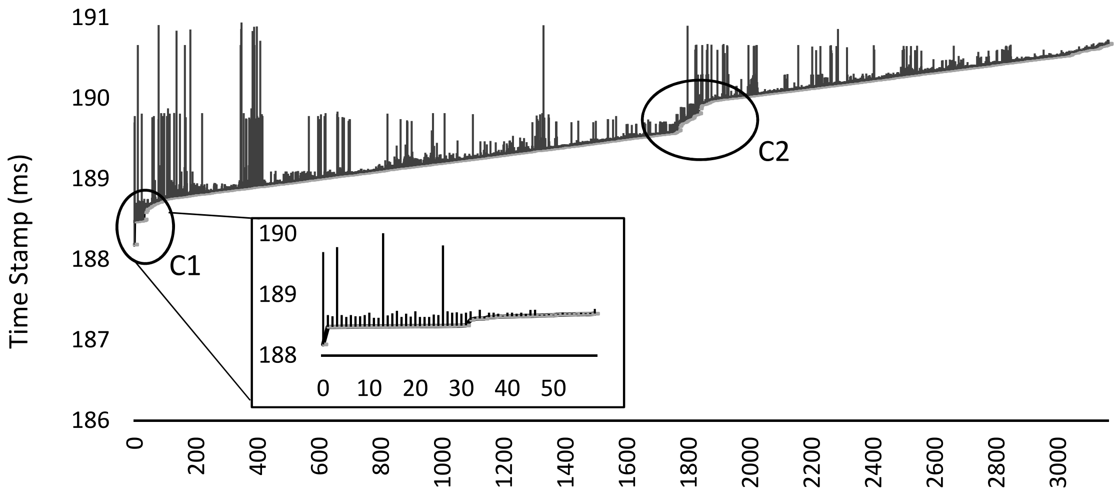


Figure 11: Kernel execution trace for BFS_citation iteration 5.

Then the time stamp of the parent and child kernels for one iteration in BFS_citation benchmark is shown in Figure 11. Each vertical line in the figure marks the start and end execution time of a kernel. The first line is for the parent kernel and the remaining are for the child kernels. The general trend shown by the figure is that kernels are scheduled and executed with increasing time stamps until completion, which conform to the fact that child kernels are launched when resources are available. There are two stages shown in the figure that present dramatic increase in time stamp, denoted by C1 and C2. C1 marks the early stage of the application, when the parent kernel starts launching several child kernels. A close look at C1 shows that 1) child kernels start execution before the parent kernel is finished and 2) child kernels are executed concurrently (31 child kernels start execution at the same time). C2 marks the stage when the TBs in the parent kernel start processing a new portion of the input vertices and generate a new round of child kernel launches. It shows that previous round of child kernel launches are gradually completed followed by the concurrent execution of newly launched kernels.

Insight. Using CDP often leads to more fine grained kernels compared to the host-side launched kernels, i.e., CDP implementations can generate a large number of child kernel launches, where often each kernel is relatively fine grained. This makes performance more sensitive to kernel level concurrency, kernel level scheduling policies, and kernel launching overhead. Alternatively, application developers may wish to be cognizant of, and sensitive to, kernel level granularity when making nested kernel calls on the GPU.

3.5 Characteristics of DFP

The above experimental evaluations of CDP implementations identify the following characteristics of DFP.

High Kernel Density: Depending on the problem size, DFP in irregular applications can show substantially high density where a large number of device kernels are launched. The high DFP density results in high *kernel launching overhead*.

Low Compute Intensity: Device kernels launched for DFP are usually fine-grained and have relatively low degrees of parallelism. Measurements across several irregular applications show that the average number of threads in each device-launched kernel is around 40 which is close to the warp size.

Workload Similarity: As DFP may exist within each thread in a kernel, and all threads are identical, the operations performed by each dynamically launched kernel are usually similar. However their instantiation may be with different degrees of parallelism. As per the nature of DFP, most device-launched kernels invoke the same kernel function but can have different configurations and parameter data.

Low Concurrency and Scheduling Efficiency: DFP generated by different threads are independent of each other and are implemented by launching device kernels through different software streams to enable concurrency. However, current kernel scheduling strategies on the GPU imposes a limit on kernel-level concurrency where the maximum number of kernels that can be executed concurrently is 32 in the GK110 architecture. As fine-grained device kernels can only be scheduled and executed concurrently up to this limit, there may not be enough warps to fully occupy the SMX. The limited warp concurrency could potentially cause either low utilization (if some of the SMXs are not assigned with any warps) or poor memory latency hiding ability (if SMXs are assigned with small number of warps).

3.6 Summary

This chapter studies the dynamically formed structured data parallelism in unstructured applications and implement them with the new CUDA Dynamic Parallelism technique on GPUs. A set of metrics are used to evaluate and analyze the potential performance benefit of the CDP implementations on the control flow behavior and memory behavior on several unstructured benchmark applications. This chapter also presents a comprehensive understanding of the efficiency of CDP in terms of runtime, memory footprint and algorithm overhead. The experiments show that CDP implementation can achieve 1.13x-2.73x potential speedup but the large kernel launching overhead could negate the performance benefit and impose a barrier to realizing a highly-efficient dynamic parallelism execution model.

The subsequent chapters of this thesis propose a solution to process DFP according to its characteristics with a set of optimizations to achieve overall performance improvement for irregular applications on the GPUs.

CHAPTER IV

DYNAMIC THREAD BLOCK LAUNCH

To address the challenges imposed by the current form of dynamic parallelism support on GPU while utilizing the DFP features in the irregular applications, this chapter introduces the Dynamic Thread Block Launch (DTBL) execution model by explaining the motivation, illustrating the mechanism, proposing the architecture extension, analyzing the potential benefits and demonstrating its performance advantage by experimental results.

4.1 Motivation

While current support of device-side kernel launch on the GPU in the form of CDP provides substantial productivity for handling DFP, the major issues of kernel launching overhead, large memory footprint, and less efficient kernel scheduling prevent the performance effective utilization of this functionality.

This chapter proposes to extend the current GPU execution model with *DTBL* where thread blocks rather than entire kernels can be dynamically launched from a GPU thread. Thread blocks (TBs) can be viewed as light weight versions of a kernel. A kernel can make nested TB calls on demand to locally exploit small pockets of parallel work as they occur in a data dependent manner. When a GPU kernel thread launches a TB, it is queued up for execution along with other TBs that are initially created by the kernel launch. Using DTBL, the set of TBs that comprise a kernel are no longer fixed at launch time but can vary dynamically over the lifetime of a kernel.

As this chapter will demonstrate, the dynamic creation of thread blocks can effectively increase the SMX occupancy, leading to higher GPU utilization. The dynamic TB launch overhead as well as memory footprint are significantly lower than that of kernel launch. Thus, DTBL enables more efficient support of irregular applications by introducing a light weight mechanism to dynamically spawn and control parallelism.

4.2 DTBL Execution Model

The execution model of DTBL allows new TBs to be dynamically launched from GPU threads and coalesced with existing kernels for scheduling efficiency. The current GPU BSP execution model is extended with several new concepts and terms to support these new features.

Figure 12 shows the execution model and thread hierarchy of DTBL. Any thread in a GPU kernel can launch multiple TBs with a single device API call (see later in this section). These TBs are composed as a single *aggregated group* utilizing a three dimensional organization similar to those of a native kernel. An aggregated group is then coalesced with a kernel - this simply means the TBs in the aggregated groups are added to the existing pool of TBs remaining to be scheduled and executed for that kernel. In fact, an aggregated group may be coalesced with the kernel of the parent thread (Figure 12a) *or* with another kernel (Figure 12b). In either case, the newly generated aggregated group execute the same function code as the kernel with which it is coalesced, and may have different input parameter values. Multiple aggregated groups can be coalesced to a single kernel.

In DTBL, coalescing is essential to increasing the TB scheduling performance due to i) TBs with the same configuration can be scheduled together to achieve the designed occupancy for the original kernel, possibly leading to higher GPU utilization and ii) coalesced aggregated groups only require one common context setup including kernel function loading, register and shared memory partitioning which can reduce the scheduling overhead. More details are described in the microarchitecture later in Section 4.3.

The kernel that is initially launched either by the host or by the device using the kernel launching API is called a *native kernel*. The TBs that compose the native kernel are *native TBs*. TBs in an aggregated group are called *aggregated TBs*. When a native kernel is coalesced with new aggregated groups, it becomes an *aggregated kernel*.

The idea of DTBL can be illustrated with two examples. The first example is Adaptive Mesh Refinement (AMR) corresponding to the execution model in Figure 12a. The DTBL

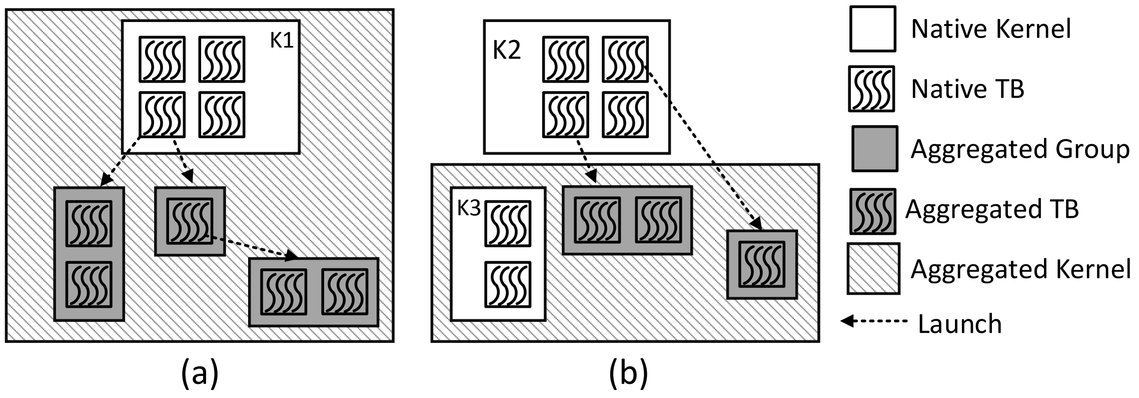


Figure 12: DTBL execution model and thread hierarchy where (a) shows the aggregated groups launched by kernel K1 are coalesced to itself and (b) shows the aggregated groups launched by kernel K2 are coalesced to another kernel K3.

implementation uses a native kernel K1 for the initial grid where each thread may launch nested aggregated groups for recursively refining the cells that are processed by the thread. All the new aggregated groups are then coalesced with K1 which become one aggregated kernel. The second example is BFS corresponding to the execution model in Figure 12b where a parent kernel K2 assigns threads to all the vertices in the vertex frontier and each parent thread may launch new TBs to expand the vertex neighbors. The kernel K3 is a native kernel previously launched by the host or the device for vertex expansion. The new TBs generated by K2 are coalesced to K3 rather than the parent.

Thread Hierarchy Within an Aggregated TB: As in GPUs today, DTBL uses a three-dimensional thread index to identify the threads in an aggregated TB. When coalesced to a native kernel, the number of threads in each dimension of an aggregated TB should be the same as that of a native TB. Therefore, aggregated TBs use the same configuration and the same amount of resources as native TBs, minimizing the overhead when scheduled on an SMX.

Aggregated TB Hierarchy Within an Aggregated Group: An aggregated group in DTBL is analogous to a device-launched kernel. Within an aggregated group, aggregated TBs are organized into one/two/three dimensions, identified by their three-dimensional TB indices. The value of each TB index dimension starts at zero. Similar to launching a device kernel,

Table 3: List of device runtime API calls for DTBL

Device Runtime API Calls	Description
`cudaGetParameterBuffer`	Reused from the original CUDA device runtime library to allocate parameter buffer for a new aggregated group.
`cudaLaunchAggGroup`	A new API call introduced by DTBL programming interface which launches a new aggregated group.

the programmers supply data addresses through parameters and use TB indices within an aggregated group as well as thread indices within an aggregated TB to index the data values used by each thread.

Synchronization: DTBL uses the same synchronization semantics as the current GPU execution model, i.e., threads within an aggregated TB can be synchronized explicitly by calling a barrier function. However, like the base programming model no explicit barrier is valid across native or aggregated TBs. Unlike the parent-child synchronization semantics in the device-kernel launching model in CDP, aggregated groups cannot be explicitly synchronized by its invoking kernel. Therefore, it is the programmers' responsibility to ensure the correctness of the program without any assumption on the execution order of aggregated groups. When various irregular applications are implemented using device kernel launching, any explicit synchronization between the child and parents is avoided due to its high overhead in saving the parent state to the global memory, so these applications can be easily adapted to the new DTBL model. A more thorough analysis of the usage of explicit synchronization is left as future work.

Memory Model: DTBL also preserves the current GPU memory model, i.e., global memory, constant memory and texture memory storage are visible to all native and aggregated TBs. Shared memory is private to each thread block and local memory is private to each thread. No memory ordering, consistency, or coherence is guaranteed across different native or aggregated TBs.

```
__global__ parent(…) {
  cudaStream_t s;
  cudaStreamCreateWithFlags(&s, …);
  void *buf=cudaGetParameterBuffer();
  …… //fill the buf with data
  cudaLaunchDevice(child, buf,
    grDim, tbDim, sharedMem, s);
}

__global__ child(…) {
}
```

(a)

```
__global__ parent(…) {


  void *buf=cudaGetParameterBuffer();
  …… //fill the buf with data
  cudaLaunchAggGroup(child, buf,
    aggDim, tbDim, sharedMem);
}

__global__ child(…) {
}
```

(b)

Figure 13: Example code segments for (a) CDP and (b) DTBL

Programming Interface: DTBL defines two device runtime API calls on top of the original CUDA Device Runtime Library for CDP as listed in Table 3. The first API call `cudaGetParameterBuffer` is the same as in the original device runtime library that is used to allocate parameter space for an aggregated group. On the other hand, the API call `cudaLaunchAggGroup` is newly defined for dynamically launching an aggregated group. Programmers can pass the kernel function pointer when calling this API to specify the kernel to be executed by and possibly coalesce with the new TBs. Similar to the device kernel launching API call `cudaLaunchDevice` in CDP, `cudaLaunchAggGroup` configures the new aggregated group with thread and TB numbers in each dimension, shared memory size, and parameters. Note that unlike a device kernel launching which should be configured with an implicit or explicit software stream to express dependency on other kernels, the aggregated thread groups are automatically guaranteed to be independent of each other. Example code segments for both CDP and DTBL implementations are shown in Figure 13 where a parent kernel launches child kernels in CDP and corresponding aggregated groups in DTBL. The similarity between the two code segments demonstrate that DTBL introduces minimal extensions to the programming interface.

4.3 Architecture Extensions and SMX Scheduling

To support the new DTBL execution model, the GPU microarchitecture is extended to process the new aggregated groups that are launched from the GPU threads. The baseline microarchitecture maintains several data structures for keeping track of deployed kernels and the TBs that comprise them. These data structures are extended to keep track of dynamically formed aggregated groups and associating them with active kernels. This is achieved in a manner that is transparent to the warp schedulers, control divergence mechanism, and memory coalescing logic. Figure 14 illustrates the major microarchitecture extensions to support DTBL. With the extended data structure and SMX scheduler, new aggregated groups are launched from the SMXs, coalescing to existing kernels in the Kernel Distributor and scheduled to execute on SMX with all other TBs in the coalesced kernel. The detailed procedure and functionality of each data structure extension are described as follows.

Launching Aggregated Groups

This is the first step that happens when the aggregated group launching API is invoked by one or more GPU threads. The SMX scheduler will react correspondingly to accept the new aggregated groups and prepare necessary information for TB coalescing in the next step.

Similar to the device kernel launching command, DTBL introduces a new aggregation operation command in the microarchitecture. This command will be issued when the aggregated group launching API calls are invoked simultaneously by one or more threads within the same warp. These aggregated group launches are then combined together to be processed by the aggregation operation command.

For each newly formed aggregated group, the SMX allocates global memory blocks through the memory controller① to store the parameters and configuration information②. The request procedure is the same as that of a device-side kernel launch. After parameters are loaded to the parameter buffer, the SMX passes the aggregation operation command to the SMX scheduler with the information for each aggregated group③.

Thread Blocks Coalescing

In this step, the SMX scheduler receives the aggregation operation command and attempts to match the newly launched aggregated groups with the existing kernels in the Kernel Distributor Entries (KDE) for TB coalescing based on aggregated group configurations. If the coalescing is successful, the SMX scheduler will push the new TBs in a scheduling TB pool for the corresponding kernel. The scheduling pool is implemented with several registers in microarchitecture to form a linked-list data structure for efficient TB scheduling. The process is implemented as a new part of the DTBL scheduling policy④ which is illustrated in Figure 15 and described in the following.

For each aggregation group, the SMX scheduler first searches the KDE to locate any existing eligible kernels that can accept the new TBs⑤. Eligible kernels should have the same entry PC addresses and TB configuration as the aggregated group. If none are found, the aggregated group is launched as a new device kernel. Experiments show that an aggregated group is able to match eligible kernels on average 98% of the time. Mismatches typically occur early, before newly generated device kernels fill the KDE.

If an eligible kernel is found, the SMX scheduler allocates an entry in the *Aggregated Group Table* (AGT) with the three-dimensional aggregated group size and the parameter address⑥. The AGT is composed of multiple Aggregated Group Entries (AGE) and serves to track all the aggregated groups. Aggregated groups that are coalesced to the same eligible kernel are linked together with the *Next* field of the AGE⑦. The AGT is stored on chip for fast accesses with a limit on the number of entries. When the SMX scheduler searches for a free entry in the AGT, it uses a simple hash function to generate the search index instead of a brute-force search. The hash function is defined as *ind = hw_tid & (AGT_size - 1)* where *hw_tid* is the hardware thread index in each SMX and *AGT_size* is the size of AGT. The

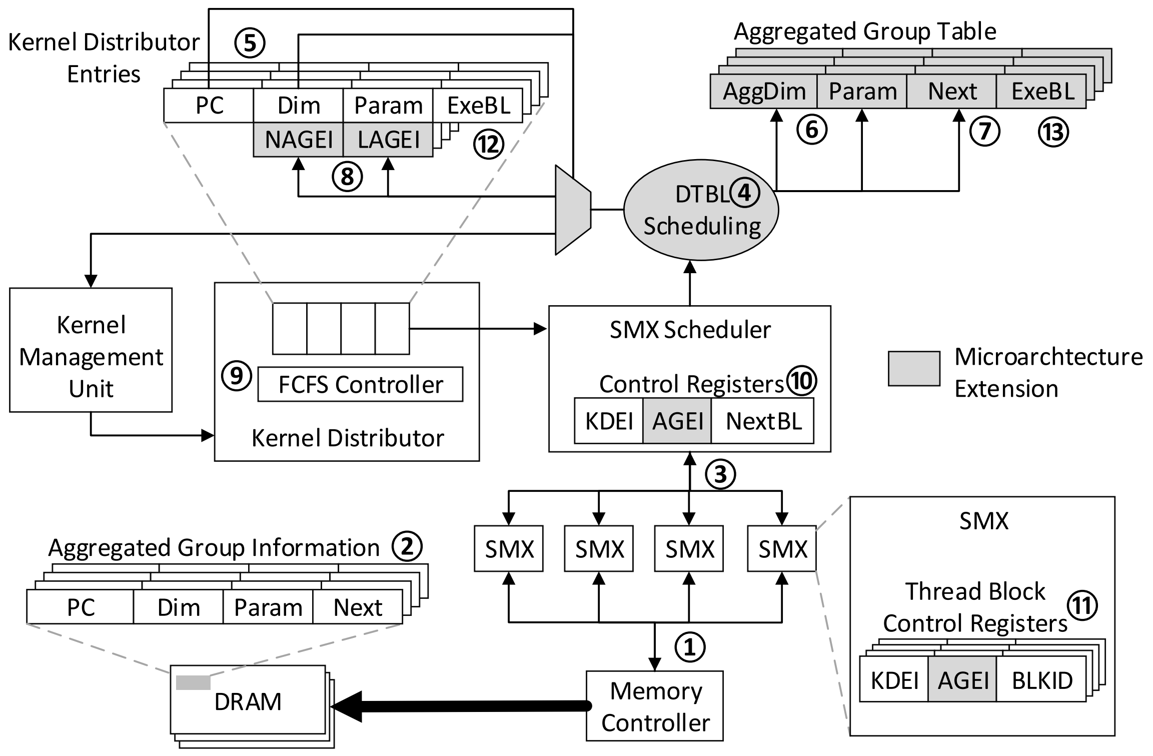


Figure 14: Microarchitecture Flow for DTBL

intuition behind the hash function is that all threads on an SMX have the same probability in launching a new aggregated group. The SMX scheduler is able to allocate an entry if the entry indexed by *ind* in AGT is free and the *ind* is recorded as aggregated group entry index (AGEI). Otherwise it will record the pointer to global memory where the aggregated group information is stored②.

Now that an eligible kernel and the corresponding KDE is found, the TBs in the new aggregated group are added to the set of TBs in the eligible kernel waiting be executed. The AGEI or the global memory pointer of the new aggregated group information is used to update the two KDE registers *Next AGEI* (NAGEI) and *Last AGEI* (LAGEI) if necessary⑧. NAGEI indicates the next aggregated group to be scheduled in the kernel. It is initialized when a kernel is newly dispatched to the Kernel Distributor to indicate no aggregated groups exists for the kernel. LAGEI indicates the last aggregated group to be coalesced to this kernel.

All the kernels in the Kernel Distributor are marked by the FCFS⑨ with a single bit

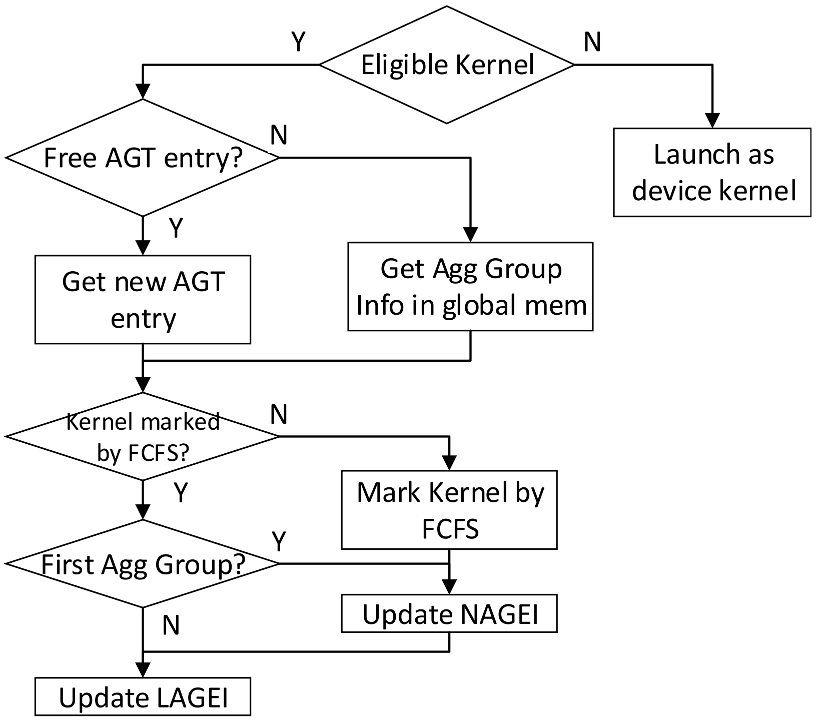


Figure 15: DTBL scheduling procedure in SMX scheduler

when they are queued to be scheduled and unmarked when all its TBs are scheduled. The FCFS controller is extended with an extra bit to indicate if it is the first time the kernel is marked by the FCFS. This is useful when the SMX scheduler attempts to update NAGEI under two different scenarios.

At the first scenario, when a new aggregated group is generated, the corresponding eligible kernel may have all its TBs scheduled to SMXs, be unmarked by the FCFS controller and only be waiting for its TBs to finish execution. In this case, the NAGEI is updated with the new aggregated group and the kernel is marked again by the FCFS controller so that the new aggregated group can be scheduled the next time the kernel is selected by the SMX scheduler.

At the other scenario, the eligible kernel is still marked by FCFS as it is either waiting in the FCFS queue or is being scheduled by the SMX scheduler. In this case, there are still TBs in the eligible kernel to be scheduled and NAGEI is only updated when the new aggregated group is the first aggregated group to be coalesced to this kernel.

Unlike NAGEI, LAGEI is always updated every time a new aggregated group is generated for the kernel to reflect the last aggregate group to be scheduled. With NAGEI, LAGEI and the *Next* field of AGE, all the aggregated groups coalesced to the same kernel are linked together to form a scheduling pool.

Aggregated Thread Blocks Scheduling on SMX

The last step in DTBL scheduling manages to schedule all the aggregated TBs on the SMXs. The SMX scheduler first determines whether the native kernel or a specific aggregated group should be scheduled according to the registers value generated by the previous step for the scheduling pool. Then it distributes the TBs in the kernel or the aggregated group to the SMXs with a set of registers to track their status. As described in the follows, this is implemented by updating the algorithm used by the baseline GPU microarchitecture to distribute and execute the native TBs.

When the SMX scheduler receives a kernel from the Kernel Distributor, it checks if it is the first time the kernel is marked by the FCFS controller. If so, the SMX scheduler starts distributing the native TBs followed by aggregated TBs pointed to by the NAGEI (if any). Otherwise it directly starts distributing the aggregated thread blocks pointed by NAGEI since the native TBs have already been scheduled when the kernel was previously dispatched by the FCFS controller. Another possibility is that the new aggregated groups are coalesced to a kernel that is currently being scheduled, the SMX scheduler will then continue to distribute the new aggregated groups after finishing distributing the TBs from the native kernel or current aggregated group. The SMX scheduler updates the NAGEI every time after finishing scheduling the current aggregated group and starts the next aggregated group indicated by the *Next* field of AGE pointed by NAGEI.

Once the SMX scheduler determines the native kernel or aggregated group to schedule, it records the corresponding index of KDE (KDEI) and AGEI in its control registers (SSCR)⑩. SSCR also has a *NextBL* field to store the index of the next TB to be distributed to the SMX. Note that since the TBs in the native kernel and the aggregated groups have the same configuration and resource usage as constrained by the DTBL execution model, the SMX scheduler can use a static resource partitioning strategy for both the native and aggregated TBs, saving the scheduling cost.

The SMX scheduler then distributes TBs to each SMX. The Thread Block Control Register (TBCR)⑪ on each SMX is updated correspondingly using the same value of *KDEI* and *AGEI* in SSCR to record the kernel index in the Kernel Distributor and the aggregated group index in the AGT so the SMX can locate the function entry and parameter address correctly for the scheduled TB. The *BLKID* field records the corresponding TB index within a kernel or an aggregated group. Once the TB finishes execution, the SMX notifies the SMX scheduler to update the *ExeBL* field in the KDE⑫ and AGE⑬ which track the number of TBs in execution.

When all the TBs of the last aggregated group marked by LAGEI have been distributed to an SMX, the SMX scheduler notifies the FCFS controller to unmark the current kernel to finish its scheduling. The corresponding entries in the Kernel Distributor or AGT will be released once all the TBs complete execution.

4.4 Overhead Analysis

The hardware overhead is caused by extra data structures introduced by the architectural extensions (shaded boxes in Figure 14). New fields in the KDE (NAGEI and LAGEI), FCFS Controller (the flag to indicate if the kernel has been previously dispatched), SSCR (AGEI) and SMX TBCR (AGEI) together take 1096 Bytes of on-chip SRAM. The size of AGT determines how many pending aggregated groups can be held on-chip for fast accesses. A 1024-entry AGT takes 20KB of on-chip SRAM (20Bytes per entry) which composes the major hardware overhead (about 0.5% of the area taken by the shared memory and registers on all SMXs). Section 4.6.2.4 analyzes the sensitivity of performance to the size of the AGT.

The major timing overhead of launching aggregated groups includes time spent on allocating parameters, searching the KDE and requesting free AGT entries. As discussed before, launching from the threads within a warp are grouped together as a single command. Therefore, the overhead is evaluated on a per-warp basis. The procedure of allocating a parameter buffer for an aggregated group is the same as that for a device-launch kernel, so the measurement is used directly from a K20c GPU. The search for eligible KDE entry can be pipelined for all the simultaneous aggregated groups launches in a warp, which takes a maximum of 32 cycles (1 cycle per entry). Searching for a free entry in AGT only takes one cycle with the hash function for each aggregated group. If a free entry is found, there will be zero cost for the SMX scheduler to load the aggregated group information when it is scheduled. Otherwise the SMX scheduler will have to load the information from the global memory and the overhead is dependent on the global memory traffic. It should be noted that allocating the parameter buffer and searching the KDE/AGE can happen in parallel, the slower of which determines the overall time overhead of aggregated group launching.

An alternative approach to the proposed microarchitecture extension is to increase the number of KDE entries so that each aggregated group can be independently scheduled from KDE. The argument is that the hardware overhead introduced by AGT could be potentially saved. However, there are also some major side effects for this approach.

First, since aggregated groups are scheduled independently, they are not coalesced so that TBs with different configurations are more likely to be executed on the same SMX. In consequence, the designed occupancy for the original kernels is less likely to be achieved and the execution efficiency could be decreased. For the same reason, the context setup overhead such as kernel function loading and resource allocation across SMXs could be increased. The context setup overhead is expected to scale with the number of aggregated group scheduled from KDE.

Second, hardware complexity and scheduling latency in the KMU and FCFS controller scales with number of KDE. For example, the number of HWQ could be increased to keep up the kernel concurrency, and the overhead for FCFS controller to track and manage the status of each aggregated group also increases linearly.

4.5 Benefits of DTBL

DTBL is beneficial primarily for the following three reasons. First, compared to device-side kernel launching, dynamic TB launches have less overhead. Instead of processing the device-side launching kernel command through a long path from the SMX to KMU and then to the Kernel Distributor, TBs are directly grouped with active kernels in the Kernel Distributor by the SMX scheduler. For irregular applications that may generate a large amount of dynamic workload, reducing the launch overhead can effectively improve the overall performance.

Second, due to the similarity of the dynamic workload in irregular applications, dynamically generated TBs are very likely to be coalesced to the same kernel which enables more TBs to be executed concurrently. Recall that the concurrent execution of fine-grained device kernels are limited by the size of Kernel Distributor. The DTBL scheduling breaks this limit as aggregated TBs are coalesced into a single native kernel that can take full advantage of the TB level concurrency on the SMX. This more efficient scheduling strategy may increase the SMX occupancy which is beneficial in increasing GPU utilization, hiding memory latency and increasing the memory bandwidth.

Third, both the reduced launch latency and increased scheduling efficiency helps to consume the dynamically launched workload faster. As the size of reserved global memory depends on the number of pending aggregated groups, DTBL can therefore reduce the global memory footprint.

4.6 Experiments and Evaluation

This section evaluates the performance of DTBL from various aspects by implementing and executing multiple CUDA irregular applications.

Table 4: GPGPU-Sim configuration parameters for DTBL simulation

SMX Clock Freq.	706MHz
Memory Clock Freq.	2600MHz
# of SMX	13
Max # of Resident Thread Blocks per SMX	16
Max # of Resident Threads per SMX	2048
# of 32-bit Registers per SMX	65536
L1 Cache / Shared Mem Size per SMX	16KB / 48KB
Max # of Concurrent Kernels	32

4.6.1 Methodology

The experiments are performed on the cycle-level GPGPU-Sim simulator [9]. The GPGPU-Sim is first configured to model the Tesla K20c GPU as the baseline architecture. The configuration parameters are shown in Table 4. The SMX scheduler is also modified to support concurrent kernel execution on the same SMX. The warp scheduler is configured to use the greedy-then-oldest scheduling policy [63]. As discussed before, the proposed microarchitecture extension is transparent to the warp scheduler so DTBL can take advantage of any warp scheduling optimization that is useful to the baseline GPU architecture.

To support the device-side kernel launch capability (CDP on K20c), the device runtime of GPGPU-Sim is extended with the implementation of corresponding API calls. The latency of these API calls which is part of the kernel launching overhead is modeled by performing the measurement on the K20c GPU with the `clock()` function and use the average cycle values from 1,000 measurements across all the evaluated benchmarks. According to the measurements, the API `cudaGetParameterBuffer` and `cudaLaunchDevice` have a linear latency model per warp basis denoted as $Ax + b$ where b is the initialization latency for each warp, A is the latency for each API called by one thread in the warp and x is the number of the threads calling the API in a warp. Note that the execution of the device API calls will be interleaved for all the warps so that some portion of the latency introduced can also be hidden by the interleaving, similar as the memory latency hiding. Besides the API latency, there is also a kernel dispatching latency (from KMU to Kernel Distributor), which

Table 5: Latency modeling for CDP and DTBL (Unit: cycles)

`cudaStreamCreateWithFlag` (CDP only)	7165
`cudaGetParameterBuffer` (CDP and DTBL)	b: 8023, A: 129
`cudaLaunchDevice` (CDP only)	b: 12187, A: 1592
Kernel dispatching	283

is measured using the average time difference between the end of the first kernel and the start of the second kernel that is dependent of the first kernel.

The accuracy of the simulation, especially the kernel launching overhead, is verified by running all the benchmarks both on the K20c GPU and the simulator and use the same correlation computation method by GPGPU-Sim. The proposed architecture extension for DTBL is implemented with overhead assignment described in Section 4.4 where the latency for parameter buffer allocation is the same as the `cudaGetParameterBuffer` API call and all other aggregated group launching latency is directly modeled by the microarchitecture extension. The latency numbers used in the simulator are shown in Table 5.

The benchmark applications used for evaluating DTBL is the same as the ones used in Chapter 3. However, some of the inputs to the benchmark are changed for a more complete performance analysis. The final list of benchmark applications are shown in Table 6. The original CUDA implementations are referred as *flat* implementations since the nested algorithmic structure is flattened and effectively serialized within each thread. An exception is the *bfs* implementation [47] where dynamic parallelism for DFP is implemented by employing TB and warp level vertex expansion techniques. For this application, the implementation uses CDP device kernels or DTBL aggregated group to replace the TB or warp level vertex expansion. The benchmarks with CDP are implemented in the way that a device kernel is launched for any DFP with sufficient parallelism available. The same methodology applies to DTBL except that a device kernel is replaced with an aggregated group for a fair comparison. Note that the data structures and algorithms of the original implementations are not changed in the CDP/DTBL implementations for a fair comparison. The proposed DTBL model for dynamic parallelism can also be orthogonal to many

Table 6: Benchmarks used in the experimental evaluation of DTBL

Application	**Input Data Set**
Adaptive Mesh Refinement (AMR)	Combustion Simulation[36]
Barnes Hut Tree (BHT) [15]	Random Data Points
Breadth-First Search (BFS) [47]	Citation Network[8] USA Road Network[8] Cage15 Sparse Matrix [8]
Graph Coloring (CLR) [19]	Citation Network[8] Graph 500 Logn20[8] Cage15 Sparser Matrix [8]
Regular Expression Match (REGX) [75]	DARPA Network Packets [45] Random String Collection
Product Recommendation (PRE) [49]	Movie Lens [28]
Relational Join (JOIN) [22]	Uniform Distributed Data Gaussian Distributed Data
Single Source Shortest Path (SSSP) [37]	Citation Network[8] Fight Network [70] Cage15 Sparser Matrix[8]

optimizations, e.g. worklist for work pulling and pushing to achieve high-level workload balance, as they can be applied in either flat or nested implementations.

DTBL only uses device runtime API for thread block launching and does not introduce any new instructions or syntax. Therefore, the CUDA compiler NVCC6.5 is used directly to compile the benchmarks. Extending the syntax to support higher-level programming interface similar as the CUDA kernel launching annotation “<<<>>>” is left as future work. The same dataset is used for both the GPU and the simulator. All the applications are executed entirely from the beginning to the end except for *regx*. The application *regx* is divided into several sections with the memory manually populated in GPGPU-Sim, and only computation kernels executed. Then all the computation kernels of the benchmarks are traced to generate the performance data.

4.6.2 Result and Analysis

This section reports the evaluation and analysis of the benchmark in various performance aspects.

4.6.2.1 Control Flow and Memory Behavior

The control flow behavior is evaluated using the warp activity percentage which is defined as average percentage of active threads in a warp as shown in Figure 16. The memory behavior is evaluated using DRAM efficiency which is computed as dram_efficiency = (n_rd+n_write)/n_activity where n_rd and n_write are the number of memory read and write commands issued by the memory controller and n_activity is the active cycles when there is a pending memory request. DRAM efficiency reveals the memory bandwidth utilization and increases when there are more coalesced memory accesses in a given period of time, as shown in Figure 17. On average, warp activity percentage of both CDP and DTBL increases 10.7% from the flat implementations and DRAM efficiency increases 0.029 or 1.14x for CDP and 0.053 or 1.27x for DTBL, demonstrating that one important benefit of both CDP and DTBL is to dynamically generate parallel workload for DFP that have more regular control flow and coalesced memory accesses.

Since both DTBL and CDP launch dynamic parallel workloads, they fundamentally behave the same in reducing control flow divergence and obtain the same amount of increase in warp activity percentage. Some benchmarks, such as *amr* and *join_gaussian*, have highly irregular computation workload and severe imbalance problem across the threads in their flat implementations and achieve most substantial increases in warp activity percentage (45.3% and 21.3%). Warp activity percentage also increases for the two *bfs* benchmarks. Although the baseline *bfs* implementation has already utilized TB-level and warp-level vertex expansion to handle dynamic parallelism, CDP and DTBL are able to use variable TB sizes to achieve even better workload balance. The benchmark *clr_graph500* does not show obvious changes and *clr_cage15* even shows a slight drop (-5.9%) because the graphs *graph500* and *cage15* already have relatively small variance in vertex degree that generate balanced workload even in the flat implementation. Launching dynamic parallel workloads for some vertices but not for others may break the original balance and cause more control flow divergence. This is consistent with the understanding that CDP and DTBL are

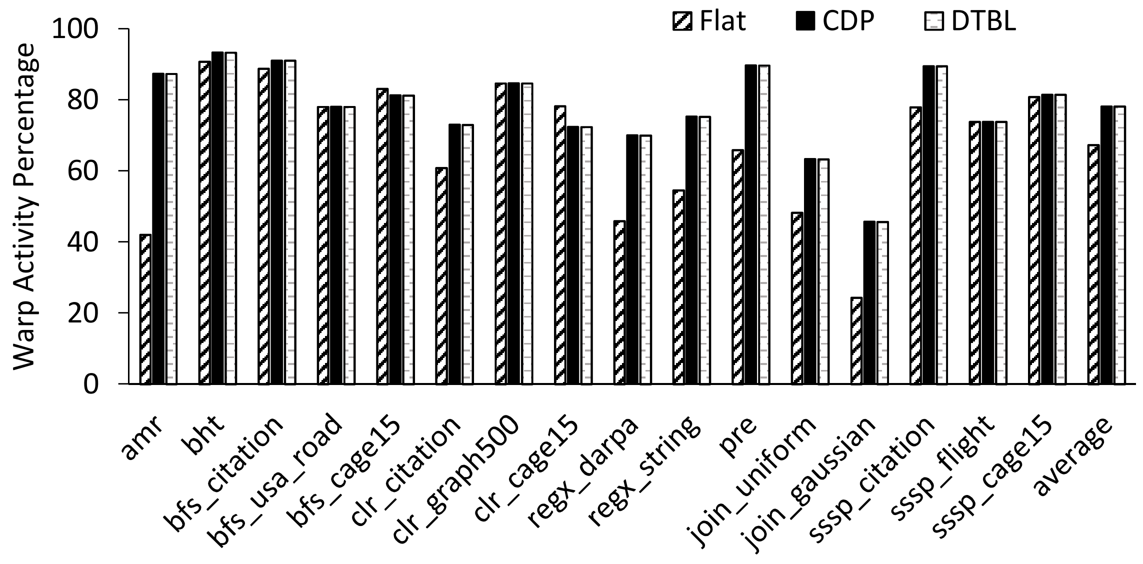


Figure 16: Average Percentage of Active Threads in a Warp

intended to work well over unbalanced workloads.

The *clr_cage15* and *sssp_cage15* are two benchmarks that achieve highest DRAM efficiency increase. In their flat implementations, as the graph *cage15* has a distributed neighbor list, the threads access vertex data far away from each other in memory and result in more non-coalesced memory accesses and memory transactions. In comparison, CDP and DTBL implement the DFP such that threads are more likely to access consecutive memory addresses. Memory irregularity could be significantly reduced in this case which is demonstrated by increasing DRAM efficiency.

4.6.2.2 *Scheduling Performance*

By coalescing dynamically generated TBs to existing kernels on the fly, DTBL is able to increase the TB-level concurrency for fine-grained parallel workloads. Lower launching latency for DTBL also contributes to the increase in the number of available TBs that can be scheduled concurrently by the SMX scheduler. Therefore, DTBL is able to outperform CDP by increasing SMX occupancy. The SMX occupancy is evaluated by measuring the average number of active warps in each cycle on all of the SMXs divided by maximum

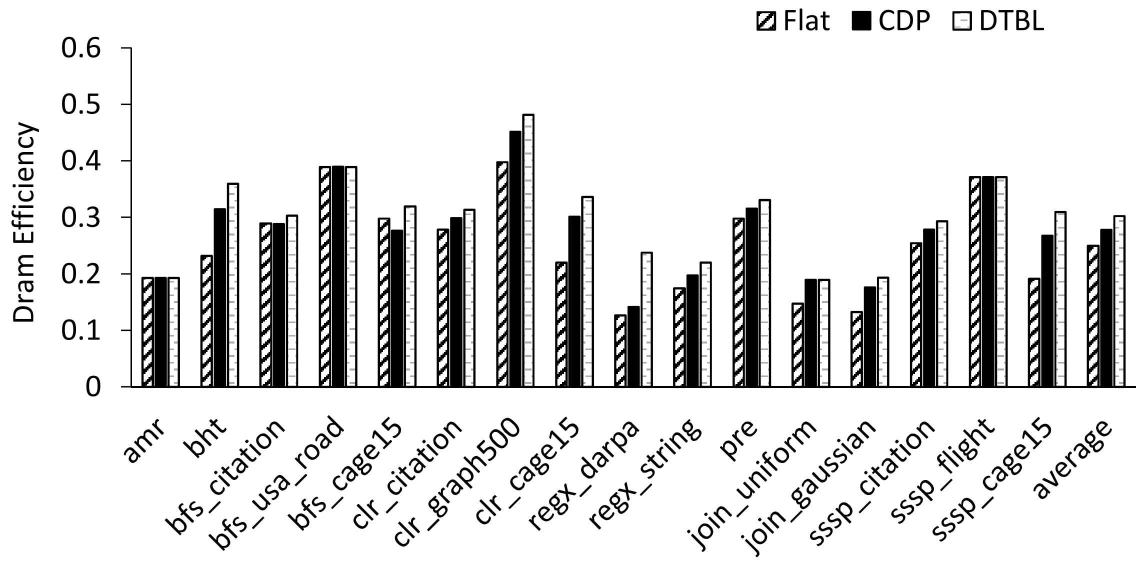


Figure 17: DRAM Efficiency

number of resident warps per SMX. The influences of scheduling strategy and launching latency are isolated by comparing the measurement with and without launching latency. The results are shown in Figure 18 where the SMX occupancy achieved by CDP and DTBL without modeling launching latency are denoted as CDP-Ideal (CDPI) and DTBL-Ideal (DTBLI) respectively. DTBLI has average of 17.9 or 1.24x increase over CDPI. The benchmark *bht* achieves the highest occupancy increases (24.6 or 1.38x) since it generates many fine-grained parallel workloads (average number of threads in a device kernel or an aggregated group is 33.4 which is close to warp size). Therefore, in its CDP implementation, the limited kernel-level concurrency on the GPU causes only a few threads to be active on GPUs which results in low SMX occupancy and utilization. DTBL, on the other hand, is able to aggregate all these fined-grained TBs together to fully utilize SMX with a higher occupancy. Other benchmarks that generate dynamic workloads with higher parallelism (coarse-grain workload) have less of a significant increase in SMX occupancy, represented by *pre* (0.46 or 1.01x) with an average 1527.9 threads in a device kernel or an aggregated group.

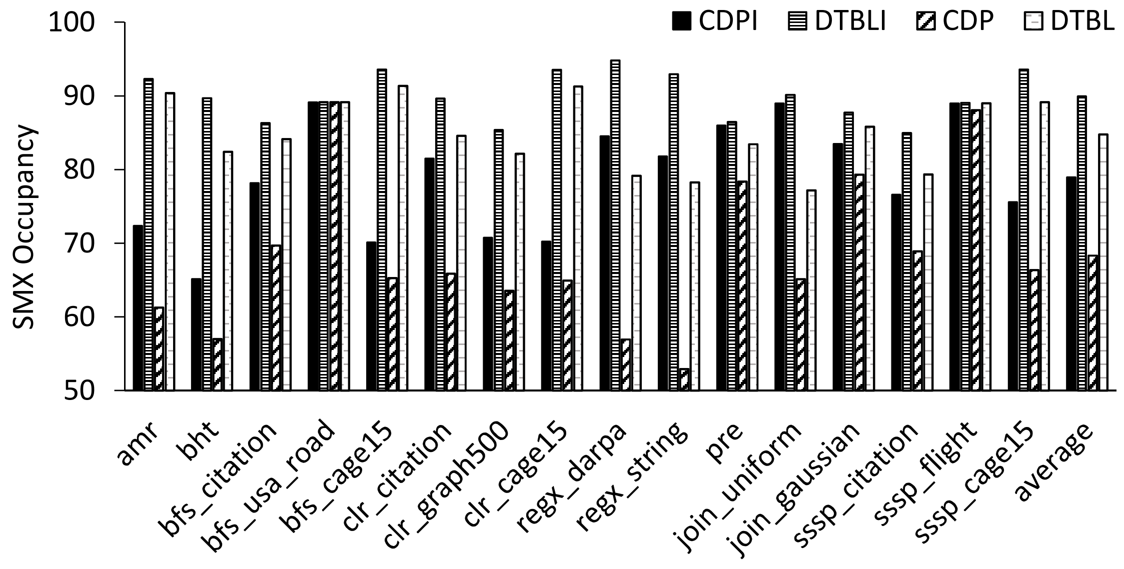


Figure 18: SMX Occupancy

If the launching latency is included for both CDP and DTBL, SMX occupancy decreases from both CDPI and DTBLI (average -10.7 or -13.5% for CDP and -5.2 or -7.6% for DTBL). The launching latency for a device kernel is higher than an aggregated group and causes a larger drop in SMX occupancy for CDP. In *regx_string*, DFP has high occurrence and generates a large number of dynamic parallel workloads. While DTBLI outperforms CDPI in SMX occupancy (11.2 or 1.14x) because of the increased thread block concurrency, the launching latency even enlarges the gap (25.4 or 1.48x). The increased SMX occupancy of DTBL also improves the DRAM efficiency as shown in Figure 17 (average 0.022 or 1.08x higher than CDP) because of the memory latency hiding capability.

The DTBL scheduling efficiency is further evaluated by comparing the average waiting time (time between launching and starting execution) and memory footprint for dynamically generated kernels or aggregated groups as shown in Figure 19 and Figure 20 respectively. Again, the waiting time is compared with and without launching latency. On average, DTBLI reduces the waiting time by 18.8% from CDPI while DTBL reduce the waiting time by 24.1% and the memory footprint by 25.6% from CDP. Similar to the SMX occupancy behavior, DTBLI of *pre* and *join_uniform* show little change in average waiting

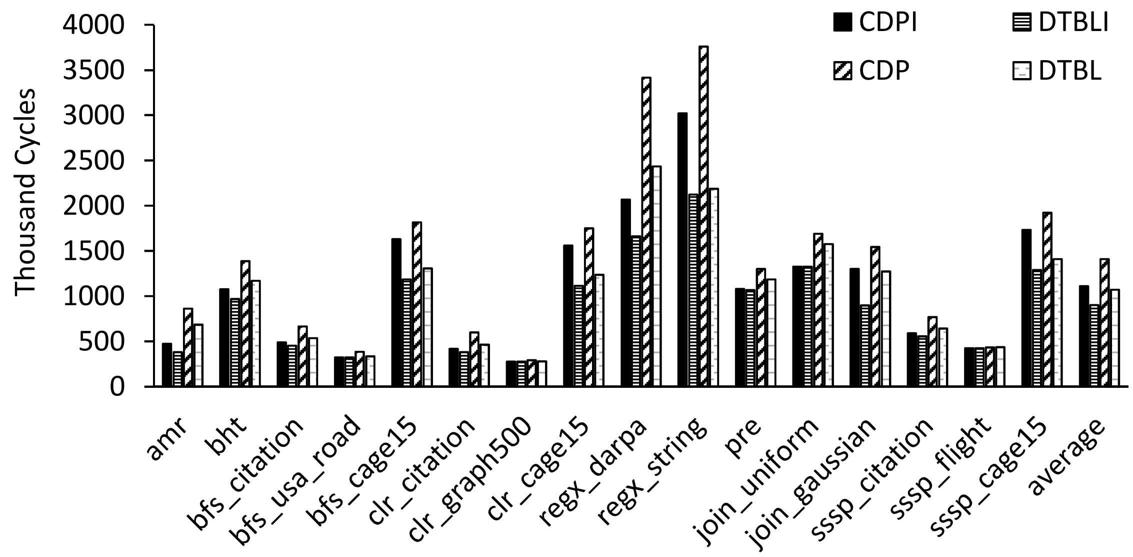


Figure 19: Average Waiting Time for a Kernel or an Aggregated Group

time compared because they generate coarse-grained dynamic workloads. The benchmark *regx_string* has the highest DFP occurrence so it benefits most from DTBL by showing the largest waiting time decrease (-41.8%) and significant memory footprint reduction (-51.2%). The benchmark *clr_graph500* does not show much change in the average waiting time when including the launching latency while its SMX occupancy is significantly affected by the launching latency because all the dynamically launched kernels or aggregated groups are forced to wait for other kernels to complete and release resources before they can be executed, which takes much longer than the launching latency. For the same reason, this benchmark also does not have any memory footprint reduction as the information of all the aggregated groups have to be saved while they are pending. One solution to this problem is to enable the spatial sharing for the native kernels and the aggregated thread groups using the software techniques introduced in [6] or hardware preemption introduced in [69]. This way the aggregated groups are able to execute on the SMX soon after they are generated and the memory reserved for holding their information could be released for new aggregated groups.

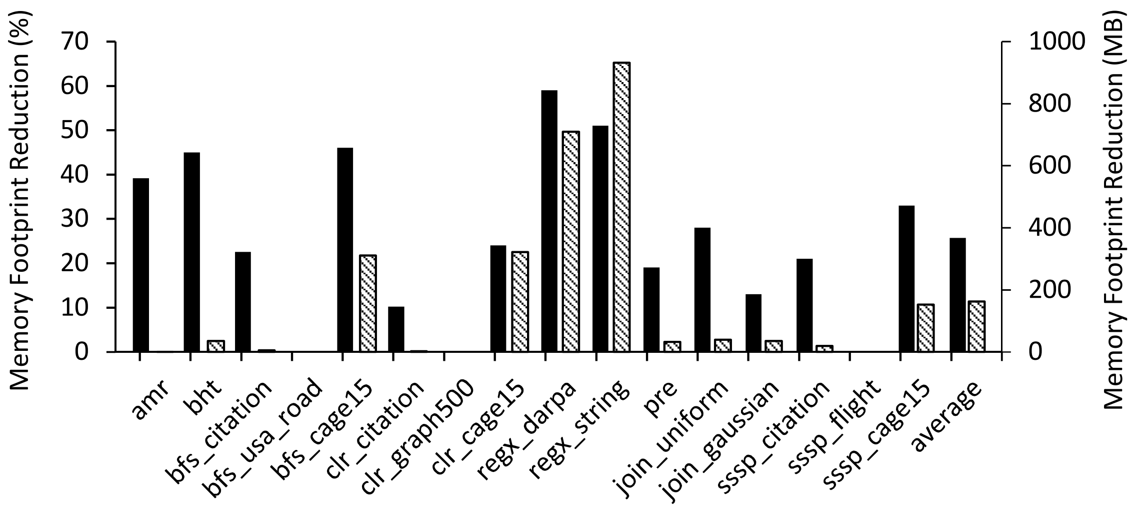


Figure 20: Memory Footprint Reduction of DTBL from CDP

4.6.2.3 Overall Performance

The overall speed up of CDPI, DTBLI, CDP and DTBL over the flat implementation is shown in Figure 21. Note that data transferring time between CPU and the GPU is excluded. As CDPI and DTBLI decrease control flow divergence and increases memory efficiency, they achieve average 1.43x and 1.63x ideal speedup respectively. However the non-trivial overhead of kernel launching negates the CDP performance gain, which results in an average of 1.16x slow down from the flat implementations. DTBL, on the other hand, shows an average of 1.21x speedup over the flat implementation and 1.40x over the CDP, which demonstrates that DTBL preserves the capability of CDP in increasing control flow and memory regularity for irregular applications while using a more efficient scheduling strategy with lower launching overhead to increase the overall performance. The benchmark *bfs_usa_road* and *sssp_flight* show very little change in the DTBL speedup. The reason is that most of vertices in the input graphs have very low vertex degree. The DFP rarely occurs in these two benchmarks so that very few device kernels or aggregated groups are launched. Therefore, both CDP and DTBL have very limited effect on the overall performance. In fact, these two benchmarks also show limited changes in other characteristics evaluated and discussed previously. Two benchmarks have slow down instead of speedup:

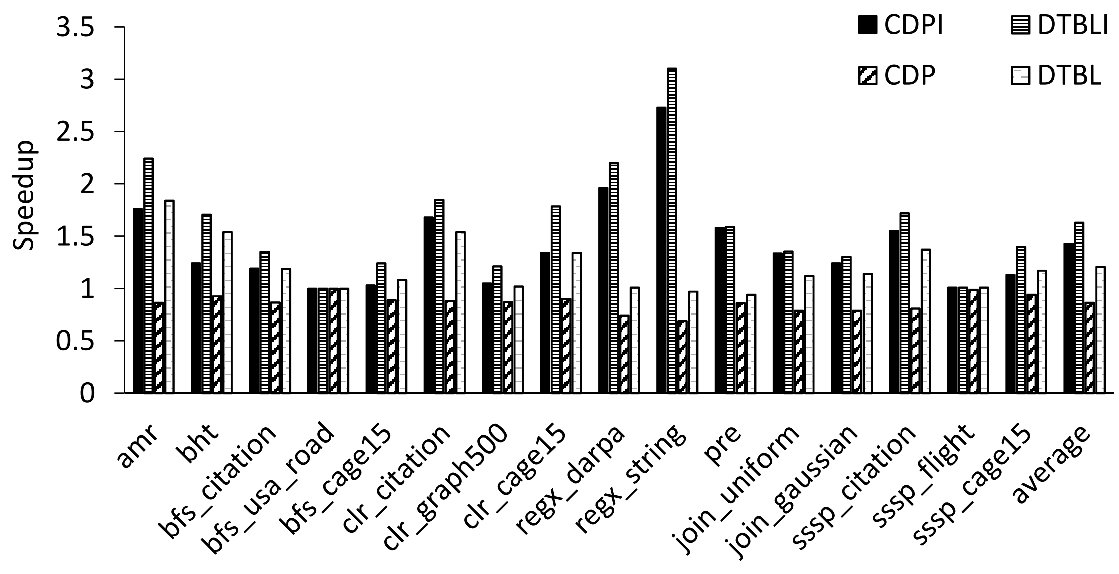


Figure 21: Overall Performance in terms of Speedup over Flat Implementation

clr_graph500 (0.97x) and *regx_string* (0.95x). The benchmark *clr_graph500* operates on the *graph500* input data set which has a very balanced vertex degree. Therefore, the flat implementation has good control flow and memory behavior. Using CDP or DTBL does not help reduce the control flow or memory irregularity but introduces extra launching overhead. For *regx_string*, while the large number of dynamically generated aggregated groups in brings significant speedup ideally (2.73x for CDPI and 3.10x for DTBLI), they also introduce substantial launching overhead even for DTBL and negates the performance gains.

4.6.2.4 Sensitivity to AGT Size

As the major architecture extension, the size of AGT determines the hardware overhead as well as the application performance. A larger AGT can increase the number of aggregate groups stored on-chip and thereby the scheduling efficiency at the cost of more on-chip SRAM. The trade off is identified by investigating the performance change over different AGT sizes as shown in Figure 23. In average, decreasing AGT size from 1024 to 512 causes 1.31x slow down and increasing to 2048 causes 1.20x speedup. Benchmarks that use relatively high number of dynamic aggregated groups such as *bht* and *regx* are more

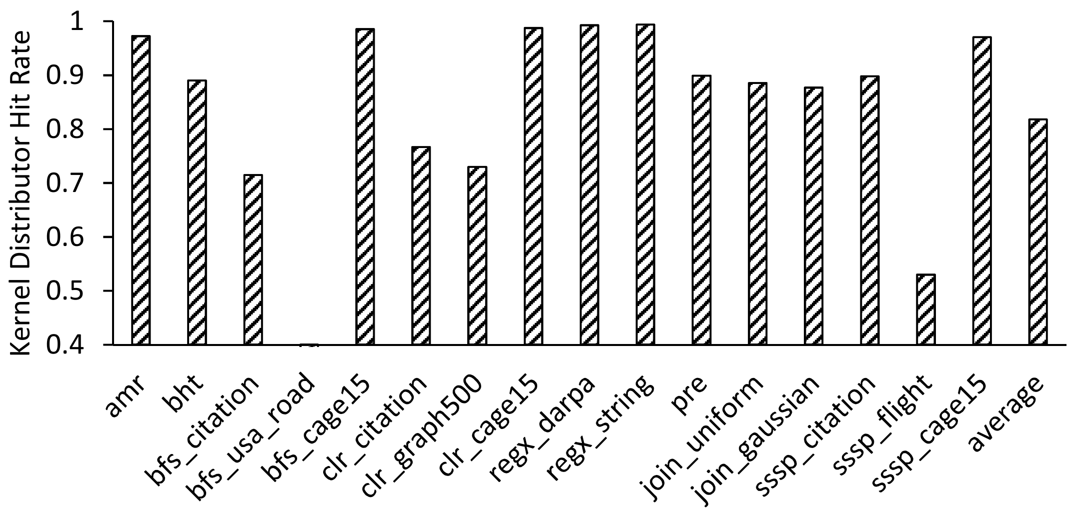


Figure 22: KDE Hitrate

sensitive to AGT size.

4.6.2.5 Sensitivity to KDE Hitrate

As the architectural extensions are designed to support DTBL, evaluation is performed for the two individual components: Kernel Distributor and Aggregated Group Register Table. For the Kernel Distributor, the KDE hit rate is measured when an aggregated group is locating an eligible kernel. For AGT, measure the entry hit rate is measured when the SMX scheduler tries to find a free entry. These evaluations enable the analysis of the aggregated groups behavior in the unstructured applications and their sensitivity to the architecture.

Figure 22 shows the KDE hit rate with an average of 0.818. Higher hit rate indicates higher DFP similarity. Recall that aggregate groups that do not hit an entry in the Kernel Distributor will be launched as a device kernel which is more expensive than coalescing with an existing kernel. For most of the benchmarks, DFP is very similar to each other so the number of kernels that exist in the life time of an application is very small since most of the aggregated groups are coalesced with the same kernel instead of launched as a new device kernel. Aggregated groups fail to locate an entry in the Kernel Distributor mainly

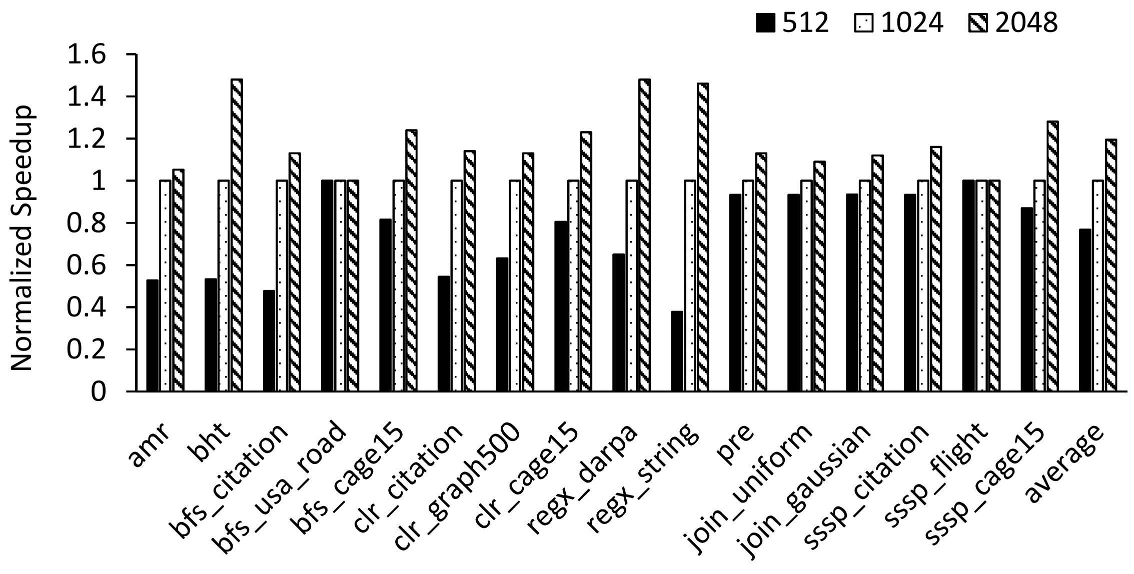


Figure 23: Performance Sensitivity to AGT Size Normalized to 1024 Entries

because of two reasons. First, no kernel that is calling the same entry function has been invoked. This usually happens at the early stage of a program especially when multiple threads are launching aggregated groups simultaneously. One solution is to launch a warm-up native kernel at the start of the application. The warm up kernel will be staying at the Kernel Distributor to receive aggregated thread blocks. Second, aggregated groups have different configurations (e.g. thread number in a thread blocks) than the kernel that resides in the Kernel Distributor. While this could be avoided by the programmers, sometime it is necessary to keep native kernels with different configuration for handling DFP with different features.

Figure 23 shows the AGT hit rate for different size of AGT. The default AGT size of the proposed architecture extension is 1024. Recall that if it fails to locate a free entry, the aggregated group information will be stored off-chip on global memory which would introduce memory latency when the aggregated group is scheduled by the SMX scheduler and the control registers are required to be updated. When the AGT size increases from 512 to 1024, the average hit rate increases from 0.622 to 0.816. Applications that generate many aggregated groups in a short period of time such as *clr_graph500* and *regx_strings* have lower hit rate. Increasing the AGT size could increase the hit rate at the cost of larger on-chip SRAM size. Similarly as the solution to reduce memory footprint, using spatial sharing on an SMX for native kernel and aggregated groups could also increase the speed of releasing entries in AGT and thereby the hit rate for a free entry. For some kernels, hit rate is low because of large number of aggregated groups are launched. This introduce latency because aggregated groups will be launched as a device kernel. Potential solution includes preemption or increase SMX sharing rate among different kernels.

4.6.3 Discussion

From the experiment evaluation and analysis, the following observations can be made for DTBL:

- Irregular applications with fine-grained DFP benefits most from DTBL scheduling scheme as fine-grained aggregated groups are coalesced to the same kernel to increase TB-level concurrency.
- Performance of irregular applications can be highly sensitive to launching latency as they usually generate large amount of dynamic work. The performance gain of DTBL is partly due to its light-weight launching and scheduling schemes.
- New programming methodology and compiler technique may be developed to facilitate the usage of the DTBL model in the way that the generation and consumption of dynamic aggregated groups are balanced to reduce hardware, runtime overhead and memory footprint.

4.7 Summary

This chapter proposes Dynamic Thread Block Launch (DTBL), a new extension to the current GPU execution model that enables dynamic thread block launching and coalescing to existing kernels on the fly. The proposed model is specifically designed to provide a more efficient solution for executing dynamically formed pockets of parallelism in irregular applications. DTBL is introduced by defining the execution model, proposing minimal modification to the programming interface and discussing the microarchitecture extension. Through experimental evaluation on various irregular CUDA applications, it is demonstrated that by increasing GPU scheduling efficiency and decreasing launching overhead, the proposed model achieves average 1.21x speedup over the original flat implementation and average 1.40x over the implementations using device-kernel launch functionality.

The subsequent chapters propose two additional optimizations for DTBL to further improve the performance from both the memory reference locality and the energy efficiency perspective.

CHAPTER V

OPTIMIZING THE PERFORMANCE OF THE DTBL MODEL

To further explore the potential optimization opportunities in dynamic parallelism execution models including CDP and DTBL, this chapter examines the memory reference locality that exists between the native TBs (or parent TBs) and the dynamic TBs (or child TBs) and proposes a new locality-aware TB scheduler.

5.1 Memory Locality in Dynamic Parallelism

Previous studies on GPU have established the importance of memory locality effects and the need for TB and warp level scheduling techniques [50][31][32][39]. While many of the insights that motivate these works are applicable, none of them address the domain of dynamic parallelism in GPUs. Dynamic parallelism involves device-side nested launches of kernels or TBs (equivalently workgroups in OpenCL). Consequently it introduces new types of locality behaviors, for example, spatial and temporal reference locality between parent kernels and child kernels, or between child kernels launched from the same parent kernel thread (sibling kernels). Modern GPU microarchitecture schedulers are designed for non-dynamic parallelism settings and are unaware of this new type of locality relationships. Existing locality-aware TB schedulers do not work across the kernel launching boundary and therefore only utilize the TB locality information within a single kernel, either parent or child.

The TBs of a device kernel (CDP) or a TB group (DTBL) are referred to as *dynamic TBs*. TBs which launch new device kernels or TB groups are the *direct parent TB*. All the TBs that are in the same kernel or TB group as the direct parent TB are the *parent TBs*. The TBs in the newly launched device kernels or TB groups are the *child TBs*. Figure 26(a) shows an example of the parent-child launching using either the CDP or the DTBL model. In this example, there are eight parent TBs (P0-P7) in the parent kernel. TB P2 generates two child TBs (C0-C1) and is their direct parent. The direct parent of the four child TBs (C2-C5) is TB P4. The notations of parent and child TBs will be used in subsequent discussion and equally applicable to both CDP and DTBL models as well as potentially other dynamic parallelism models as long as they retain the TB-based BSP execution model.

The memory reference locality that exists between the parent and child TBs in the course of exploiting dynamic parallelism is examined on a GPU. Parent-child locality provides an opportunity for optimizing performance that is not exploited by existing TB schedulers on current GPUs, and is the major motivation of the proposed LaPerm TB scheduler.

5.1.1 Spatial and Temporal Locality

Researches [10][62][17] have shown that while it is common to observe the existence of reference locality at certain time during the execution of irregular applications, it usually occurs in a way that is non-uniform, fine-grained, nested, and dynamic. In structured applications, (e.g., many scientific codes) inter-thread locality often leads to effective coalescing of memory references and consequent efficient use of memory bandwidth. In contrast, the non-uniform occurrences of locality in irregular applications makes it difficult to exploit the peak memory bandwidth. However it has been shown in Chapter 3 that the use of dynamic parallelism can convert intra-thread locality to uniform inter-thread locality which in turn can lead to increased coalescing of memory accesses and thereby effective use of memory bandwidth. For example, expanding the neighbors of a vertex in a graph problem is often done by a single thread leading to intra-thread locality across outgoing edges. With dynamic parallelism, a child TB can expand each vertex concurrently designated by the parent thread. Thus, intra-thread locality of the parent is converted to inter-thread locality of the child TB. The work in this chapter focuses on the shared structures between parent and child which can lead to locality of references between the parent threads and the child threads.

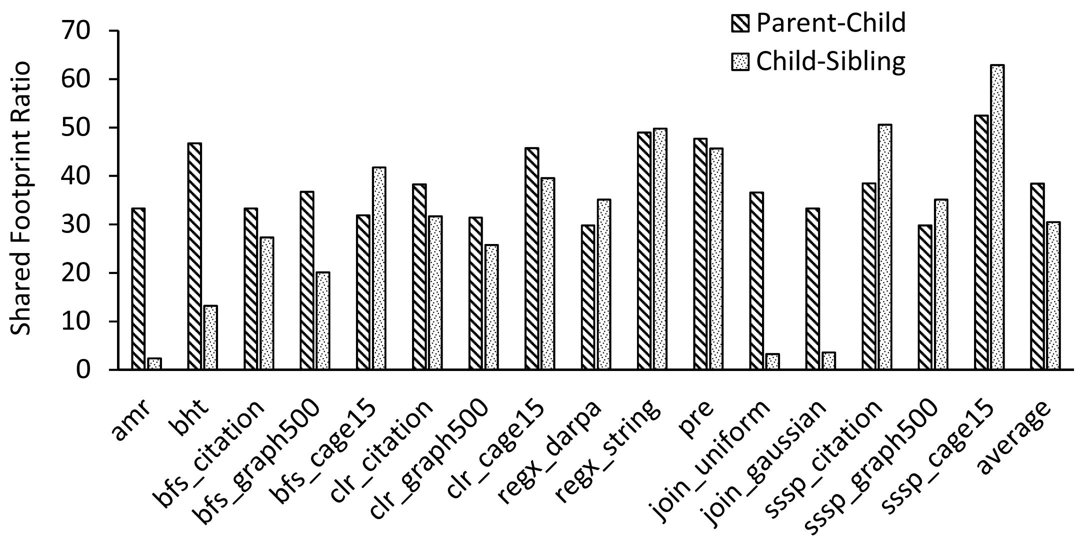


Figure 24: Shared footprint ratio for parent-child and child-sibling TBs.

The existence of such locality is demonstrated by examining the memory access patterns of the direct parent and child TBs in multiple benchmarks described in Table 7. The examination process is performed between each direct parent TB and all of its child TBs, as well as between each child TB and all of its sibling TBs (the TBs that are launched by the same direct parent). The memory access patterns are application-dependent regardless of whether the CDP or the DTBL model is used. To quantify the memory access patterns and reveal the potential parent-child locality, 1) the set of memory references that the direct parent and all of its child TBs make are recorded to compute their respective sizes as p and c in units of a 128-byte cache block, 2) the memory references that are shared between the direct parent and all of its child TBs are identified to compute the total size as pc cache blocks, 3) the ratio pc/c as the *shared footprint ratio* is computed for parent-child. Similarly, the memory references made by a single child TB and all of its sibling TBs are recorded respectively with size co and cs, the memory references shared by them are identified with size cos and the ratio cos/cs is the *shared footprint ratio* for child-sibling. Figure 24 shows the results with an average shared footprint ratio of 38.4% for parent-child and 30.5% for child-sibling. It should be noted that data locality also exists among

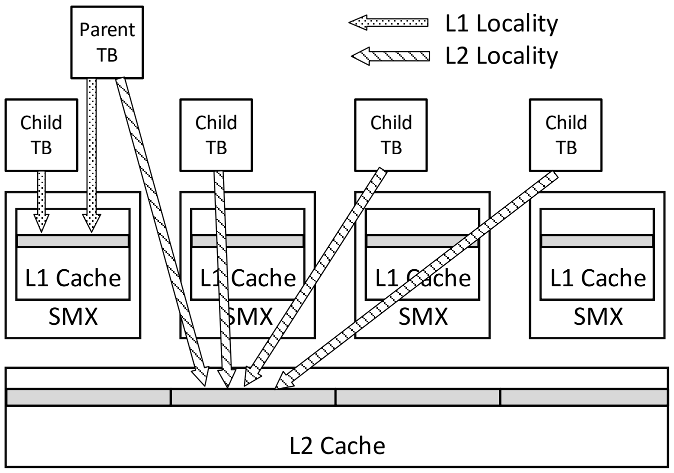


Figure 25: Parent-Child Locality and Potential Impact on L1 and L2 Cache

the parent TBs, but is significantly less than parent-child or child-sibling data reuse. Analysis shows that the average shared footprint ratio for parent TBs is 9.3%. Therefore this work focuses on schedulers that can utilize the parent-child TB data reuse. Higher shared footprint ratio reveals better potential locality between the direct parent and the child TBs which may exist both spatially and temporally as described in the following:

Temporal Locality: A common practice in using the dynamic parallelism model for irregular applications is that the parent TB performs the necessary computation to generate the data, passes the data pointers (usually stored in the global memory) to the child and invokes the child to continue the computation. The reuse of the parent-generated data by the child TBs results in good temporal locality as long as the execution of the child TBs is "soon enough" after the parent. The parent-child shared footprint ratio shown in Figure 24 demonstrate the potential existence of such temporal locality.

Spatial Locality: Spatial locality may exist either between the direct parent TB and the child TBs or between different child TBs. This is usually because the computations of either the parent or the child can access memory locations that are relatively spatially close. For example, using a common data structure such as Compressed Sparse Row (CSR) for the graph problem where neighbor vertices are stored in consecutive addresses in the memory, different child TBs may explore subgraphs that are stored closely to each other in the memory. Compared with parent-child locality, the child-sibling locality can have higher variation as shown in Figure 24, depending on the benchmark characteristics or even the input data. For example, the input graphs *citation_network* and *cage15* exhibit more concentrated connectivity as vertices are more likely to connect to their (spatially) closer neighbors. Therefore, with the CSR data structure and its memory mapping, the child-sibling shared footprint ratio for the graph benchmarks that take these two input graphs are higher than *graph500* where vertices can connect to other vertices all over the graph, resulting in child TBs dealing with more distributed memory accesses. It is even more apparent in the benchmark *amr* and *join* that the child TBs are always working on its own memory region with virtually no data reference from other child TBs, causing the lowest shared footprint ratio among all the benchmarks.

While the intra TB locality of the child TBs with dynamic parallelism can result in more coalesced memory accesses that can leverage the global memory of the GPU memory hierarchy, the locality between the parent and child TBs provides an opportunity for improved memory performance in terms of L1 and L2 cache behavior as shown in Figure 25. L2 cache performance can be increased if locality exists among the TBs that are executed closer in time. Furthermore, execution of these TBs on the same SMX may even have a positive impact the L1 cache performance. However, exploiting such potential cache behaviors is by itself not straightforward and can largely depend on the GPU SMX scheduler. Blelloch et al. [12][13] develop a theoretical model for scheduling fine-grained nested parallel tasks onto parallel architectures with multi-level caches. This chapter is a practical implementation of such a theory on the GPU architecture while also taking into account multiple specific features that are unique to GPU such as the TB level concurrency realized by TB interleaving scheduling.

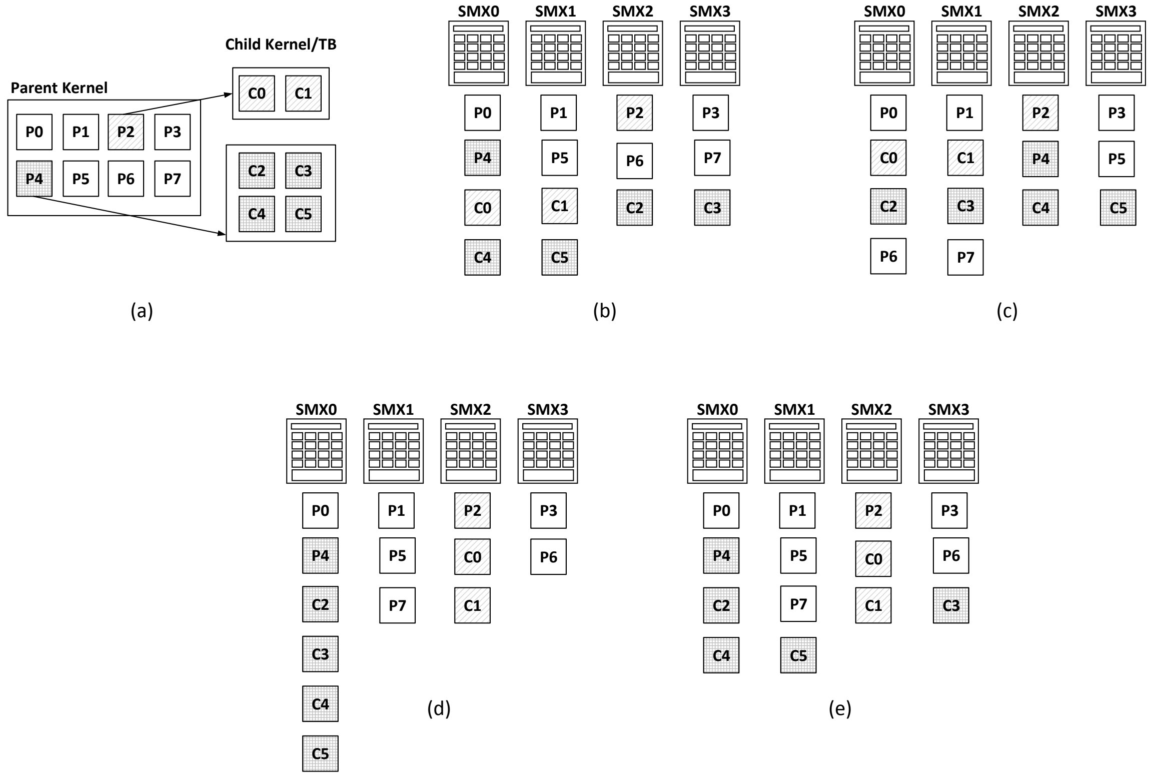


Figure 26: (a) An example of the parent-child kernel/TB launching, (b) its TB scheduling results using the Round-Robin TB scheduler, (c) TB Prioritizing (*TB-Pri*), (d) Prioritized SMX binding (*SMX-Bind*) and (e) Adaptive Prioritized SMX Binding (*Adaptive-Bind*).

5.1.2 Round-Robin TB Scheduler

The SMX scheduler on current GPUs adopts the round-robin (RR) TB scheduling policy which is designed for fairness and efficiency. This is the baseline scheduling policy used in this chapter. This policy works well for structured applications. However, with the ability to dynamically launch kernels or TBs, this policy fails to exploit parent-child locality or child-sibling locality for dynamic TBs.

Figure 26(b) illustrates the effect of the RR policy for dispatching parent and child TBs to the SMXs for the example shown in Figure 26(a). The 8 parent TBs (P0-P7) and 6 child TBs (C0-C5) are executed on a GPU that has 4 SMXs (SMX0-SMX3). Each SMX is able to accommodate one TB. In the baseline GPU architecture, the KDU employs a FCFS kernel scheduler for all the parent and child kernels while the SMX scheduler only dispatches child TBs after the parent TBs. Therefore, the child TBs (C0-C5) will be scheduled after parent TBs (P0-P7). Furthermore, TBs are dispatched to SMXs in a round-robin fashion, so all the parent TBs and child TBs are distributed evenly across all the SMXs (assuming each parent TB and child TB is able to complete execution at the same pace) as shown in Figure 26(b). There are two major issues with the resulting TB distribution in terms of the impact on locality:

- Child TBs do not start execution soon after their direct parents. After TB P2 is executed, the SMXs are occupied by TBs (P4-P7) before P2's child TBs (C0-C1) can be dispatched. TBs (P4-P7) may pollute the L1/L2 cache and make it impossible for TBs (C0-C1) to reuse the data generated by TB P2 directly.

- Even if a child TB is scheduled soon enough after its direct parent TB, such as TB (C2-C3), they are not dispatched on the same SMX as its direct parent. Therefore, it is difficult to utilize the L1 cache of each SMX to exploit the parent-child or child-sibling locality.

The above two issues are exacerbated in real applications where the parent kernel generally is comprised of many TBs so that child TBs have to wait even longer before they can be dispatched and executed. The long wait may potentially destroy any opportunity to utilize the parent-child locality information.

Therefore, this work proposes *LaPerm*, a locality-aware TB scheduler which is specifically designed to improve locality behavior when employing dynamic parallelism on GPUs. As the rest of this chapter will demonstrate, LaPerm leverages the spatial and temporal locality between and among parent and child TBs leading to better memory system performance, and therefore overall performance for irregular applications that employ dynamic parallelism.

5.2 LaPerm Scheduler

In this section, the LaPerm TB scheduler is introduced which is comprised of three scheduling decisions: TB Prioritizing, Prioritized SMX Binding and Adaptive Prioritized SMX Binding. The scheduling decisions differ in the specific forms of reference locality that they exploit and may showcase different performance benefits for applications with different characteristics. LaPerm applies to the dynamically generated TBs both from device kernels in CDP as well as the TB groups in DTBL. Architecture extensions are also proposed to support LaPerm on GPUs.

5.2.1 TB Prioritizing

To address the issues with the RR TB scheduler for dynamic parallelism, a TB Prioritizing Scheduler (*TB-Pri*) is proposed where dynamic TBs are assigned a higher priority so that they can be dispatched to SMXs before the remaining TBs of the parent kernel or TB group. The parent TBs are given an initial priority and the launched child TBs are assigned a priority of one greater than that of the parent TBs. This priority assignment process can be nested to accommodate nested launches from the parent TBs to a maximum level *L* of the child TBs. Any nested launch level that exceeds *L* will be clamped to *L*. The goal of

TB-Pri is to start the execution of dynamic TBs as soon as they are launched by the parents to facilitate the leverage of temporal locality.

Figure 26(c) shows an example of applying *TB-Pri* to Figure 26(a). For the purpose of illustration, the following description refers to the process of scheduling four TBs to the four consecutive SMXs starting from SMX0 to SMX3 as *one round* of TB dispatching, one cycle for each TB on an SMX. Assume the parent TBs (P0-P7) are assigned with priority 0, then the child TBs (C0-C5) are assigned with priority 1. The first round of TB dispatching stays the same as the RR scheduler where TBs (P0-P3) are distributed to SMX0-SMX3. As the child TBs (C0-C1) are generated by TB P2 and assigned higher priority than TBs (P4-P7), they will be dispatched to the SMXs before (P4-P7) in the second round, resulting in C0 on SMX0, C1 on SMX1, P4 on SMX2 and P5 on SMX3. Since TB P4 generates another four dynamic TBs (C2-C5), they will be scheduled on SMX0-SMX1 in the third round before the remaining two parent TBs P6 and P7 which are dispatched in the final round. Compared with the RR scheduler, child TBs (C0-C1) and (C4-C5) are scheduled earlier (in the second and third round instead of the third and the fourth round), which reduces the time gap from their direct parents and increases the possibility of better cache behavior because of the temporal locality. As child TBs can be scheduled to all the SMXs on the GPU, L2 cache performance increase can be the major benefit.

Architecture Support. To support *TB-Pri*, the kernel and TB scheduler are extended such that they can manage TBs with different priority values. The newer generation of NVIDIA GPUs support prioritized kernel launches [57], where kernels assigned with higher priority can be scheduled first and preempt the kernels with lower priority using the technique described in [69]. This is realized through multiple queues with different priority values, each of which contains the kernels with a specific priority value as shown in Figure 27(b). These priority queues are stored in the global memory and managed by KMU which dispatches kernels to KDU from the queues with higher priority followed by those with lower priority. Thus the SMX scheduler will also distribute TBs from kernels with

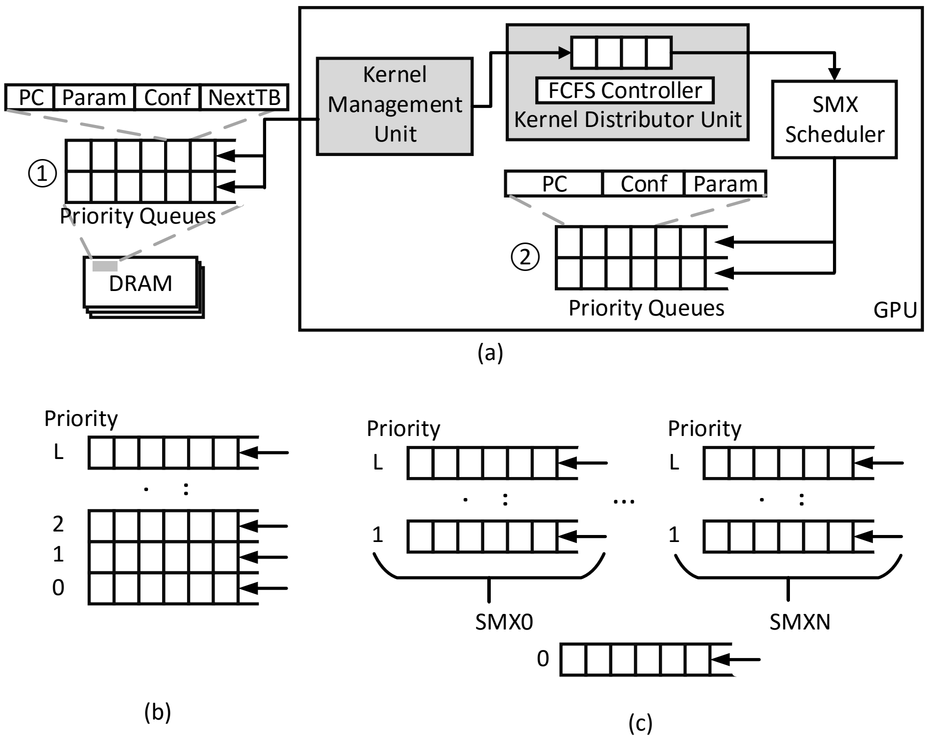


Figure 27: Architecture Extension for LaPerm (a), the priority queues used by *TB-Pri* (b) and the SMX-bound priority queues used by *SMX-Bind* and *Adaptive-Bind*.

higher priority to the SMX before those with lower priority. Preemption happens when higher-priority kernels are generated after lower-priority kernels start execution. In this case, when a TB from the lower-priority kernel finishes execution, the SMX scheduler will dispatch the waiting TBs from the higher-priority kernel to take up the freed capacity.

As shown in Figure 27(a), *TB-Pri* for both CDP and DTBL can also use the priority queues to manage the device kernels. Each entry of the priority queue contains information of the device kernel or TB groups including PC, parameter address, thread configuration and the next TB to be scheduled. The host-launched kernels stays in the lowest priority queue 0. In CDP, the priority queues are stored in the global memory①. The child kernels are assigned to the queue whose priority value is greater than its direct parent priority by one so that TBs from the child kernels are able to be dispatched before the remaining parent TBs. In the same priority queue, the newer kernels are appended to the tail so the priority queue itself is FCFS. The same priority queue structures are also used for DTBL to store the dynamic TB group information. As proposed in [72], DTBL uses both the on-chip SRAM and the global memory to store the dynamic TB group information when they are generated from the SMXs. These TB group tables are reused to store the priority queues (① for global memory and ② for on-chip SRAM) so that the on-chip SRAM ensures fast access to the TB group information from the SMX scheduler while the priority queues stored in the global memory serve as the overflow buffer.

Issues. Although *TB-Pri* leverages temporal locality and moves the child TB execution earlier, soon after the direct parent, the TB may be scheduled on any SMX. This can only increase the L2 cache performance. In the example Figure 26(c), TB (C0-C1) are still executed on different SMXs than its direct parent TB P2, therefore the L1 cache on SMX2 still cannot be utilized for parent-child data reuse. A similar observation applies to child TB (C2-C5) of the direct parent TB P4. TB C4 is now executed on SMX2 immediately after P4, which exhibits better locality than in Figure 26(b) and facilitates better L1 cache utilization, but the remaining child TBs (C2, C3, C5) are distributed across all the SMXs so that both the parent-child and child-sibling locality improvement is limited to the L2 cache behavior.

5.2.2 Prioritized SMX Binding

To utilize the entire GPU cache hierarchy more effectively, especially the L1 cache for data reuse, *TB-Pri* is extended so that the child TB should also be bound to the specific SMX that is used to execute its direct parent. This policy is referred to as Prioritized SMX Binding or *SMX-Bind*. The SMX binding directs the SMX scheduler to dispatch the child TBs such that they can use the same L1 cache on the SMX that is used by the direct parent.

Figure 26(d) shows the scheduling result using *SMX-Bind* for the parent-child launch structure in Figure 26(a). *SMX-Bind* identifies that child TBs (C0-C1) are launched by TB P2 from SMX2 so (C0-C1) are bound and dispatched on the same SMX2. Similarly, child TBs (C2-C5) are bound by their direct parent P4's executing SMX which is SMX4. The binding process ensures that C0 and C1 are scheduled in the second and third round to SMX2 and (C2-C5) are scheduled from the third to the sixth round to SMX0. All the remaining parent TBs are still dispatched using the original round-robin scheduling scheme. As the child TBs are now always scheduled on the same SMX as the direct parent, L1 cache can be well utilized to exploit the parent-child and child-sibling locality.

Architectural Support. The priority queues used for *TB-Pri* are extended to support *SMX-Bind* as shown in Figure 27(c) where the priority queues from 1 to L are used for each of the SMXs (SMX0-SMXN shown in Figure 27(c)). The priority queue 0 is shared by all the SMXs and reserved to store the information of the top-level parent kernels (host-launched kernels). A simple duplication of the original priority queues in Figure 27(a)) for the N SMXs would cost (N-1) times more hardware overhead, therefore, the extended architecture evenly divides the original priority queues in into N priority queues, each associated with one SMX with the expectation that TBs are evenly distributed across the SMXs. For each newly generated device kernel or TB group, the SMX scheduler will push its information to the priority queues that are associated with the SMX occupied by the direct parent. For each SMX, the TB dispatching process only fetches TBs from the associated priority queues until all the associated priority queues are empty so that new parent TBs can be fetched from priority queue 0. Note that in some GPUs, SMXs are divided into multiple clusters where each cluster possess more than one SMX and the L1 cache is shared by all the SMXs in a cluster [56]. In this case, *SMX-Bind* scheduling scheme associates the priority queues with the entire cluster and the newly generated TBs will be bound to any SMX in the cluster. Within each cluster, the round-robin dispatching strategy is employed for the TBs fetched from the priority queues.

Issues. In an ideal case, dynamic TBs can be evenly distributed across all the SMX to avoid any fairness issues and ensures that the evenly divided priority queues among SMXs are used in a balanced and efficient manner. However, as shown in Figure 26(d), it is possible that some parent TBs may have more nested launch levels or more child TBs (e.g. TB

P4 and its four child TBs) than others. Restricting all these child TBs to a single SMX (or SMX cluster) may result in the idling of other SMXs and low overall execution efficiency. In Figure 26(d), SMX1 and SMX2 are idle after the third round of scheduling and SMX3 is idle after the second round, creating an unbalanced SMX workload. In irregular applications, it is very common that the launching patterns including nesting levels and child TB numbers vary from one parent TB to another, increasing the possibility that *SMX-Bind* could suffer from the SMX workload imbalance issue.

5.2.3 Adaptive Prioritized SMX Binding

To solve the load imbalance issues in *SMX-Bind* and increase the overall execution efficiency while preserving the cache performance benefits, the *SMX-Bind* scheduling scheme is further optimized to incorporate a more flexible TB dispatching strategy which is referred as Adaptive Prioritized SMX Binding (*Adaptive-Bind*). *Adaptive-Bind* still first dispatches prioritized child TBs to their bound SMX followed by other lower-priority parent TBs. At some point, both the prioritized child TBs bound to one SMX and all the parent TBs have been dispatched. *Adaptive-Bind* will then cross the SMX boundary and dispatch child TBs that are supposed to be bound to other SMXs to the current SMX if it has enough available resource to execute these child TBs. In this process, the dispatching scheme effectively put all child TBs bound to other SMXs as the backup TBs of the current SMX. The backup TBs can be viewed as TBs with the priority even lower than the top-level parent TBs which has priority 0. The goal here is to generate a more balanced TB distribution across all the SMXs to avoid any SMX idleness, which can result in balanced data reuse with increasing SMX utilization. This scheduling policy is balancing the tradeoff between exploiting reference locality within the cache hierarchy with utilization of the spatially distributed SMXs.

The scheduling results of *Adaptive-Bind* on Figure 26(a) is shown in Figure 26(e). Until the third round, the TB dispatching of *Adaptive-Bind* is the same as that of *SMX-Bind* as shown in Figure 26(d). The difference starts on SMX3 in the third round. As no child TBs

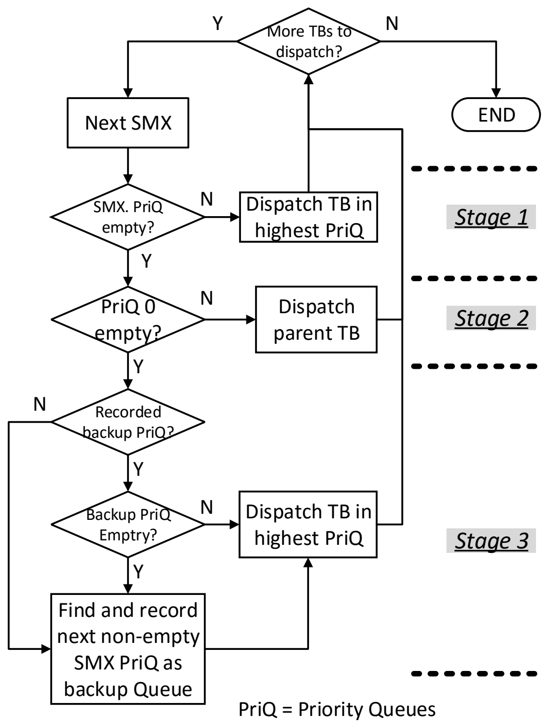


Figure 28: LaPerm Scheduler Flow Chart

are bound to SMX3 and all the parent TBs have been dispatched, *Adaptive-Bind* fetches the next TB from P4's child TBs – TB C3 which was originally bound to SMX0 – and executes it on SMX3. A similar procedure applies in the fourth round of TB scheduling on SMX1. The result shows that TB P2 and child TBs (C0-C1), and TB P4 and child TBs (C2, C4) are scheduled on the same SMX while the remaining child TBs are scheduled across all the SMXs. Compared with *SMX-Bind*, the performance of L1 may decrease due to less parent-child data reuse but it is compensated for by better SMX workload balance and thereby a potential positive impact on the overall execution efficiency.

Architectural Support. *Adaptive-Bind* still employs the same SMX-bound priority queues that are used by *SMX-Bind*. However, an extended SMX scheduler is designed to manage the priority queues and dispatch TBs. The complete *Adaptive-Bind* TB scheduler operation flow is shown in Figure 28 which is implemented as an extension to the SMX scheduler used by current GPUs. The LaPerm scheduler starts by following the normal routine of an SMX scheduler to check if there are more TBs from KDU to dispatch and execute. Then it checks all the SMXs, one for each cycle, and selects the candidate TB for the current SMX in three progressive stages: 1) highest-priority TB in the current SMX's priority queues, 2) parent TB in the global priority 0 queue and 3) highest-priority TB in the backup queues. Stage 2 only happens when the current SMX's priority queues are empty while stage 3 only happens when both the current SMX's priority queues and priority 0 queue are empty.

Note that in stage 3, when *Adaptive-Bind* selects priority queues of one SMX as the backup queues for the current SMX, it will focus on scheduling TBs from the chosen backup queues until they are also empty. As shown in Figure 28, the backup queues will be recorded each time they are selected and reused next time when stage 3 is invoked if they are not empty. The major reasons for this fixed backup scheme are that i) the TBs from the backup queues are also more likely to be scheduled on the same SMX which may help leverage their locality and ii) although the SMX scheduler is able to schedule TBs with different configurations on the same SMX, it may incur the overhead of resource initializing such as register and shared memory partitioning. Focusing on priority queues of one SMX can effectively minimize such overhead as each entry of a priority queue is either a kernel or a TB group that contains TBs using the same configuration.

For simplification, Figure 28 illustrates the LaPerm scheduler in DTBL model where TB groups are launched and pushed to the priority queues and directly scheduled by the SMX scheduler. In CDP model, new kernels are pushed to the SMX-bound priority queues stored in the global memory and dispatched by KMU to KDU and then to the on-chip SMX-bound priority queues used by the SMX scheduler. Therefore, The LaPerm scheduler also involves extension to the KMU kernel scheduler where it checks all the SMX-bound priority queues in a round-robin fashion (one SMX a time), dispatches the kernel with the highest priority if there is an available KDU entry and store its information in the corresponding priority queue of its bound SMX. The KDU entry number (currently 32 on GPUs that supports CDP) limits the dynamic kernels and thereby dynamic TBs that are available to be dispatched by the LaPerm SMX scheduler within a time frame. If the KDU is filled up with the 32 concurrent kernels, newly generated kernels cannot be dispatched from the KMU to the KDU even if they have higher priority. This is also a known limit in the kernel preemption context [69] where the kernels that are available to be preempted are limited to the ones that stay in KDU. In contrast, all the dynamic TBs in a DTBL model are coalesced to kernels in KDU so they are always visible to the LaPerm scheduler. As a result, TB dispatching with LaPerm on CDP may not always be able to find the highest-priority TB and achieve the optimized results in terms of locality and cache performance.

5.2.4 Impact of Launching Latency

LaPerm is built on the assumption that the child TBs can be executed early enough after the direct parent TBs to utilize the temporal locality and spatial locality. However, an important issue in the dynamic parallelism model is the launching latency of the child TBs especially in CDP [74], which can i) cause a long wait before the child TBs can actually be dispatched by the LaPerm scheduler, ii) introduce a lengthy time gap between the parent and child and iii) kill any potential parent-child locality. The DTBL model [72] along with any other future developments in dynamic parallelism models with better architectural, memory system, runtime, driver support may further reduce the launching latency and make full use of LaPerm. Section 5.3 analyzes the impact of launching latency on LaPerm scheduler performance.

5.2.5 Overhead Analysis

The major hardware overhead is caused by the priority queues used by LaPerm and the SMX scheduler extension shown in Figure 27. The SMX-bound priority queues that are stored in the global memory can have flexible size and be allocated during the runtime. They are indexed per SMX and per priority level. The on-chip SMX-bound priority queues are stored in a 3K bytes SRAM for each SMX (about 1% of the area cost by the register file and shared memory) and is able to store 128 entries (24 byte per entry). For an L-level priority queue, (L-1) index pointers are employed to separate these 128 entries to store TBs using the decreasing order of priority. The priority queue 0 shared by all the SMXs needs additional 768 bytes (32 24-byte entries) on-chip SRAM storage. Note that for CDP, the number of entries of the on-chip priority queues is limited to 32 per SMX the same as the KDU entry number.

The major timing overhead comes from pushing new dynamic TBs into priority queues and the LaPerm TB dispatching process. For CDP, generating new device kernels already incurs the overhead in storing the new kernel information in the global memory [57] which is the memory access latency. Pushing them to the priority queues stored in the global memory does not introduce additional overhead. Dispatching kernels by the KMU from the priority queues to the KDU may incur maximum extra L cycles where L is the maximum priority levels. The overhead is caused by the searching of the highest-priority kernel where in the worst case, all the L priority queues have to be searched, one cycle for each. For DTBL, inserting a new TB group to the on-chip priority queues introduces the searching overhead of the 128-entry queue to locate the insert position according to TB group's priority, which can be 128 cycles in the worst case. However, this searching overhead can be hidden by the setting up process of the TB groups such as allocating parameter buffer. If the on-chip priority queue is full and the new TB groups have to be stored in the overflow priority queues in the global memory, the overhead would be the global memory access latency which can also partly be hidden by the TB group setting up process. Finally, the dispatching process of LaPerm TB is designed such that all the three stage searches can be finished within one cycle just as the baseline TB scheduler.

5.2.6 Discussion

The LaPerm scheduler is designed in a manner that is transparent to the warp scheduler, therefore it may be combined with any warp scheduler optimization such as [63][64]. Specifically, warp schedulers described in [31] also take into account the locality between different TBs and seek higher memory system utilization including bank-level parallelism, row buffer hit rate and cache hit rate by using a TB-aware warp grouping and prioritizing approach. Such warp schedulers can be leveraged by LaPerm to achieve even better memory system performance.

While many TB scheduling strategies are designed for the regular BSP model and may not apply by their own under the dynamic parallelism model, they can be certainly implemented as an optimization to LaPerm. For example, the TB scheduler introduced in [32] can dynamically adjust the dispatching TB number on each SMX to avoid too much memory contention. In LaPerm, the relatively small L1 cache (maximum 48 KB) may result in not fitting enough reusable data of the parent and child TBs, which can benefit from the incorporation of such contention-based TB control strategies.

This work does not consider different data reuse patterns across child TBs, the impact of data reuse distance between the parent and the child TBs as well as that of the different hardware parameters such as cache size, thereby any scheduling optimization accordingly which could be implemented with commensurate runtime and hardware support. The LaPerm scheduler is a first step in scheduling approaches based on understanding data-reuse in dynamic parallelism that provides insights to help address these problems.

5.3 Experiments

In this section, the three different decisions of the LaPerm scheduler are evaluated with multiple benchmark applications. Some key insights are provided from the evaluation results.

5.3.1 Methodology

The LaPerm scheduler is implemented on GPGPU-Sim that are extended with CDP and DTBL. The configuration of GPGPU-Sim is the same as that for DTBL simulation which is shown in Table 4.

The benchmark applications used to evaluate the LaPerm scheduler are adapted from

Table 7: Benchmarks used in the experimental evaluation for LaPerm scheduler

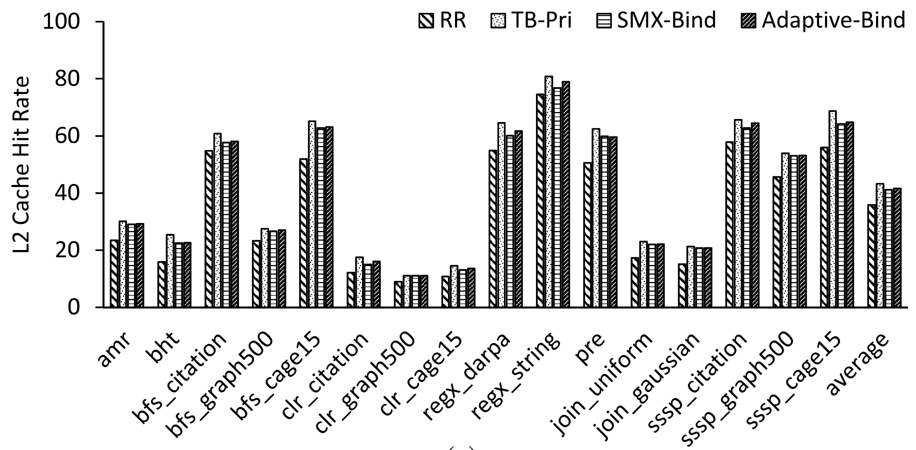
Application	**Input Data Set**
Adaptive Mesh Refinement (AMR)	Combustion Simulation[36]
Barnes Hut Tree (BHT) [15]	Random Data Points
Breadth-First Search (BFS) [47]	Citation Network[8] Graph 500 Logn20[8] Cage15 Sparse Matrix [8]
Graph Coloring (CLR) [19]	Citation Network[8] Graph 500 Logn20[8] Cage15 Sparser Matrix [8]
Regular Expression Match (REGX) [75]	DARPA Network Packets [45] Random String Collection
Product Recommendation (PRE) [49]	Movie Lens [28]
Relational Join (JOIN) [22]	Uniform Distributed Data Gaussian Distributed Data
Single Source Shortest Path (SSSP) [37]	Citation Network[8] Graph 500 Logn20[8] Cage15 Sparser Matrix[8]

the ones used for DTBL with different input sets as shown in Table 7. The reported results include the overhead from both CDP/DTBL as well as the proposed LaPerm scheduler.

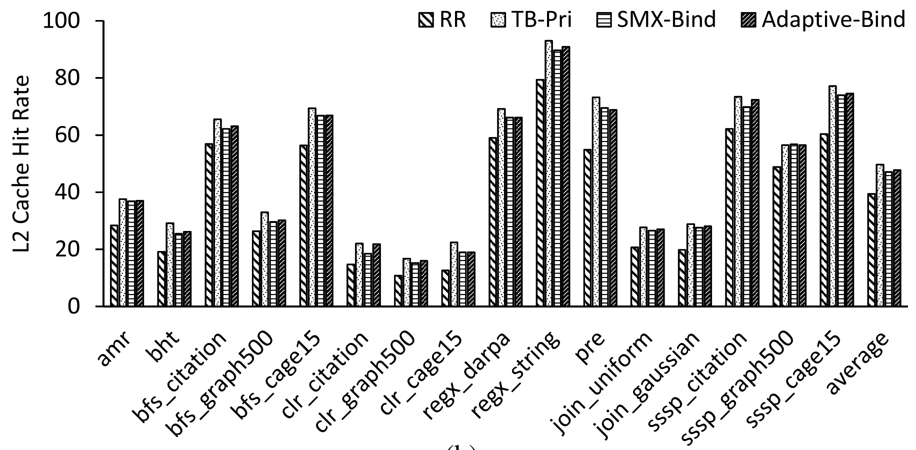
5.3.2 Result and Analysis

This section reports the evaluation and analysis of the benchmark in various performance aspects. As the main focus of LaPerm is the memory system performance especially L1 and L2 cache, the cache hit rate is used as the metrics. The impact of LaPerm on the IPC (instruction per cycle) metrics is also analyzed to evaluate the overall performance of the applications. All the evaluations are performed both for the CDP and DTBL model. Figure 29 and Figure 30 show the L2 and L1 cache hit rate respectively for the original CDP and DTBL using the RR TB scheduler as well as the three different schemes employed by LaPerm. Figure 31(a) and Figure 31(b) show the IPC normalized to the original IPC of CDP and DTBL implementations with RR scheduler respectively.

Performance of *TB-Pri*. As discussed in Section 5.2.1, the goal of *TB-Pri* is to increase the cache hit rate by prioritizing child TBs earlier after parent TBs. This is demonstrated



(a)



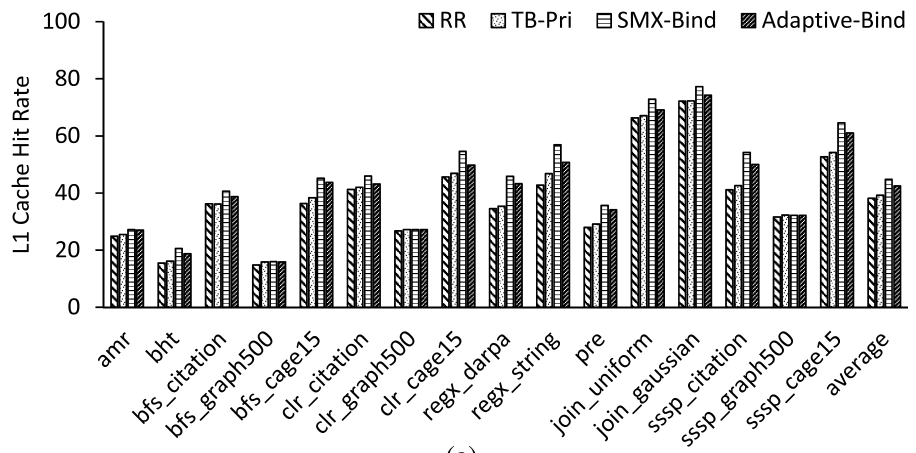
(b)

Figure 29: L2 cache hit rate when applying LaPerm to (a) CDP and (b) DTBL.

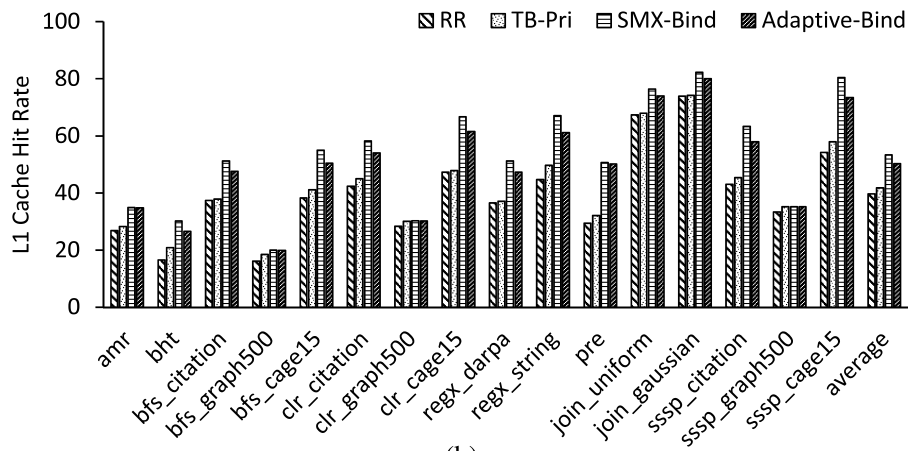
by an average increase of 6.7% (CDP) and 8.7% (DTBL) for L2 cache hit rate and 1.1% (CDP) and 2.1% (DTBL) for L1 cache hit rate over RR scheduler. Together they also result in 4% and 13% normalized IPC increase for CDP and DTBL respectively.

Some of the benchmarks that achieve the highest L2 cache hit rate are *pre* and all the graph applications (*bfs*, *clr*, *sssp*) with the *cage15* input. These benchmarks generally have more dynamic child TB launching and higher parent-child shared footprint ratio as shown in Figure 24 and benefit more if the child TBs are able to reuse the data from the parent or the sibling TBs.

Performance of *SMX-Bind*. Although *TB-Pri* does not target L1 cache performance, there is still a slight increase in the L1 cache hit rate. This is because child TB prioritization



(a)



(b)

Figure 30: L1 cache hit rate when applying LaPerm to (a) CDP and (b) DTBL.

can result in a few child TBs coincidentally being dispatched to the same SMX as the direct parent TB. This dispatching pattern is reinforced by *SMX-Bind* to achieve a L1 cache hit rate increase shown as 6.6% on average for CDP and 13.6% on average for DTBL.

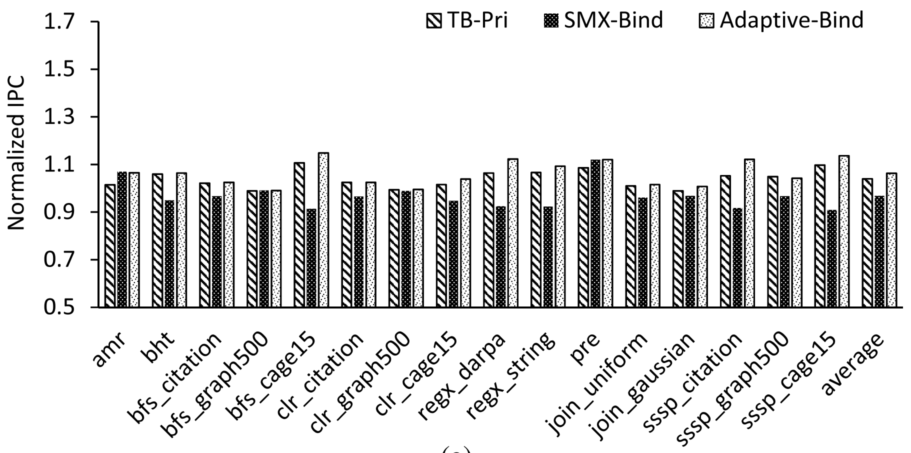
The applications *pre* and *sssp_cage15* are again among the ones that achieve the highest L1 cache hit rate. In addition, *regx_string* also exhibits good L1 cache performance benefit. These applications have the characteristics that the workload performed by the child TBs focus on a relatively small memory region. For example, the production recommendation process of *pre* tends to search products that are highly related and thereby stored closer to each other in the memory. As a consequence, these applications can generate more closer memory accesses and higher child-sibling shared footprint ratio. When all these sibilant

TBs are scheduled on the same SMX, they fundamentally increase the data reuse which result in substantial L1 cache hit rate increase.

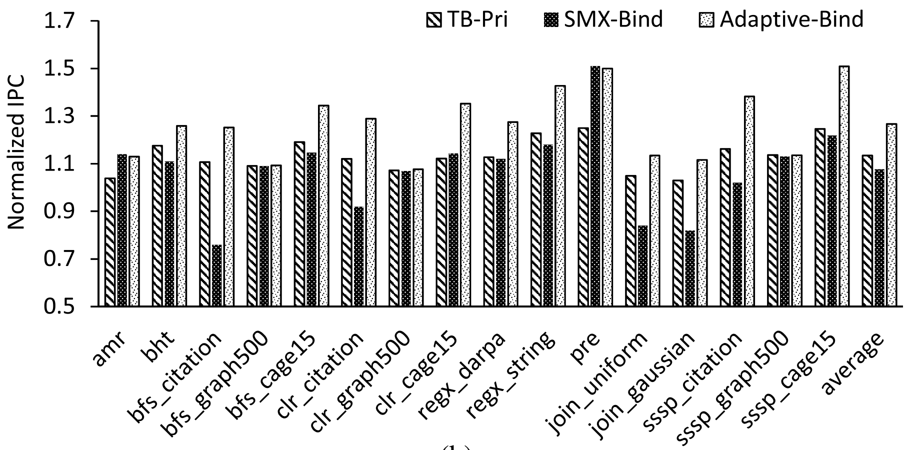
In contrast, for the graph applications with *graph500* as input, *SMX-Bind* does not have any obvious L1 cache hit rate change from *TB-Pri*. Although these applications present some shared footprint ratio between the child TBs and their direct parent, the locality actually can also exist between any arbitrary non-direct parent TB and child TBs. The reason is that *graph500* is a graph with high and balanced connectivity that are evenly distributed across all the vertices. The data used by one parent TB exploring some of the vertices can be effectively reused by child TBs generated by a different parent exploring other vertices. The consequence of increased locality and cache performance from such data reuse patterns have already been captured by *TB-Pri*. Binding child TBs to specific SMX does not necessarily generate a higher L1 cache hit rate.

One major side effect of *SMX-Bind* is the SMX workload imbalance which may result in IPC decrease. Compared with *TB-Pri*, the average normalized IPC decreases 9% for CDP and 5% for DTBL. For some of the DTBL applications (*bfs_citation*, *clr_citation*, *join*) and almost all of the CDP applications, normalized IPC even drops below 100% indicating performance loss from the baseline implementations with the original RR TB scheduler. Applications suffer from larger IPC loss generally have a more imbalanced child TB launching patterns, i.e. some parent TBs may have substantially more child TBs and nested launching level than others, causing a long execution tail when these TBs are exclusively restricted to an SMX.

Performance of *Adaptive-Bind*. By using the adaptive SMX binding approach provided by *Adaptive-Bind*, the SMX workload imbalance side effect brought by *SMX-Bind* is minimized, which results in overall normalized increase of 6% for CDP and 27% for DTBL at the cost of some L1 cache hit rate decrease (by 2.3% for CDP and by 3.1% for DTBL compared with *SMX-Bind*). The study shows that IPC is impacted by L1 hit rate and load balancing – in fact IPC improvements due to the latter are greater than IPC reductions due



(a)



(b)

Figure 31: Normalized IPC when applying LaPerm to (a) CDP and (b) DTBL.

to the drop in L1 rate. The results demonstrate that *Adaptive-Bind* effectively combines the benefits of prioritizing child TB execution, SMX binding and load-balance TB scheduling to achieve cache and overall performance gains for irregular applications that are implemented with the dynamic parallelism model. As a representative, application *sssp_cage15* achieves the highest IPC gain (11% for CDP and 51% for DTBL).

It is interesting to see some of the applications, such as *amr* and *pre*, have their normalized IPC increase from *SMX-Bind* and keep the value in *Adaptive-Bind* without any obvious further increase. The reason is that they have a more balanced kernel launching patterns among the some or all of the parent TBs. For example, *amr* has TBs in the grid centers to simulate the combustion patterns which all have similar temperature distribution, requiring similar refinement performed by the child TBs. Binding their child TBs to the SMX occupied by the direct parent will generate good L1 cache performance without causing many workload imbalance issues. Therefore, *SMX-Bind* would itself be a reasonable scheduling strategy for these applications to achieve IPC increase and does not require the SMX re-balancing process from *Adaptive-Bind*.

There are slight L2 cache hit rate changes compared with *TB-Pri* and *SMX-Bind*. In fact, increasing (decreasing) L1 cache hit rate may result in fewer (more) memory accesses falling into L2 cache, which could change the L2 cache behavior. According to the experiments, these changes do not affect L2 cache hit rate substantially and are not the major factors in affecting the overall performance.

5.3.3 Impact of Different Dynamic Parallelism Models

The microarchitecture and runtime differences of dynamic parallelism models such as CDP and DTBL can have impact on the effectiveness of the LaPerm scheduler. One of the major differences is the launching latency as described in Section 5.2.4. The evaluations for both CDP and DTBL reveal that generally LaPerm in DTBL shows better cache performance and greater IPC increase (27% versus 6% in CDP) largely due to the fact that the high launching latency of the child kernels precludes LaPerm from timely dispatch to be executed closer to the parents in time. As for some applications such as *bfs*, parent TBs usually only have a small amount of work to do, long child launching latency leaves LaPerm no choice but only to schedule the remaining parent TB first before any child TBs arrive to fill the time gap.

Recall that CDP implementation today is subject to the 32 concurrent kernel limit in the KDU, reducing the number of child TBs that are available for LaPerm to schedule. As a result, the opportunity for LaPerm to perform an optimized TB prioritization using *TB-Pri* is dramatically reduced. The limit on the scheduling of TB candidates also causes poorer SMX imbalance for *SMX-Bind* as it is more likely to dispatch the available TBs to only a few SMXs but not other, which is the reason of the poor IPC of CDP that is even lower than using the original round-robin TB scheduler. As opposed to CDP, DTBL use dynamic TB coalescing to break the KDU limit and increase TB level concurrency, which makes LaPerm a more effective and efficient solution for the TB scheduler.

5.3.4 Insights

The experiments and results show that the three different scheduling decisions employed by LaPerm have various performance impacts on applications with different characteristics. Some of the insights include:

- *TB-Pri* uses child TB prioritization to increase L2 cache performance and is specifically useful for applications where the locality is not restricted to the direct parent and its child TBs but also between multiple parents and their child TBs. Such locality facilitates the data reuse across different SMXs.

- For applications with more restricted locality between direct parent and child TBs, *SMX-Bind* is able to show the most obvious L1 cache performance improvement. On the other hand, the overall IPC may be optimized only when there are many parent-child launchings with similar workload to achieve SMX balance.

- There is a basic tradeoff between exploiting parent-child and child-sibling locality, and achieving higher SMX utilizations. For most irregular applications which show varying parent-child launching behavior across different parent TBs, *Adaptive-Bind* is the TB scheduler to achieve both the best cache performance and the balanced SMX workload which results in overall IPC increase.

5.4 Summary

This chapter proposes a thread block scheduler, LaPerm, specifically designed for dynamic parallelism execution models on GPUs. The idea behind LaPerm is that the memory locality exists between the parent and child thread blocks that cannot be effectively exploit by existing round-robin TB scheduler on current GPUs. LaPerm employs three different scheduling decisions with new microarchitectural extensions to utilize such parent-child locality and improve the cache performance on GPUs. LaPerm is evaluated on a cycle-level GPU simulator with several CUDA irregular applications that are implemented with dynamic parallelism execution models, and demonstrate that by increasing both the L1 and L2 cache performance, LaPerm is able to achieve 27% IPC improvement over the original round-robin TB scheduler.

CHAPTER VI

IMPROVING THE POWER EFFICIENCY OF THE DTBL MODEL

As the last research part of this thesis, this chapter examines the SMX occupancy patterns of the irregular applications implemented with the DTBL model from the power dissipation perspective and proposes a new power saving optimization that incorporates a flexible TB diversion and scheduling strategy as well as an opportunistic DVFS mechanism.

6.1 Impact of SMX Occupancy Patterns on Power Dissipation

This section compares the SMX occupancy patterns of the DTBL model with the general BSP GPU model for processing dynamic parallelism in the irregular applications and makes the observation that the SMX occupancy variance in DTBL implementation provides the potential opportunities for an power saving optimization.

6.1.1 DTBL vs. Persistent Thread (PT) Model

The DTBL model enables more flexible TB dispatching and execution by allowing them to be generated on demand to process the dynamic work. This is in contrast to the typical flat implementations for dynamic parallelism applications represented by PT model, where a fixed number of TBs are dispatched to the SMXs and stay on the SMXs for the lifetime of a kernel. The number of TBs used in the PT model is chosen such that maximal occupancy can be achieved by the SMXs which is determined by the thread number, register number and share memory usage by a single TB. Theoretically, the maximal occupancy that can be achieved is 100% which means maximal number of concurrent warps that can be executed on a SMX is reached. Accordingly, the PT model maintains a steady SMX occupancy during the lifetime of a kernel. These TBs are then used to process the work in the kernel: the work can be either assigned at the very beginning of the kernel, or generated dynamically by these TBs during their processing. A global software queue is employed to manage these work. TBs constantly check the global queue to consume any existing work and append new work to the queue. Processing dynamic work requires power consumption from the SMXs which consists of the static leakage power and the dynamic power from all the GPU components such as register files, shared memory, execution units, main memory, etc. [41]. Note that the SMX cannot be completely idle even if there is no effective dynamic work available to process as it is still necessary for the TBs to perform the global queue checking operations iteratively.

The DTBL model, on the other hand, can result in a higher variance in the SMX occupancy compared with the PT model. This is because the amount of dynamic work generated can differ from time to time. Unlike the PT model, TBs in the DTBL will only be generated and dispatched to the SMX when dynamic work is available. Therefore, DTBL implementations can have different number of concurrent warps running on a SMX at different stages of the execution. This also includes the possible case where a SMX can be completely idle at one time as there is not enough dynamic work to saturate all the SMXs on the GPU. Note that the SMX still consumes a small amount of dynamic power as well as non-negligible static leakage power even if it is completely idle, which is referred combined as *SMX idle power* [41].

As a case study, Figure 32 and Figure 33 show the SMX occupancy phase behavior and total GPU power consumption during a vertex expansion kernel for the fourth iteration of the BFS application implemented with the DTBL and the PT model, repsectively. The data are obtained from the cycle-level simulator GPGPU-Sim [9] with the power model simulator GPUWattch [41]. All the measurements are done with a sampling rate of 500 cycles. In each figure, the SMX occupancy is shown for two of the SMXs on a GPU (SMX 3 and SMX 11) while the power consumption is shown for the entire GPU.

The comparison between the the DTBL and the PT models show that the SMX occupancy for the DTBL model has a much higher variance during the execution. They both start with a constant SMX occupancy as the parent kernels are preparing the data for

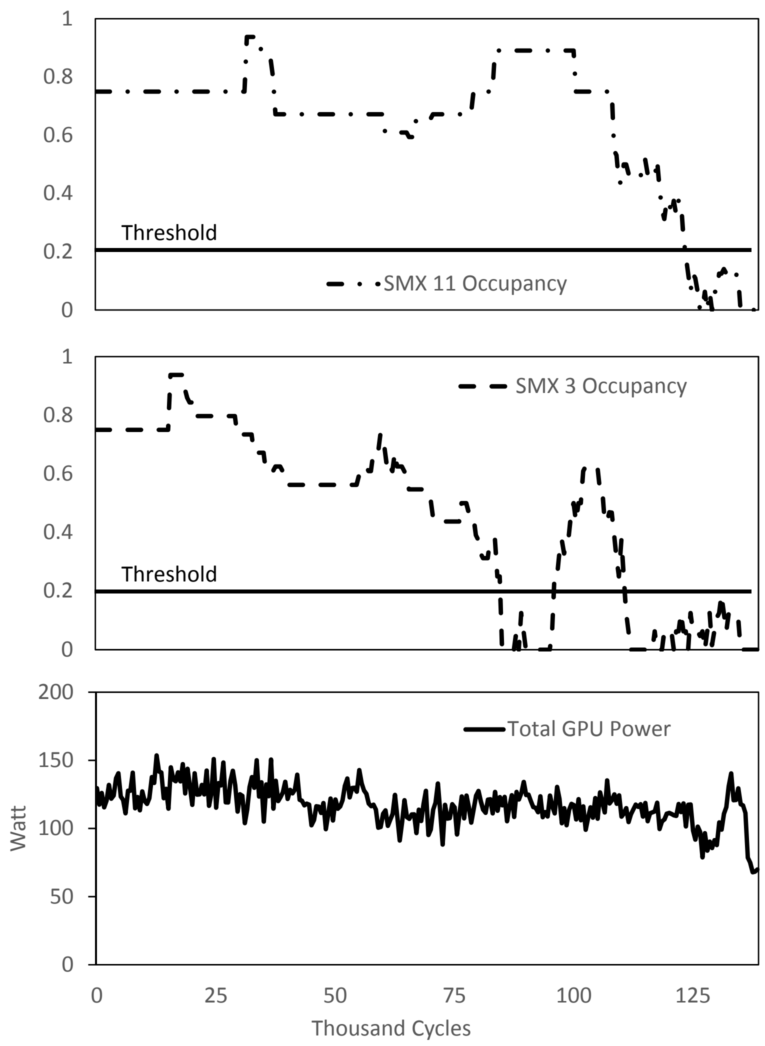


Figure 32: A case study for the SMX occupancy and power consumption in one vertex expansion kernel of BFS with DTBL implementation

dynamic work. Subsequently, the SMX occupancy of the DTBL implementation varies depending on the dynamic parallel degree in the vertex expansion operation and thereby the speed of parent kernel generating new dynamic TBs while the SMX occupancy of the PT implementation stays the same. Both implementations show a tail where SMX occupancy gradually decreases to zero to indicate the completion of the kernel. Generally, for the benchmark applications in Table 7, the SMX occupancy has a 23.5% higher variance with the DTBL implementation than the PT implementation.

When examined from the power dissipation perspective, Figure 32 and Figure 33 show

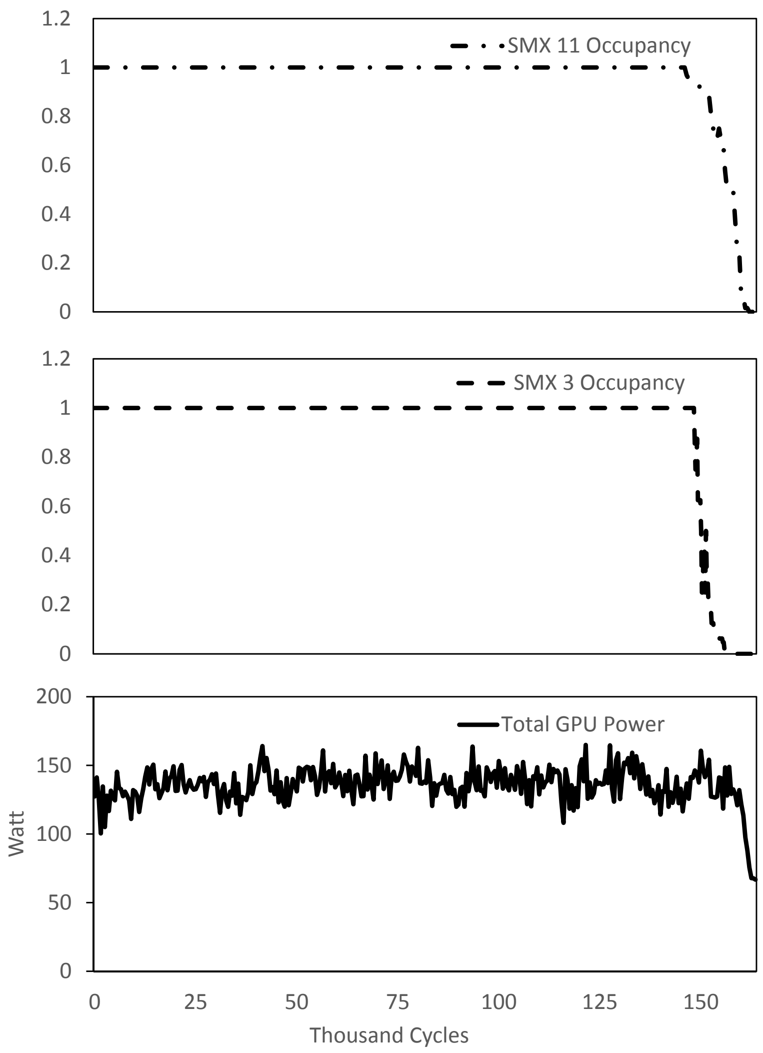


Figure 33: A case study for the SMX occupancy and power consumption in one vertex expansion kernel of BFS with PT implementation

the following features of the DTBL and the PT implementations:

1. The DTBL implementation has an average lower power consumption than the PT implementation (117.6W vs. 135.96W). This is mainly due to the SMX occupancy variance which results in lower SMX utilization while there is smaller amount of dynamic work. Generally, DTBL implementations of the benchmark applications achieve a 12.7% power consumption decrease compared with PT implementations.

2. DTBL also has a lower total energy consumption compared with PT. The lower average power consumption of DTBL contributes to the lower total energy consumption.

It should also be noted that the vertex expansion kernel implemented with DTBL performs better in terms of the total execution time compared with PT. This is shown as smaller total cycle count in Figure 32 than in Figure 33 (138.5K vs. 163.5K).

3. The power consumption during the final phase of the execution of both DTBL and PT gradually decreases, which corresponds to the decrease of the SMX occupancy during this phase. Eventually the power consumption reaches the *SMX idle power* when all the SMXs are completely idle.

6.1.2 SMX Occupancy Bubble for DTBL Implementation

Aside from the fact that DTBL implementations already exhibit lower power dissipation than the PT implementations, this section further studies the SMX occupancy behavior to find any other possible opportunities for power saving.

In Figure 32, the occupancy of SMX 3 may drop close to zero during the execution (between cycle 85K and 96K) and then resume to a higher value later. This is in contrast to the occupancy of SMX 10 which is generally maintained at a high level except towards the end of the execution. To quantify such SMX occupancy drop, a threshold is introduced and the period where the SMX occupancy drops below the threshold is referred as *SMX Occupancy Bubble*. Note that the occurrence of an *SMX Occupancy Bubble* may differ for different thresholds. Figure 32 shows a threshold of 0.2 and there is one *SMX Occupancy Bubble* on the SMX 3 which lasts for 11K cycles.

In general, Figure 34 shows the occurrence percentage of *SMX Occupancy Bubble* for the 0.0625 SMX Occupancy threshold (4 concurrently executing warps on a SMX). The 16 benchmark applications are divided into two groups according to their *SMX Occupancy Bubble* behavior. The applications in the first group has an average of 32.7% occurrence as they generally have highly dynamic work generating patterns. The applications in the second group only has an average of 4.1% occurence because i) they may generate dynamic work in a more balanced pattern, represented by all the graph applications with *graph500* input as it has uniform vertex degree, ii) there are large amount of dynamically generated TBs which may saturate the GPU, represented by the two *regx* applications as they are dealing with highly-intensive data set, or iii) the generated dynamic work are coarse-grained with high number of threads which could result in high SMX occupancy for the entire execution period, represented by *pre* and *join_uniform* due to the nature of their implementation schemes.

From the power dissipation perspective, *SMX Occupancy Bubble* is a period which potentially has low SMX utilization and long execution unit stalls or even complete SMX idleness. Compared with the period where SMXs are highly-utilized, *SMX Occupancy Bubble* would consume less power. This also includes the case where an SMX is completely idle and therefore only consumes *SMX idle power* just as the final phase of the execution shown in Figure 32 and Figure 33. Note that in some cases, the SMX utilization could still be high even during the *SMX Occupancy Bubble* where there is at least one warp running. This could happen in a compute-bound application where there is much fewer or even no SMX execution unit stalls caused by memory instructions and therefore the execution units are always busy. Generally, *SMX Occupancy Bubble* imposes the following effect on the dynamic power and leakage power as described in the GPU power model:

1. *SMX Occupancy Bubble* would result in the dynamic power decrease of the SMX. However, even an idle SMX would consume dynamic power to maintain the operation of its active components, such as the control logic that maintains the SMX scheduler registers so that it can be resumed to its normal active status when new TBs are ready to be scheduled.

2. The leakage power of the SMX stays the same during the *SMX Occupancy Bubble* as it is independent of SMX utilization.

Ideally, applying power gating to an idle SMX by turning it completely off could save

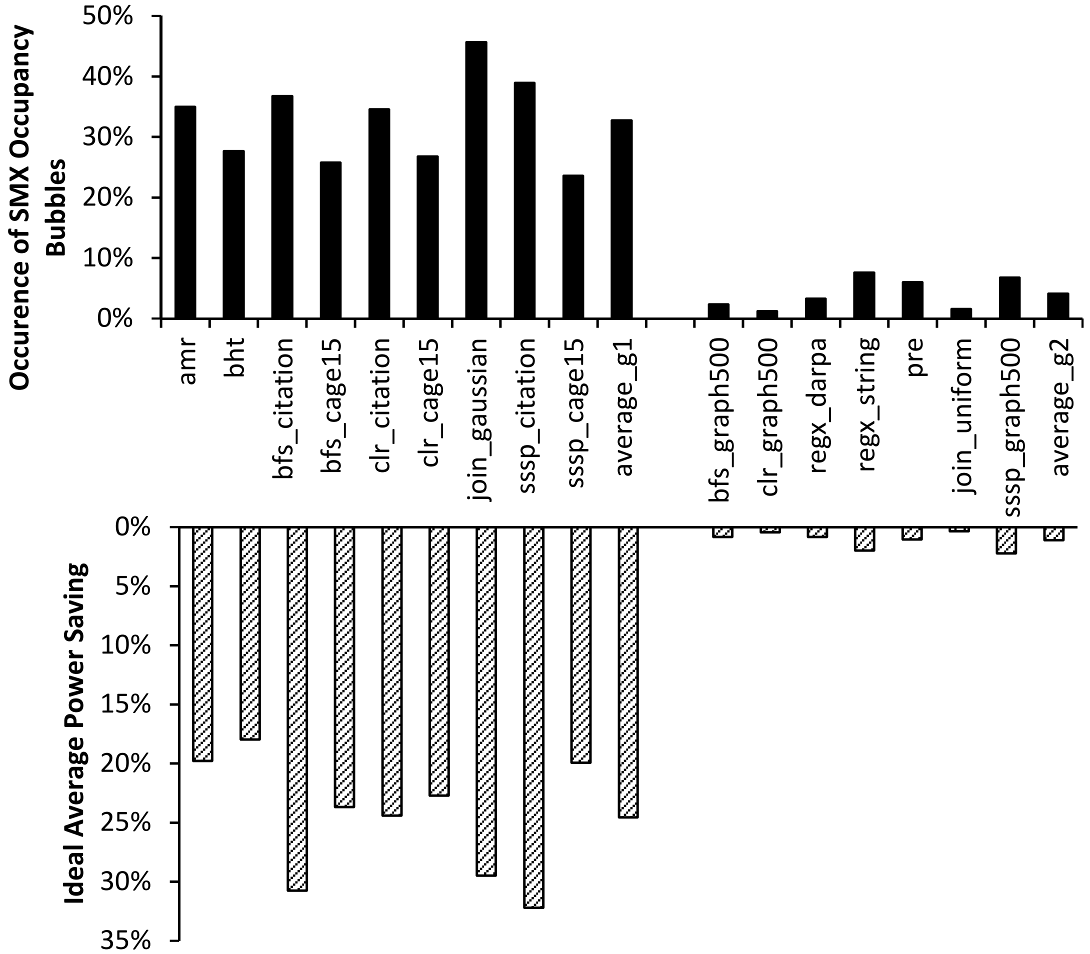


Figure 34: The occurrence percentage of *SMX Occupancy Bubble* for threshold of 0.0625 (4 warps) and ideal average power saving

both the dynamic power as well as the leakage power. Figure 34 shows the potential average power saving in this ideal case. The idea is to assume that the SMX is completely idle during the *SMX Occupancy Bubble* so that there is only *SMX idle power* consumption. Applying ideal power gating would further reduce the *SMX idle power* to zero. In this process, the original power consumption is measured through GPUWattch while the *SMX idle power* is the preset value in the GPUWattch configuration file according to their original verification on the real hardware [41]. The two groups of benchmark applications exhibit different power saving behavior as correlated to their *SMX Occupancy Bubble* occurrence. While the second group only has an average of 1.1% power saving, the first group has an average of 24.5% power saving when both the dynamic power and the leakage power are eliminated for the idle SMX. The potential opportunity of power saving for the first group of applications is the major motivation of a new power saving optimization for DTBL. However, there are two challenges that should be addressed in such an optimization.

1. The SMX is not completely idle during the entire *SMX Occupancy Bubble* in the actual execution process of an application. The running warps on the SMX should be relocated before power gating can be properly applied to turn off the SMX.

2. An ideal power gating is not possible on current GPUs. Alternatively, DVFS is able to set an SMX to the lowest P-state to save both the dynamic and leakage power. Applying DVFS would require careful design methodology to minimize the overhead while achieve best energy efficiency.

6.2 SMX Occupancy Bubble-based Power Saving Optimization for DTBL

To address the above challenges, this section proposes a power saving optimization for DTBL which is composed of two stages: 1) a dynamic TB diversion and scheduling strategy that can decrease the SMX occupancy to zero during the SMX Occupancy Bubble when necessary so that the SMX is completely idle and 2) an opportunistic DVFS mechanism that set the idle SMX to the lowest P-state to save both the dynamic and leakage power while imposing minimal performance impact.

6.2.1 Dynamic TB Diversion and Scheduling

The purpose of dynamic TB diversion and scheduling for DTBL is to further reduce the SMX occupancy during the *SMX Occupancy Bubble*, ultimately to zero, so it could facilitate the usage of the power saving techniques such as power gating or DVFS. Three steps are necessary in this diversion and scheduling process: *TB Diversion*, *SMX Draining* and *SMX Resuming*. The detailed description of these steps are as follows:

TB Diversion. TB Diversion relocates the TBs that are to be scheduled on the current SMX to another SMX. It does not affect the TBs that are currently executing on the SMX but only guarantees no new TBs will be dispatched to the current SMX.

The TB diversion process is done for each SMX. It starts by detecting the occurrence of *SMX Occupancy Bubble* according to a threshold T and activated at the first time sample when the SMX occupancy falls below the threshold T. At the same time, the status of the remaining SMXs are examined to determine if there exists an SMX that 1) does not need TB diversion at the moment, 2) SMX occupancy below 100% and 3) enough resources available for scheduling the relocated TB on the current SMX. Requirement 1) would guarantee the diversion process free of deadlock especially during a period when all SMXs have low occupancy (e.g. execution tail) so that at least one SMX can be served as the relocation destinations for all the remaining SMXs. Requirements 2) and 3) would locate at least one SMX that can be served as the relocation destination for the current SMX. If any of these three requirements could not be satisfied, the TB diversion process will abort. Otherwise, it will start dispatching the new TBs to the remaining SMXs on the GPU as if the current TB is not available.

SMX Draining. While TB Diversion would stop any potential SMX occupancy increase on the current SMX, the SMX Draining process will ensure the SMX occupancy reaches zero. The idea is simply for the SMX to wait until the current running TBs to finish. Alternatively, an aggressive TB Diversion strategy could employ TB preemption [69] instead of SMX Draining. However, the overhead of saving and resuming TB states could be nontrivial and cause dramatic performance degradation. Therefore, this research only focuses on the low-overhead SMX Draining strategy. However, unlike TB preemption, a TB could take arbitrarily long time to finish during SMX Draining, although it eventually would conclude unless there is a deadlock. Therefore, the proposed SMX Draining process sets a waiting period for W time samples as the maximum draining time. If the executing TBs on the current SMX could not complete after the waiting period, both the SMX Draining and the TB Diversion will abort, which means new TBs will be dispatched to the current SMX as they normally would. A successful SMX Draining process, on the other hand, would end with a complete idle SMX where the power saving techniques can be applied.

There are two cases where the SMX Draining process is not necessary. In the first case, when the TB diversion process starts, the SMX occupancy of the current SMX is already zero. As no new TBs will be dispatched to the current SMX, SMX Draining could also be avoided. The second case is related to the application of the DVFS power saving technique. Since DVFS does not completely shut down an SMX as power gating, it is not always mandatory to ensure the SMX is idle before applying DVFS. The details will be discussed in Section 6.2.2.

SMX Resuming. A succesful TB diversion and SMX draining would effectively eliminate the availability of the current SMX to potentially save power when the workload intensity reduces during the execution of a DTBL kernel. However, when the workload intensity resumes, the TB diversion process should also be stopped so that the current SMX could be utilized again. The SMX Resuming process will be made effective in this case. It happens when there are TBs waiting in the GPU but no SMX is available either because all of 100% SMX occupancy or there is not enough resource on the SMX available for the next TB. As a consequence, the idle SMX would be resumed to consume the new TBs. If there are multiple idle SMXs which have been through the TB diversion process, they will be resumed one at a time at each time sample if necessary.

Figure 35 illustrates an example of dynamic TB diversion for one vertex expansion kernel of BFS with DTBL implementation. It shows the SMX Occupancy of SMX 3 and SMX 11 before the diversion just the same as Figure 32. There are two TB Diversion processes for SMX 3. The first one happens at the beginning of the *SMX Occupancy Bubble*. Since the SMX Occupancy of SMX 3 is already zero, there is no SMX Draining process. The TB

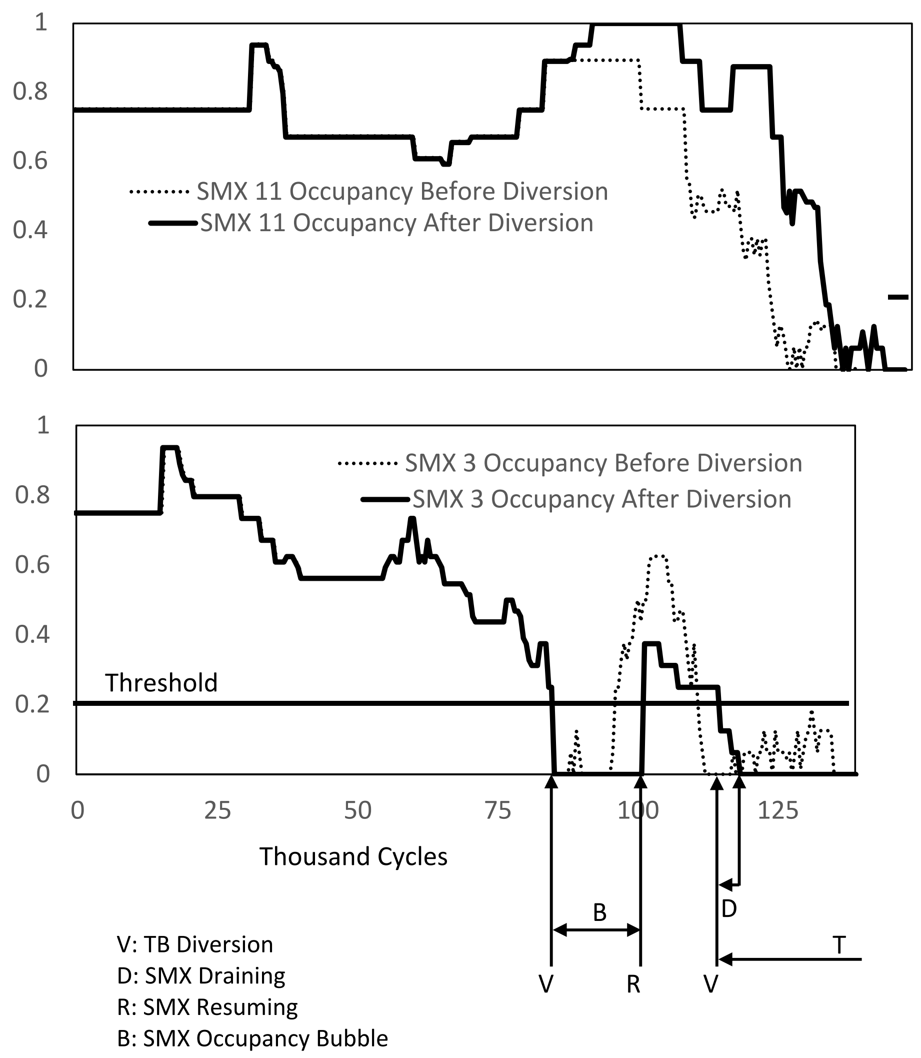


Figure 35: A case study for the Dynamic TB Diversion for one vertex expansion kernel of BFS with DTBL implementation

diversion of SMX 3 and other SMXs cause an increase in the SMX occupancy of SMX 11, which eventually reaches 100%. Subsequently, SMX 3 detects more workload on the GPU and performs the TB Resuming to leave the SMX idle status.

The second TB diversion process happens when the SMX Occupancy of SMX 3 drops below the threshold again around cycle 115K. The SMX draining process is activated this time to gradually reduce the SMX Occupancy to zero. It is interesting to notice that this is actually the execution tail of this kernel, so there is no SMX resuming as the relocated TBs will not saturate other SMXs (e.g. SMX 11). Generally, during the execution tail of a kernel, TB diversion process may or may not happen depending on whether or not the SMX occupancy of all the SMXs drop below the threshold at the same time sample. It is also possible that all the TBs will eventually be diverted to one SMX. The SMX Resuming process generally does not happen.

It should also be noted that in Figure 35, SMX 11 takes longer time to finish all the TBs when the TB Diversion is applied. This imposes a negative impact on the performance of the DTBL implementation, which brings up the question of how to trade off between the performance degradation and energy saving. Section 6.3 will evaluate the performance impact and energy efficiency of the proposed research in further details.

6.2.2 Opportunistic DVFS

After the TB diversion process reduces the SMX occupancy to zero, the power saving techinque can be applied. As discussed before, ideal power gating technique could completely eliminate the dynamic and leakage power of an idle SMX to achieve substantial power saving. This research examines a more practical DVFS strategy adopted by the GPU Power simulator GPUWattch [41] which sets the SMX to the lowest P-state with low frequency to reduce dynamic power and scales the voltage with frequency using a predictive technology model [2] to reduce leakage power. The goal is to demonstrate the effectiveness of the proposed power saving optimization for DTBL when incorporating its execution phase behavior.

Applying DVFS. DVFS is applied after the SMX Draining process completes (if it is necessary). The transition between different SMX P-states and voltages would require time *P*, after which the SMX will be set to the lowest P-state and scaling voltage. When SMX Resuming is required due to increasing workloads on the GPU, DVFS will make another transition to the normal P-state which is the highest frequency with corresponding voltage scaling. Again, such a transition would take time *P*.

GPU-level and SMX-level DVFS. GPU-level DVFS sets the entire GPU to the lowest P-state and scaling voltage while SMX-level DVFS only works on a single SMX. While the latter is the major technique used in this research combining with the TB diversion process, the former could be useful when all the SMXs are idle during the execution tail.

DVFS with SMX Draining. SMX Draining is not mandatory with DVFS. The DVFS can be applied right after the TB diversion while the SMX occupancy is not zero. The argument is that the negative performance impact brought by lower SMX frequency and voltage would not last long as the workload remaining on the SMX is minimal (occupancy is limited by the threshold used in the SMX diversion) while the upcoming power saving could be beneficial. Again, the tradeoff between the performance and power saving is discussed in Section 6.3.

6.2.3 Incorporating Dynamic TB Diversion and DVFS

Figure Figure 36 shows the high-level flow for the SMX Occupancy Bubble-based power optimization for DTBL that combines both the dynamic TB diversion and opportunistic DVFS. This flow will be performed on each SMX through the SMX scheduler at every time sample. It starts by checking if the SMX is currently in a DVFS transition. This step is to ensure that any DVFS decision, either the GPU-level or the SMX-level, made at an earlier time sample is completed as the transition time could be several time samples. Then the SMX scheduler will examine if all the SMXs are in the idle state, which is specifically designed for any execution tail so that the GPU-level DVFS can be applied accordingly. The remaining steps in the optimization flow is divided into two parts: the left part corresponds to the process of enforcing TB Diversion on an SMX with or without SMX Draining, which eventually leads to the lowest P-state setting through DVFS; the right part corresponds to the processing of resuming the normal status of an SMX by resetting it to the normal P-state. The TB scheduling for this SMX will be resumed after the P-state transition completes.

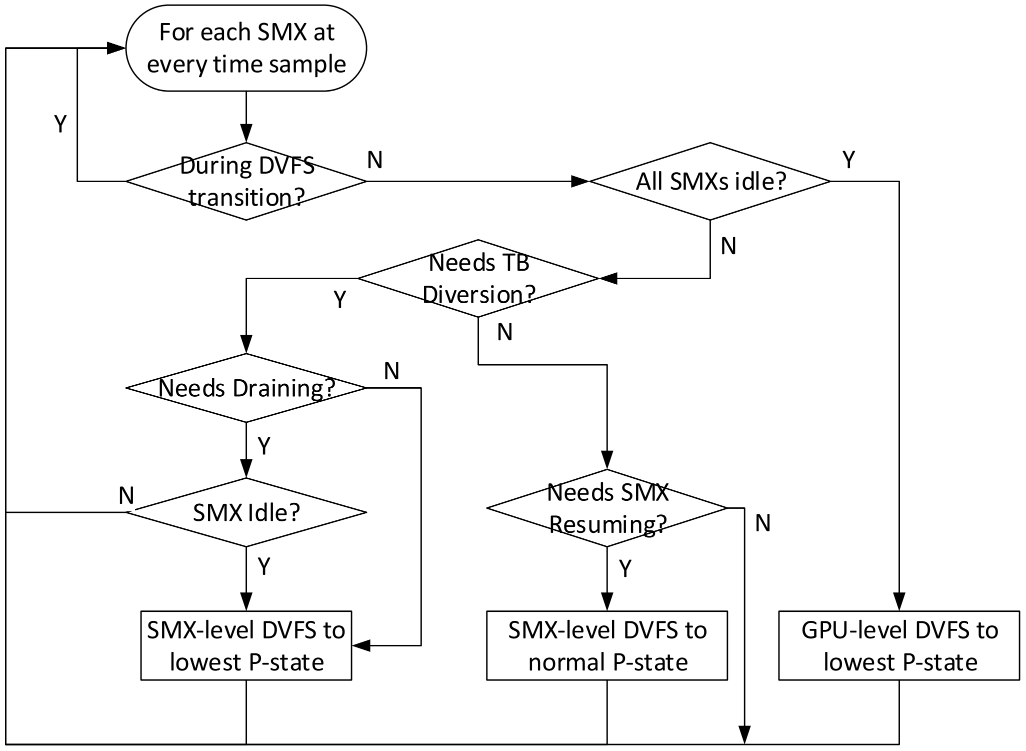

For each SMX at every time sample
During DVFS transition?
Y
N
All SMXs idle?
Y
N
Needs TB Diversion?
Y
N
Needs Draining?
N
Y
SMX Idle?
N
Y
Needs SMX Resuming?
N
Y
SMX-level DVFS to lowest P-state
SMX-level DVFS to normal P-state
GPU-level DVFS to lowest P-state

Figure 36: High-level Flow Chart for the SMX Occupancy Bubble-based Power Optimization for DTBL

6.2.4 Architecture Extension

The support of the proposed power saving optimization requires the current GPU microarchitecture to be extended, specifically for enabling TB diversion and DVFS at different levels according to different SMX status. As shown in Figure 37, the SMX scheduler is extended with both the new Diversion Control Registers and the extra scheduling logic for TB scheduling and SMX status monitoring. The Diversion Control Registers are composed of three registers: diversion status (DVR), draining status (DRN) and DVFS control (DVFS). Each bit of the DVR and DRN indicates the current TB Diversion and SMX Draining status of an SMX. The first bit of the DVFS register indicates if the GPU-level DVFS is activated while the remaining bits are used for SMX-level DVFS of each SMX. There is no separate register for the SMX Resuming status as it can be indicated by a clear DVR bit and a valid SMX-level DVFS bit – there is no TB Diversion on the SMX but DVFS is activated which

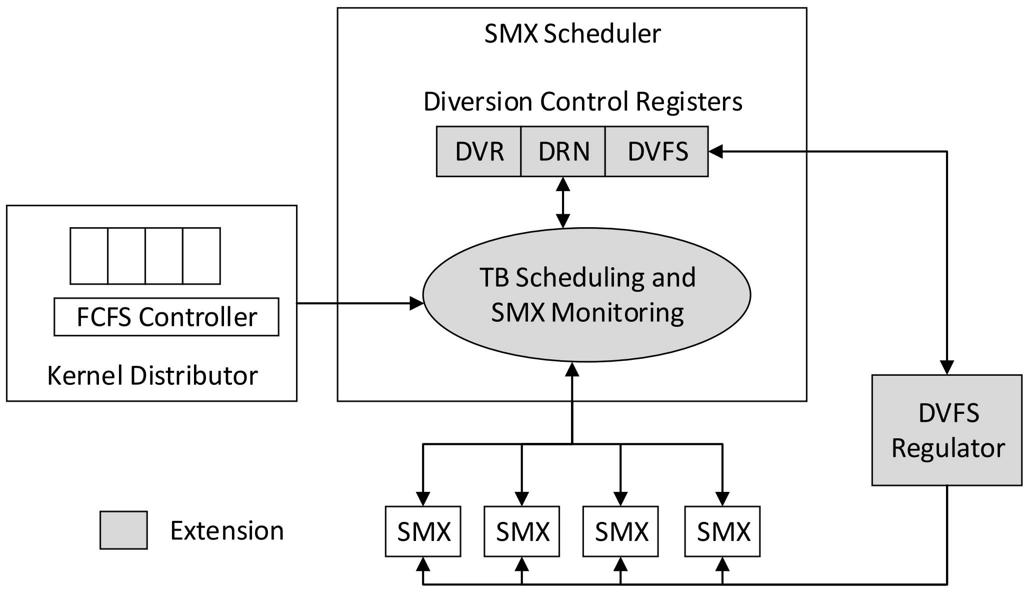


Figure 37: Architecture Extension for the SMX Occupancy Bubble-based Power Optimization for DTBL

resume the SMX to its normal P-state. The new SMX Monitoring logic in the SMX scheduler will set/reset the values of the Diversion Control Registers according to Figure 36 by examining the SMX Occupancy of all the SMXs. The new TB scheduling logic is responsible to dispatch the TBs from the kernels in the Kernel Distributor (both the native TBs and the dynamic TBs in DTBL) to the SMXs according to the values of DVR, that is, the SMX with a set DVR bit will not be used as the scheduling destination. The DVFS Regulator can be either on-chip or off-chip which could result in different transition time *P*. It is designed in a way that the voltage and frequency of each single SMX can be controlled separately to enable SMX-level DVFS. The decision of using DVFS Regulator to set the SMX to the normal P-state or the lowest P-state is made with the DVFS control register in the SMX scheduler. The DVFS regulator can also update the DVFS control register after the transition is completed.

The proposed microarchitecture extension introduces area overhead through the new control registers and scheduling logic. The Diversion Control Registers together take 6 bytes assuming a 13-SMX Kepler GPU. The TB Scheduling and SMX Monitoring logic is implemented as a state machine on top of the original SMX Scheduler which would be the large area overhead. The microarchitecture extension itself does not introduce extra scheduling cycles as the TB Scheduling and SMX Monitoring process will be integrated into the original SMX Scheduling process and only happen at the designated time sample. However, the scheduling process may introduce performance overhead to the application which will be evaluated in Section 6.3.

6.2.5 Discussion

The power saving optimization proposed in this chapter is based on the execution phase behavior of the applications implemented with DTBL. As the dynamic parallelism in DTBL exhibits substantially different behavior in terms of TB execution than a traditional GPU BSP execution model, the strategy here is to focus on the TBs scheduling and SMX occupancy that lead to different power consumption consequence. Therefore it can be integrated with many other GPU power models and optimizations that are applicable to the general GPU applications. Specifically, techniques in controlling GPU components such as registers [3][40], warps [5][78], caches [76] and execution units [4] can be leveraged for fine-grained power optimization. Different power models [44][29] can be used for more accurate power and energy consumption estimation and therefore optimization.

6.3 Experimental Evaluation

This section evaluates the proposed optimization for multiple CUDA applications implemented with DTBL from both the power saving and the performance impact perspective. Detailed analysis of the experiments are presented with multiple sensitivity studies and insights discussion.

6.3.1 Methodology

The proposed SMX Occupancy Bubble-based power saving optimization is evaluated on GPGPU-Sim [9] with the integrated power simulator GPUWattch [41]. The configuration of GPGPU-Sim is set to model the microarchitecture of the Tesla K20c GPU with the DTBL execution model and LaPerm TB scheduler. This configuration is used as the baseline in the experiments. The power simulation parameters for GPUWattch are configured according to the default setting of GPUWattch, where the DVFS setting uses the P-states which has a peak of 700MHz and a minimum of 100MHz. The scaling voltage with the frequency is from 1V to 0.55V according to the 45nm predictive technology model [2]. The remaining section referred 100MHz/0.55V as the lowest P-state and 700MHz/1V as the highest P-state which is the default state.

The sampling rate for the SMX occupancy evaluation, TB diversion and power optimization operation is 500 cycles the same as the on-chip DVFS transition time used in GPUWattch [41]. The SMX Occupancy threshold T for TB Diversion is set to 0.0625 or 4 warps. This number is chosen in the way such that there are enough *SMX Occupancy Bubbles* in the execution for the power optimizations while there is minimized overhead introduced by the DVFS transition. The waiting period W for SMX draining is 10 time samples or 5000 cycles. In the experiments, the SMXs are able to be drained to complete idleness within this waiting period for up to 95.4% of the *SMX Occupancy Bubbles*. The experiments use two different DVFS transition times P: 1 time sample or 500 cycles for a fast on-chip DVFS regulator and 20 time samples or 10000 cycles for an off-chip DVFS regulator the same as GPUWattch [41]. While the major results in this section are reported with the fast on-chip DVFS regulator, the sensitivity study discusses the impact of an off-chip DVFS regulator with longer transition time.

This section uses the same set of benchmark applications as shown in Table 7. As described in Section 6.1.2, these 16 benchmarks are divided into 2 groups according to their potential power saving when ideal power gating technique is applied. As the proposed SMX Occupancy Bubble-based power saving optimization will affect the performance of the application which may change the total execution time, the subsequent section uses energy savings instead of power saving as the metric to evaluate the energy efficiency of

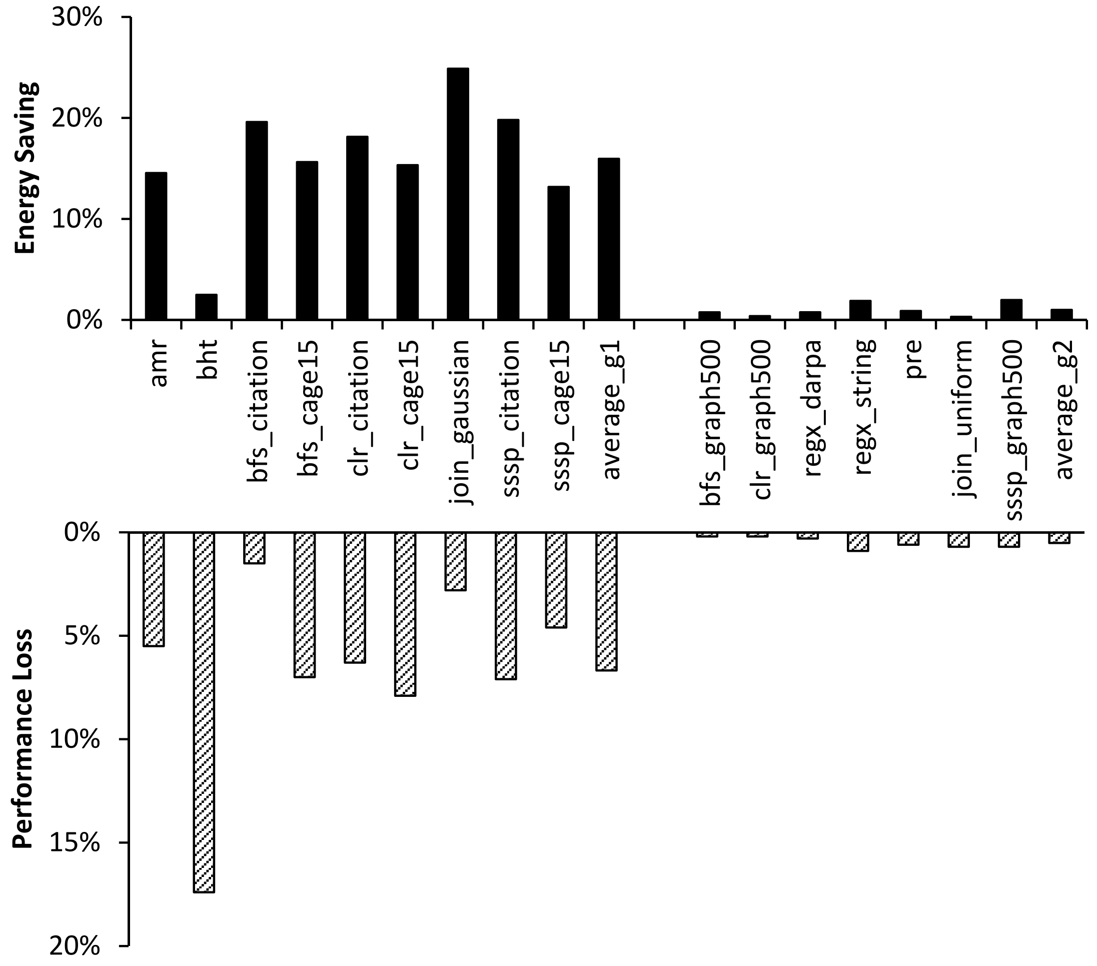


Figure 38: Energy Savings and Performance Loss for the SMX Occupancy Bubble-base Power Optmization (*P* = 500 cycles)

the applications.

6.3.2 Energy Savings and Performance Impact

Figure 38 shows the energy savings and performance loss of the proposed power saving optimization for DTBL. The average energy savings is 15.9% for the applications in group 1, which indicates the effectiveness of the optimization in using the SMX Occupancy Bubble with DVFS, and thereby improving the overall energy efficiency. Recall that the energy savings with an ideal power-gating technique is 24.5%. The application *join_gaussian* has the highest energy savings 24.9% which is only 4.6% lower than its ideal counterpart. This is due to the fact that 1) *join_gaussian* has high occurrence of SMX Occupancy Bubbles

(45.7%) which has been proven to lead to great energy savings in the ideal case and 2) the SMX Occupancy Bubbles are concentrated and therefore can last for a relatively long period which facilitates the application of DVFS without introducing too much transition time overhead. In contrast, although the application *bht* has relatively high SMX Occupancy Bubble occurrence (27.7%) and therefore 18.0% ideal energy saving, the actual simulated energy savings for the proposed optimization is only 2.5%. A close look at the *bht* reveals that it has fast-changing execution phase behavior, i.e. the SMX Occupancy Bubbles are short and distributed. It also requires frequent SMX Resuming shortly after TB Diversion as workload intensity recovers quickly after the SMX Occupancy Bubbles. Therefore, the effective time of the SMX under the lowest P-state is short, leading to limited energy saving.

The average performance loss for group 1 applications is 6.7%. As discussed in the earlier sections, the performance loss comes from the fact that relocating TBs may cause some of the SMXs to have longer execution tails than the others, leading to the execution time increase of the entire kernel. The DVFS transition time is also a major contributor to longer total execution time. The benchmark *bfs_citation* and *bht* are the benchmarks with the least and the most performance loss (1.5% and 17.4%), respectively. In fact, they demonstrate the impact of SMX Bubble Occupancy patterns on the performance. For benchmarks such as *join_gaussian* and *bfs_citation*, the concentrated SMX Occupancy Bubbles result in not only longer time an SMX could be set into the lowest P-state, but also fewer DVFS transitions with less transition overhead. On the other hand, the benchmark *bht* suffers from long overall DVFS transition time which could introduce substantial performance overhead. Note that the longer execution time also has a negative impact on the total energy efficiency, which is another cause that *bht* has the lowest energy savings among the all.

Recall that there are very few SMX occupancy bubbles in the second group of applications due to multiple reasons, therefore, the proposed optimization only has a limited impact on both the energy savings (average 1.0%) and performance loss (average 0.5%).

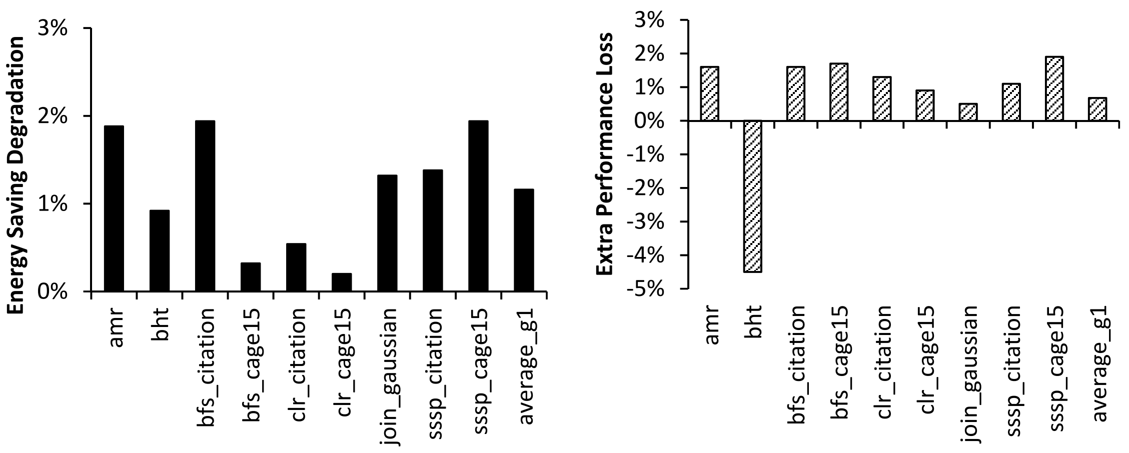
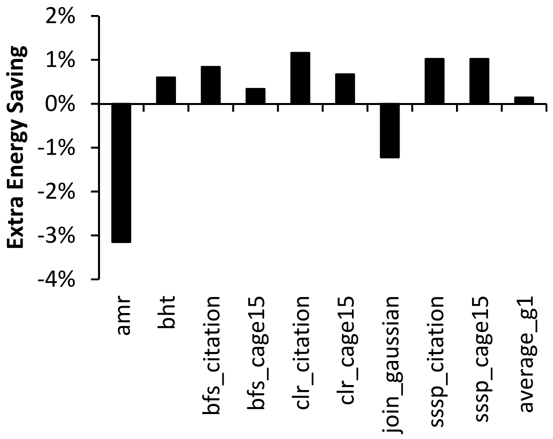


Figure 39: Energy savings degradation and extra performance loss for off-chip DVFS regulator with the 10000 cycle transition time compared with on-chip DVFS regulator with the fast 500 cycle transition time

6.3.3 Sensitivity of DVFS Transition Time

As the DVFS transition time could affect the energy efficiency, this section performs a sensitivity analysis of the transition time. Figure 39 illustrates the extra energy savings degradation and performance loss of group 1 applications when using a off-chip DVFS regulator with the 10000 cycle transition time.

The result shows that the average energy savings degradation is 1.2% with an average performance loss of 0.6%. The long DVFS transition time reduces the effective time that the idle SMX is under the lowest P-state while also prevents it from processing more dynamic work as the TB Diversion is turned on, which may cause oversubscribing of the remaining SMX and the decrease of overall energy efficiency. However, it is also worthwhile to notice the performance gain for *bht* (4.5%) compared with the case where a faster DVFS regulator is used. The long transition time actually makes it impossible to capture every SMX Occupancy Bubble during its execution, which effectively reduces the overhead of TB Diversion and Resuming and results in shorter execution time.



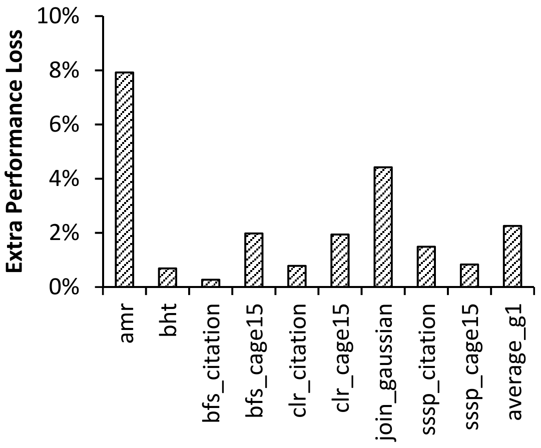


Figure 40: Extra Energy savings and extra performance loss when SMX Draining is turned off (P = 500 cycles)

6.3.4 Impact of SMX Draining

When SMX Draining is turned off during the proposed optimization, DVFS can be applied immediately after TB Diversion starts, which could lead to extra power saving. However, DVFS on non-idle SMX also leads to longer execution time for the active TBs and overall performance loss. Together they affect the energy efficiency as shown in Figure 40. All the group 1 applications benefit from extra energy savings except for *amr* and *join_gaussian* with 3.2% and 1.2% energy savings degradation, respectively. Both applications are compute-bound with TBs that can execute for a long time. Applying DVFS without SMX Draining results in substantial loss in performance (7.9% and 4.4%) and energy efficiency.

6.3.5 Insights

The above experiments results and analysis show that there are various aspects that could affect the effectiveness of the proposed power saving optimization. Some of the insights are as follows:

- Both the occurrence percentage and phase behavior of SMX Occupancy Bubbles have major impact on the energy savings and performance. Concentrated and long

SMX Occupancy Bubbles result in better energy efficiency while distributed and short SMX Occupancy Bubbles could introduce high overhead that negates any potential benefits of the proposed optimization.

- The DVFS Transition time may tradeoff between achieving better power saving and catching the fast-changing SMX Occupancy Bubbles phase behavior. Further optimizations can be proposed to accommodate different DVFS transition time, such as introducing the minimal time the DVFS should be active once it is enabled.

6.4 Summary

This chapter presents an optimization for the DTBL execution model from the power dissipation perspective. The concept of *SMX Occupancy Bubble* is introduced and demonstrated through the comparison between the execution phase behavior between the DTBL and the PT models to reveal the potential power saving opportunities for DTBL applications. The SMX Occupancy Bubble-base power saving optimization is then proposed to incorporate a dynamic TB diversion and scheduling strategy followed by a flexible DVFS application scheme to reduce both the dynamic and leakage power during the SMX Occupancy Bubble and achieve better energy efficiency. Experiments on a set of irregular applications with DTBL implementation show the proposed optimization is able to achieve an average of 15.9% energy savings with 6.7% performance loss.

CHAPTER VII

CONCLUSION

This thesis seeks to address an important question of how to efficiently map the emerging data-intensive applications with irregular data structures onto the GPUs that employ the regular BSP execution model. These applications, represented by graph processing, relational computing and machine learning, are characterized by their unstructured control and memory behavior which would lead to low compute utilization when implemented on the GPU. This thesis concludes that an extension to the current GPU execution model with the capability of dynamic thread block launching augmented by a set of optimizations in improving the scheduling and energy efficiency is an effective and efficient solution for the GPU implementations of the irregular applications with fine-grained dynamic parallelism. This conclusion is demonstrated by the presentation of the following research problems, their results and important insights.

First, a set of irregular CUDA applications are evaluated to present and characterize **D**ynamically **F**ormed Pockets of Structured **P**arallelism (DFP). This is performed through GPU implementations with the CUDA Dynamic Parallelism (CDP) features introduced by recent generations of NVIDIA GPUs. DFP has three important features of dynamic parallelism in irregular applications: high dynamic workload density, low compute intensity and workload similarity. The characterization study shows that a GPU execution model that supports dynamic workload generation would potentially benefit the irregular applications with DFP in terms of better control flow and memory behavior as well as higher productivity. It also shows that the implementations with dynamic kernel launches through CDP suffer from low GPU utilization and high kernel launching overhead which would negate any performance benefit and impose a challenge in utilizing the dynamic parallelism execution model.

Second, the Dynamic Thread Block Launch (DTBL) extension to the current GPU execution model is proposed and evaluated. The major achievement of DTBL is a lightweight, efficient programming model and corresponding microarchitecture/runtime support for DFP. The ability of launching fine-graind TBs from a GPU thread on demand instead of heavy-weight GPU kernels helps address the most two important performance issues in support of dynamic parallelism on the GPU: increasing SMX execution efficiency and minimizing launching overhead. Through careful design of the DTBL semantics and the innovative TB coalescing strategy, the proposed new execution model extension demonstrates its capability of supporting fine-graind dynamic parallelism in DFP with significantly reduced overhead compared with CDP.

Third, the LaPerm memory locality-aware TB scheduler is proposed as an optimization to the DTBL model. The motivation of such a scheduler is that there is a new type of memory reference locality relationship in dynamic parallelism between the parent TBs and the child TBs as well as the sibling child TBs in terms of substantial data reuse. The current round-robin TB scheduler fails to catch such locality as it is designed for applications implemented with non-dynamic parallelism execution models. The LaPerm TB scheduler introduces three-level decisions to utilize the memory hierarchy such as L1/L2 caches for the parent-child and child-child locality to improve the overall memory system performance. As an optimization to the original DTBL model, LaPerm demonstrates it effectiveness in further increasing the performance of irregular applications when implemented with dynamic parallelism.

Fourth, the DTBL model is evaluated from a power dissipation perspective and a new energy saving optimization is proposed. The evaluation compares the DTBL implementation with the PT implementations that employ the regular GPU BSP models and concludes that there exists larger SMX Occupancy variance in the execution of DTBL applications due to the generation of dynamic workloads. The existence of SMX Occupancy Bubble where the SMX occupancy falls below a threshold can be potentially utilized to reduce energy consumption for the DTBL applications. The proposed optimization employs a new dynamic TB diversion and scheduling strategy to turn the SMX Occupancy Bubbles into SMX idle periods, followed by an opportunistic DVFS scheme to further reduce the dynamic power and leakage power, which eventually leads to the increase of overall energy efficiency for the DTBL execution model.

In summary, the rapid-growing irregular application domain results in an urgent demand for new GPU execution models to address the newly raised challenges and issues in performance, energy, reliability, etc. This thesis takes a step towards this end by proposing the DTBL execution model along with the associated optimization in memory scheduling and energy efficiency to accommodate the dynamic parallelism that is observed in irregular applications. While DTBL demonstrates its efficiency in various aspects, there are still many questions to be answered. Exploring further opportunities in supporting dynamic parallelism on the GPU is both promising and imperative.

REFERENCES

[1] “Movielens.” http://movielens.umn.edu.

[2] “Predictive technology model.”

[3] Abdel-Majeed, M. and Annavaram, M., “Warped register file: A power efficient register file for gpgpus,” in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 412–423, IEEE, 2013.

[4] Abdel-Majeed, M., Wong, D., and Annavaram, M., “Warped gates: gating aware scheduling and power gating for gpgpus,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 111–122, ACM, 2013.

[5] Abdel-Majeed, M., Wong, D., Kuang, J., and Annavaram, M., “Origami: Folding warps for energy efficient gpus,” in *Proceedings of the 2016 International Conference on Supercomputing*, p. 41, ACM, 2016.

[6] Adriaens, J., Compton, K., Kim, N. S., and Schulte, M., “The case for gpgpu spatial multitasking,” in *Proceedings of 18th International Symposium on High Performance Computer Architecture (HPCA-18)*, 2012.

[7] Anderson, J. A., Lorenz, C. D., and Travesset, A., “General purpose molecular dynamics simulations fully implemented on graphics processing units,” *Journal of Computational Physics*, vol. 227, no. 10, 2008.

[8] Bader, D. A., Meyerhenke, H., Sanders, P., and Wagner, D., “10th dimacs implementation challenge: Graph partitioning and graph clustering,” 2011.

[9] Bakhoda, A., Yuan, G., Fung, W., Wong, H., and Aamodt, T., “Analyzing cuda workloads using a detailed gpu simulator,” in *Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’09)*, 2009.

[10] Beamer, S., Asanovic, K., and Patterson, D., “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC’15)*, 2015.

[11] Bergstrom, L. and Reppy, J., “Nested data-parallelism on the gpu,” *ACM SIGPLAN Notices*, vol. 47, no. 9, pp. 247–258, 2012.

[12] Blelloch, G. E., Gibbons, P. B., and Matias, Y., “Provably efficient scheduling for languages with fine-grained parallelism,” *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 281–321, 1999.

[13] Blelloch, G. E., Gibbons, P. B., and Simhadri, H. V., "Low depth cache-oblivious algorithms," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pp. 189–199, ACM, 2010.

[14] Burtscher, M., Nasre, R., and Pingali, K., "A quantitative study of irregular programs on gpus," in *Proceedings of 2012 IEEE International Symposium on Workload Characterization (IISWC'12)*, 2012.

[15] Burtscher, M. and Pingali, K., "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," *GPU computing Gems Emerald edition*, p. 75, 2011.

[16] Che, S., Beckmann, B. M., Reinhardt, S. K., and Skadron, K., "Pannotia: Understanding irregular gpgpu graph applications," in *Proceedings of 2013 IEEE International Symposium on Workload Characterization (IISWC'13)*, 2013.

[17] Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., and Skadron, K., "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of 2010 IEEE International Symposium o nWorkload Characterization (IISWC'10)*, 2010.

[18] Chen, G. and Shen, X., "Free launch: Optimizing gpu dynamic kernel launches through thread reuse," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[19] Cohen, J. and Castonguay, P., "Efficient graph matching and coloring on the gpu," in *GPU Technology Conference*, 2012.

[20] Coon, B. W., Nickolls, J. R., Lindholm, J. E., Stoll, R. J., Wang, N., and Choquette, J. H., "Thread group scheduler for computing on a parallel thread processor." US Patent 8,732,713, 2014.

[21] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S., "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU3)*, 2010.

[22] Diamos, G., Wu, H., Wang, J., Lele, A., and Yalamanchili, S., "Relational algorithms for multi-bulk-synchronous processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles andPractice of Parallel Programming (PPoPP'13)*, 2013.

[23] DiMarco, J. and Taufer, M., "Performance impact of dynamic parallelism on different clustering algorithms," in *SPIE Defense, Security, and Sensing*, 2013.

[24] Fung, W. W., Sham, I., Yuan, G., and Aamodt, T. M., "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, 2007.

[25] Gaster, B. R. and Howes, L., "Can gpgpu programming be liberated from the data-parallel bottleneck?," *Computer*, vol. 45, no. 8, pp. 42–52, 2012.

[26] Ge, R., Vogt, R., Majumder, J., Alam, A., Burtscher, M., and Zong, Z., "Effects of dynamic voltage and frequency scaling on a k20 gpu," in *2013 42nd International Conference on Parallel Processing*, pp. 826–833, IEEE, 2013.

[27] Gupta, K., Stuart, J. A., and Owens, J. D., "A study of persistent threads style gpu programming for gpgpu workloads," in *Proceedings of Innovative Parallel Computing (InPar'12)*, 2012.

[28] Herlocker, J. L., Konstan, J. A., Borchers, A., and Riedl, J., "An algorithmic framework for performing collaborative filtering," in *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1999.

[29] Hong, S. and Kim, H., "An integrated gpu power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 280–289, ACM, 2010.

[30] Jiao, Q., Lu, M., Huynh, H. P., and Mitra, T., "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 1–11, IEEE Computer Society, 2015.

[31] Jog, A., Kayiran, O., Chidambaram Nachiappan, N., Mishra, A. K., Kandemir, M. T., Mutlu, O., Iyer, R., and Das, C. R., "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 2013.

[32] Kayiran, O., Jog, A., Kandemir, M. T., and Das, C. R., "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*, 2013.

[33] Kerr, A., Diamos, G., and Yalamanchili, S., "A characterization and analysis of ptx kernels," in *Proceedings of 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, 2009.

[34] Khronos, "The opencl specification version 2.0," 2014.

[35] Kim, J. and Batten, C., "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, 2014.

[36] Kuhl, A., "Thermodynamic states in explosion fields," in *14th International Symposium on Detonation, Coeur d'Alene Resort, ID, USA*, 2010.

[37] Kulkarni, M., Burtscher, M., Casçaval, C., and Pingali, K., "Lonestar: A suite of parallel irregular programs," in *Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, 2009.

[38] Lee, H., Brown, K., Sujeeth, A., Rompf, T., and Olukotun, K., "Locality-aware mapping of nested parallel patterns on gpus," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, 2014.

[39] Lee, M., Song, S., Moon, J., Kim, J., Seo, W., Cho, Y., and Ryu, S., "Improving gpgpu resource utilization through alternative thread block scheduling," in *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[40] Lee, S., Kim, K., Koo, G., Jeon, H., Ro, W. W., and Annavaram, M., "Warped-compression: enabling power efficient gpus through register compression," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 502–514, ACM, 2015.

[41] Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N. S., Aamodt, T. M., and Reddi, V. J., "Gpuwattch: enabling energy optimizations in gpgpus," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[42] Li, D., Wu, H., and Becchi, M., "Nested parallelism on gpu: Exploring parallelization templates for irregular loops and recursive computations," in *Proceedings of 2015 44th International Conference on Parallel Processing (ICPP)*, 2015.

[43] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P., "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, ACM, 2009.

[44] Ma, X., Dong, M., Zhong, L., and Deng, Z., "Statistical power consumption analysis and modeling for gpu-based computing," in *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[45] McHugh, J., "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory," *ACM Transactions on Information and System Security*, vol. 3, no. 4, pp. 262–294, 2000.

[46] Meng, J., Tarjan, D., and Skadron, K., "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 235–246, 2010.

[47] Merrill, D., Garland, M., and Grimshaw, A., "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, 2012.

[48] Mosegaard, J. and Sørensen, T. S., "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of the 11th Eurographics Conference on Virtual Environments*, pp. 105–111, Eurographics Association, 2005.

[49] Nadungodage, C. H., Xia, Y., Lee, J. J., Lee, M., and Park, C. S., "Gpu accelerated item-based collaborative filtering for big-data applications," in *Proceedings of 2013 IEEE International Conference on Big Data*, 2013.

[50] Narasiman, V., Shebanow, M., Lee, C. J., Miftakhutdinov, R., Mutlu, O., and Patt, Y. N., "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO-44)*, 2011.

[51] Nasre, R., Burtscher, M., and Pingali, K., "Data-driven versus topology-driven irregular computations on gpus," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 463–474, IEEE, 2013.

[52] NVIDIA, "Gpu performance state interface."

[53] NVIDIA, "Hyperq sample," 2012.

[54] NVIDIA, "Nvidia's next generation cuda compute architecture: Kepler gk110," 2012.

[55] NVIDIA, "Cuda profiler user's guide version 5.5," 2013.

[56] NVIDIA, "Nvidia geforce gtx 980 whitepaper," 2014.

[57] NVIDIA, "Cuda c programming guide," 2015.

[58] NVIDIA, "Cuda dynamic parallelism programming guide," 2015.

[59] Orr, M. S., Beckmann, B. M., Reinhardt, S. K., and Wood, D. A., "Fine-grain task aggregation and coordination on gpus," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA-41)*, 2014.

[60] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and others, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 66, 2010.

[61] Podlozhnyuk, V., "Black-scholes option pricing," 2007.

[62] Rhu, M., Sullivan, M., Leng, J., and Erez, M., "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013.

[63] Rogers, T. G., O'Connor, M., and Aamodt, T. M., "Cache-conscious wavefront scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, 2012.

[64] Rogers, T. G., O'Connor, M., and Aamodt, T. M., "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, pp. 99–110, 2013.

[65] Saad, Y., *SPARSKIT: A basic toolkit for sparse matrix computations*. Research Institute for Advanced Computer Science, NASA Ames Research Center Moffet Field, California, 1990.

[66] Solomon, S. and Thulasiraman, P., "Performance study of mapping irregular computations on gpus," in *Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[67] Steffen, M. and Zambreno, J., "Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, 2010.

[68] Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and Hwu, W.-m., "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[69] Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M., "Enabling preemptive multiprogramming on gpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA-41)*, 2014.

[70] Visualizing.org, "Global flight network."

[71] Wang, F., Dong, J., and Yuan, B., "Graph-based substructure pattern mining using cuda dynamic parallelism," in *Intelligent Data Engineering and Automated Learning–IDEAL 2013*, pp. 342–349, 2013.

[72] Wang, J., Rubin, N., Sidelnik, A., and Yalamanchili, S., "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," in *Proceedings of the 42nd Annual International Symposium on Computer Architecuture (ISCA-42)*, 2015.

[73] Wang, J., Rubin, N., Sidelnik, A., and Yalamanchili, S., "Laperm: Locality aware scheduler for dynamic parallelism on gpus," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA-43)*, 2016.

[74] Wang, J. and Yalamanchili, S., "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *Proceedings of 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*, 2014.

[75] Wang, L., Chen, S., Tang, Y., and Su, J., "Gregex: Gpu based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pp. 366–370, IEEE, 2011.

[76] Wang, Y., Roy, S., and Ranganathan, N., "Run-time power-gating in caches of gpus for leakage energy savings," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 300–303, EDA Consortium, 2012.

[77] Wu, H., Diamos, G., Cadambi, S., and Yalamanchili, S., "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, 2012.

[78] Xu, Q. and Annavaram, M., "Pats: Pattern aware scheduling and power gating for gpgpus," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 225–236, ACM, 2014.

[79] Yang, Y. and Zhou, H., "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*, 2014.

[80] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X., "On-the-fly elimination of dynamic irregularities for gpu computing," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 369–380, 2011.

[81] Zhang, P., Holk, E., Matty, J., Misurda, S., Zalewski, M., Chu, J., McMillan, S., and Lumsdaine, A., "Dynamic parallelism for simple and efficient gpu graph algorithms," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, 2015.