UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving Performance and Energy Efficiency of GPUs through Locality Analysis

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

Doctor of Philosophy

in

Computer Science

by

Devashree Tripathy

September 2021

Dissertation Committee:

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

The Dissertation of Devashree Tripathy is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Finishing this dissertation marks the end of a major chapter of my life and the beginning of a new innings. I thank you all from the bottom of my heart, for your unquestionable support, continuous encouragement and endless patience, that helped me navigate my PhD Journey.

First of all, I would like to thank my advisor, Dr. Laxmi Bhuyan for believing in me and granting me the freedom to pursue the research projects. His technical, editorial and general advice, and mentoring was pivotal in the completion of the dissertation. I would also like to thank my dissertation committee members, Dr. Daniel Wong for his support and endless help in my research, Dr. Zizhong Chen and Dr. Nael Abu-Ghazaleh for their support and guidance as this dissertation shaped from a proposal to a complete study. Their constructive feedback has helped me immensely to improve the quality of this dissertation.

Next, I would like to thank my amazing collaborators, Dr. Manoranjan Satpathy, Amirali Abdolrashidi, Hadi Zamani Sabzi and Debiprasanna Sahoo. Especially, Amirali for listening to my crazy ideas and improvising those, brainstorming with me, Bashar for standing by me through thick and thin, and Debiprasanna for collaborating and mentoring me on various occasions. I would like to thank all the professors, mentors, co-workers, teachers, students, friends in my life for being there for me and believing in me. I thank my labmates and fellow graduate researchers : Hadi Zamani Sabzi, Liang Zhou, Mehmet Esat Belviranli, Chih-hsun Chou, Keval Vora, Amlan Kusum, Quan Fan, Rameswar Panda, Anguluri Rajasekhar, Kiran Ranganathan, Marcus Chow, Ali Jahanshahi, Sourav Panda, Madhurima Chakraborty and Chengshuo Xu (Bruce).

Finally, most importantly, I would like to express my gratitude towards my family for their unconditional support during my pursuit of this arduous journey. Especially, I thank my parents, Dr.Chitaranjan Tripathy and Minakshi Tripathy for providing me a strong education foundation to begin with and rowing the seeds of research aptitude since childhood, my brother, Nachiket Tripathy for always motivating me, my in-laws, for their love, support and encouragement, last but not the least, thanks to my best friend and my dear husband Bibek Mishra without whose unconditional love and support this dissertation would not have been possible. He actually underwent this journey along with me by my side, and sometimes was a part of it more than me. He not only encouraged me to think out of the box and aim for the best in life, but also stayed committed to the PhD process with me till the end, by putting up with my countless work-hours and keeping me focused on the priorities when I had too many things on my plate.

This dissertation includes content published in the following proceedings:

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

2. **Devashree Tripathy**, Amirali Abdolrashidi, Quan Fan, Daniel Wong and Manoranjan Satpathy. "LocalityGuru: A PTX Analyzer for Extracting Thread Block-level Locality in GPGPUs." 15th International Conference on Networking, Architecture, and Storage (NAS), October 2021.

3. **Devashree Tripathy**, Hadi Zamani Sabzi, Debiprasanna Sahoo, Laxmi Narayan Bhuyan and Manoranjan Satpathy. "Slumber: Static-Power Management for GPGPU Register Files." ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), August 2020.

To all taxpayers who subsidized my education.

ABSTRACT OF THE DISSERTATION

Improving Performance and Energy Efficiency of GPUs through Locality Analysis

by

Devashree Tripathy

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2021
Prof. Laxmi Bhuyan, Chairperson

The massive parallelism provided by general-purpose GPUs (GPGPUs) possessing numerous compute threads in their streaming multiprocessors (SMs) and enormous memory bandwidths have made them the de-facto accelerator of choice in many scientific domains. To support the complex memory access patterns of applications, GPGPUs have a multi-level memory hierarchy consisting of a huge register file and an L1 data cache private to each SM, a banked shared L2 cache connected through an interconnection network across all SMs and high-bandwidth banked DRAM. With the amount of parallelism GPUs can provide, memory traffic becomes a major bottleneck, mostly due to the small amount of private cache that can be allocated for each thread, and the constant demand of data from the GPU's many computation cores. This results in under-utilization of many SM components like register file, thereby incurring sizable overhead in the GPU power consumption due to wasted static energy of the registers. The aim of this dissertation is to develop techniques that can boost the performance in spite of small caches and improve power management techniques to boost energy saving.

In our first technique, we present *PAVER*, a priority-aware vertex scheduler, which takes a graph-theoretic approach towards thread-block (TB) scheduling. We analyze the cache locality behavior among TBs and represent the problem using a graph representing the TBs and the locality among them. This graph will then be partitioned to TB groups that display maximum data sharing and assigned to the same SM by the locality-aware TB scheduler. This novel technique also reduces the leakage and dynamic access power of the L2 caches, while improving the overall performance of the GPU.

In our second study, *Locality Guru*, we seek to employ the JIT analysis to find the data-locality between structures at various granularity such as threads, warps and TBs in a GPU Kernel using the load register's address tracing through a syntax tree. This information can help make smarter decisions for a locality aware data-partition and scheduling in single and multi-GPUs.

In the previous techniques, we gained performance benefit by exploiting the data-locality in the GPUs, which eventually translates to static energy saving in the whole GPU. Next, we analyze the static energy saving of the storage structures like L1 and L2 caches by directly applying power management techniques to save power during the time they are idle.

Finally, we develop, *Slumber*, a realistic model for determining the wake-up time of registers from various under-volting and power gating modes. We propose a hybrid energy saving technique where a combination of power-gating and under-volting can be used to save optimum energy in the register file depending on the idle period of the registers with a negligible performance penalty.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The massive parallelism provided by general-purpose GPUs (GPGPUs) is in demand from many areas of industry and research. Possessing numerous compute threads in their streaming multiprocessors (SMs) and enormous memory bandwidths as high as 1555 GB/s [105], GPGPUs have become the de-facto accelerator of choice in many scientific domains. To support the complex memory access patterns of applications, GPGPUs have a multi-level memory hierarchy consisting of an L1 data cache private to each SM, a banked shared L2 cache connected through an interconnection network across all SMs and high-bandwidth banked DRAM.

With the amount of parallelism GPUs can provide, memory traffic becomes a major bottleneck for present-day GPUs, mostly due to the small amount of private cache that can be allocated for each thread, and the constant demand of data from the GPU's many computation cores. With the ever-increasing data size of GPU applications, and each thread having to process more data, simply increasing the cache sizes is not a viable option, since

the additional area will incur extra cost and overhead. This means that smaller L1 and L2 caches are much more likely to suffer from cache thrashing, i.e. eviction of cache lines which could have been used by other execution units. Cache thrashing can lead to more cache operations, which means more energy consumption, and tremendous under-utilization of the other GPU resources, resulting in under-performance [33, 67, 141, 160, 162]. To minimize this, all threads need to utilize the shared cache memory spaces with each other as efficiently as possible. Efficient use of the memory system as well as co-locating the compute and data together is important for exploiting the massive computational capability offered by GPGPUs to their full potential. A key approach to efficiently improve the memory bandwidth is *data locality* – (i) increasing the data reuse within the SM at the thread, warps and thread block (TB) level, thereby reusing the L1 cache lines before it is evicted; and (ii) placing the data close to the computation so as to reduce the communication across multiple SMs within a GPU or even multiple GPUs. There is no generalized technique which exploits the data locality present in various applications to improve the efficiency of cache and memory system usage. There have been prior research on addressing the memory bottleneck issues, including the prefetching [75, 128], cache management [147], locality-aware schedulers [87, 148], memory-level parallelism [40, 165, 168] etc. All these techniques need the data locality information which is either known a priori using profiling techniques, or learnt and predicted during the kernel execution. For extracting the locality information, we use profiling in section 1.1 and a PTX analyzer at the JIT time in section 1.2.

Along with the challenge to maintain high GPU throughput and performance by mitigating the memory bottleneck issue, the leakage power dissipation has become one of

the major concerns with technology scaling. The GPGPU storage structures like caches and register file have grown in size over last decade in order to support the parallel execution of thousands of threads. Given that each thread has its own dedicated set of physical registers, these registers remain idle when corresponding threads go for long latency operation. Existing research [2, 39] shows that the leakage energy consumption of the storage structure can be reduced by either power gating or reducing to a data-retentive low-leakage voltage (Drowsy Voltage) to ensure that the data is not lost while not in use. Sections 1.3 and 1.4 in this thesis introduce to evaluate these techniques and explore new architecture to further increase energy saving.

1.1 Data-Locality Aware Thread Block Scheduler

The most common relationships between the threads sharing the cache are read-after-write (RAW) and read-after-read (RAR) cases. RAW constitutes data dependency among tasks, e.g. thread blocks (TBs), also known as *structured parallelism*. In this case, a certain execution order among the tasks will be formed and the execution time will be bound to a *critical path*. Exploiting this order can improve the performance substantially. There have been numerous works on structured parallelism in CPUs [7, 37, 45, 127], and more recently in the realm of GPUs [5, 17, 148]. In particular, they try to exploit the data locality between parent and child TBs.

RAR, or local data sharing, on the other hand, happens in *unstructured parallelism*, in which the tasks are independent, and thus free to be executed in any order. Its impact is more prominent when there is no write-allocate policy in place. As a result, the program's

outcome would be correct regardless of task ordering and location of execution. However, processes can still subtly affect each other in terms of shared resources and, by extension, the performance, in contrast to the more explicit sharing in structured parallelism. To make better use of the cache data, the data locality of the parallel-run tasks must be observed and considered with respect to a given cache architecture. Research works have addressed this issue in the multi-core CPU area [62, 159], and Wang et al [148] improved cache performance with respect to data-reuse involving parent-child thread blocks in GPUs. To the best of our knowledge, there has been no research on exploiting data locality in unstructured parallelism in GPUs.

At the TB level, there have been attempts to take advantage of locality based on a specific data access pattern. In [81], Lee et al. propose Block CTA Scheduling (BCS) which naively assigns two consecutive TBs to the same SM. However, their approach works only for row-major applications, i.e. applications optimized to run with row-major data structures, such as matrix multiplication, n-body, and hotspot. Since the grid structure of the tasks is application-specific, PAVER addresses this problem with a more generic graph-based approach to improve performance and memory efficiency. We do so by creating a graph of TBs using their data sharing statistics, where the vertices represent the TBs and a weighted edge between two TB denotes the number of shared data locations between them.

Designing a graph structure requires us to know the access footprint per TB in the application a priori and then deciding which TBs to group in the same SM to maximize the cache utilization. Hence, it is necessary to analyze the cache access characteristics before the execution. Compiler-assisted methods can extract locality information directly from

the code itself e.g. static compiler analysis, which can be used to optimize cache bypassing, warp scheduling, thread throttling, etc. [67, 69, 91]. We propose PAVER (**P**riority **A**ware **V**ertex Schedul**ER**), a TB scheduler that can utilize the RAR information between the TBs while allocating them to the SMs. In PAVER, we propose a just-in-time compilation approach to gather the data sharing statistics among the TBs which run the same kernel and are able to use the same allocated memory. The JIT analysis is accomplished in a GPU after compilation and before the kernel launch [104]. We partition the graph in order to assign TB groups with the most locality to SMs. We explore various graph partitioning policies, such as k-way and recursive bi-partitioning algorithms based on METIS graph partitioning software [63], and Prim's maximum spanning tree (MST)-based algorithm [117]. The partitioning is stopped when the maximum number of TBs is allocated to an SM, as determined by the application and resources. To improve load balancing between SMs, we also incorporate a task-stealing process to move a TB from one SM queue to another when all the TBs in the latter finish early. This ensures a decent load balance in the final phase of the execution.

Our partitioning algorithm ensures maximum cache sharing within an SM for L1 locality, high L2 locality among the SMs, and also ensures load balancing between the SMs. In a SM, maximum allowed concurrent TBs are sent for execution in the form of bunches of 32 threads each, called a warp, to a warp scheduler. This scheduler checks for ready warps within the pool of available warps and once a TB has finished execution, another TB assigned to that SM starts executing. There are several warp scheduling policies with the baseline policy as Greedy Then Oldest (GTO) in which a single warp is prioritized for

execution until it hits a long latency operation after which it is replaced by the warp which was assigned to the core for the longest period of time. PAVER is able to gain significant speedup by reusing the L1 and L2 cache data. However, extracting the locality information and then deriving insights to design efficient TB scheduler can be infeasible and tedious for larger workloads and unknown applications.

1.2 A PTX Analyzer to Extract TB Level Locality

Paver obtains the data locality information through profiling. Such profiling work has also been done for both CPUs [8, 62, 144, 164, 167] and GPUs [38, 130] for extracting the memory and execution time traces. However, as the profiling step has tremendous overhead, research has been started recently to speed up the GPU profiling through sampling and other techniques [62]. Profiling requires the user to run the code in order to detect opportunities to improve the performance. For input dependent applications, the profiled data extracted might not be valid for a different input data or other executions. To minimize the user burden of extracting the profiled data apriori, we propose to automate the locality extraction process by analyzing the intermediate IR (PTX), as discussed in the following sub-section.

Some of the recent works use programming language constructs to associate the thread to a mapped data in order to extract the data address range accessed by a thread [22, 23, 37, 134]. However, these approaches add to the programmer's burden of learning a new language and using it to rewrite the program. Locality Descriptor [146] expresses the data locality using program semantics, which needs the programmer to explicitly specify the static tile dimensions, compute and data mapping and data sharing pattern. TAFE [118]

estimates the thread address footprints of the static as well as dynamic data-dependent applications before kernel launch. The thread to data address index range relation coefficients are extracted by manual inspection of the application source code.

A benefit of analyzing a flat IR such as PTX code over the CUDA source code is that it has fewer program constructs and simpler program semantics. For example, CUDA constructs such as "for loop" and "while loop" that are syntactically different, but semantically equivalent, tend to correspond to similar PTX. Similarly, the conditional "if-else" and "ternary" statements that have *syntactically different* but *semantically equivalent* CUDA source code are compiled to similar PTX code. PTX representation uses an assembly-like structure for a virtual GPU architecture, and is only higher-level than the SASS format which executes on a specific GPU architecture.

In LocalityGuru, we aim to derive the relationship between the thread blocks and the memory addresses accessed by them. A detailed compiler analysis is performed on the PTX intermediate representation to extract the data locality in terms of number of common data elements shared between all thread block pairs in a kernel. The static analysis helps us determine the stride distance between two consecutive thread blocks in the grid as well as the data-dependency between the TBs due to the loop iterations. In LocalityGuru, we perform a detailed static compiler analysis to automatically extract the thread to index range relationship from the intermediate representation (IR) of the source code (in PTX format). We analyze the PTX code at JIT compilation time before the kernel launch and perform the detailed static index analysis to derive the equation for the thread/TB mapping to data element indices accessed. We validate the results of the TB locality graph obtained through

automated LocalityGuru PTX analyzer by comparing with the profiling data-locality results. Our approach imposes zero timing overhead on the kernel execution time.

1.3 Static-Power Management for the Caches

In the previous sections, the increased speedup saves the static energy of the entire GPU as the application aided by data locality analysis finishes sooner compared to the default case. However, due to long memory latency, and lack of parallelism, the GPU storage structures are under-utilized. Next, we explore the static energy saving of the storage structures like Caches and Register Files by leveraging the reuse distance or temporal locality distance between the subsequent accesses to the Caches.

In the past, the CMOS transistors used to have negligible leakage power dissipation. However, as the feature size decreases, the subthreshold leakage power dissipation has increased. As the processor technology shrinks below 0.1μm, the leakage power dominates the total power consumed by the circuit [71]. In recent years, the primary objective of the chip design is shifting from achieving highest performance to achieving high performance as well as energy efficiency (Performance per Watt). This metric is very important in the case of the de-facto accelerators like GPUs which are power hungry while delivering high throughput. There is a lot of static energy saving opportunity in the caches as they are idle in most of the gpu execution cycles.

There has been a lot of work on saving the CPU cache power using undervolting and power gating [6, 39, 70, 95]. The caches can be switched to either state-rentive "drowsy" low leakage mode or state-destroying power gated mode. In the drowsy mode ($0 < V_{drowsy} <$

V_{DD}), the contents of the caches are not lost, however, in the power gated mode (V = 0 Volts), the contents of the caches are wiped out as the transistors are discharged below the minimum threshold voltage needed for data retention. Trimodal switch [111] is used to switch the target circuit (cache) into one of the three states: ON (V_{DD}) , OFF (0 Volts) or drowsy (V_{drowsy}). In [39], the drowsy control signal is used to switch a idle cache line to the drowsy mode regardless of its access history. Another paper [70] extends this technique for instruction caches along with the next target sub-bank predictor, where only one sub-bank is active and the rest are in drowsy mode. Some other papers [6, 95] use the cache access pattern and cache line re-use distance information to switch the cache line to drowsy state or OFF state. Once the cache line is turned off, they fetch the lost cache data from the lower level memory hierarchy. In the GPU cache leakage energy savings context, one of the recent works [151] proposes a cache-level power management technique to switch the cache array to drowsy or OFF mode when there are no pending access request to the cache.

Our proposed cache power management design "Snooze" under-volts the L1 and L2 caches when there are no pending accesses in the queues (Address Generation Unit→L1, Interconnect→L1, Interconnect→L2, and DRAM→L2), which leads to significant leakage energy savings with negligible performance penalty. The caches are undervolted at various granularities like cache bank-level and cache line or block-level and it is observed that the line-level gating yields more leakage energy savings compared to the bank-level gating as a cache line has longer idle period and is less frequently accessed compared to a cache bank on a average.

1.4 A Static-Power Management Technique for the Register Files

Over the past decades, the GPUs have continued to scale up in terms of number of concurrent threads and cores. In order to support the faster context switching among the active threads, the register file size per core has also grown in size. The register file is currently the largest SRAM structure on the die and hence consumes the most leakage energy compared to L1 and L2 caches. One of the main limitations of the General Purpose Graphic Processor Units (GPGPUs) is the heterogeneity in workloads with different degrees of parallelism resulting in major resource under-utilization [5, 160]. This results in wasted energy especially in case of the register files, which constitute 18% of the total GPU chip power consumption and 32% of the streaming multi-processor's (SM) leakage power [54, 86].

One way to save energy is to power gate the unused components when they are not in use. However, the memory components like register files lose data when power-gated. Authors of [2, 39] propose a drowsy mode, where the state of the memory cells or passive units are retained by lowering the voltage to a minimum voltage, called drowsy voltage. The active units or the functional units are turned off, i.e. read and write accesses cannot be performed. However, switching the component to deep sleep (drowsy) mode can result in performance penalty due to high wake-up latency. Under-volting is a technique to set an optimum voltage level (less than V_{DD}) that has less wake-up latency as compared to the deep sleep state that mitigates the performance penalties. However, the undervolting level of the functional elements will be different. Our detailed modelling of under-volting the registers using the trimodal switch [111] with HSPICE [136] shows that the drowsy

voltage energy break-even time (EBT) can be as high as 13 clock cycles (cc), therefore, the performance penalty incurred can not be neglected. Hence, we argue that there is a need for an intermediate under-volting level with less transition time. In this thesis, we propose two undervolting levels of 0.3 Volt and 0.9 Volt referred to as deep sleep and shallow sleep state respectively. The shallow state can be used when the idle period falls much below the EBT for the drowsy state. The register is **under-volted** to a state retentive voltage if (a) next access to the register is a read access, or (b) idleperiod length is larger than the corresponding energy break-even time (EBT).

We use static compiler analysis to determine the type of register's next access and, thereby, predict the idle period length of the register. Then we determine the under-volting level based on the idle period length and EBT. Though all the prior approaches [2, 150] under-volt the registers associated with a warp, we show that the registers can instead be power-gated if the contents of the registers are not useful anymore. Since the register inter-access distance is about 789 clock cycles on an average [2], power-gating the idle registers which are waiting to be written leads to significant leakage energy benefits. It should be noted that power-gating is only possible if the idle-period length is more than the power-gating break-even time. The registers can be **power-gated** only under the following circumstances:

(a) register is not allocated to any thread block (TB),

(b) next pending access to the register is "Write", as write resets the register contents.

(c) The warp has finished executing the kernel. Since the register shall be re-allocated when next thread-block is allocated to the SM, the physical registers for the warp

can be power-gated till all the other warps for the TBs finish executing the kernel. Thereby saving maximum static energy while in the OFF state.

We propose Slumber, a hybrid energy saving technique where a combination of power-gating and under-volting can be used to save optimum energy depending on the idle period of the registers with a negligible performance penalty.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 summarizes the related works in different categories like locality based resource partitioning, locality aware scheduling techniques,and power management techniques in CPUs and GPUs. In chapter 3, we describe our graph theoritic approach to analyse the data locality among the thread blocks (TB) in a kernel and design various TB ordering and grouping strategies to gain speedup. Chapter 4 discuss automated compiler based approach to extract the locality information of any unknown application at the Just-In-Time compilation time. The JIT PTX analyzer overcomes the limitations of profiling approach incase of larger workloads and unknown applications. Chapter 5 and Chapter 6 discuss the proposed power-management technique for the GPU Caches and Register File respectively. Different undervolting techniques are employed depending on the idle period length and the storage structures are switched on just before their predicted accesses to mitigate any potential performance penalty. Finally, Chapter 7 concludes this dissertation by giving a summary of our work as well as directions for future work.

Chapter 2

Related Work

This chapter covers the literature related to the studies proposed in this thesis. First, we provide the related work for locality in GPUs, resource based partitioning and the locality aware warp and thread block scheduling techniques. Next, we discuss the compiler assisted locality information extraction works, and then, we summarize the general power management solutions for cache and register file energy efficiency.

2.1 CPU Cache Locality

The idea of using data locality to improve the performance and reduce the memory bottleneck started in the realm of CPUs and continued to multicores. As the cores have become more complex, so have the cache hierarchies, which means that any unnecessary cache action can now leave more of a negative impact on the performance and the energy efficiency [159]. According to [164], even though exploiting inter-core cache locality is in progress, it should not be without taking intra-core locality into account, for it could actually

perform worse than only exploiting intra-core locality. There have been analyses of the tradeoff between cache reuse and vectorization on CPUs, but in the end, they should be used in the right place and it mainly depends on the application type and the architecture [129]. Kandemir et al [62] argue that different cache hierarchies in different architectures makes it difficult for the programmer to optimize the application for all architectures, and presents a compiler-based method in which loop iterations are assigned to different cores and scheduled based on the cache topology and cache access pattern.

In [164], Zhang et al. use a compiler-based strategy to create a computation block dependency graph, targeting data reuse on multicore CPUs. Their observation on data reuse balance leads them to develop a task mapping and scheduling policy that balances inter-core and intra-core data reuse.

Task stealing has also been incorporated and explored in recent literature. Yoo et al [159] propose a locality-aware scheduler for unstructured parallel applications in a multi-core CPU which increases the speedup and reduces the energy consumption for a 32-core system by 2.05x and 47% respectively, and shows that the benefit will increase as the number of cores increase, an additional 1.83x for 1024 cores. They capture the data sharing of an application using special programming APIs and use this information to create a task sharing graph. Then, they generate task groups to be launched on each core keeping the cache topology in consideration. They also perform task reordering to capture temporal locality and task stealing for load balancing. The work proposed by Lifflander et al [92] increases cache locality for recursive programs by tracking data reuse opportunities and, using work stealing, interleaving the execution of the function that can use them.

2.2 GPU Cache Locality

Koo et al [74] categorize the load instructions into deterministic, where the address is calculated using thread ID, thread block ID, etc., and non-deterministic (from user input, etc.) Deterministic loads are observed to have a more coalesced access pattern in a stark contrast to the non-deterministic loads which can create far more reservation fails in the cache. They then suggest solutions to alleviate the issue with such loads, such as pre-fetching for indirect addresses [77], reworking the cache hierarchy, and assigning neighboring TBs with data locality to the same SM, the last one being the focus of our work in Chapter 3. Vijaykumar et al [146] has also shown the potential of exploiting locality by proposing the 'Locality Descriptor', which enables definition of locality abstractions on software by the programmer, and utilization thereof by the hardware, improving performance by an average of 26.6% when exploiting locality in the caches. Their work also enables using hardware techniques to improve the performance, such as TB scheduling and cache management. Our work lifts the requirement of software abstraction definition from the programmer, and instead seeks to use a generalized compiler-based approach to extract the block locality among the TBs and utilize them in the SM task assignment stage.

2.2.1 Locality aware TB Scheduling

Making use of the locality among thread blocks can prove challenging since very little is known about the exact underlying TB scheduling architecture. Several cache locality algorithms and structures have been proposed for GPGPUs in recent years which include some form of a thread block scheduler within. As mentioned earlier, most of them have

a rather naive approach as they develop heuristics based on workload behavior (e.g. data layout) to exploit data locality. Another drawback is that many of them target specific structures only, e.g. grid applications [142]. Our work (Chapter 3), however, focuses on a more generalized approach to exploit data locality where there is no need to know the application's access patterns, making it effective for applications with all types of structures.

In [87], Ang Li et al. develop a locality-aware TB clustering framework that can exploit inter-TB locality for GPU applications, obtaining an average speedup of 46% and 41% on GTX570 (Fermi) and GTX1080 (Pascal) respectively. This framework can be integrated into a GPU compiler and used on existing GPUs. In their work they argue that inter-TB locality can be exploited only in applications whose algorithmic behavior is known beforehand. They create a new kernel with TBs having inter-TB locality clustered together.

Chen et al [25] propose a hardware-software approach for applications with structural data access, both row- and column-major applications. It checks the address ranges of the ready TBs and issues the TB with the maximum overlapping address range with the TBs already executing on that SM, increasing data reuse and improving the performance by 9% over the BCS scheduling [81]. The maximum overlapping address is determined with the assumption that each TB accesses a continuous 2D space in the cache, whereas our work has a more generalized approach. Also, in [137], Abdulaziz et al. devise a sharing-aware TB scheduler Based on their observation that around 70% of data sharing takes place between consecutive thread block IDs. It assigns TB groups of consecutive TB IDs to the SMs while maintaining load balance among SMs. To aid the scheduler, they also devise a cache replacement policy in L1 and L2 levels such that the L1 cache tries to maximize the number

of cache blocks private to that SM, while L2 cache maximizes the number of cache blocks shared across SMs. This prevents cache block duplication in L1 and L2. Most of their performance benefits come from efficient cache replacement policies as compared to their TB scheduling policy.

The idea of temporal locality could also be extended to dynamic parallelism [59]. Wang et al [148] propose a locality-aware scheduler specifically for dynamic parallelism, in which the TBs belonging to child kernel will be scheduled on the same SM as those of the parent kernel are on, with 27% performance improvement. This paper shows that similar to parallel TBs, locality among parent and child TBs can also be high and therefore exploited.

2.2.2 Locality aware Warp Scheduling

Augmenting the warp schedulers exists in many works for different applications, including exploitation of data locality within the TB. In [121], the warp scheduler rearranges the access patterns of different warps as well as the threads in the warps to reduce the L1 cache misses and thrashing significantly, resulting in 24% performance improvement. The same authors proposed [122] which checks the control flow of the execution and divergences, and uses a predictive approach to schedule threads such that the L1 cache size usage and the likelihood of thrashing is minimized, resulting in a 26% improvement over [121].

Oh et al [107] propose a locality-aware warp scheduler coupled with a pre-fetching scheme, yielding 31.7% performance improvement over the baseline. In this work, cache access patterns are analyzed and warps accessing the same cache line in the same time frame are grouped. If the first warp of the group hits the cache, the rest of the group will also be prioritized under the assumption that their accesses would also be hits, increasing data

utilization before eviction. Should the first warp miss, the pre-fetcher would attempt to pre-load the data for the other warps in the group as well.

It must be pointed out that we consider only spatial locality among the TBs because the temporal localities depend on the warp scheduling policies inside a streaming multiprocessor (SM). These are best handled by warp scheduling policies, such as CCWS [121], which can work orthogonal to the TB scheduling policies. When applied with PAVER (Chapter 3), they will further improve the performance.

2.3 Compiler Assisted Locality Analysis

Prior works have proposed various methods to improve performance leveraging the data locality in different applications. In some works, the programmer provides hints in the code which would be used to optimize locality [18, 118, 124, 146]. Locality Descriptor [146] lets the user express and use the data locality information in GPUs through software to allow optimization for the programmer, combined with hardware to leverage the data locality in the application. Sometimes, new programming languages have been proposed to support the expression of data locality [16, 22, 23, 37, 134, 143, 157].

Compiler analysis has already been used by several works for data locality [65, 88, 94, 109]. Index analysis in compilers has been utilized to perform loop transformations in the source code targetted at improving data locality [14, 131, 158]. However, in Chapter 4, we choose to utilize the PTX intermediate code which is architecture-agnostic and holds the control and data flow information better than source code, which may not be always available to the user.

CODA [68] uses static analysis to compute the stride distance between two consecutive thread blocks (TBs) at runtime. This information is used to determine the size of the data accessed by a TB and ensure that the TBs and the data they access are co-located on the same GPU. Though this approach captures the TB index range in case of the 2D regular grid applications exhibiting strided accesses, a more detailed analysis of the code is needed to account for other access patterns in various applications. LADM [65] classifies the TB locality pattern in the application into one of seven patterns using static compiler analysis to check for loop variance of array indices. However, this process uses the *CUDA source code*, which may not be always available to the user, whereas LocalityGuru seeks to find the relationships between the GPU registers through PTX analysis. The static analysis helps us determine the stride distance between two consecutive thread blocks in the grid as well as the dependency of the data accessed by a TB on the loop iteration. TAFE [118] allows the programmer to send static kernel data and dynamic memory information to the device in order to extract their data locality information, and contains hardware to track data-dependent accesses, reducing the software overhead. Therefore, indirect memory accesses can be obtained, albeit through user APIs in the code. However, in Chapter 4, we choose to utilize the PTX intermediate code which is architecture-agnostic and holds the control and data ow information better than source code, which may not be always available to the user.

2.4 CPU Power Optimization Techniques

Power gating (PG) and Dynamic Voltage Scaling are two highly effective techniques proposed in literature to reduce the energy consumption in the processors. PG power management technique turns off the functional units by gating their supply voltage to reduce the power at circuit level when they are unused. A tri-mode switch [111] can be used to drive the target circuit (combinational or sequential logic) in one of three states ON, OFF and drowsy. Drowsy mode preserves the state of the circuit block in the low-leakage mode were as the power gating is a state-destroying mode where the data in the circuit is lost. The ability of data retention in drowsy mode has been the reason for wide use of the trimodal switched in implementing the data-retentive power gating designs.

Several techniques to save the leakage power in caches have been extensively studied [6, 39, 70, 95]. Drowsy cache approach to save energy in the data cache caches has received significant attention [39]. Each cache line has three states: ON,OFF and drowsy. A "drowsy" control signal is used to switch the idle cache line to drowsy state (state-retentive). Similarly, [70] employed similar drowsy leakage current reduction technique to save the leakage energy in the instruction caches. [6, 95] uses the prior knowledge of cache access pattern and cache line re-use distance to switch the cache line to drowsy state or OFF state. Once the cache line is turned off, they rely on the lower level memory hierarchy to get the required lost cache data.

Register file power management has received significant attention. Previous works have focused on reducing the register file power consumption from circuit level [49], micro-architecture level [28] till the software level [11, 115]. The number of active registers used

were reduced using compile time register file partitioning and recompilation in [44]. Then the register file was partitioned into two sections: active and inactive. The inactive registers were switched to drowsy mode and the register file power is saved as the applications use a reduced set of the registers (active).

2.5 GPGPU Power Optimization

Prior works have proposed architectural modifications to increase the idle-period length as well as save energy in specific GPU components such as caches [151], execution units [3, 125, 156, 163] and register file [2]. μ C -States [64] identifies the non-bottleneck components in the GPU data-path based on their utilization. The authors propose to clock-gate the data-retentive components and power-gate the non-data-retentive ones. They turn-off or tune-down half of the width of non-bottleneck resources leading to energy savings. Wang et. al. [149] propose architectural-level power-gating mechanisms for the graphics pipeline to save leakage power in shader clusters, fixed-function geometry units and execution units.

The access patterns of the GPU cache are different from CPU cache as it suffers from high miss rate and low locality. Wang et al. [151] propose a static energy saving technique for the L1 caches and L2 caches in GPUs. They switch the L1 cache (private) to a SM into the state preserving low leakage mode when there are no ready warps at the warp scheduler. Similarly, L2 is switched to low leakage mode when there are no pending requests. However, when the kernel exits, the caches are no more accessed and hence, are put to OFF state. [152] aggressively switches the cache lines to low power mode and also

bypasses the L1 data cache in some cases to save on the leakage power.

The register file (RF) resources are generally over provisioned to meet the high performance targets, however, their average utilization remains low [2, 141]. To address the under utilization issue and save the leakage power, several novel RF power management techniques like register sharing [53, 55], re-purposing the RF as cache or shared memory [35, 73, 77] and power gating the idle registers [2, 48, 82] have been proposed. The prior work "warped register file" [2] uses a similar concept as "Drowsiness" [39] to place the allocated registers in the drowsy state by default; the state is changed to ON when they are accessed. The un-allocated registers are in OFF state. They save the leakage energy of the RF by switching the registers to drowsy mode after each accesses. Due to branch divergence and limited parallelism, there can be several idle threads in warp.They identify the idle threads and save the dynamic energy by not charging the bit and work lines of the registers of idle threads. [82] use a register compression scheme to save register file power. The energy required to access a register file is reduced by reducing the number of register banks needed to store the warp registers. They also power gate a register bank when none of the bank entries are allocated for a register. [48] proposes a hybrid power gating control for mobile GPUs which captures both long and short-term unused cycles of the register file and uses the appropriate power gating mode to reduce the leakage energy consumption.

Chapter 3

PAVER: Locality Graph-based Thread Block Scheduling for GPUs

This chapter presents a locality graph based thread block scheduler, PAVER, to elegantly solve the complex problem of exploiting the data locality in applications exhibiting unstructured parallelism. PAVER proposes a novel graph-theoretic approach for TB scheduling on GPUs based on the cache sharing behavior between the TBs. The following outlines the organization of this chapter:

- In Section 3.1, we explore the baseline architecture and show how locality awareness can benefit the application performance.
- In Section 3.2, we generate the *TB locality graphs* of various benchmarks through analyzing memory access behaviour of individual thread blocks. We also present details of the JIT compiler analysis, adopted in this chapter.

- In Section 3.3 we design TB scheduling policies, starting from MST, and improve it to k-way partitioning and recursive graph bi-partitioning. We compare these partitioning techniques to understand its effect on L1 and L2 locality.
- In Section 3.4 we explain the PAVER runtime and TB scheduling and how task stealing can help with an application's load balancing. We further discuss the architectural support to store the TB-groups produced by various TB-partitioning strategies and using them to guide TB scheduling.
- In Section 3.5, we evaluate PAVER and show that our proposed scheduler can achieve average speedup of 29%, 49.1%, 41.2% compared to LRR in Fermi, Pascal and Volta architectures. Benchmarks selected from widely used benchmark suites Parboil [135], Rodinia [24], Polybench [116], ISPASS [1], and CUDA SDK [100] were analyzed and classified into high, low and no inter-TB locality categories.

3.1 Background and Motivation

TB scheduling is managed by the GigaThread engine in Nvidia GPUs [84]. Despite efforts to approximate the behavior of the TB scheduler [98], there are few official details on it. The baseline TB scheduler is assumed to be using a loose round-robin (LRR) policy, as empirically observed in prior work [87]. A round-robin policy implicitly takes advantage of locality among consecutive TBs that can simultaneously access the L2 cache. To capture both L1 and L2 locality, some prior works assign TBs at the granularity of a group of consecutive TBs (typically groups of two) to an SM [81, 137]. These prior techniques specifically exploit the two-dimensional data grid with locality typically occurring between TBs in the same

row. Such a policy is bound to fare poorly if there is large column-wise communication, or even worse, if the communication between TBs is arbitrary.

As a novelty, we attempt to establish an order in thread block execution by using a graph-based approach to maximize data locality. This method is completely generic and is able to extract locality patterns through graph-theoretic approaches regardless of the application's data layout or algorithmic behavior. To this end, we determine the locality among all the thread blocks per kernel before the kernel launch, and create a weighted graph pertaining to that kernel (for multi-kernel applications, a graph is generated for each kernel). The vertices are annotated with thread block IDs and the edge weights represent the number of shared data references between two TBs. The higher the weight of an edge between two nodes, the higher is the locality between the two TBs. We will later explore various locality graph partitioning techniques in order to assign TBs to SMs to maximize cache reuse.

3.1.1 Baseline GPGPU architecture

Table 3.1: Fermi, Pascal, Volta GPU specifications (for evaluation)

Architecture	Fermi	Pascal	Volta
# of SMs	15	28	80
Max # of TBs/Warps/Threads per SM	8/48/1536	32/32/2048	32/64/2048
L1/Shared Mem. Cache Size per SM	16/48 KB	48/96 KB	32/96 KB
Total L2 Cache Size	768 KB	3MB	4.5MB
Core Frequency	700 MHz	1 GHz	1.2 GHz

In this work, we use Nvidia GTX480 (Fermi [101]), Nvidia TITANX (Pascal [102]) and Nvidia TITANV (Volta [103]) architectures for evaluation purposes. Our technique exploits application-level characteristics and can be generally applicable to all architectures. Table 3.1 describes the specifications of GTX480, TITANX and TITANV, and Figure 3.1

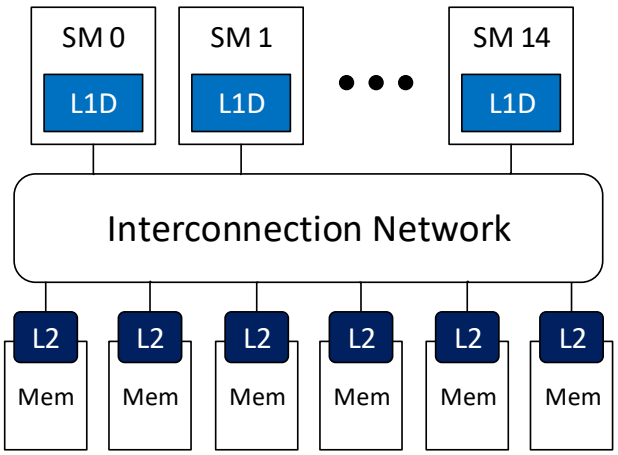

Figure 3.1: GPU memory hierarchy

depicts the memory hierarchy layout for a generic GPU. As it can be seen, every streaming multiprocessor (SM) has an L1 cache, and there is an L2 cache per memory channel that is shared by all the SMs. Whenever a load operation attempts to access a data which is already cached, it is considered a *cache hit*. Otherwise, it results in a *cache miss*, and the cache will request the data from the lower memory in the hierarchy. A miss can occur due to the data simply never being present (cold miss), or due to the data block being evicted from cache because of cache size limit (capacity miss), or set conflicts (conflict miss), or becoming stale after another 'sibling' cache has modified it. In Fermi, however, since there is a write-evict policy [106], we do not experience cache invalidations due to coherence protocols.

All load/store units (LDST) in the SMs have a memory coalescing unit to reduce unnecessary cache accesses. To further accommodate memory coalescing and avoid extra memory traffic, an SM's L1 data cache has a miss status holding register (MSHR) table, which sends the request to the L2 cache and holds the pending data block request while it is being loaded. If another thread tries to access the same block again, it is called a *hit*

reserve or MSHR hit. In this case, the operation will wait for the request in the MSHR to be finished. There can be MSHR hits in a cache only if there is data sharing across warps. If the MSHR is full, any more requests will be a miss and named a *reservation fail* [13].

3.1.2 L1/L2 hit and miss distribution

There are two main types of locality: temporal and spatial. Temporal locality refers to the same data in the cache used at different times, while spatial locality involves different units accessing different parts of the same data block, e.g. two TBs accessing two adjacent elements in the same cache line. We define a term called *"block locality"* to capture the usage frequency of any data element inside a data block either due to temporal or spatial locality during the execution. If the GPU knows which data block is needed by which specific TB, it can schedule the appropriate TB before the data block is evicted. GPU execution can benefit from a planned scheduling, which maximizes data locality in all caches. If TBs that use distant parts of the memory also share the same SM, it will lead to unnecessary L1 evictions and thrashing, hurting the performance. Similarly, if TBs executing on different SMs lack block locality, they will suffer from L2 eviction.

We measured the L1 data cache and L2 cache hit and miss distribution for different benchmarks and observed that on an average, 33.6% of all misses in L1 are conflict or capacity misses. More conflict or capacity misses lead to more frequent L2 cache accesses, which may in turn result in L2 evictions. Also, it was noted 9% conflict + capacity misses in L2 on average. Our method will reduce the average cache miss rate due to both conflict and capacity misses.

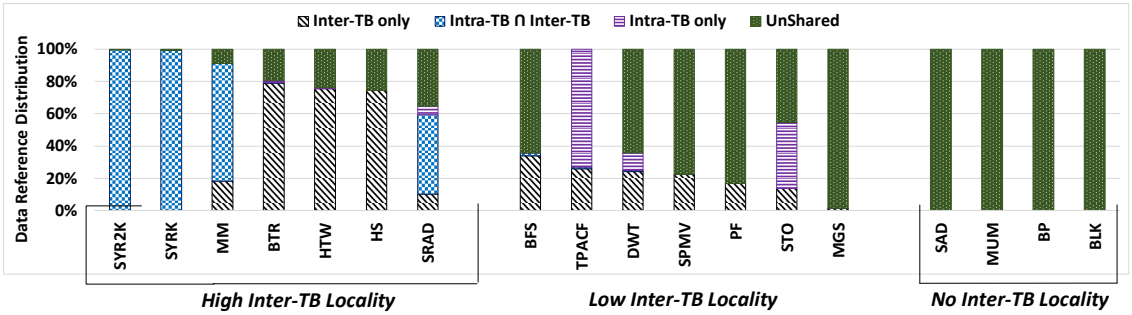

Figure 3.2: Data reference sharing distribution

Data Reference Distribution: Figure 3.2 shows the data reference distribution for different benchmarks. The y-axis shows the normalized total number of global read accesses in an application as a percentage. We categorize data accesses into the following type of references:

- Unshared - single warp in a TB accesses a data ;
- Intra-TB - multiple warps within a TB access same data ;
- Inter-TB - multiple TBs access same data ; and
- Intra-TB ∩ Inter-TB - data block is accessed by multiple warps within a TB, and across multiple TBs.

As it can be seen, the benchmarks SYRK, SYR2K, MM, BTR, HTW, HS, SRAD have heavy inter-TB data sharing; BFS, TPACF, DWT, SPMV, PF, STO and MGS have low inter-TB sharing; SAD, MUM, BP and BLK have no inter-TB data sharing. Therefore, if TBs sharing the data are assigned to the same SM, we can increase the L1 hits, thereby reducing the number of L2 accesses and lowering L2 conflict misses, and improving IPC. On

an average, 27% of data sharing between TBs is observed. Hence this work targets exploiting inter-TB and intra-TB ∩ inter-TB references by cleverly scheduling the TBs instead of the naive LRR TB scheduler.

3.2 Generating Locality Graphs

3.2.1 PAVER overview

Figure 3.3 overviews the **PAVER** framework, a **P**riority-**A**ware **V**ertex schedul**ER**. It is a locality-aware thread block scheduling (TB) framework, which is guided by locality graph analysis. The load address ranges for a TB are extracted from the PTX code. The locality graph is constructed from the extracted locality information, where the vertices of the graph represent the TB ID and the edge weight represents the number of common shared data-references accessed by those TBs. The TB graph capturing the inter-TB locality is partitioned such that each partition contains TBs having maximum locality among them (explained in Section 3.3). After this stage, we enforce the execution of a TB partition in an SM through hardware support. A locality-aware TB scheduler assigns the TBs to the SMs in the order specified by the partitioning algorithm (explained in Section 3.4).

3.2.2 Locality Graph:

The data reference sharing information is used to generate a locality graph in the form of an adjacency matrix. In this matrix, every element in location (i, j) specifies the number of data references shared between TB_i and TB_j. We represent the graph by an adjacency matrix is symmetrical around the diagonal (undirected graph). The examples

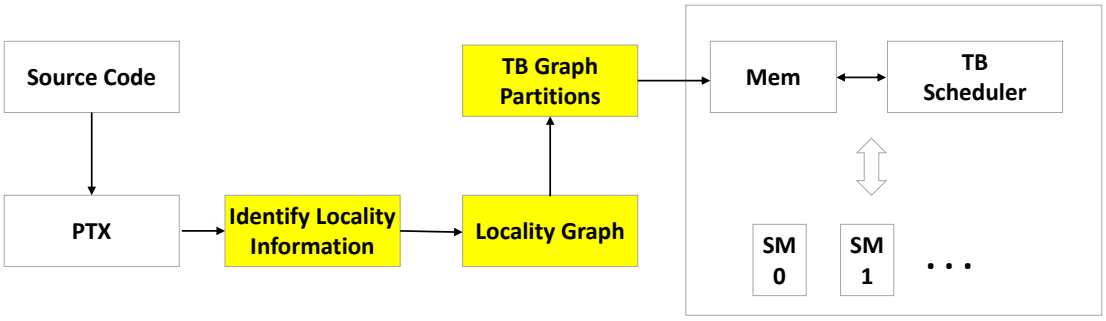

Figure 3.3: PAVER Overview: Paver generates the locality graphs by identifying the locality information in the PTX code at JIT. The TB-graph partitions are then fed to TB scheduler at run-time to leverage the inter-TB data locality.

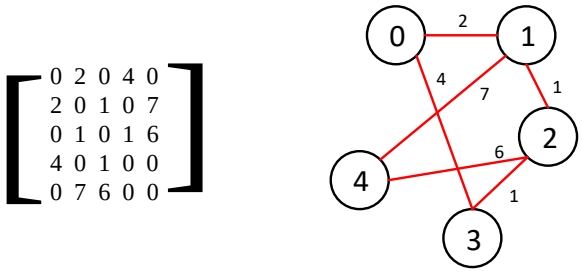

Figure 3.4: An example of an adjacency matrix and the corresponding locality graph

of an adjacency matrix and the corresponding locality graph are shown in Figure 3.4. The nodes are numbered as per the TB ID, and node weights represent the number of instructions executed by the node. In our case, the weights of all nodes are the same, because all the TBs execute the same number of static instructions in an SIMD manner.

Figure 3.5 visualizes the adjacency matrix representing the locality graph for different applications. Both X and Y dimensions represent the TB numbers. The sharing (edge) between them is represented by a point in the figure, where the color density of the point represents weight on the particular edge. It may be observed that maximum sharing

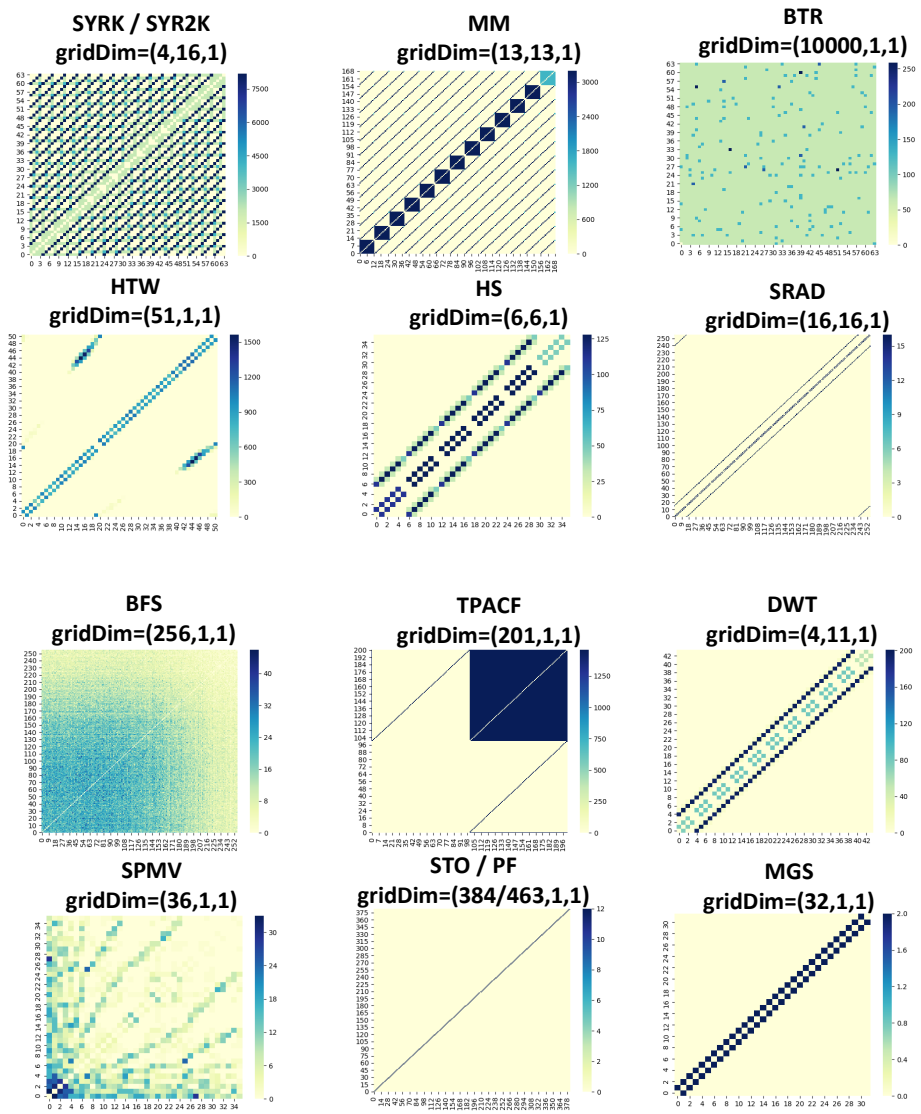

Figure 3.5: Adjacency matrix of Various Applications.

between two TBs may occur when they are adjacent (MGS, STO, PF, HTW) or when they are far away (BFS, BTR, SPMV). Though the prior work in [81] assumes that consecutive TB have max locality, this is not necessarily true. Sometimes there is sharing among adjacent TBs in the same row and same column of the 2D Grid (SYRK, SYR2K, MM), same row and first column of 2D Grid (HS, DWT, SRAD), same row of 1D Grid (MGS, STO, PF, HTW) or arbitrary (BFS, BTR, SPMV). A generic graph representation accounts for all these cases.

3.2.3 Identifying Locality Information

In order to run CUDA applications, the CUDA code must first be parsed into the PTX intermediate representation (IR) during compilation. In this stage, the code is converted into a quasi-Assembly structure, with instructions using input/output registers, indirect addressing, etc. However, after conversion to PTX, the code is still not ready for execution on real hardware, as GPUs require a code format specific to their architecture, i.e. the SASS representation. The conversion from PTX to SASS is performed via just-in-time (JIT) compilation at the time of application load [104]. This is when some of the remaining unknown parameters are resolved that might be dependent on user input during the kernel launch in order to specify the kernel's characteristics, such as input/output arguments and pointers, grid and block sizes, etc. When the kernel is converted to SASS format with all the necessary parameters, only then can it run on the target GPU. In our work, we aim to perform analysis on the PTX code before the kernel launch in order to extract locality information from the kernel PTX.

Profiling has been extensively used in CPUs [8, 144, 167]. In a recent paper for GPUs [38], Ocelot [30] was used to instrument the PTX code. The instrumentation involves inserting a device function call to gather the memory trace of the entire program in order to detect the uncoalesced accesses in the code. Alternatively, Shen et al. [130] use an instrumentation engine, built on top of LLVM [78] to place bits of code on both the host and device sides to track statistics such as memory reuse distance. The code will then be translated into PTX. However, profiling requires the user to run the application once to extract all the relevant information, which is not desirable if the data size is large. To

minimize the user burden, preprocessing the code before execution is needed. In PAVER, we propose a JIT-based static analysis to extract TB locality information from each kernel. In a CUDA application, inputs and outputs are given to the CUDA kernel call as base pointers initialized through `cudaMalloc()`. The kernel would then read the data from the input data structures and write the results in the outputs. In order to generate a locality graph for our application, the accessed read addresses will need to be extracted in order to determine address in the global memory, accessed by a TB.

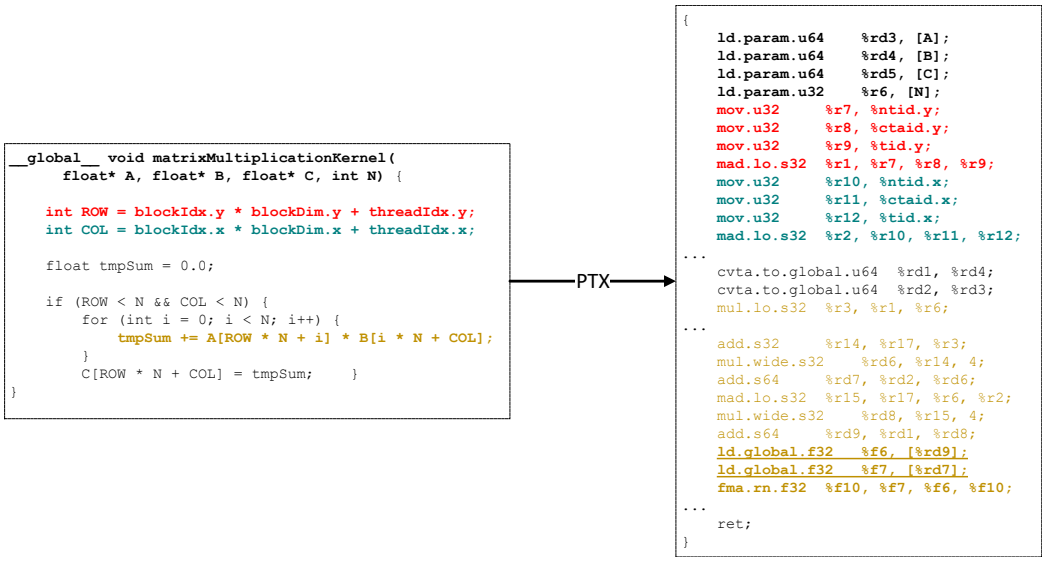

Figure 3.6: Matrix multiplication source code (left) and some of its corresponding PTX representation (right)

Figure 3.6 shows an example of the matrix multiplication code in CUDA converted into PTX. Some of the corresponding codes on both sides have been highlighted with matching colors. It is in our interest to extract the information from the global memory read instructions (e.g. `ld.global`, underlined) for each TB in order to determine the locality

Table 3.2: Data reference sharing across TBs for various applications from [1, 24, 100, 135].

Benchmark	Description	*SpScore*	Total TB	Total Data references	Shared TB	Degree of sharing
HS	Hotspot	0.995	1849	524288	1849	0.009194
PF	Pathfinder	0.995	463	2100000	463	0.036717
STO	StoreGPU	0.994	384	49164	384	0.044271
BFS	Breadth first search	0.006	3907	7811036	3907	0.066406
SRAD	(Speckle Reducing Anisotropic Diffusion	0.977	16384	4196352	16384	0.066406
TPACF	Two Point Angular Correlation Function	0.745	201	148388	201	0.084577
MM	Matrix-multiply	0.857	169	1024	169	0.100592
SYR2K	Symmetric rank-2k	0.414	256	196608	256	0.265625
SYRK	Symmetric rank-k	0.414	256	131072	256	0.265625
HTW	Heart Wall	0.923	51	78135	51	0.333333
MGS	Merge Sort	0.939	32768	8454144	32768	0.375
BTR	B+ Tree	0.000	10000	674287	10000	0.0017
DWT	Discrete wavelet	0.862	4096	65536	4096	0.386364
SPMV	Sparse-Matrix Dense-Vector Multiplication	0.430	765	6981392	765	0.472222
MUM	MummerGPU	1	196	1055946	0	-
SAD	Sum of Absolute Differences	1	1584	25344	0	-
BLK	Black scholes	1	480	6000000	0	-
BP	Back-propagation	1	4096	1114112	0	-

among all TBs in the kernel. In order to extract memory access information, a JIT analysis is necessary, since the arguments in a kernel call may not be known until the kernel is finally being called. In addition, some of the kernel parameters, such as kernel grid size, block size and input data size, may also be unknown before the call, e.g. if they are dependent on user input. However, once the kernel is called, all the information stated above will be available. Therefore, each thread's unique parameters, such as thread ID and block ID, would also be known at that time, and would remain the same throughout the kernel's execution, which simplifies our analysis. Once the analysis is complete for the called kernel, we have each TB's memory access locations, which can then be used to construct a locality graph. The exact values of the matrix size N and the base addresses of input/output matrices

A, B, C are known after *malloc* in GPU. When the grid size and TB size become available, so do the ranges of the existing thread-specific values, namely $threadIdx$ and $blockIdx$. Therefore, ROW and COL (colored red and teal) expressed in terms of these two values can readily be determined for each thread at JIT compilation. Also, we can locate the global memory access instructions and identify which elements of the arrays are accessed by each TB, thereby, determining the value range of the memory accesses per TB. Here, for example, with ROW and N being known and i iterating from 0 to N, we know all the possible values of index $ROW * N + i$, and thus all elements of A being read in the kernel by a particular thread. Similarly, all the elements of matrix B and the thread blocks accessing them can be known. Any $TB_k \in \{0, 1, ..., gridDim.x * gridDim.y * gridDim.z\} (= U_{TB})$ accesses a set of elements in matrices A and B. For A and B, the access sets for TB_k will be $A_k = \{i + N(\lfloor \frac{TB_k}{gridDim.x} \rfloor blockDim.y + j) \mid i \in [0, N), j \in [0, blockDim.y)\}$ and $B_k = \{(i.N(TB_k \% gridDim.x) blockDim.x + j) \mid i \in [0, N), j \in [0, blockDim.x)\}$. Using this, we can obtain the common set of elements in matrices accessed by every TB_i and TB_j, denoted by sets (A_i, B_i) and (A_j, B_j). Therefore, the number of common data elements accessed by TB_i and TB_j in the locality graph will be: $L(TB_i, TB_j) = |A_i \cap A_j| + |B_i \cap B_j|$. Table 3.2 showcases the characteristics of the benchmarks used in this work. Note that *Sparsity score* is expressed as: $SpScore = 1.0 - \frac{non-zero\ elements\ in\ matrix}{total\ elements\ in\ matrix}$. *Shared TB* refers to the number of TBs, which share at least one data reference. The reported statistics are averaged over all the kernels for an application. It may be noticed that the *degree of sharing* (ratio of shared blocks to shared TBs) varies widely depending on the application. The applications with large sharing are likely to benefit from proper TB scheduling.

In PAVER, our focus is *static memory analysis* at JIT, or analysis of memory locations available before the execution of the kernel, such as device variable addresses, immediate values, and kernel parameters. As an example, if `A` is an input kernel argument, it counts as a static memory location if we use an index that is available at JIT, such as `A[0]`, `A[i]` (where `i` is a loop parameter), `A[tid]`, etc. At this time, we do not analyze *non-static memory locations*, which can only be known during run-time, e.g. pointer chasing. An example of a non-static memory location is `A[A[0]]`, since the value stored in `A[0]` cannot be known except at run-time, which is outside the scope of this work.

Overhead

The JIT analysis is done before the kernel-launch and hence does not affect the kernel-execution time, but it increases the *host side time*. The functions like initialization, memory allocation (malloc), cudaMalloc, cudaMemcpy, defining the gridDim and blockDim contribute towards the host side time. Usually, the kernel load is executed in CPU before the GPU execution starts and is negligible. Since the JIT analysis is done right after the grid and block dimensions are defined, the JIT overhead is calculated as : $Overhead = \frac{JIT\ time}{host\ side\ time\ in\ baseline + JIT\ time} * 100$. The JIT overhead for some applications with high inter-TB locality SYRK, SYR2K and MM are 0.26%, 0.2% and 0.01%, respectively. Note that device to host cudaMemcpy has been excluded from the overhead calculation as it does not delay the kernel load time. The overhead will reduce further if the device to host cudaMemcpy time is included.

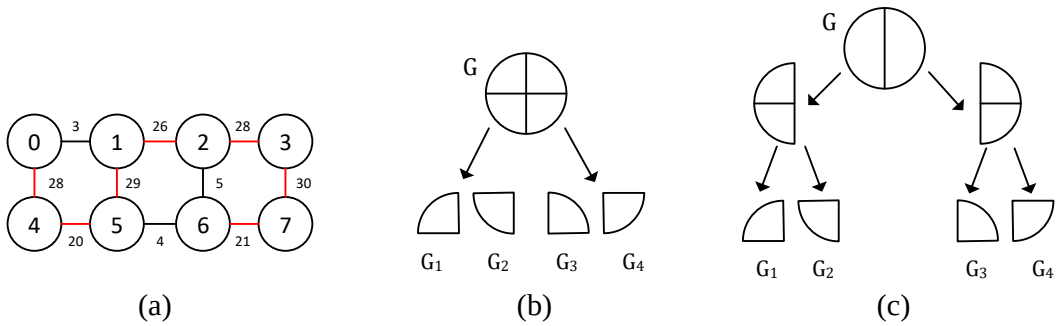

Figure 3.7: Overview of TB grouping and ordering approaches. (a) maximum spanning tree. (b) k-way partitioning. (c) recursive bipartitioning.

3.3 PAVER Thread Block (TB) Scheduling

In this section, we explore three different graph-based locality analysis techniques to partition and group thread blocks to guide TB scheduling, namely; a naive approach using maximum spanning tree (MST); a k-way partition-based method to improve the hit rate of L1 caches; and a recursive bipartitioning-based approach to account for both L1 and L2 cache performances. Figure 3.7 shows an overview of each grouping strategy. TB grouping and ordering is done at the just-in-time (JIT) compilation and then the TB partitions are passed onto the GPU's global memory to guide TB scheduling.

3.3.1 Maximum Spanning Tree-based TB Scheduler (MST-TS)

In our first approach, we map this problem into a variant of the traveling salesman problem (TSP), which aims to traverse all the vertices (TBs) in the graph with minimum (or maximum) traveling cost. In our case, we can leverage the TSP problem to capture all the significant cases of data sharing in terms of maximum edge weights in the graph G.

The heuristic we use to solve TSP is the maximum spanning tree (MST). The MST could be constructed using either Prim's [117] or Kruskal's algorithm [76]. In this work, we use Prim's MST. Once the MST is constructed, we have a path connecting all the TBs. An example of an MST solution (red lines) is displayed in Figure 3.7 (a), where each node represents a TB. We then partition consecutive TBs in the MST into N groups of size x, where x is equal to the number of TBs that can run concurrently in the SM and N is the number of SMs in the GPU architecture. N is limited by the hardware resources (registers, shared memory, number of threads, maximum number of TBs in an SM, etc.). After kernel launch, the first TB-group of size x is assigned to a SM, thereafter, the subsequent TB groups assigned to the SM have one TB each. For example, if our N is 2 and x is 2, then we have 6 groupings of (0,4), (1,5), (2), (3), (7) and (6). This assignment aims at achieving high L1 locality and load-balancing across the SMs.

This approach captures more inter-TB locality than BCS and LRR. BCS groups two consecutive TBs into a pair and assigns them to the same SM. This approach will not work for the applications having column wise locality in a 2D grid or arbitrary locality pattern. Additionally, the TB pair assignment is delayed till the TB contexts for the pair is available, leading to resource starvation in the SM. MST-TS overcomes the SM-under utilization issue observed in BCS by assigning a TB as to the SM as soon as the context becomes available and using a graph-based representation which accounts for all types of locality patterns. However, it only captures one-dimensional sharing without considering data sharing between more than one TB. Thus, we need a more generalized approach.

3.3.2 *k*-way Partition-based TB Scheduler (Kway-TS)

Although the MST ordering could enhance the data locality within TB groups, it considers a TB that has the highest data sharing on the path and ignores other connections to a TB that may also share data. Given a graph $G(V, E)$, E' is the set of edges belonging to the MST. In the MST ordering, we only order the TBs based on the data sharing on E'. A partition based on the complete edge set E should produce better inter-TB locality-aware groupings for L1 cache locality inside the SM.

We present k-way partitioning, in which k is the total number of partitions equal to the number of SMs. We evenly partition the entire set of TBs to all the k SMs considering data sharing. For example in Figure 3.7 (b), we partition the graph G into 4 groups to be assigned to 4 SMs. Graph-partitioning tools, such as METIS [63] or Chaco [47] can be used, to partition the graph in a load-balanced manner while maximizing the sum of edge weights within partitions. In this work, we utilize METIS for graph partitioning. The main advantage of this method over MST-ordering is that the graph is partitioned in a way that the groups have the highest connectivity, i.e. TBs have the highest locality within a group. Therefore, it leads to a much higher L1 locality. Since all the TBs in a partition cannot execute concurrently due to resource limitation in an SM, *we re-order the TBs in each partition using Prim's MST such that the subset of TBs in each partition executing concurrently has maximum locality with each other.*

One disadvantage of k-way partitioning is that L1 data locality is maximized within each partition. However, each partition is executed in an SM over the entire kernel runtime, with multiple partitions without locality running simultaneously on different SMs. The

locality between the SMs may be lost leading to L2 thrashing. This type of scheduling prioritizes L1 locality over L2 locality.

Algorithm 1 Recursive bipartitioning

```
Let Q be the queue of TB groups
Let L be the list of TB groups for SM scheduling
Q.push(G_0)
while Q not empty do
    G_i =Q.front()
    Q.pop()
    (G_i^0, G_i^1)=METIS.partition(G_i, n = 2)
    if G_i^0.size() < maxTB then
        move G_i^0 to list L
    else
        Q.push(G_i^0)
    end
    if G_i^1.size() < maxTB then
        move G_i^1 to list L
    else
        Q.push(G_i^1)
    end
end
```

3.3.3 Recursive Bi-partition-based TB Scheduler (RB-TS)

The problem mentioned before with k-way partitioning inspires us to design the recursive bi-partitioning scheduling for TBs to guarantee maximum data locality across L1 and L2 and load balance between TB groups. As shown in Figure 3.7 (c), we recursively partition the graph G into two parts until the partition size is less than the maximum allowed TBs in each SM (which we denote as x from Section 3.3.1). This essentially creates a binary tree where the leaf nodes (G_1, G_2, G_3 and G_4) are prioritized from left to right. This preserves L1 locality by TBs grouped within the same leaf node, and preserves L2 data locality by concurrently scheduling adjacent groups on different SMs that share the same parent.

The pseudocode for our recursive bipartitioning algorithm is shown in Algorithm 1. Q is the queue to store the TB groups, which need further partitioning. L stores all the final leaf TB groups in the partition tree. At every iteration, we pop out one sub-graph G_i from the queue and partition it into two different TB groups. If G_i is smaller than the TB capacity of an SM, we get one final TB group. Otherwise, the algorithm pushes it to the back of the queue Q and waits for further bipartitioning. Our algorithm uses METIS to achieve recursive bipartitioning with load balancing.

3.4 PAVER Runtime

In this section, we will discuss the generalized PAVER Runtime, which schedules thread blocks (TBs) based on graph-based TB grouping strategies. Once the TB groups have been created and re-ordered at JIT compilation, the TB scheduler uses them at runtime to schedule TBs among SMs. For PAVER TB Scheduling policies, a global queue (located in global memory) is used to store the pointers to the TB groups. We assume that the maximum number of concurrent TBs executing on an SM is x, which is dependent on the kernel resource requirement like registers, shared memory, local memory and number of threads in an TB.

3.4.1 Hardware Implementation

The architectural modification needed for supporting PAVER is shown in Figure 3.8. Once the kernel is loaded, the TB groups and ordering within each group are stored in the global memory as an array of arrays. The *global queue* stores the array of partitions.

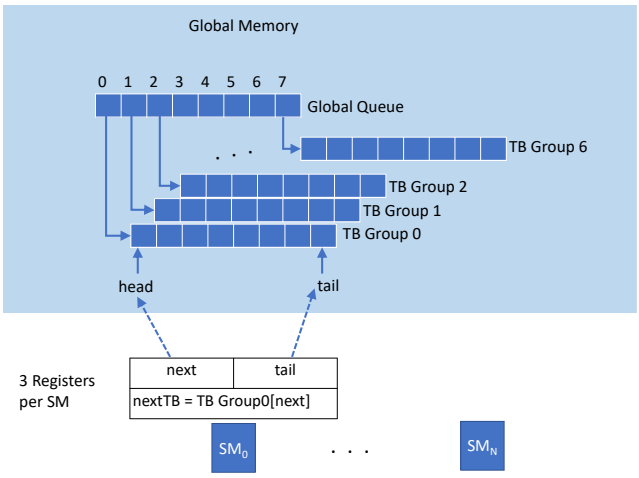

Figure 3.8: Storing TB Groups in Global Memory: Once the TB-Groups are generated by different graph-partitioning strategies (MST, Kway, Recursive Bipartitioning), they are stored in the global memory. A global queue (located in global memory) is used to store the pointers to these TB groups. Each SM is associated with two registers (*next, tail*) which point to the TB group's *head* (initially) and *tail* assigned to that SM, respectively. Once the current TB from the TB group is issued to the SM and starts executing, the *next* register value is updated to point to next TB in the TB group and the next TB register is loaded with the new value i.e. TB group[*next*]. This next TB register guides the thread block scheduler.

Each entry in the array corresponds to a partition and points to the head pointer of an array that stores all the TB groups of that partition in order. The number of TB groups and size of each TB group differs by the partitioning technique used. Each SM is associated with two registers (*next, tail*) which point to the TB group's *head* (initially) and *tail* assigned to that SM, respectively. Another 2-byte register in the SM stores the *next TB ID* i.e. TB group[*next*]. Once the current TB from the TB group is issued to the SM and starts executing, the *next* register value is updated to point to next TB in the TB group and the next TB register is loaded with the new value. This next TB register guides the thread block scheduler.

Storage Overhead:

The 3 extra registers per SM to store *next, tail and nextTB* incur an area overhead of 64-bit * 2 (for next and tail) + 16-bit (for nextTB) i.e. 18 Bytes. So, total storage overhead for Fermi (15 SMs), Pascal (28 SMs) and Volta (80 SMs) are 270 Bytes, 504 Bytes, and 1440 Bytes respectively. This overhead is negligible compared with the area of other on-chip storage structures in Fermi/Pascal/Volta GPU such as L1 cache (240KB/720KB/480KB), shared memory (1920KB/3840KB/3840KB) and Register File (1920KB/14MB/40MB).

Timing Overhead:

Any timing overhead would occur when the nextTB is yet to be loaded from memory. However, this operation is off the critical path as the fetching occurs while the TBs are executing on the SM. The only time the penalties occur is when there is a free TB context available in an SM but there is no ready TB to be issued. However, this scenario is very rare as the time taken for loading a TB from memory is very small compared to the TB execution time.

3.4.2 Task Stealing

Towards the end of the execution run, if the scheduling of a group to an SM leads to load imbalance, we utilize task stealing to balance out the load of the last group amongst SMs. Note that since PAVER focuses on RAR locality, rather than RAW, it will not preclude cases where issued thread blocks on a busy GPU are dependent on still-unissued TBs which can result in a deadlock.

When there are no more unassigned groups, we employ *task stealing* to improve performance through load-balancing. Ideally SMs should finish their workloads all at the same time. However, there are workloads of different sizes on each SM. Therefore, an SM could finish early and stay idle while the other SMs are still working on their TB groups. With task stealing, however, we take a TB assigned to a busy SM (but not yet issued) and reassign it to a free SM. By tapping into the freed resources, we can make sure that the SMs are utilized as much as possible until the application's termination, resulting in additional performance.

Algorithm 2 Task Stealing Algorithm

```
Max_WaitingTB=0
Average_WaitingTB=0
for SM in range(0, total SM) do
    WaitingTB= SM.tail - SM.next
    if WaitingTB > Max_WaitingTB then
        Max_WaitingTB=WaitingTB
        DonorSM=SM
    end
    Average_WaitingTB += WaitingTB
end
Average_WaitingTB /= total SM
Stolen TB count = Max_WaitingTB - Average_WaitingTB
return DonorSM, Stolen TB count
```

Task stealing is employed in k-way-TS and RB-TS. In MST-TS, the TB-Group size is 1, hence a SM does not have more than 1 TB waiting in the TB group. When all the tasks in the TB group are exhausted and SM has a free TB context, it is marked as recipient SM. While, the SM with maximum number of waiting tasks in the TB group is the donor SM. The "WaitingTB" of each SM is determined by the number of waiting tasks in its TB group (obtained from $SM.next$ and $SM.tail$). The granularity of tasks stolen is determined as: $Max_{WaitingTB}$ - $Average_{WaitingTB}$.

If *Stolen TB count* > 2, it still captures some locality. Task stealing affects the L1 locality within the donor SM but the overall kernel execution saves the extra cycles by load-balancing the SMs. Also, the locality at L2 level is still preserved.

Storage Overhead: For the task-stealer, 3 16-bit registers are used to store the $Max_{WaitingTB}$, $Average_{WaitingTB}$, *Stolen TB count* which accounts for 6 Bytes. This overhead is negligible compared with the area of other on-chip storage structures in Fermi, Pascal and Volta GPU.

Control logic Overhead: The task stealer uses a very simple control circuit consisting of 3 adders, 1 comparator and 1 divider. The area and power overhead of these control logic would be negligible compared to the GPU chip area.

3.4.3 Generalized Runtime Algorithm for all TB Policies

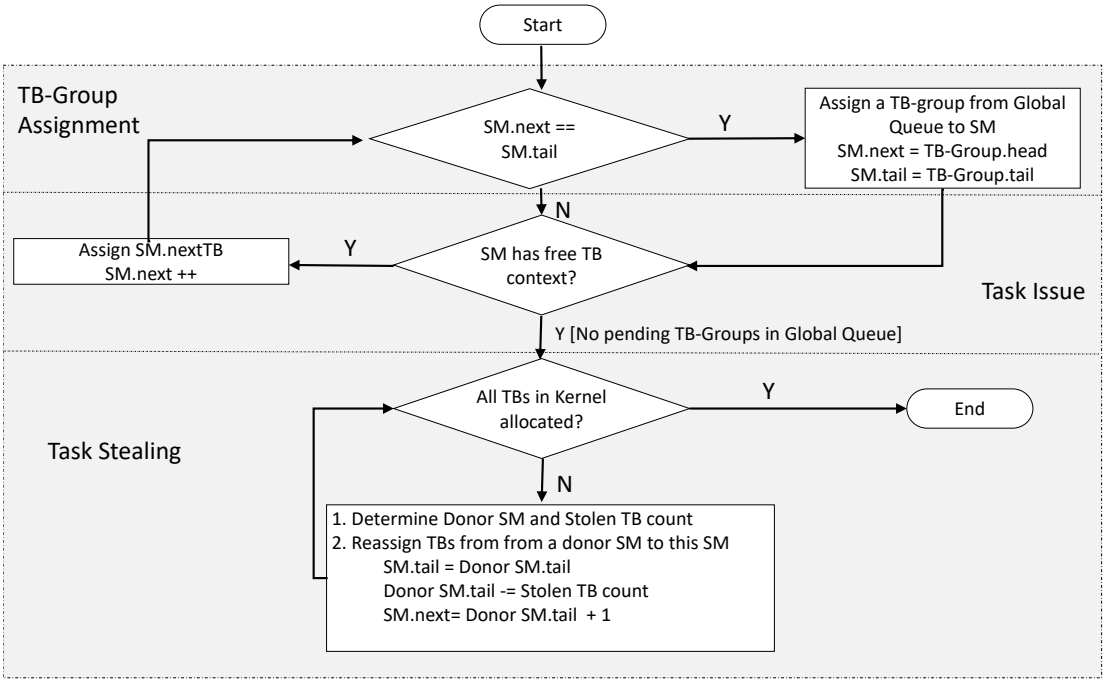

Figure 3.9: PAVER Runtime Flowchart

Once the kernel starts executing, TB scheduler assigns the TB groups from the global queue to the SMs in a round-robin manner. As shown in Figure 3.9, once a TB group is assigned to an SM, the $SM.next$ and $SM.tail$ are initialized to point to the TB group's head and tail, respectively. Due to limited hardware resources, a limited number of TBs can be issued and executed concurrently in an SM. A TB (stored in $SM.nextTB$) from the *assigned TB group* is *issued* to the SM as soon as a free TB context is available. Upon TB issue, SM.next is updated to point to next TB in the assigned TB-Group. TB-scheduler then fetches the $nextTB$ to be issued, located at TB Group[SM.next] and fills up the $SM.nextTB$ register. If $SM.next == SM.tail$, then all the TBs in the TB group are exhausted. The PAVER TB scheduler will then assigns another available TB group to the SM.

When all the entries in the global queue are exhausted, the load imbalance occurs, as an SM has exhausted its TB group and has a free TB context available, while other busy SMs have TBs waiting in their assigned TB groups. In this scenario, the task stealing is enabled. The donor SM and number of TBs to steal are determined as per the task stealing algorithm (described in Algorithm 2). The tasks are stolen from the tail of the donor TB group and the tail of the donor SM is updated. After the task stealing, $SM.next$ and $SM.tail$ are updated accordingly. For example: There are 4 waiting TBs in the TB-Group of the donor SM indexed as (TB_0, TB_1, TB_2, TB_3), $DonorSM.next = TB_0$, $DonorSM.tail = TB_3$ and $Stolen\ TB\ count = 2$. After Task-stealing, recipient $SM.tail$ is updated to donor SM.tail i.e. TB_3. Donor $SM.tail$ is decremented by the stolen TB count and points to TB_1. Recipient SM.tail points to ($DonorSM.tail + 1$) i.e. TB_2.

3.5 Evaluation

3.5.1 Methodology

We use GPGPU-Sim [13] with simulation parameters in Table 3.1 to model GTX480 Fermi, TITANX Pascal and TITANV Volta GPUs. The warp scheduling policy follows a greedy-then-oldest (GTO) policy [121]. Our thread block scheduling technique can be run with any warp scheduler, but we find GTO to provide the best performance. Apart from our MST-TS, k-way-TS and RB-TS, we also implement the Loose Round Robin (LRR) and Block CTA Scheduler (BCS) scheduling policy as the baseline. We did not compare with warp scheduling policies, such as CCWS [121], because they are orthogonal to TB scheduling and can be applied to PAVER to further improve the performance.

LRR scheduler selects one SM at a time, and assigns a TB in a sequential order. LRR is one of the the fairest schedulers. However, since non-adjacent TBs are assigned to the same SM, there is less locality among them and their data accesses may cause the L1 cache to suffer from thrashing. Meanwhile, the locality will mostly remain on the L2 cache level because adjacent TBs will execute at the same time but in different SMs. BCS [81] is similar to LRR with one difference: it schedules two neighboring TBs at the same time with the assumption that the highest locality among TB occurs in neighboring pairs. This specifically benefits 2D grid-type workloads only. Since BCS will only schedule if the SM has enough resources to fit in two new TBs, it can potentially lead to performance reduction due to lack of resources. In the worst case, if no free context for a TB pair can be found during the execution, it may add unnecessary cycles to the execution by causing task serialization, whereas it could avoid such cases by filling any free context in the SM without pairing the

remaining TBs in that cycle. BCS has better L1 locality than LRR, but it can be improved much more. The L2 locality remains the same as LRR.

3.5.2 Benchmarks

Benchmarks shown in Table 3.2 are selected from Parboil [135], Rodinia [24], Polybench [116], ISPASS [1], and CUDA SDK [100] benchmark suites. All these suites are widely used in evaluating a GPU architecture's performance. Out of all the kernels in these suites, we used a subset that have high as well as low inter-TB locality. However, we also used a few benchmarks with little to no locality to test the impact of our TB scheduling policies to see if they are affected in any way. As shown in Figure 3.2, The benchmarks such as SYR2K, SYRK,MM, BTR, HTW, HS and SRAD which have at least 50% Inter TB Data-references (Inter-TB + Intra-TB $\cap$ Inter-TB) are considered high-TB locality. The benchmarks with less than 50% Inter TB Data-references are considered low-TB locality. The benchmarks which show low-TB locality are BFS, TPACF, DWT, SPMV, PF, STO and MGS. The benchmarks without any inter-TB locality are SAD, MUM, BP and BLK.

3.5.3 Results

3.5.4 Speedup Results

Figure 3.10 displays the kernel execution time as well as JIT Analysis Overhead of our different TB scheduling policies with respect to LRR and BCS. The x-axis shows the benchmark name, and the y-axis shows the Execution time + JIT Overhead w.r.t. LRR. The results have been normalized to the LRR TB scheduler. The TB policies LRR and

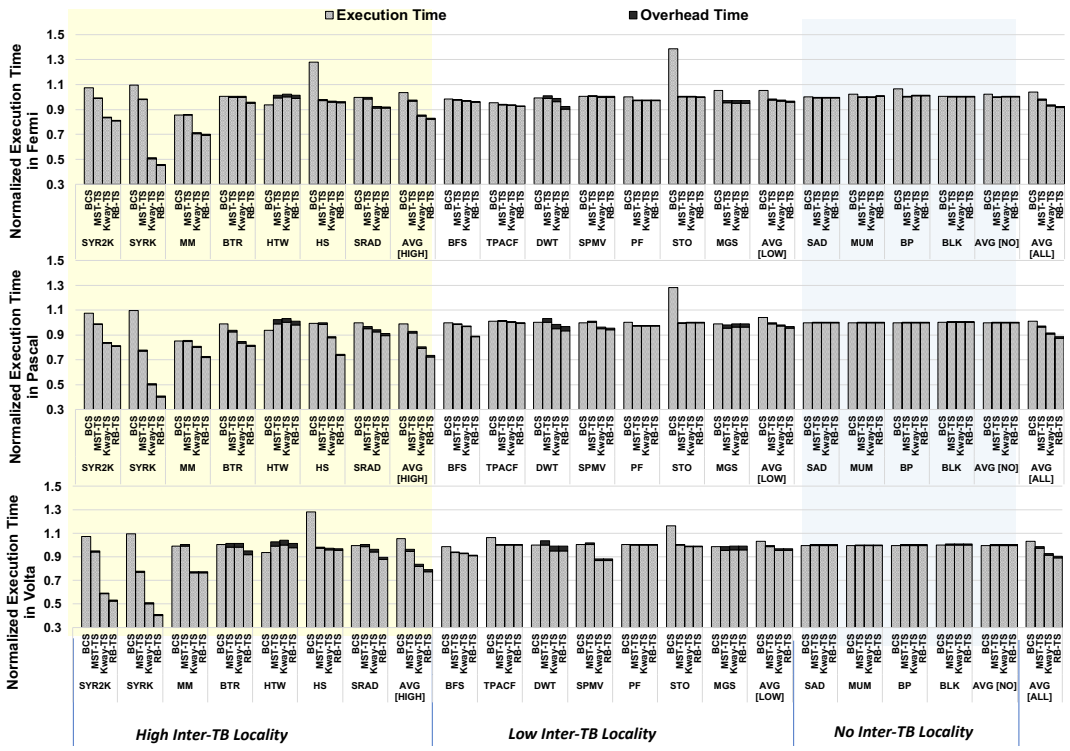

Figure 3.10: Kernel Execution Time and JIT Analysis Overhead of BCS, MST-TS, k-way-TS, RB-TS normalized w.r.t. baseline TB scheduling policy (LRR), on Fermi (top row), Pascal (middle) and Volta architectures (bottom).

BCS do not involve JIT analysis, hence, JIT time is excluded in BCS result. GPU kernel execution time is calculated using the kernel execution cycles and core frequency (from Table 3.1). On an average, the JIT overhead is observed to be a very negligible fraction (1%) of the total kernel execution time.

Figure 3.11 displays the speedup of our different TB scheduling policies with respect to LRR and BCS. The x-axis shows the benchmark name, and the y-axis shows the speedup w.r.t. LRR. The results have been normalized to the LRR TB scheduler. The speedup of PAVER TB scheduling policies over the baseline (LRR) is calculated as $Speedup = \frac{kernel\ execution\ time\ in\ baseline}{kernel\ execution\ time\ in\ PAVER+JIT\ analysis\ time}$. In Fermi (Figure 3.11 (top)), for

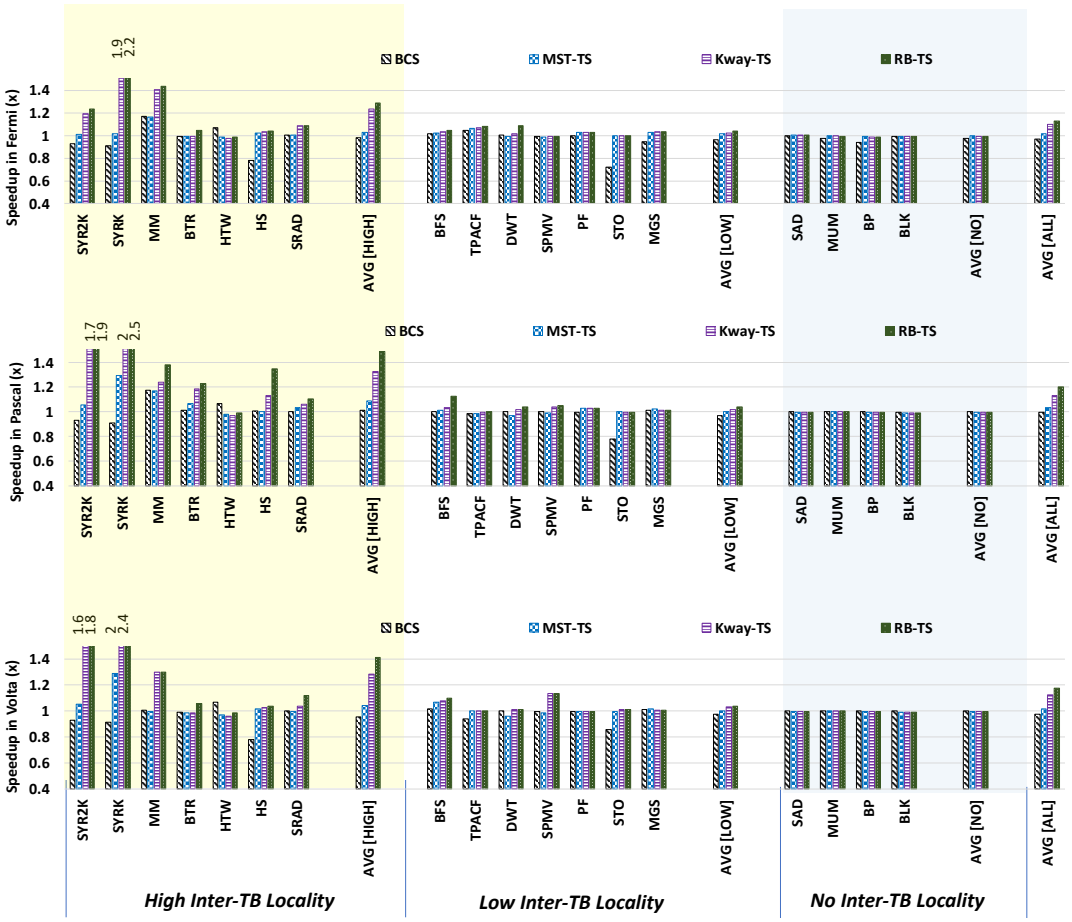

Figure 3.11: Speedup of BCS, MST-TS, k-way-TS, RB-TS normalized w.r.t. baseline TB scheduling policy (LRR), on Fermi (top row), Pascal (middle) and Volta architectures (bottom).

applications with high inter-TB locality, on an average, the TB policies: Maximum Spanning Tree-based TB Scheduler (MST-TS), k-way partition based TB Scheduler (k-way-TS) and Recursive Bipartition-based TB Scheduler (RB-TS) have 2.8%, 23.8% and 29% speedups respectively as compared to baseline LRR. The average speedup for applications with low inter-TB locality are for MST-TS, k-way-TS and RB-TS are 1.8%, 2.5% and 3.8% respectively. We also tested our TB policies on the benchmarks SAD, MUM, BP and BLK which do-not have any inter-TB locality to see if the IPC is affected by the new TB

scheduling policies. It is observed that the performance of our proposed TB policies for these applications is reduced by a negligible amount 0.15%,0.3% and 0.5% for MST-TS, k-way-TS and RB-TS respectively as compared to the baseline LRR scheduler. Overall, considering the applications from different inter-TB locality categories (high, low and no-sharing), MST-TS, k-way-TS and RB-TS show an average speed-up of 1.8%, 10.1% and 12.6% respectively. PAVER was evaluated for the recent architectures Pascal and Volta. Our graph-based TB policies MST-TS, k-way-TS and RB-TS achieve an average speedup of 10.8%, 32.5% and 49.1% for high inter-TB benchmarks; 0.2%, 1.8% and 3.7% for low inter-TB benchmarks; and 3.3%, 13.32% and 20.49% overall for different inter-TB locality applications in Pascal (Figure 3.11 (middle)). We observe that PAVER fares well in Volta (Figure 3.11 (bottom)), where MST-TS, k-way-TS and RB-TS achieve an average speedup of 4.5%, 28.6% and 41.2% for high inter-TB benchmarks; 0.4%, 3.5% and 3.8% for low inter-TB benchmarks; and 0.18%, 12.4% and 17.4% overall for different inter-TB locality applications.

The high inter-TB locality benchmarks get significant improvement in IPC through our graph-based TB scheduling policies, upto 2.2x for SYRK benchmark (Fermi). Note that our work is orthogonal to warp scheduling techniques, and therefore adding a warp scheduler [58, 99] on top of the TB scheduler could further increase the speedup.

In the case of BCS, if the maximum thread block (TB) per SM limit is an odd number, it leads to thread block throttling, which means some of the thread blocks could not be jointly assigned due to the lack of free TB contexts in the SM. For example, in the STO application, there are 384 thread blocks and yet 3 thread blocks executing on the SM at the same time (maximum concurrent TB execution per SM depends on the resources like

registers, local memory, shared memory, constant memory consumed by each TB). The BCS suffers drastically because every SM is assigned to execute 2 thread blocks instead of 3 as in the baseline. Similarly, in HS benchmark 3 TBs execute concurrently in a SM and lead to severe throttling incase of BCS TB policy which leads to around 22% slower execution than LRR. Similarly, for benchmarks like SYR2K, SYRK, BP and MGS even if the maximum TB per SM is even i.e. 6, 6 ,6 and 8 respectively, whenever a TB finishes execution in a SM, BCS does not start executing the next TB immediately. Rather, it waits for 2 TB contexts to be freed up so that a pair of TB can start executing. This leads to SM resources starvation and inferior performance by increasing the execution time of SYR2K, SYRK, BP and MGS by 7%, 9% ,6% and 5% respectively w.r.t LRR. Our policies, however, do not lead to throttling as evident in Figure 3.11. In addition, BCS assigns every two consecutive TBs to the SM based on the assumption that the neighboring TBs would always have a high inter-TB locality, which does not hold for 2- and higher-dimensional grid applications, where the locality could be between TBs in a column. Our graph-based approaches, however, are generalized methods and take all types of data access patterns into account.

It may be pointed out that prior work [87] covers a limited pattern behavior which has locality along the X-dimension or Y-dimension of the grid. In our work, any locality pattern (shown in Figure 5) can be analyzed in form of a graph and partitioned among the SMs to leverage maximum locality within the SM. Applications such as BTR, BFS and SPMV having irregular data-locality pattern application have been analyzed through our graph-based approach and gives significant speedup for the Fermi, Pascal and Volta architectures as shown in Figure 3.11. We get significant performance benefits compared

to the baseline LRR for the unstructured applications BTR (4.52% for Fermi, 22.8% for Pascal and 5.9% for Volta), BFS (4.4% for Fermi, 12.8% for Pascal and 9.85% for Volta); and SPMV (-0.5% for Fermi, 4.9% for Pascal and 13.7% for Volta).

Effect of Task Stealing

When all the thread blocks on an SM finish early and the SM is idle, the thread blocks on a busy SM are reassigned to the free SM for the load balancing purpose. Task stealing is beneficial for only k-way-TS and RB-TS approach as in these approach the TB groups are pre-assigned to the SM without accounting for the actual execution time of each SM. How-ever in the MST-TS the load-balancing is done implicitly when the TB from the MST are assigned to the SM in a round-robin fashion. Task stealing re-balances the workload of each SM when the kernel is on the verge of finishing the execution and some of the SMs have already finished their work. Average Performance benefits of 3% and 2% were observed for k-way-TS and RB-TS respectively w.r.t no task stealing case in Fermi.

Effect of Cache Size

Increasing the cache size shall reduce the capacity misses in the cache. However, the cache in GPUs tend to be limited in size. Since in the Fermi architecture 64KB of RAM had configurable partitioning between shared memory and L1 Cache, we increased the L1 Cache size from 16 KB to 48 KB to see the effect of the cache size on the performance of applications. With increased cache size PAVER[RB-TS] outperforms the baseline configuration by 16%.

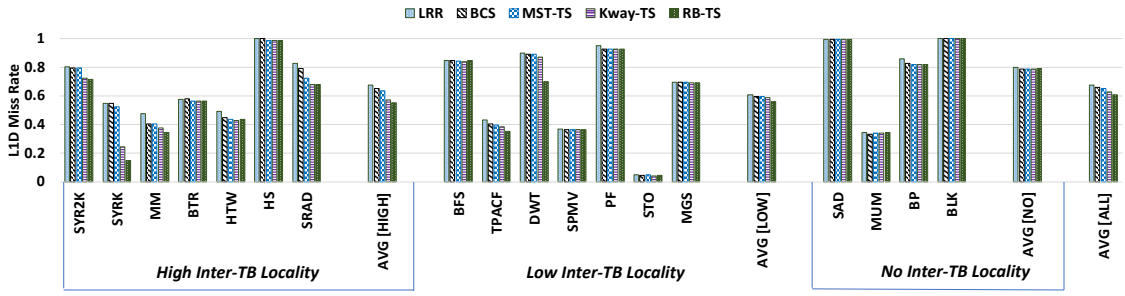

Figure 3.12: L1 miss rate comparison of LRR, BCS, MST-TS, k-way-TS and RB-TS in Fermi.

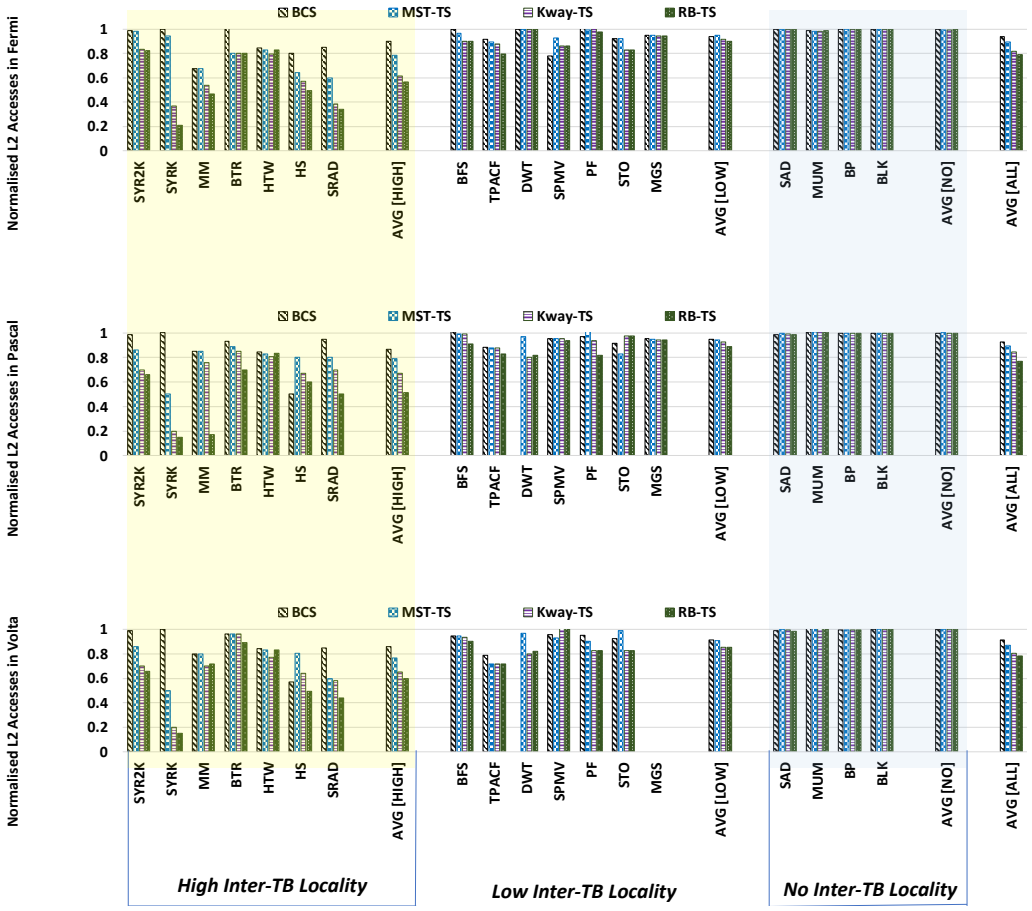

Figure 3.13: L2D access comparison for BCS, MST-TS, k-way-TS and RB-TS normalized w.r.t. LRR for applications with high, low and no inter-TB locality, on Fermi (top), Pascal (middle) and Volta architectures (bottom).

3.5.5 L1 Misses

Figure 3.12 shows the L1 miss rate of different TB scheduling methods explored in this work; MST-TS, k-way-TS, RB-TS, BCS and LRR. It can be seen that our recursive bipartitioning method reduces the L1 miss rate over LRR by 43.3% (high inter TB locality), 10% (low inter TB locality) and 21% (all applications), contributing to the speedup. The L1 miss rate shown in Figure 3.12 is proportional to the L2 accesses shown in Figure 3.13 (Fermi). Hence, we excluded the L1 miss rate results for Pascal and Volta as normalised L2 accesses accounts for that.

The average conflict and capacity miss for all the benchmarks is 9% and the cold miss constitutes 91% of the total miss at L2. Hence, we observe very minimal reduction in L2 misses as majority of them are cold misses, however the accesses to L2 are reduced by 21% as the graph-based TB policies reduce the misses at L1. Reduced accesses to L2 cache and shorter execution time leads to energy saving. Figure 3.13 shows the L2 accesses for all the TB scheduling policies normalized to baseline LRR. In Fermi (Figure 3.13(top), TB policies MST-TS, k-way-TS and RB-TS lead to reduced L2 accesses of 78.4%, 61.4% and 56.7% for high inter-TB benchmarks; 95%, 91.88% and 90.22% for low inter-TB benchmarks; and 89.6%, 81.7% and 79.3% overall for all applications. In Pascal (Figure 3.13(middle)), TB policies MST-TS, k-way-TS and RB-TS lead to reduced L2 accesses of 79%, 67% and 51.59% for high inter-TB benchmarks; 94.2%, 92.6% and 89.1% for low inter-TB benchmarks; and 89.6%, 84.3% and 76.94% overall for different inter-TB locality applications. L2 transactions are reduced for Volta (Figure 3.13 (bottom)) , where MST-TS, k-way-TS and RB-TS reduce accesses by significant fraction of 76.4%, 65% and 59.8% for high inter-TB benchmarks;

90.9%, 85.5% and 85.3% for low inter-TB benchmarks; and 87%, 80.5% and 78.3% overall for all applications.

It is noteworthy that in both k-way partitioning (k-way-TS) and recursive bipartitioning (RB-TS), in cases where the number of TBs are less than the total SM capacity, i.e. total TBs that all SMs can hold, all thread blocks are assigned to their respective SMs immediately after the kernel's initialization. This means that both schedulers would perform the same. Locality captured in each partition is compared k-way-TS and RB-TS. Locality is expressed in-terms of the sum of the edge weights of the sub-graph in the partition. The more is the edge weight of each partition, more is the locality captured. In k-way the partition size is huge and the concurrently running TB might not necessarily have the maximum sharing within the partition. How-ever in RB-TS we reach a sweet-spot where all the concurrently running TB i.e. all the TB inside a partition are likely to have maximum sharing. This is why we observe lower L1 miss rates in RB-TS as compared to k-way-TS.

Comparison with cache-fit graph partitioning policy

A prior work [27] has employed cache-fit policy on SPMV application using edge-partitioning and kernel-splitting. To perform a comparison, we evaluated SPMV application in Parboil benchmark suite for cache-fit policy using GPGPU-Sim. Through benchmarking of the SPMV execution, the TB size was found out to be 15 KB. Since the default L1 size in Fermi is 16KB, 1 TB is mapped to the SM in the cache-fit policy. In our policy, we had employed 5 TBs based on the register and threads usage per TB. Similarly, the number of TBs per SM was reduced to 3 and 2 for Pascal and Volta architectures so that the working set of the concurrent TBs fits into the L1 cache. Overall, we observed that the normalized

speedup is reduced in the cache-fit policy compared to the baseline LRR policy. They are 58% (Fermi), 91.24% (Volta) of the baseline. Reducing the number of concurrent TBs in the SM in cache-fit policy helps reduce the pressure on L1 cache, resulting in lower L1 cache misses. However, it also results in the under utilization of a SM and starvation of the other SM resources resulting in increased execution time. In Pascal, the number of TBs mapped to SM is set to 3 due to a larger L1 size, resulting in a speed-up of 109.73% over baseline. However, it may be reminded that the RB-TS policy in PAVER is even better than the baseline LRR by 99.4% (Fermi), 104.94% (Pascal) and 114.9% (Volta). In cache-fit policy, the normalised L1 miss rate is reduced compared to the baseline, 42.72% (Fermi), 87.32% (Pascal), 100.6% (Volta) of the baseline, similarly, in RB-TS, the normalized L1 miss rate is reduced compared to the baseline, 86.18% (Fermi), 94% (Pascal), 99% (Volta) of the baseline. However, the L1 miss-rates are reduced in our RB-TS for all architectures without under-utilizing the other SM resources, resulting in a high speedup.

3.6 Conclusion

In this chapter, we have shown that the inter-TB locality can be exploited to improve the performance substantially. Unstructured parallelism is a part of many GPU applications today and newer architectures should take advantage of the data locality among the thread blocks. To this end, we performed compiler analysis to measure the data reference sharing among TBs for various applications. Then we proposed three generalized graph-based TB scheduling policies based on MST, k-way partitioning, and recursive bipartitioning. Our scheduling techniques reduce L2 accesses by 43.3%, 48.5%, 40.21% and increase the average

performance speedup by 29%, 49.1% and 41.2% . We believe the results can be further improved by taking the time of data reference sharing into consideration.

Chapter 4

LocalityGuru: A PTX Analyzer for Extracting Thread Block-level Locality in GPGPUs

In this chapter, we present LocalityGuru, a detailed static compiler analyzer to automatically extract the thread to index range relationship from the intermediate representation (IR) of the source code (in PTX format). This chapter makes the following contributions:

- We analyze the PTX code at JIT compilation time before the kernel launch and perform the detailed static index analysis to derive the equation for the thread/TB mapping to data element indices accessed.

- We validate the results of the TB locality graph obtained through automated LocalityGuru PTX analyzer by comparing with the profiling data-locality results. Our approach imposes *zero timing overhead* on the kernel execution time.

The chapter is organized as follows: Section 4.1 describes the PTX analysis background. Section 4.2 discusses the process of constructing the syntax trees and using them to extract the locality behaviour of the kernel at the Thread-Block (TB) level. Our results are discussed in Section 4.3. We conclude in Section 4.4.

4.1 Background

Static analysis for a GPU application has to take into account the parallel nature of thread block executions and data accesses. Otherwise, it is little different from a static analysis meant for a sequential application.

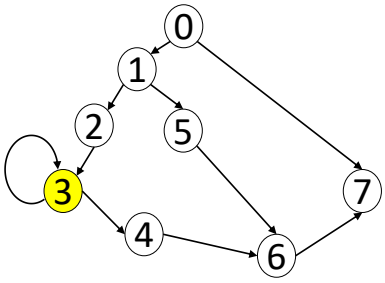

Figure 4.1: Control flow graph of the PTX basic blocks (bb) for matrix multiplication. Basic block 3 (highlighted) contains the `ld.global` instructions and has a self-loop.

Target architecture-independent PTX programs have an assembly-language-style syntax with instruction operation codes (opcodes) and operands. A common way of

representing PTX code is through the use of control flow graphs (CFGs). Example of CFG for matrix multiplication is shown in Figure 4.1. Nodes of a CFG represent basic blocks, which are sequences of instructions without any control flow statements in between. Edges are directed from a source basic block to a target basic block showing the possibility for control to jump from the source block to the target block. There can be up to two possible edges from a given basic block (in case of a conditional jump, leading to a *true block* and a *false block*.) A great deal of information can be derived from the CFG structure, including the registers accessing a certain memory block, their order, and sometimes the frequency, of those accesses. In this work, we aim to extract this information before runtime in order to help with the GPU's cache management and locality-aware task scheduling.

4.2 LocalityGuru

This section discusses a PTX analysis based approach to determine the locality of threads, warps and TBs at just-in-time (JIT) compilation time using syntax trees.

4.2.1 PTX Analysis with Syntax Trees

Abstract syntax trees (referred to as *syntax trees* in the paper) are used to represent the syntactic structure of PTX code. The trees of the programming constructs like arithmetic and logical expressions, and control flow statements are grouped into operators (root nodes) and operands (leaves of the root nodes). For example, the syntax for instructions in PTX are as follows: $opcode.type \quad dst, src1, src2[, src3]$, where *.type* represents the type of the

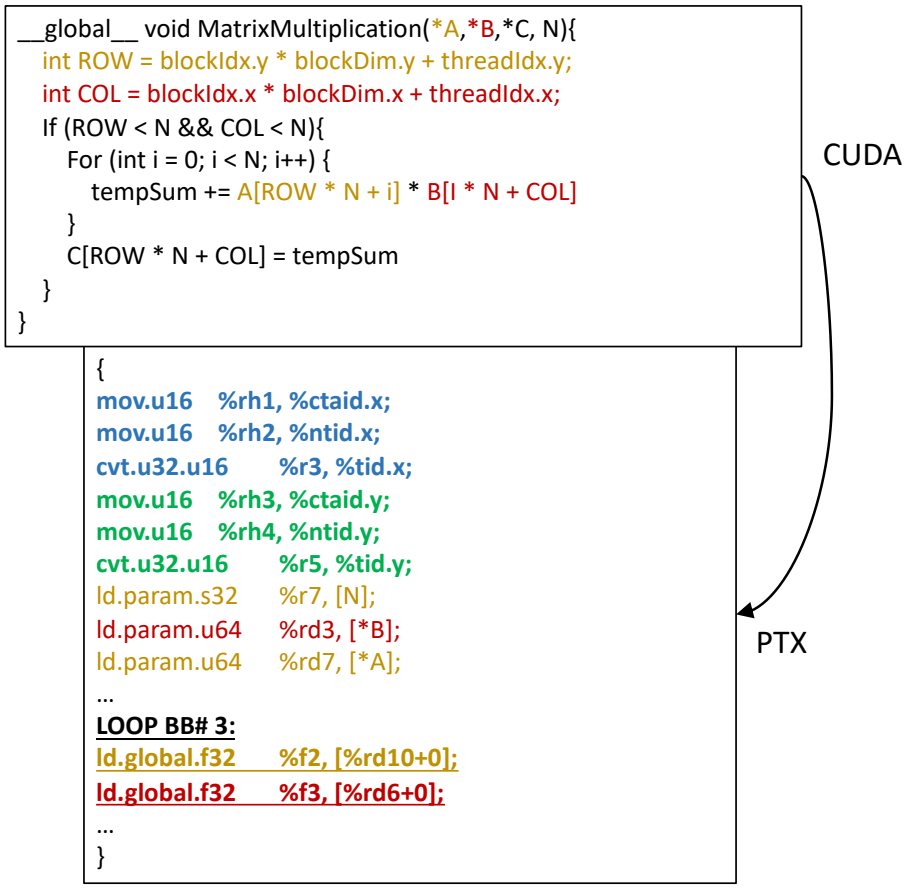

Figure 4.2: CUDA source code and PTX IR for for Matrix Multiplication Application

source operands. ($.type \in \{.u16, .u32, .u64, .s16, .s32, .s64\}$). The syntax tree expression for the instruction will have the opcode as the root node, destination operand (dst) as the result, and source operands 1 and 2 ($src1$ and $src2$) as the left and right child nodes respectively. Note that PTX also supports a third source operand for some operations, such as multiply-and-add (`mad_op`), in which $dst = (src1 * src2) + src3$. The construction of the syntax trees is explained in Section 4.2.2.

Figure 4.2 shows the example of the translation of the CUDA source code into the PTX intermediate representation in the matrix multiplication application. The CUDA

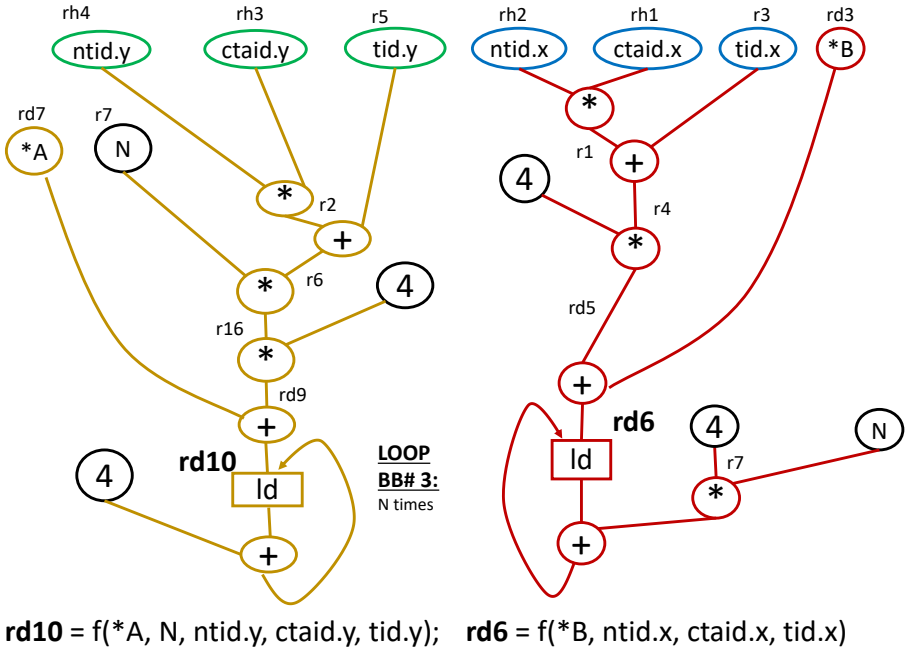

Figure 4.3: Abstract Syntax for Matrix Multiplication Application

source code is converted to its PTX representation during the offline compilation phase, and the kernel parameters are ready just before the kernel launch during the JIT compilation. After the kernel arguments are known, PTX is converted to architecture-specific SASS for execution. The advantage of doing locality analysis at PTX instead of SASS is that PTX is hardware agnostic and has added information from the kernel parameters. LocalityGuru works at the JIT compilation time to extract the thread-level address-data footprint using the abstract syntax tree. In some applications, the kernel parameters such as the `GridDim`, `BlockDim` and input data size `N` are dependent on the user input and are determined only at JIT-compilation time. In matrix multiplication, the exact values of the matrix size `N` and the base addresses of input/output matrices `A`, `B`, `C` are known after `malloc` in GPU. When the grid size and TB size become available, so do the ranges of the existing thread-specific

Algorithm 3 Syntax Trees for global load registers

```
// ST:Syntax Tree
function BuildSyntaxTree(I, ST)
    if dst(I) in S then
        node ← leaf_node_in_ST
        node.opcode ← I.opcode
        Erase dst(I) from S
        node.left ← create_new_node(I.src1)
        node.right ← create_new_node(I.src2)
        node.third ← create_new_node(I.src3)
    end
function TraceBasicBlocks(BB, ST)
    if BB has predecessors then
        for BB_j in BB_predecessors do
            for I in BB_j_inst_reversed do
                BuildSyntaxTree(I, ST)
            end
            TraceBasicBlocks(BB_j, ST)
        end
    end
// The main function
function GetAllSyntaxTrees(PTX_instructions)
    for I in PTX_instructions do
        if I.opcode is ld.global then
            // ld_reg = src(I)
            // Initialize syntax tree for load register:  ld_reg
            ST ← create_new_node(ld_reg)
            Insert ld_reg into S  for I_i in BB_inst_reversed do
                // BB is the basic block containing global load Instruction
                BuildSyntaxTree(I_i, ST)
            end
            TraceBasicBlocks(BB, ST)
        end
    end
    return ST
```

values, namely `threadIdx` and `blockIdx`. The load address range accessed per thread is stored in the source operand of the global memory read instructions, e.g., `ld.global`. The control flow graph (CFG) derived from matrix multiplication's PTX is shown in Figure 4.1. We start from the basic block with the `ld.global` instruction (BB 3) and recursively parse the instructions in all its predecessors until the leaf nodes consist of immediate values and kernel parameters only.

Some of the corresponding codes in PTX (Figure 4.2) and syntax tree (Figure 4.3) are highlighted using matching colors. The statements used for calculating the syntax tree of matrix A are in brown color and the matrix B are in red color across CUDA , PTX as well as syntax tree edge and node color annotations. After locating the global memory access instructions, an abstract syntax graph (ASG) is constructed for each `ld.global` source register (`rd10` and `rd6` from Figure 4.3) to identify which elements of the arrays (`A` and `B`) are accessed by each thread, thereby determining the value range of the memory accesses per thread/TB. In the ASG, the leaf nodes are the registers storing the values known at kernel-launch time, such as `&A`, `&B`, `N`, `GridDim` and `BlockDim`. Using the syntax tree, `rd10` is expressed as a function of `&A`, `N`, `ntid.y` (same as `BlockDim.y`), `ctaid.y` and `tid.y` i.e. $\texttt{rd10} = \texttt{\&A} + [ROW * \texttt{N} + i]$ in CUDA. The kernel parameters in `.x` and `.y` dimension are in blue and green respectively in the PTX (Figure 4.2) and comprise of the leaf nodes in syntax tree (Figure 4.3) of `rd6` and `rd10` respectively.

Using this, we can obtain the common set of elements in each matrix accessed by every TB, e.g., A_i, for TB_i, etc. Therefore, the number of common data elements accessed by TB_i and TB_j in the locality graph (Figure 4.4 - matrix multiplication) will be:

$$L(TB_i, TB_j) = |A_i \cap A_j| + |B_i \cap B_j|$$

4.2.2 Syntax Tree Construction

Algorithm 3 shows the pseudo-code of the syntax tree for global load source registers. We locate the global load source registers in the PTX code by tracing the

Algorithm 4 Locality Graph from Syntax Tree

```
// ST:Syntax Tree
// ST_loop:Syntax Tree for loop_variable
// L:Locality Graph for TB
// Map:  TB_Address_map
// reg:  global load register
1  function EvalST_loop (ST, Map)
2      if loop in reg.BB then
3          while EvalST(BB_label) != 0 do
4              Update loop_iterator in ST(BB_label)
5              Update leaf_node reg in ST_loop(reg)
6              Map[TB].insert(EvalST(reg))
7          end
8      end
9  function GetTBAddressMap(ST)
       // Assign kernel parameters to leaf nodes in ST
10     ntid ← BlockDim forall ctaid in GridDim do
11         TB ← get_tb_id(ctaid, GridDim)
12         forall tid in BlockDim do
               // Filter out idle threads in False branch BB
13             for P not in BB_False_branch do
                   // P: Predicate
14                 if !  EvalST(P) then
15                     forall offset(reg) do
16                         Insert (EvalST(reg) + offset) into Map[TB] // BB with loops
17                         EvalST_loop(ST, Map)
18                     end
19                 end
20             end
21         end
22     end
23     return Map
   // The main function
24 function GetLocalityGraph()
25     Map ← GetTBAddressMap(ST)
26     forall ctaid in GridDim do
27         if TB_i != TB_j then
28             L[TB_i][TB_j] = Map[TB_i] ∩ Map[TB_j]
29         end
30     end
31     return L
```

`ld.global` instructions and constructing a syntax tree (ST) for each (lines 19-29). For each instruction I_i in the basic block containing register *ld_reg*, `BuildSyntaxTree()` looks back for the instruction with *ld_reg* as its result. A node is then created for I_i and inserted into ST (lines 1-9). After iterating through a BB, all its predecessors are iterated recursively

(with no loops) to trace the source of the register using `TraceBasicBlocks()`, and the tree is updated accordingly (lines 10-18).

Handling Branches

Every basic block having more than one successor ends with a branch instruction. The true branch leads to BB 3, while the false branch avoids it. A conditional branch instruction in PTX uses a predicate to dictate whether it should execute the following basic block. During the locality graph construction shown in Algorithm 4 (Line 13), if the predicate value points to the false branch for a thread, it is removed from the locality calculations.

In many cases, not all the threads are assigned a task due to the input data size not being a multiple of number of threads, resulting in some threads in the last row or column of the grid idling throughout the kernel (*idle threads*). Thus, while generating the locality graph, the idle threads need to be filtered out in the analysis. In our matrix multiplication example, the threads executing BB 5 and BB 7 are idle threads (Figure 4.1). In order to determine whether to skip idle threads, we should build syntax trees for the predicates in the predecessor basic blocks to identify the threads executing the false branch basic blocks.

Handling Loops

In our matrix multiplication example, BB 3 has a self-loop (i.e. a branch in which the predecessor is same as successor) which points back to its beginning (Figure 4.1). The loop has an iterator which is incremented by 1 in every loop iteration and ends the loop when it reaches the value of `N`. In every loop iteration, `rd6` and `rd10` represent `&A` $+ [ROW * \mathtt{N} + i]$

and $\&B + [i * N + COL]$ respectively where i is the loop iteration count. A separate syntax tree (ST_{loop}) is constructed per loop variable which captures the formula for updating the loop variables in each iteration. We evaluate the syntax tree for the loop as in Algorithm 4 $EvalST_{loop}()$ (Line 1), where the leaf nodes of the syntax tree are updated with the values of the loop variables in the previous iteration, and then evaluate the syntax tree result for the predicate at the end of the loop to decide whether to continue to the next loop iteration by branching to its start, or exit the loop, as shown in Algorithm 4 (Line 3).

Handling Address offsets and data-hazards in the source register

Multiple global load instructions may use the same source register with different address offsets. In this case, we look for any data hazard due to register write into that register between the two instructions. If there is a data hazard, then separate syntax trees are constructed for it. Otherwise, we construct only one syntax tree, and store the register name and address offset in a map. During the syntax tree evaluation phase in Algorithm 4 (Line 15), the address offset is added to the calculated value of the syntax tree for the register.

4.2.3 Locality Graph from Syntax Tree

Algorithm 4 shows the pseudo-code of obtaining the locality graph from the syntax tree. The STs constructed in Algorithm 3 are used to map thread block ID to memory addresses it has accessed ($TB\_Address\_map$) (lines 9-23). Since we already represented the global load source registers in terms of kernel parameters in the syntax tree, a set of memory addresses can be calculated for every thread ID and TB ID. All the cases of *idle threads*,

address offsets in the source register, and loops are taken into account while evaluating the syntax tree to get the address range as described in Section 4.2.2. Once $TB\_Address\_map$ is populated with values for all TBs and all global source registers, the locality graph L is constructed for each kernel by intersecting the read memory address sets of TB pairs (lines 24-31).

The syntax trees are constructed per PTX file. However, since each kernel can have different program body and kernel parameters, the locality graph is constructed per kernel which also accounts for the changed input data per kernel.

4.2.4 Summary

The syntax tree derives the thread-to-memory-addresses-accessed relationship in terms of thread ID, block ID and other kernel parameters. This information can be used to capture inter-thread, inter-warp, inter-TB locality within the same kernel as well as across multiple kernels. Though we have shown an example to capture the locality due to memory read of the same data, LocalityGuru can also be used for data-dependency analysis among TBs in multiple kernels. In that case, each TB shall have a $TB\_Address\_map$ for both read and write accesses. This inter-kernel TB-level data dependency (or producer-consumer relationships) can be used for hardware optimization techniques like TB scheduling as discussed in [4, 51, 148]. At the intra-kernel level, LocalityGuru can aid in the optimization techniques proposed in [5, 25, 81, 87, 139, 154].

The applications evaluated in this chapter (Section 4.3) contain thread ID-dependent accesses. Our technique can also be extended to be used for the indirect accesses where the result of one memory access (thread ID-dependent primary access) is used to calculate the

address of the next memory access (secondary access). The primary data structure can be passed to the PTX analyzer as an argument to extract the access patterns [118, 124].

4.3 Results and Discussion

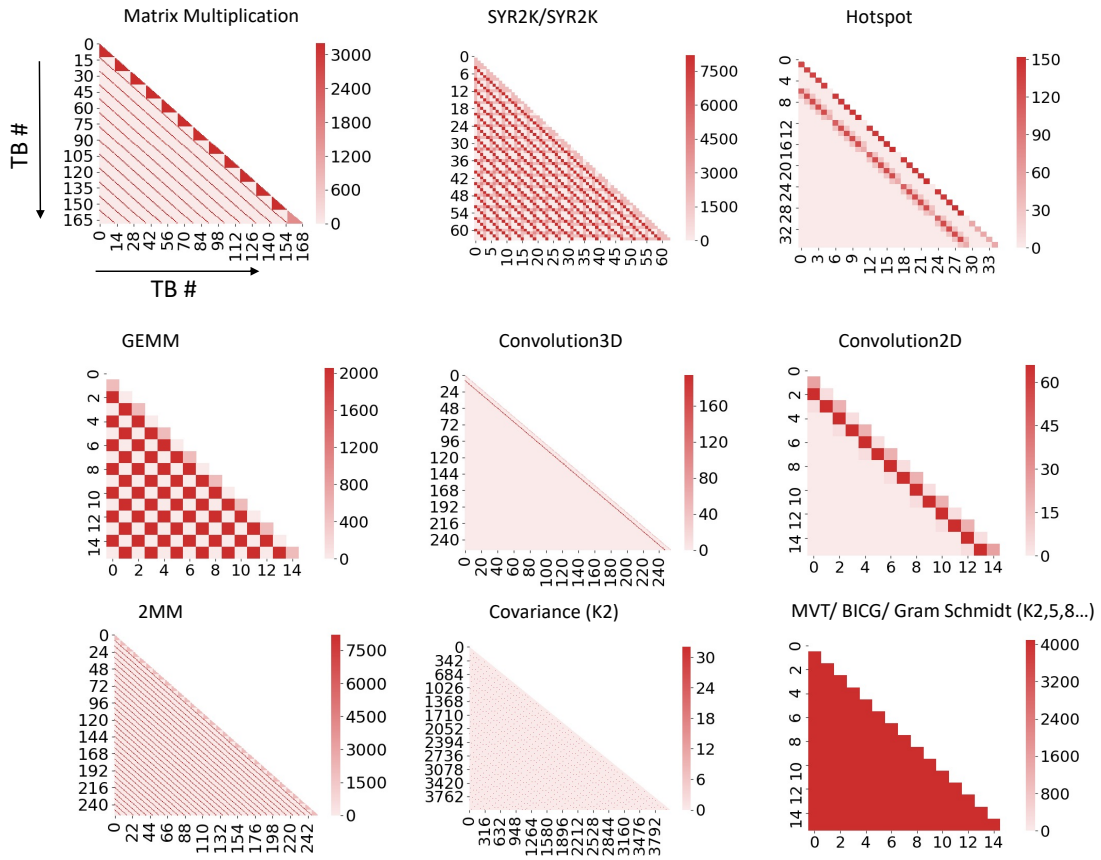

Figure 4.4: TB locality graph results for different applications. The numbers in brackets represent the Kernel #. The adjacency matrix representation of locality graph is symmetric and has been shown as a lower triangular matrix. The TB # are shown in the x axis (increasing order) and y axis (decreasing order). The number of common memory addresses accessed by any two TBs is shown as the red color intensity in the heatmap. The more intense red color refers to more data-locality among the TBs.

4.3.1 Methodology

We implement and perform the PTX analysis for thread-level address footprint using the built-in PTX parser in GPGPU-Sim [13]. Our analysis algorithm is generic and can be implemented in any compiler framework that supports PTX, such as LLVM [78] and GPUOcelot [36]. It is fully automated to do the locality analysis on any unknown application in CUDA or OpenCL which generates PTX. In our experiments, 12 benchmarks used were selected from 3 different benchmark suites NVIDIA CUDA SDK [100] (Matrix Multiplication), Rodinia [24] (Hotspot) and Polybench [116] (SYRK, SYR2K, 2MM, GEMM, BICG, Covariance, Convolution 3D/2D, MVT, Gram-Schmidt).

4.3.2 Understanding the Patterns

The terminology used in our discussion is as follows:

bx: `blockIdx.x`, by: `blockIdx.y`,

tx: `threadIdx.x`, ty: `threadIdx.y`,

$index_x$: (bx * `blockDim.x` + tx),

$index_y$: (by * `blockDim.y` + ty)

If a global load address is only dependent on $index_x$ *or* $index_y$, it is referred to as an $index_x$-only or $index_y$-only pattern here. When the address is dependent on $index_x$ *and* $index_y$, it is referred to as $index_{xy}$ pattern. When the address is dependent on $c_x index_x + c_y index_y$, it is referred to as $index_{x+y}$ pattern, where c_i, c_j are constants.

In matrix multiplication, for input size of 200, `GridDim`=13x13 and `BlockDim`=16x16, the accesses to input matrices are `A`$[index_y * 200 + i]$ and `B`$[i * 200 + index_x]$ where $i \in \{0, ..., 199\}$. Therefore 13 consecutive TBs in the same column of the TB grid share $16 \times 200 = 3200$ data elements (shown as the triangles in the diagonal of the matrix multiplication in Figure 4.4). Similarly, the TBs in the same row of the TB grid share 3200 data elements, which is shown as accesses by a strided distance of 13 (i.e. `GridDim.x`) in the figure.

In 2MM, for input size `N` of 256, `GridDim`=8x32 and `BlockDim`=32x8, the read accesses to matrices are `A`$[index_y * \mathtt{N} + (0...\mathtt{N}-1)]$, `B`$[(0...\mathtt{N}-1) * \mathtt{N} + index_x]$ and `C`$[index_x * \mathtt{N} + index_y]$. The TBs in the same row of the TB grid exhibit $index_y$-only pattern and share `blockDim.y` $* \mathtt{N} = 2048$ elements and in the same column exhibit $index_x$-only pattern and share `blockDim.x` $* \mathtt{N} = 8192$ elements. No sharing is observed for matrix `C` which has $index_{x+y}$.

Similarly, in GEMM, for input size `N` of 64, `GridDim`=2x8 and `BlockDim`=32x8, as the matrices `A` and `B` have similar access patterns as 2MM and matrix multiplication, row-wise sharing among the TBs is `blockDim.y` $* \mathtt{N} = 512$ elements and in the same column share `blockDim.x` $* \mathtt{N} = 2048$ elements.

In case of Gram-Schmidt, kernels 2/5/8 have `GridDim`=4 and `BlockDim`=256, and accesses input matrices `a`$[index_x * \mathtt{N} + \mathtt{k}]$ and `r`$[\mathtt{k} * \mathtt{N} + \mathtt{k}]$ where `k` is a kernel paremeter. Here as matrix `r` indices are not a function of bx or by, it would be accessed by all the TBs and an $index_x$-only pattern of sharing would be observed. Now since the TBs are arranged in a 1D grid, we observe that the locality graph is fully connected i.e. every TB is connected to rest

of the TBs in the kernel. Similarly, MVT has 1D kernel and accesses matrices $\texttt{x1}[index_x]$, $\texttt{a}[index_x * \texttt{N} + 0...\texttt{N} - 1]$ and $\texttt{y}_1[0...\texttt{N}]$. $index_x$-only pattern sharing in a 1D Kernel (along the x-axis) in applications like Gram-Schmidt, MVT and BICG results in fully connected locality graph.

In case of SYR2K, for input size N of 256, GridDim=8x32 and BlockDim=32x8, the read accesses to matrices are $\texttt{A}[index_y * \texttt{N} + i]$, $\texttt{A}[index_x * \texttt{N} + i]$, $\texttt{B}[index_x * \texttt{N} + i]$, $\texttt{B}[index_y * \texttt{N} + i]$ and $\texttt{C}[index_x * \texttt{N} + index_y]$, where $i \in \{0, ..., \texttt{N} - 1\}$. Here, the sharing pattern for both the matrices A and B is $index_{xy}$. No sharing is observed for matrix C. Since there are some common elements from $index_x \cap index_y$, we observe a different pattern for $index_{xy}$ as compared to $index_x$ only or $index_y$ only.

In case of Convolution2D, the kernel is 2D with GridDim=2x8 and BlockDim=32x8 and the read accesses to matrices are $\texttt{A}[(index_x + i) * \texttt{N} + index_y + j]$ where $i, j \in \{-1, 0, 1\}$. It has a stencil computation behavior where every pixel computation involves read accesses to all its neighboring pixels in xy-plane. Hence we observe that every TB shares elements with its neighboring TBs in both dimensions and has $index_{x+y}$ pattern.

In Covariance, kernel 2 is 2D with GridDim=64x64 and BlockDim=32x8, the read accesses to matrices are $\texttt{mean}[index_x + 1]$ and $\texttt{data}[index_y * (\texttt{N} + 1) + index_x]$, where $i \in \{0, ..., \texttt{N}\}$. We observe the TB-to-data mapping is in column-major order, hence the slope is reversed compared to the row-wise data mapping locality pattern seen in MM, GEMM and 2MM.

4.3.3 Validating the Results

In order to validate the results of LocalityGuru, the locality graphs were constructed for all kernels using profiling by running benchmarks in GPGPU-Sim and recording the memory addresses accessed by each TB. We perform an element-wise comparison between the resulting locality graph from LocalityGuru and the simulator's generated locality graph. No differences were noted.

4.4 Conclusion

In this paper, we aim to derive the relationship between the thread blocks and the memory addresses accessed by them. A detailed compiler analysis is performed on the PTX intermediate representation using syntax trees to extract the data locality in terms of the number of common data elements shared between all thread block pairs in a kernel. Our locality analysis technique can be employed at multiple granularities such thread-, warp- or TB-level in a GPU kernel. This information can be leveraged to help make optimizations for locality-aware data-partition, memory page data placement, prefetching, cache management and TB as well as warp scheduling in single or multi GPUs.

Chapter 5

Snooze: Cache Leakage Energy Management in GPGPUs

In this chapter we describe the cache static power management technique which leads to leakage energy savings in GPU caches when they are idle during the workload execution. Our technique makes the following contributions:

1. The cache state transition latency for wake-up and sleep are hidden micro architecturally by predicting when the next access, hence there is negligible performance penalty for employing the undervolting states.

2. The state retentive drowsy mode keeps the contents of the cache safe for the next access. Though the functionalities like cache read and writes cannot be performed in the drowsy mode, but the data stored in the caches is not lost.

3. The gating logic is connected to a 32 Bytes cache line, so that break even time is very short (2 cycles). The caches can be undervolted even for the short idle periods greater than 2 cycles, thereby leading to maximum leakage energy savings.

4. The caches are undervolted at various granularities like cache bank-level and cache line-level and it is observed that the line-level gating yields significant leakage energy savings compared to the bank-level gating as a cache line has longer idle period compared to cache bank on a average.

The chapter is organized as follows: Section 5.1 describes GPGPU cache background and provides motivation for this work. Section 5.2 discusses the proposed power-management technique for the GPU caches including the power mode transition and hardware support. Our results are discussed in Section 5.3 and we conclude in Section 5.4.

5.1 Background and Motivation

For better performance, the fast high-leakage transistors are being used, on the other hand, the cost issues favour energy efficient designs. In this era of green computing, the energy efficient processor design is gaining importance. The modern processors are using increasingly large sized SRAM structures like caches and register files. With each CMOS technology generation, there has been a increase in the the leakage energy consumption of these structures as shown in Figure 5.1. The subthreshold leakage power is a major issue for all transistors, but it is a critical problem for the on-chip caches which are a growing fraction with the recent GPU generations. The leakage power is dominating the dynamic power consumption of the circuit as the gate length reduces. In the newer GPUs, the number of

streaming multiprocessor (SM) are increasing, thereby increasing the cumulative L1D cache size and L2 cache size to meet the memory demands of the increasing concurrently executing threads. Figure 5.2 shows the average power consumed in GPU for various applications. L2 caches consume around 4.5% of the total GPU power and the L1 caches have been categorized in "Other" section and comprise of 7.2% along with the other components such as shared memory, load store unit etc.

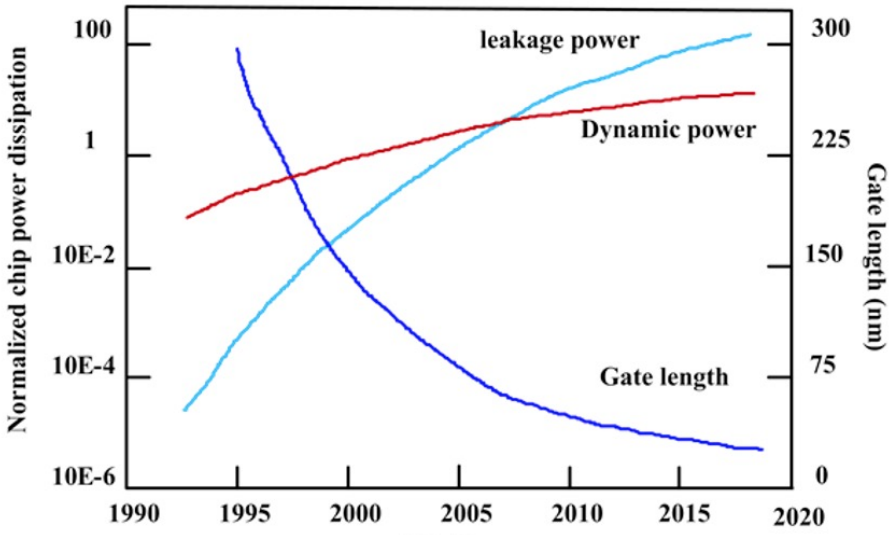

Figure 5.1: With the reduction in the feature size in the recent technology nodes, the leakage power dominates the dynamic power [19]

Table 5.1 shows the configuration of L1 and L2 caches in the GPUs. The L1 data cache is a private, per-SM, non-blocking non-coherent first level cache for memory accesses. The L1 cache is not banked and is able to service two coalesced memory request per SM core cycle. An incoming memory request to L1 data cache is 128 Bytes or smaller. L1D access latency is one cycle and it write hits lead to eviction of the block, write misses are

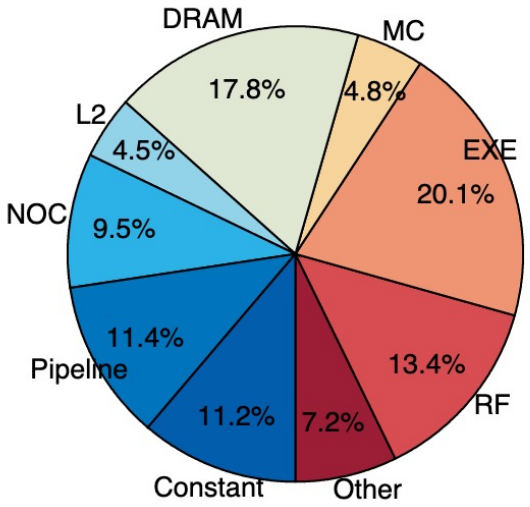

Figure 5.2: Average Power Consumption for GTX 480 [86]

Table 5.1: GPU cache configuration in different architectures

Caches	Fermi	Pascal	Volta
L1D size per SM	16 KB	48 KB	64 KB
Number of L1D per GPU	15	28	80
L1D cache line size	128 bytes	128 bytes	128 bytes
L1 configuration	32 sets, 4-way	64 sets, 6-way	64 sets, 8-way
L2 size	768 KB	3 MB	4.5 MB
Number of L2 Banks per GPU	12	24	24
L2 cache line size	128 bytes	128 bytes	128 bytes
L2 configuration	64 sets, 8-way	64 sets, 16-way	64 sets, 24-way

write no-allocate policy. Upon L1D miss, one miss request per SM cycle is inserted in to FIFO L1→Interconnect (ICNT) queue. L2 banks are accessed through the interconnection network. Each bank is connected to a memory sub-partition. The memory request packets in ICNT→L2 queue access the L2 bank. Upon L2 miss, the memory request is pushed to L2→DRAM queue.

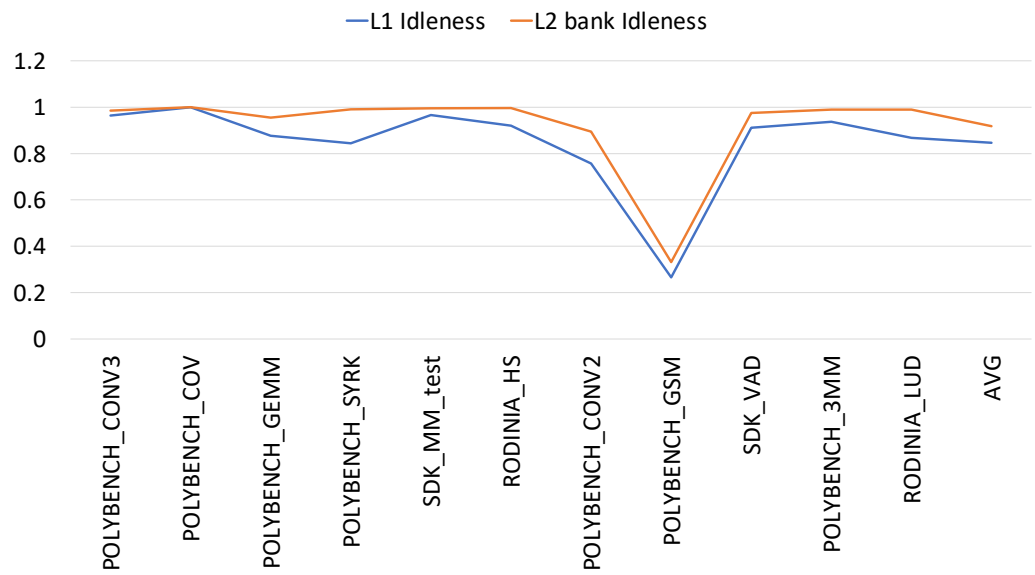

Figure 5.3: Idleness period normalized to the total execution time for L1 and L2 Bank

Table 5.2: Power management modes for 32 Bytes Cache line. [151]

State	Wakeup Delay (in clock cycles)	Static Power
ON	0	100%
Sleep	2	22.8%
OFF	2	1.5%

Figure 5.3 shows the Idleness period for the L1 and L2 banks for different applications. We profiled the accesses to the cache and recorded the time stamp for each access. The idleness is calculated as the cumulative sum of the reuse distance between each consecutive accesses to the cache, and then normalized to the total execution cycles. The idleness can be interpreted as the reciprocal of the utilization. It is evident from the figure that overall the caches are heavily underutilized and there is more scope of reducing the leakage energy consumption for the L2 banks compared to the L1 banks are they are less utilized.

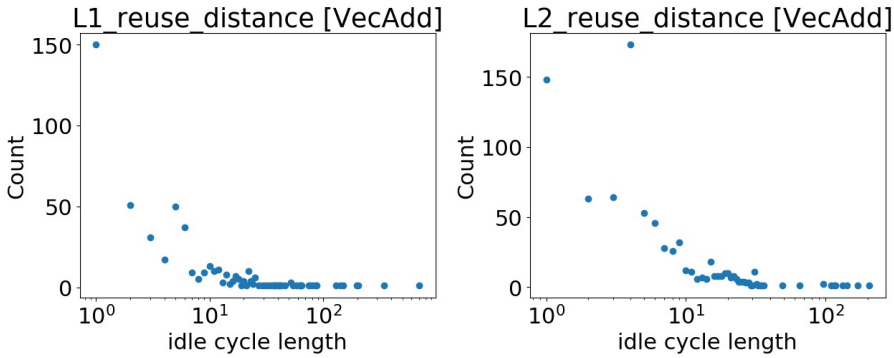

Figure 5.4: On the left figure the different colors show the numbers for different (a) L1 (SM 0....14) and (b) L2 banks (0...12) Observations: Most idle cycles are shorter than 10 cycles.

Table 5.2 shows the mode transistion latency from modes Sleep or OFF states to ON state. The transistion latency and associated leakage power characterization of the 32 Byte SRAM array is obtained from [151]. The latency of OFF $\rightarrow$ ON state is 1.954 ns. Assuming 1 GHz frequency, the OFF $\rightarrow$ ON latency is found to be 2 cycles. The leakage power consumed by the 32 Byte SRAM array at Sleep and OFF modes is 22.8% and 1.5% of the leakage power consumed at ON state. Figure 5.4 shows the reuse distance of (a) L1 caches and (b) L2 cache bank for Vector Add application. In general most frequently seen idle cycles are shorter than 10 cycles. As the wake-up delay from Sleep states to ON state is 2 cycles, mode transition of the caches when the the idle cycles less than 2 cycles can lead to added delay in accessing the caches. This shall eventually lead to performance degradation. Hence, we need to look up in the access queues for the caches and make sure that the caches are not transitioned to the sleep mode when there is a pending memory access in next 2 cycles.

5.2 Snooze Design

5.2.1 Hardware Support

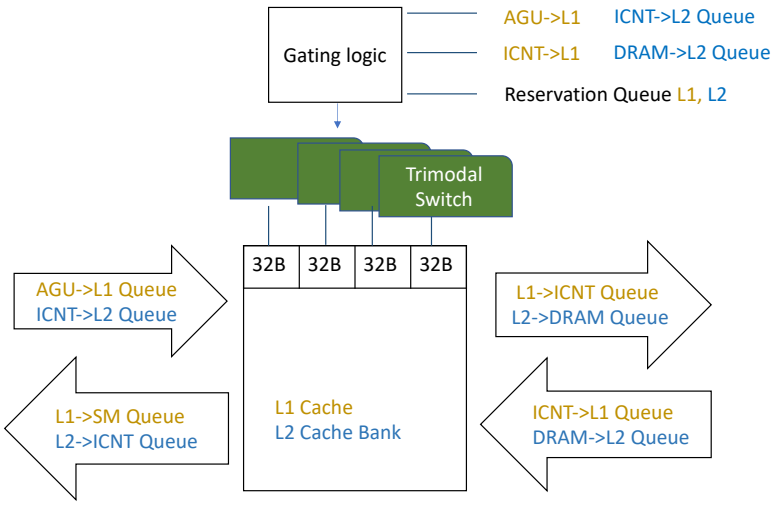

Figure 5.5: 32 Bytes Cache lines are connected to a trimodal switch having 3 states (ON, Sleep and OFF). The structures for L1 and L2 caches are color annotated in Brown and Blue respectively.

Figure 5.5 shows the hardware support for the Snooze in L1 cache and L2 bank. The power gating is enabled using the trimodal switch [111] and a gating logic. A trimodal switch provides 3 output voltages for OFF, ON and Sleep mode to a 32 Byte cache block based on the 2-bit control signal provided by the Gating logic [151]. The gating logic enables the voltage modes depending on the input queues Address Generation Unit (AGU) to L1, Interconnect (ICNT) to L1 and reservation queue for L1; and ICNT to L2, DRAM to L2, reservation queue for L2 respectively. The purpose of using reservation queue is to speculate the need to wake up the cache when there are memory requests waiting in the AGU→L1 and ICNT→L2 queue. As explained in section 5.2.2, the cases where there is are reservation

fails due to MSHR full or there are no replaceable blocks in the cache set because all block are reserved by prior pending requests, we need to keep track of the requests resulting in the fails. We use a 32 entry queue for storing the base addresses which resulted in the reservation fail status. In this case, the set of base address of AGU→L1 queue.top() or ICNT→L2 queue.top() is stored in the reservation queue and is erased later when the base address is filled up during ICNT→L1.pop() or DRAM→L2.pop() operation. This strategy helps using in avoiding the unnecessary wake up of the L1 cache or L2 banks even when AGU→L1 and ICNT→L2 queues are *not* empty. The total hardware overhead incurred by the Reservation Queue L1 per SM as well as each L2 bank is 6 bits for sets * 32 entries i.e. 24 Bytes. We use 6 bits for storing 64 sets in L1 and L2. The total hardware storage overhead per GPU shall be 24 Bytes * (15 L1 caches + 12 L2 Banks) = 648 Bytes i.e. around 0.05% of the total L1 and L2 size in the GPU.

5.2.2 Power Mode Transition

The Finite State Machine for the various power modes *ON*, *Sleep* and *OFF* for *Snooze* is shown in Figure 5.6. When the kernel starts execution, all the caches are *OFF* at ❶. When the memory load/store instruction is issued to the load-store unit, the Address Generation Unit (AGU) calculates the base address to be accessed by each thread and coalesces the accesses from multiple threads or warps and add them to AGU→L1 access queue. Once the memory request is added to the AGU→L1 queue, the L1 is transitioned to *ON* mode through ❷. The time lag between the request being added to the queue and the L1 access is around 3-5 cycles which is sufficient to hide the wake-up delay of 2 cycles in L1

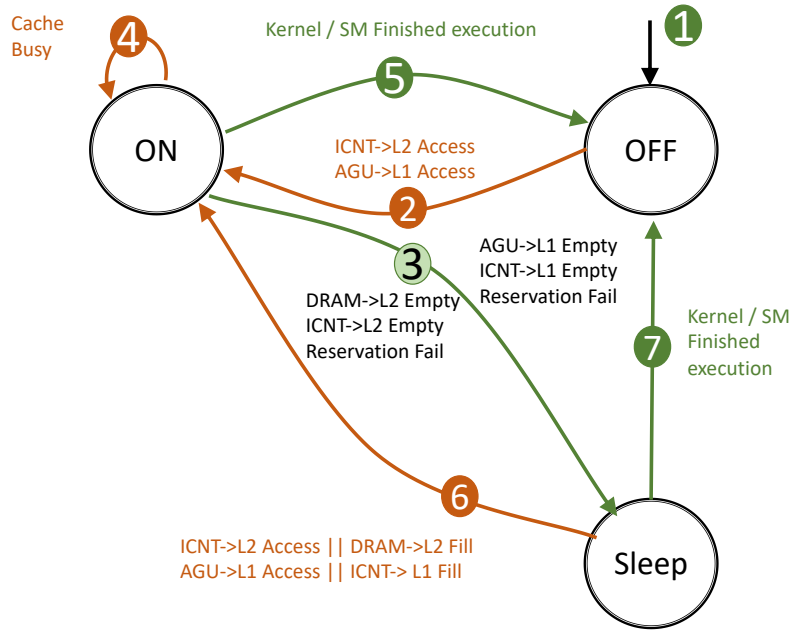

Figure 5.6: State Transition Diagram for L1 cache and L2 banks. 3 power states are ON, Sleep and OFF.

cache. As long as there is a request in AGU→L1 and the L1 cache is being accessed, it is busy serving the requests and stays in ❹. However, when the AGU→L1 and ICNT→L1 queues are empty, the L1 transitions to *Sleep* State through ❸. There are scenarios where the AGU→L1 is not empty, but still there are no accesses to L1 due to *reservation fail*. The reservation fail results if either the miss status holding register (MSHR) is full or there are no replaceable blocks in the cache set because all block are reserved by prior pending requests. If the L1 cache access results in a miss, the request reaches the L2 bank through the interconnection network which then passes on the request to ICNT→L2 Bank queue. A request in ICNT→L2 Bank wakes up the L2 cache through ❷. Once the L2 bank serves the request, it checks for the pending requests in ICNT→L2 and DRAM→L2 queue. If they are empty, it transitions to *Sleep* through ❸. Like in case of L1, the scenario when there

is reservation fail at L2 bank and the ICNT→L2 queue is not empty, the L2 switches to *Sleep* mode and stays there till a fill request is made to the bank. Incase of L2 miss, the request is pushed to L2→DRAM queue. Once a request reaches the DRAM→L2 queue, L2 cache wakeup for the fill request through ❻. The same mechanism happens at the L1 fill requests where the L1 wakes up when the request is pushed to ICNT→L1 queue. Towards the end of kernel execution, once all the TBs in the SM finish, the L1 caches (private) to SM are switched to *OFF* mode through ❺ and ❼. This happens when there is load imbalancing among the SMs. When the kernel exits the GPU, all the L2 banks are switched *OFF* through ❺ and ❼. This saves static energy for the caches waiting for the subsequent kernel to be launched to the GPU.

5.3 Results and Discussion

5.3.1 Methodology

The proposed power management technique for saving leakage energy was evaluated using GPGPU-Sim v3.2.1 [13] based on Fermi-like configuration with 15 SM. Each SM comprises of a private L1 cache. Each SM has two warp schedulers using greedy-then-oldest warp scheduling policy [121]. There are 6 DRAM memory channels and each channel has two sub-partitions, each sub-partition has a L2 bank. The cache block size for both L1 and L2 cache is 128 bytes. Each L1 cache size is 16 KB and L2 bank size is 64 KB.

For evaluating the effectiveness of the techniques on the recent state-of-art architectures, we evaluated SNOOZE on Pascal and Volta-like configuration (Table 5.1). In

our experiments, 20 benchmarks used were selected from 4 different benchmark suites ISPASS [13], Nvidia CUDA SDK [100], Rodinia [24] and Polybench [116].

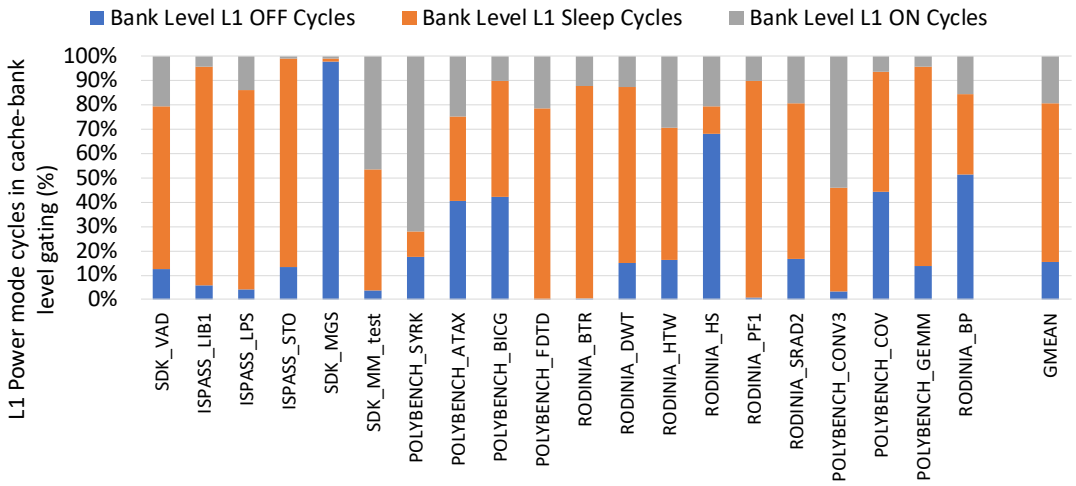

Figure 5.7: % distribution of the cycles in different states in L1 cache for cache bank-level gating in Fermi

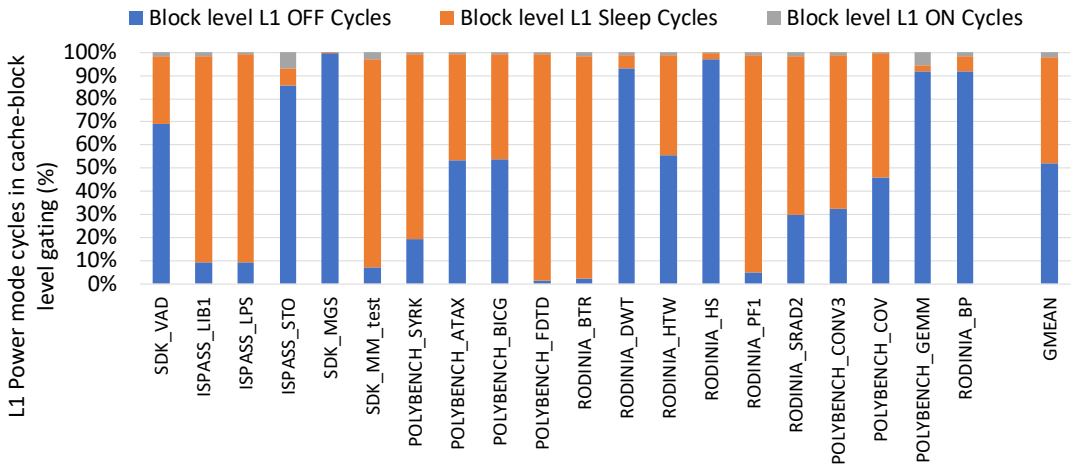

Figure 5.8: % distribution of the cycles in different states in L1 cache for cache block-level gating in Fermi

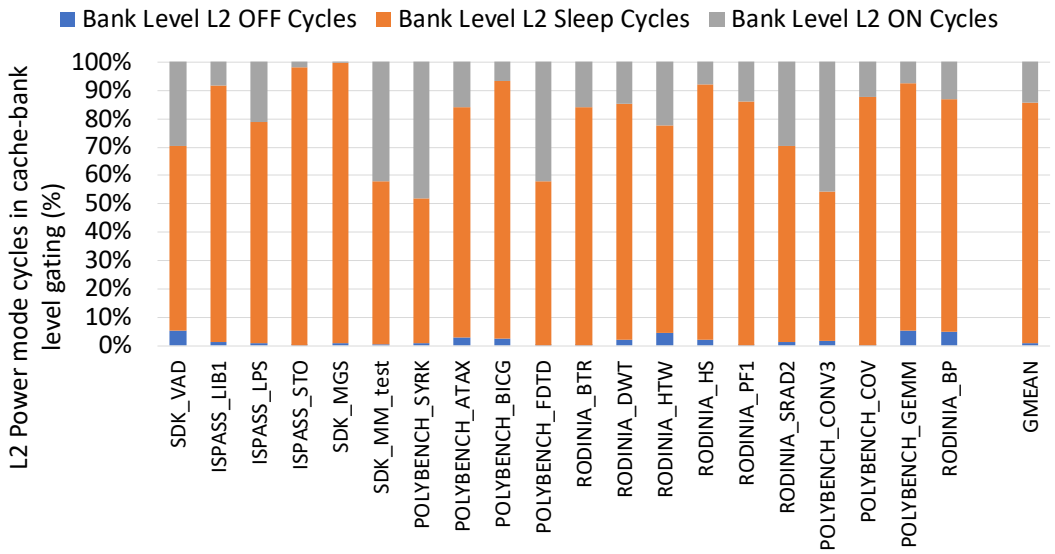

Figure 5.9: % distribution of the cycles in different states in L2 cache Bank for cache bank-level gating in Fermi

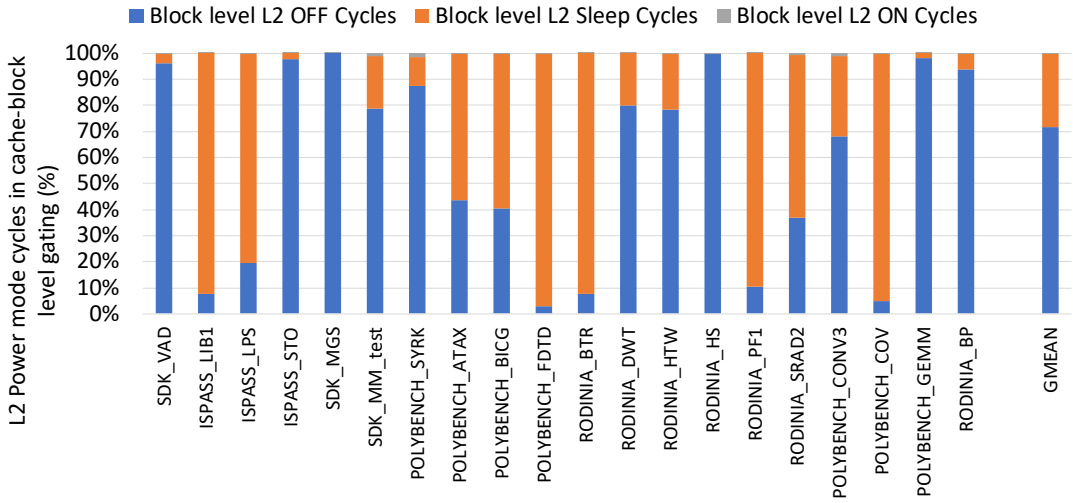

Figure 5.10: % distribution of the cycles in different states in L2 cache Bank for cache block-level gating in Fermi

5.3.2 Energy Savings

The % distribution of the cycles in different states (OFF, Sleep and ON) for L1 (bank-level and block-level) and L2 (bank-level and block-level) are shown for Fermi

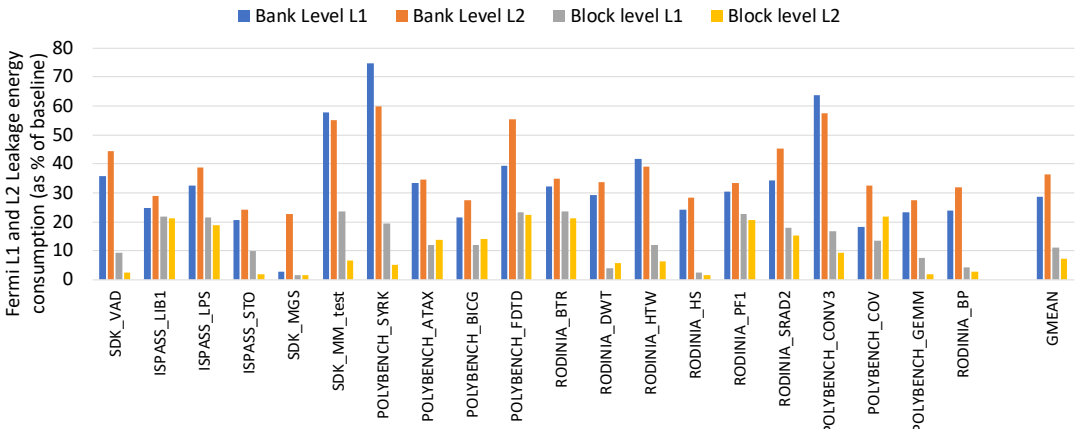

Figure 5.11: Fermi Leakage Energy Consumption for L1 and L2 caches using Cache bank-level vs block-level gating as a % of the default leakage energy consumed without any energy saving technique.

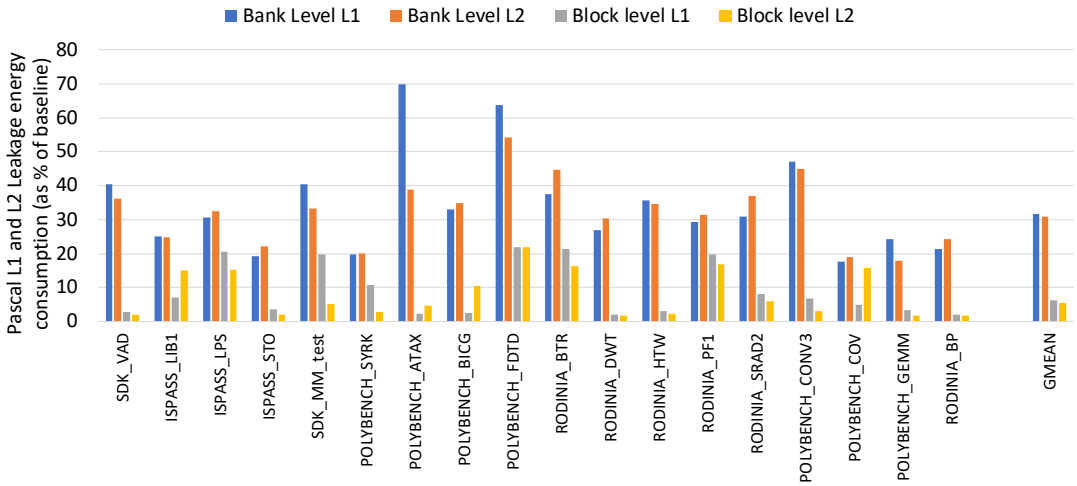

Figure 5.12: Pascal Leakage Energy Consumption for L1 and L2 caches using Cache bank-level vs block-level gating as a % of the default leakage energy consumed without any energy saving technique.

architecture in Figure 5.7, Figure 5.8, Figure 5.9 and Figure 5.10, respectively. The geometric mean distribution of OFF, Sleep and ON cycles for L1 bank-level gating is 9.95%, 42.61% and 12.50%, L1 block-level gating is 27.72%, 24.50% and 1.187%, L2 bank-level gating is 0.86%, 76.81% and 12.78% and L2 block-level gating is 37.49%, 14.85% and 0.08%. The percentage

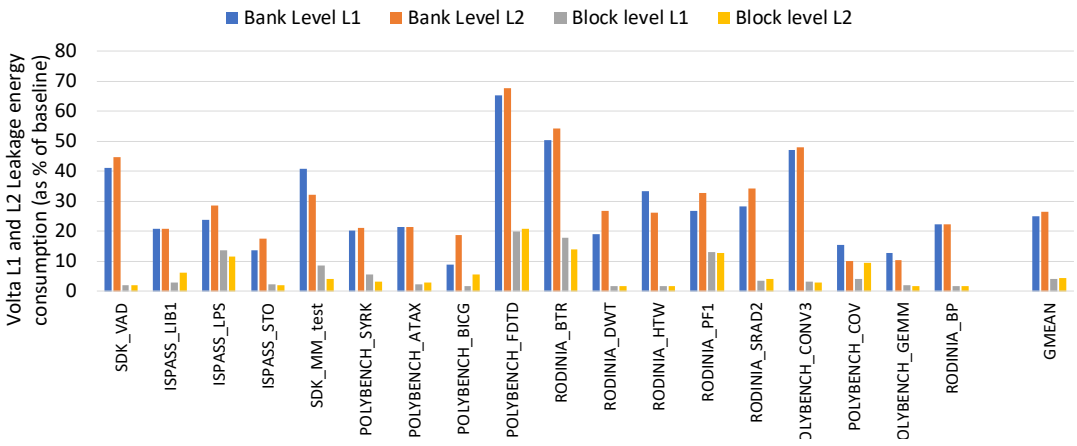

Figure 5.13: Volta Leakage Energy Consumption for L1 and L2 caches using Cache bank-level vs block-level gating as a % of the default leakage energy consumed without any energy saving technique.

distribution of cycles in different states affect the overall leakage energy consumption. In Fermi, the 16 KB L1 cache contains 128 cache blocks, where each block is 128 Bytes. In bank level gating, the L1 cache has to stay on when any of the 128 blocks is accessed. However, in case of block level gating, since each cache block is controlled independently, a given cache block wakes up only when it is about to be accessed, while the rest of the cache blocks in the L1 cache are in Sleep, OFF or ON mode as per the state of the next pending request. It is observed that both L1 and L2 block level gating had reduced overall ON cycles and increased Sleep and OFF cycles. This increase in OFF and Sleep cycles results in very low leakage energy consumption in case of block level gating as compared to bank level gating.

Figure 5.11, Figure 5.12 and Figure 5.13 show leakage energy consumption for L1 and L2 caches under bank and block-level gating techniques in Fermi, Pascal and Volta architectures. The cache configuration for different architectures is shown in Table 5.1. The leakage energy consumption is calculated as the cumulative sum of (number of cycles in a *State*) * (leakage power consumed in the State). In Fermi, the leakage energy consumption

in L1 is reduced to 28.56% for bank-level and 11.17% for block level gating, in L2 is reduced to 36.27% for bank-level and 7.31% for block level gating. In Pascal, the leakage energy consumption in L1 is reduced to 31.62% for bank-level and 6.21% for block level gating, in L2 is reduced to 30.87% for bank-level and 5.37% for block level gating. In Volta, the leakage energy consumption in L1 is reduced to 25.03% for bank-level and 4.03% for block level gating, in L2 is reduced to 26.49% for bank-level and 4.28% for block level gating. With the newer architectures, as the cache size increases, both the bank-level and block-level gating techniques result in more leakage energy savings. Comparing the bank-level and block-level gating results, block-level outperforms the bank-level gating in terms of leakage energy savings. This is because, the block level technique results in more number of OFF cycles (accounts for 1.5% of the ON mode leakage energy) compared to the bank level gating.

5.4 Conclusion

In this chapter, we propose SNOOZE, a technique to save the leakage power of L1 and L2 cache in GPU, based on power-gating and under-volting. The three power modes for cache are designed to derive static energy savings benefit during different occasions during GPU processing. Our micro-architectural technique for waking up the caches by observing the pending requests queue successfully hides the latency and ensures no significant negative impact on performance. Our results show that the idle cycles of L1 and L2 cache take up a considerable portion of the total execution cycles, which leads to the drastic reduction in leakage power consumption from 100% to 28.56% for bank-level and 11.17% for block level gating in L1, to 36.27% for bank-level and 7.31% for block level gating in L2.

Although using the sleep states we were able to save significant power in the caches, still the main energy hungry component in the GPUs i.e. the Register File power needs to be taken care of. Hence in the next chapter we will explore the power management technique to save the static power in the Register File.

Chapter 6

Slumber: Static-Power Management for GPGPU Register Files

In this chapter, we propose *Slumber*, a static-power management technique, for the register files of GPUs which uses various under-volting levels and power-gating at the run-time to save maximum leakage power. This work makes the following contributions:

- We propose a novel power management technique named *"Slumber"* that enables multiple levels of under-volting as well as power-gating based on a static compiler analysis to determine the type of next register access. The length of the idle period is determined using our run-time technique.

- We implement the proposed techniques using the GPGPUSim and obtain a static energy saving of 94% with negligible performance degradation of 1.2% on an average. (Section-6.3)

The chapter is organized as follows: Section 6.1 describes GPGPU register file architecture and motivation for this work. Section 6.2 discusses the device level power modeling of the GPU registers to determine the wake-up latency for power-gating and different under-volting levels. Section 6.3 discusses the proposed power-management technique for the GPU Register File. Our results are discussed in Section 6.4 and we conclude in Section 6.5.

6.1 Motivation

The current state-of-art GPU design focuses on the performance and throughput of the applications being executed. The warp scheduler strives to make the best use of all the available resources by fast context-switching between the warps. When one warp is stalled, another warp swaps in and starts executing to boost the resource utilization. However, in most practical scenarios the GPGPU resources are under-utilized primarily due to imbalanced workload distribution. The large register file accounts for the larger fraction of the on-chip storage in terms of area and leakage power. As shown in Figure 6.1, the register file size has exceeded the size of the the other storage structures like L1D, Shared Memory and L2 Cache. For example, in NVIDIA Pascal, 14.3 MB of register file accounts for 63% of the on-chip storage area. In the current work, we focus on the register file which consumes 32% of the Streaming Multi-Processor's (SM) total leakage power and hence, is the biggest contributor of the static power in a GPGPU core [86]. The behavior of the

applications not only lead to resource under-utilization but also sub-optimal power usage. In the existing literature, there is almost no power-saving features besides the coarse-grain DVFS and clock-gating [20, 34].

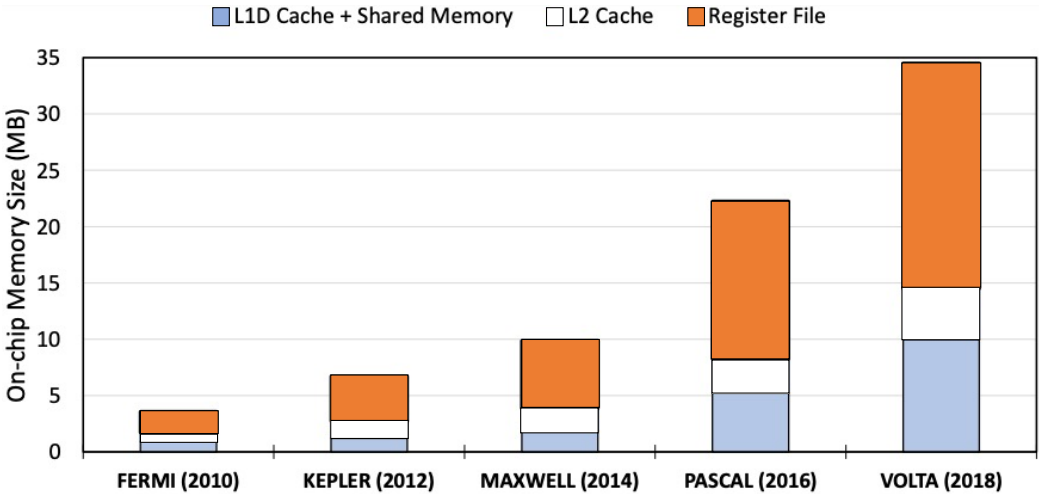

Figure 6.1: The Register File size has out grown the cumulative size of L2 Cache, L1 cache and Shared memory in GPUs over the years. Static Energy consumed by the storage structures is proportional to their sizes.

6.1.1 Register File Organization

In the baseline (Fermi GPGPU Architecture), 128 KB *Register File* comprises of 4 single-ported banks, which is further subdivided into 8 sub-banks operating in parallel (each 128 bits wide, i.e., 4 registers per entry). Each sub-bank has 256 entries. A warp having 32 threads accesses the same entry in all the 8 sub-banks within a bank in parallel to access the register values for all the threads [34]. The registers allocated to a thread belong to the same sub-bank. The register file can be under-volted at different levels of granularity: whole

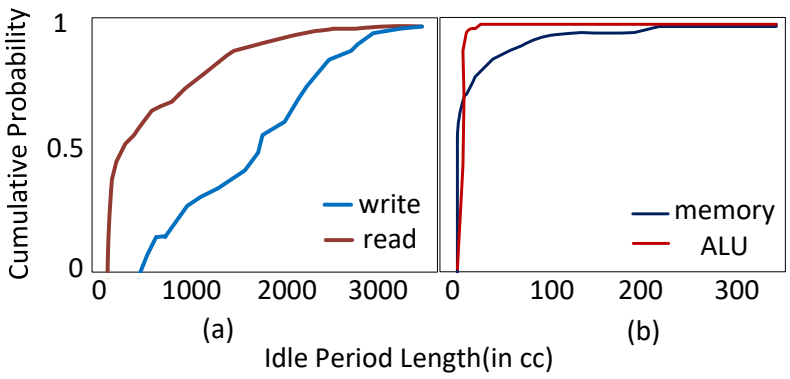

Figure 6.2: GPGPU registers Idle Period Distribution for Vector Add application: (a) Cumulative Density Function (CDF) for the reuse distance of the register-writes and register-reads. (b) CDF of the time difference between the cycle when the instruction is scheduled by the warp scheduler and the cycle when the register is updated.

register file, register file bank, registers allocated per thread, or individual 32-bit Register. A more fine-grained under-volting leads to maximum static power savings in the component but it comes at the cost of extra hardware complexity. Hence, the granularity at which under-volting needs to be done has a significant impact on the power usage and performance.

Cumulative probability of occurrence of the write and read idle period length for a given register shows that, on an average, the idleness period of the register used in write in the next access is greater than register used in read in the next access as shown in Figure 6.2(a). The write idle period refers to the inter-access distance of a register between two consecutive accesses where the later access is a register write operation. Similarly, the read idle period refers to the inter-access distance where the later access is a register read operation.

Utilization is calculated as the ratio of the number of clock cycles the unit was accessed and the total clock cycles taken for the execution of the application. GPU registers

are heavily under-utilized. The less the utilisation, more the opportunity of energy saving. If the accesses are less apart from each other with time, then the idle period length is less. The register file can be under-volted at different levels of granularity: whole register file, register file bank, registers allocated per thread, or individual 32-bit Register. A more fine-grained under-volting leads to maximum static power savings in the component but it comes at the cost of extra hardware complexity. Hence, the granularity at which under-volting needs to be done has a significant impact on the power usage and performance. Our experiments show that the average utilization of the Register File, register Bank, all registers associated with a warp, and each 128 Byte register are 32.7%, 16.3%, 3.9% and 0.2% respectively. This imbalance is mainly because of the load imbalance across the GPU cores, warp branch divergence, irregular memory access patterns and cache contention. In this work, we target on saving leakage energy at the register granularity as they have the least utilization.

The ***registers to be read*** in next accesses can not be power-gated as they are state-retentive, hence they are ***undervolted***. However, the ***registers to be written*** in the next access can be ***power-gated or undervolted*** depending on the idle period length. If the idleness period is more than the power gating break-even time then they are power-gated else they are undervolted.

The main challenge in under-volting or power gating is determining the time to initiate the wake-up of the component; inefficient wake-up policy can lead to sleeping on the registers in the critical path of the GPU pipeline execution and hence, incur heavy performance penalty.

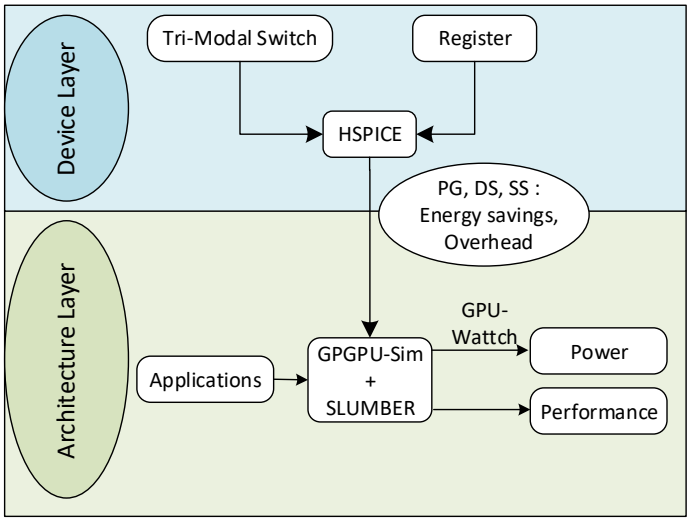

Figure 6.3: Slumber Overview showing Cross-layer methodology.

6.1.2 Conservative wake-up

Latency of the non-memory instructions execution (ALU-Int/Float, SFU) are deterministic. So, the output register is switched to "Sleep mode" depending on the instruction execution latency and the wake-up is initiated at time calculated as instruction latency - Wakeup latency of sleep mode. However, determining the wake-up initiation time for the memory instructions is challenging as the memory instruction execution latency is non-deterministic and depends on various factors like mshr-free-entries, L1 hit/miss, L2 miss, DRAM access and queuing delay. Hence, the status of the memory instruction is tracked [L1 hit/miss, L2 miss] and the memory operation latency is assumed to be the access latency of L1 (for L1 hit), access latency of L2 (for L1 miss), or access latency of DRAM (for L2 miss). Accordingly, wake-up is initiated at the minimum memory access latency - wakeup latency of sleep mode.

6.1.3 Slumber Overview

The overview of the proposed static power reduction technique has been shown in Figure 6.3. The device layer implementation (details in Section 6.2) consists of the accurate modeling of the different power reduction modes (PG: Power gating, DS: Deep Sleep and SS: Shallow Sleep) and the associated leakage energy savings and overhead in terms of wake-up latency to the ON mode using the Tri-modal switch [111] and 128 Byte (32 set of 6T SRAM) register using 45 nm technology. The architecture layer consists of modifying the baseline GPGPU-Sim [13] to incorporate the architectural modifications of Slumber and integrate the different power modes and overheads from the device layer simulation. We evaluate various applications from commonly used GPU Benchmark suites [13, 24, 100, 116] using the modified GPGPU-Sim for the power and performance results.

6.2 Static Power Reduction Metrics Estimation

Static power can be reduced by two techniques: **Power gating (PG)** and **Under-volting (UV)**. We shall discuss the static power savings metrics: performance and power overhead associated with each technique in the following section.

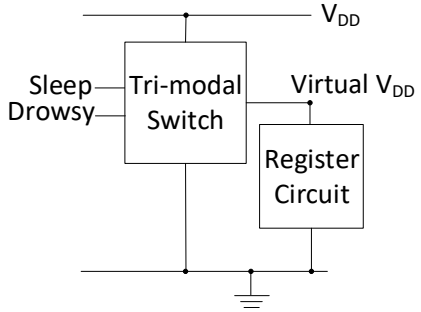

Figure 6.4: Varying the voltage across the register using Tri-modal switch.

Figure 6.4 shows a tri-modal switch used to apply different voltages (Virtual V_{DD}) across the target register [111]. For power-gating, the Virtual V_{DD} is set to "Zero" such that the voltage drop across the register is negligible and we have maximum leakage power savings. For under-volting, Virtual V_{DD} is set to a voltage such that ($0 < VirtualV_{DD} < V_{DD}$) and the lower the Virtual V_{DD}, the more is the leakage power saving. The output of the tri-modal switch (Virtual V_{DD}) is controlled by the signals "Sleep" and "Drowsy" and the width of the transistor MS as described in [111].

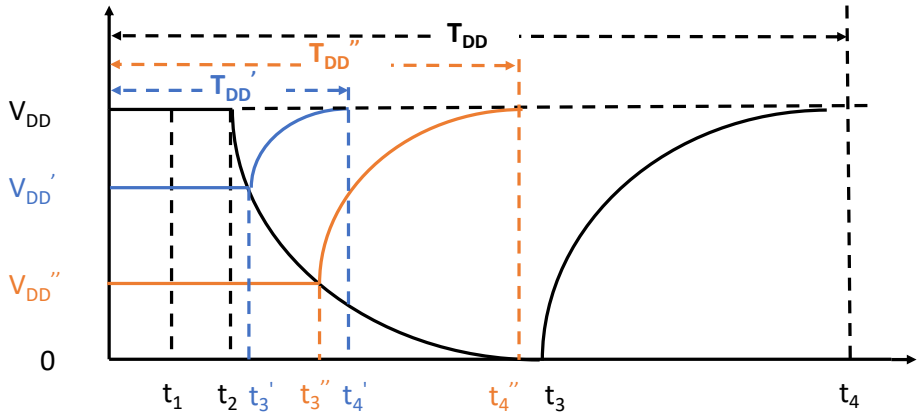

Figure 6.5: Target circuit state transition during power gating interval.

When the target register is idle, a *sleep* signal = 1 and *drowsy* signal = 0 are applied to the power gating switch so that the Virtual V_{DD} is set to *zero*; the target circuit is turned off (OFF state). Alternatively, while waking up the target circuit, the sleep signal = 0 so as to set Virtual V_{DD} to V_{DD}. However, power-gating introduces the energy overhead due to the switching of transistors to OFF/sleep state. This implies that the target circuit should now sleep for at least break-even time ($t_{breakeven}$) to compensate for the energy overhead incurred ($E_{overhead}$). For example, Figure-6.5 shows the voltage transition when

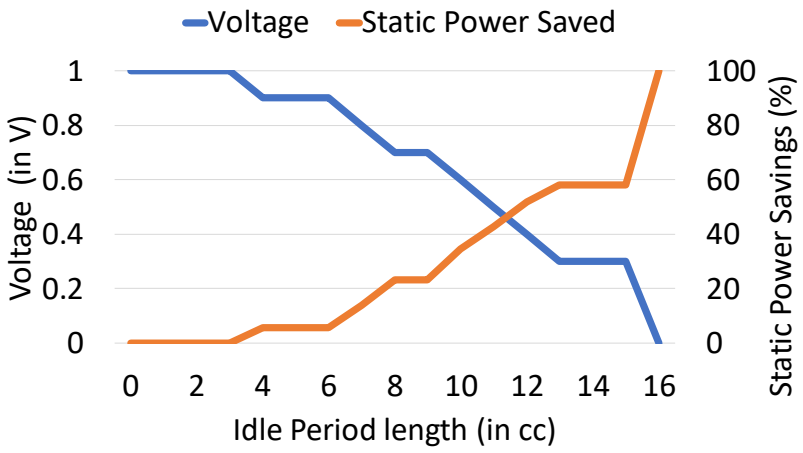

Figure 6.6: Determination of optimal under-volting level and associated static power savings based on the idle period length.

the circuit is switched from ON (Voltage = V_{DD}) to OFF (Voltage = 0) at time t_2 and back to ON at time t_4. The target circuit does not switch to OFF state immediately, the voltage across the capacitive load in the target circuit completely discharges at t_3. The circuit stays in OFF state from t_3 till t_4 and then the wake up is initiated at t_4. As seen in the ON to OFF transition, the circuit capacitance charges to V_{DD} slowly and the circuit is completely ON at t_5. The break even time can calculated by taking the difference of t_4 and t_2, i.e., $t_{breakeven} = t_4 - t_2$; $t_{breakeven}$ is defined as the minimum time period the target circuit should sleep in-order to save power. At $t_{breakeven}$, energy saved (E_{saved}) is same as energy overhead ($E_{overhead}$). $t_{detect}(= t_1)$ is the time taken by the control circuit to make a decision to power gate the target circuit. Finally, $t_{fall}(= t_3 - t_2)$ is the transition time to turn-off and $t_{wake-up}(= t_5 - t_4)$ is the transition time to wake up the target circuit.

On the other hand, if we undervolt to a level V_{DD}' as shown in Figure 6.5, the transition times (t_3'-t_2') and (t_5'-t_4') are much shorter. Then the required idle period (T_{DD}')

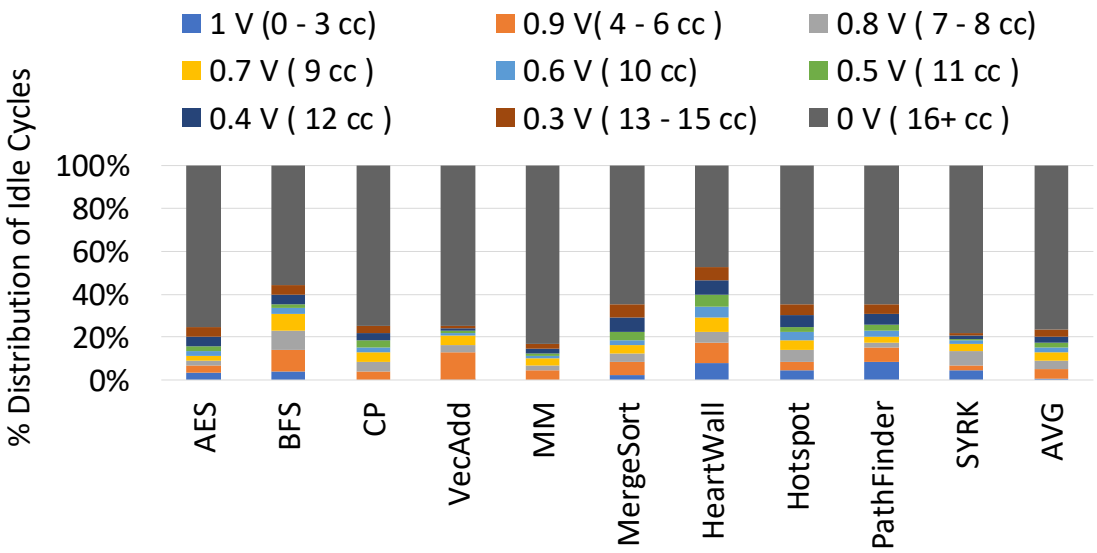

Figure 6.7: Idle Period Distribution based on length

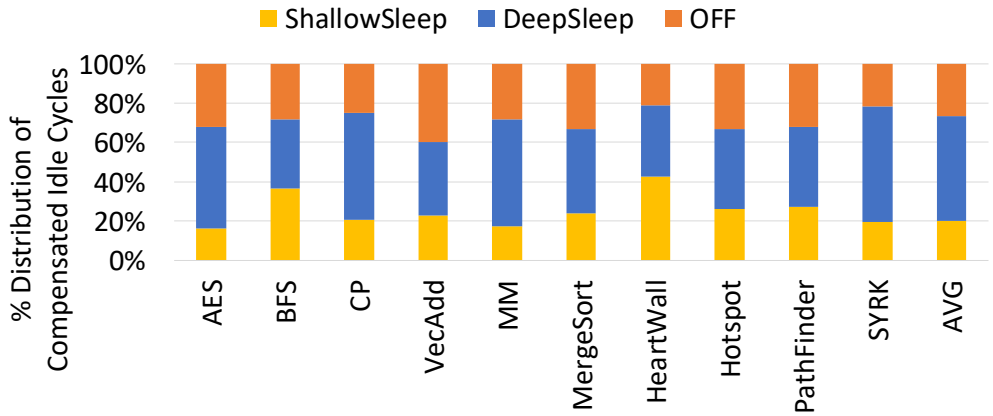

Figure 6.8: Idle Period Distribution across Sleep States

can be considerably reduced at the expense of a little more energy. Target circuit state transition during power gating interval is shown in Figure-6.5. We obtain the values of t_{detect}, t_{fall} and $t_{wake-up}$ for the components using HSPICE simulations and calculate the break even time (T_{DD}) for power-gating, $T_{DD}{}'$ for undervolting to level $V_{DD}{}'$ and $T_{DD}{}''$ for

undervolting to level V_{DD}''. The undervolting level V_{DD}' is termed as shallow sleep state and V_{DD}'' is called deep sleep state in our work.

The two main considerations for any static power reduction technique are: (a) typically, there is a performance overhead associated with transitioning between different low-power states; ideally, the transition overhead should not affect the overall performance of the application, (b) the storage structures should be state-retentive at the low power mode to preserve the content in such a way that it is ready to use when the circuit is powered back to ON state.

We explore the state transition delay of the registers at various voltage levels using the methodology described above. The transition time from different voltages to ON voltage (1V) and the static power saving for the registers as a percentage of the maximum static power consumed in the ON state are shown in Figure 6.6. Primary Y -axis shows voltage applied across the Register (Virtual V_{DD}) for optimal static power savings. Secondary Y -axis shows static power consumed by the registers as a percentage of the maximum static power consumed while at V_{DD} in ON state. The drowsy state, considered in the previous papers [2, 39], corresponds to a voltage of 0.3 V that takes an idle period of 13 cycles to be enabled. The % distribution of the idle cycles based on their length are shown in Figure 6.7. Over 75% of the idle cycles have a length of more than 16 cc and rest of the idle cycles are evenly divided across the voltage categories of 0.3 V - 0.9 V. Although we can theoretically put a register to power gating after 16 cycles, that is not possible in case the next access is a register read. We have to put it in deep sleep mode. Also, incorporating multiple sleep states shall incur high area overhead, hence, we define a shallow sleep state (SS) that works

Table 6.1: Power management modes for Register File.

State	Wakeup Delay (in clock cycles)	Static Power
ON	0	100%
Shallow Sleep (SS)	4	94%
Deep Sleep (DS)	13	42%
Power Gating (PG)	16	0%

at 0.9 V, takes only 4 cycles but can save 6% static power, compared to when the register is on. We call the drowsy state as the deep sleep (DS) state.

Table 6.1 shows different power management states used in SLUMBER, their wake-up delays and static power consumed. The voltage across the component in Figure 6.4 for the ON state is 1V, shallow sleep state is 0.9V, deep sleep state is 0.3V and power gating or OFF state is 0V. Considering these states, the latency distribution for different applications is shown in Figure 6.8. The distribution gives an idea about the possible energy saving in Slumber. On a average, 26.3% of the compensated idle cycles are in OFF state, 53.5% are in Deep Sleep State and 20.1% are in shallow sleep state. Even though the idle cycle length of over 75% of the idle cycles is more than 16 cycles, but only 26.3% can be power gated, as for the read accesses, the register cannot be power-gated.

6.3 Slumber Design

6.3.1 Compiler-generated Hints

The first step determining the undervolting level for a register is identifying the type of next access to the register. We task the compiler to parse through the kernel code and assign a 1-bit flag (0: if next access is write, 1: if next access is read) to each register in

each instruction in each kernel. This is,in general, a difficult problem at compile time, as the kernel code consists of loops. Hence we approximate as follows: if the iteration count of a loop is statically resolvable, we can determine the type of next access and we use that to assign the flags to the register. If the iteration count is not a resolvable constant, we conservatively assign a fixed flag (1) to each register access in instructions outside the loop. We use the same approach for nested loops.

6.3.2 Power-Gating and Under-volting Opportunities in Register File (RF).

Since the idle cycles for register read cannot be power-gated irrespective of the idle cycle length, we insert 1 bit flags per register at the compile time stating the type of the next access to the operand registers is read/write. We observed that registers can be undervolted and power-gated in primarily three different conditions:

(1) Register write waiting on a compute operation:

The compute operations have a deterministic execution latency once they enter the functional unit pipeline. For example, the latency of Integer multiplication operation is 6 cc [13] (predicted idle period of the corresponding register is 6 cc), hence, the corresponding registers are switched to the shallow sleep (SS) state where there is a energy saving of 2 cc after accounting for the switching overhead to SS state. Similarly, in case of integer division operation where the latency is 153 cc [13], the registers are switched to OFF state. There can be additional wait periods at the collector unit prior to the execution in functional unit as only one out of multiple ready warps can be issued per cycle to initiate execution in the

functional unit. The warps also wait at the dispatch stage if there is a write conflict at the result bus since the result bus is shared between all the compute functional units. We do not account for these delays as we use a conservative wake-up policy.

(2) Register write waiting on a memory operation:

At the write back stage, the scoreboard releases the output registers for a warp after writing into the registers for the finished warp. A 2 bit-counter per warp (MSHR Tracker) can be used to keep track of the L1-hit, L1-miss and L2-miss for a warp. Upon a L1-hit, counter is set to 0, in case of a L1-miss the counter is set to 1 and in case of a L2 miss it is set to 2. Usually, the hit access latency of L2 cache and DRAM is in order of hundred cycles. Since the access latency for L2 and DRAM cache is much higher than the wake-up latency of the OFF state, the registers are switched OFF in case of a L1-miss or L2-miss. No state change is there in case of LI-hit. The counter values are used to determine the minimum memory access latency. The idle period of the corresponding registers is predicted to be same as this minimum memory access latency. The register is woken up after a time = *minimum memory access latency - OFF state latency.* The minimum memory access latency of L1, L2 and DRAM was determined using microbenchmarking [153].

(3) Register idle as the warp has finished executing the kernel:

The registers associated with a warp become idle after the warp has finished executing the kernel and waiting for the other warps in the same thread-Block to finish execution. The physical registers for the warp can be power-gated till all the warps for the thread-blocks finish executing the kernel. This is because the register contents shall be re-

allocated when next thread-block is allocated to the Streaming Multiprocessor (SM). Another scenario when the registers are idle is when the last batch of thread-blocks are executing in another SM. Since all the thread-blocks in the current SM have finished execution, they can be power-gated.

6.3.3 Power Mode Transition

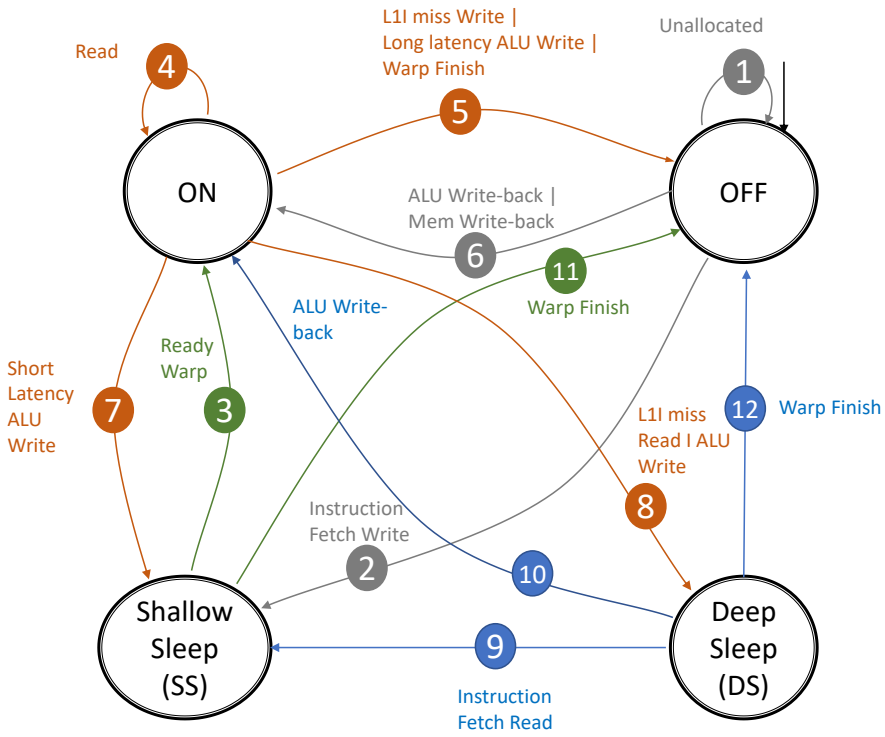

Figure 6.9: FSM of the Slumber Algorithm.

The Finite State Machine for the various power mode transition in Slumber is shown in Figure 6.9. The SASS code for an illustrative example is shown below. Using the compiler hints (determined in Sec 6.3.1), the registers with next access type "read" or

"un-determined" are marked as red and next access type "write" are marked as blue.

SASS Code with compiler hints:

PC_1: GLD R0 [R], [0x1];

PC_2: GLD R1 [R], [R0];

PC_3: IMAD R2 [R], [0x6], R0 [W], R1 [-];

PC_4: IADD R0 [-], [0x4], R2 [-];

When the kernel starts execution, all the registers are in *OFF* mode. When a thread-block is assigned to a Streaming Multiprocessor (SM) by the thread-block scheduler, the physical registers are allocated. The registers not assigned to any thread-block continue in *OFF* mode through ❶. PC_1 suffers a cold-miss in L1 instruction cache (L1I) and L2. The counter associated with registers R0 is set to DRAM access latency. Once the R0 counter attains a value of 12, R0 starts transitioning into *Shallow Sleep (SS)* mode through ❷. Once warp-scheduler signals PC_1 to be ready, R0 is switched ON through ❸. The default warp scheduling policy is modified to determine the three future warps to be scheduled in-addition to the current scheduled warp in order to hide the *SS* to *ON* latency of 4 cc. The registers for the future to-be-scheduled ready warps (identified by the warp-scheduler) are switched from the *SS* to *ON*.After the warp-scheduler issues PC_1 and it enters the load-store unit in the pipeline, R0 continues to stay in ON state ❹ till the L1 cache access status is determined. Incase the L1 cache access is a hit, R0 is updated; However, incase of a L1 miss, the R0 transitions into OFF state ❺ and associated counter is set to a value determined as per sec 6.3.2. Wake-up of R0 is initiated before the memory request completes (tracked by the R0 counter) through ❻. After the write-back to R0 is complete, its next state-transition

is determined depending on the predicted idle period and type of the next access to the register (read/write) as follows:

The first step for determining the idle period length of the register which is to be used in another instruction is to check if the next instruction is ready to be issued to functional unit.

(a) If next instruction (PC_2) is ready, the the register R0 continues to be in ON state through ❹.

(b) If next instruction (PC_2) is not-ready, but exists in instruction-buffer or L1I, then R0 transitions to *SS* state through ❼. Its wake-up is initiated later by the warp-scheduler (similar to PC_1 described earlier).

The mechanism to predict if the instruction exists in L1I is described later in section 6.3.4. If the instruction is not present in L1I, then the access to L2 is a long-latency operation. Hence, the register R0 can be switched to (c) *DS* through ❽ if the next access type is "read",or, (d) *OFF* through ❺ next access type is "write". The wake-up of register R0 is initiated through ❾ or ❷ by the associated counter. Similarly, after PC_3 is issued to the Special-Function-Unit (SFU) by the warp-scheduler, the input operand registers R0 and R1 transition as per the discussion above (similar to PC_1). However, depending on the latency of the ALU operation, the output register R2 remains in the *ON* state (through ❹ for branch operation (latency is 2 cycles)) or transitions to a low-leakage state and back to *ON* (*SS* through ❼ and ❸ for integer add (latency is 6 cycles), *DS* through ❽ and ❿ for integer-max operation (latency is 15 cycles)), *OFF* through ❺ and ❻for integer-division operation (latency is 153 cycles)). Finally, When a warp finishes execution of the kernel, all

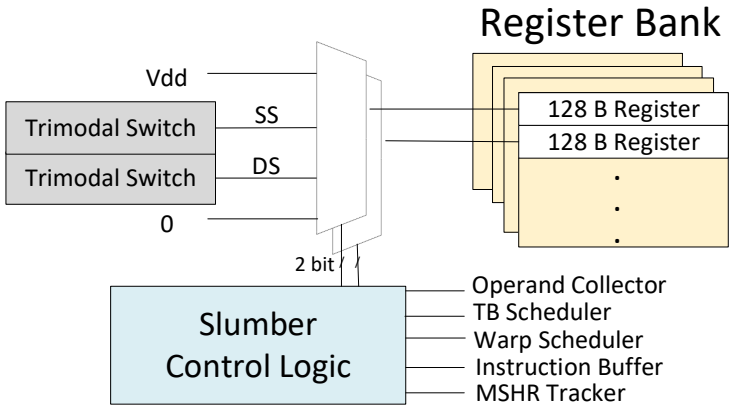

Figure 6.10: Illustration of GPU Registers connected to the voltage rail and tri-modal switch output using mux and Slumber control Logic in order to enable Power Gating (V = 0), Under-volting modes(SS: Shallow Sleep, DS: Deep Sleep) and ON state (V = V_{dd}).

the registers are transitioned from any other mode (*ON* through ❺, *SS* through ⓫, *DS* through ⓬) to *OFF* mode.

6.3.4 Re-Architecting Register File for Slumber

In this section, we illustrate how the baseline register file architecture is modified to incorporate SLUMBER power modes shown in Figure 6.10. The power modes are enabled by connecting each 128 byte register to a multiplexer (mux). Each register is associated with a 10-bit counter to wake-up after the predicted idle period since the maximum idle period length results from DRAM access is 400-600 cc [13]. The mux has 4 input signals: Vdd (Supply Voltage), Deep Sleep Voltage, Shallow Sleep Voltage (Both generated by varying the configuration of the Trimodal switch), 0 (Ground) ; 2-bit control signal (from the Slumber Control Logic) to select between 4 different power modes. The Slumber Control Logic decides the Power mode for a given register based on the input from the MSHR tracker

(L1 or L2 miss), Instruction Buffer (next registers to be in the Shallow Sleep mode), Warp Scheduler (registers for the future warps in On mode), Operand collector and Thread-Block Scheduler. We introduce a hardware fetch-unit-list to determine whether a PC exists in L1I. The 16-entry fetch-unit-list to keep track of the starting PC address in each cache line, where each entry is 8 Bytes (PC size). Each cache line in L1I serves 16 consecutive PC requests. If there is an L1I hit, the instruction gets loaded into instruction buffer in 2 clock cycles. However, wake-up delay from OFF or DS state is much higher. So, we use a similar mechanism as in Section 4.1.2 to determine the idle period length of the registers and initiate their transition into SS state.

Overheads

The counters associated with each 128 Byte register introduces the largest area overhead of 10 bit x 1024 i.e. 1.25 KB. The counters along with area overhead of Slumber Control Logic, MSHR Tracker (16 Bytes), multiplexers, tri-modal switches and the fetch-unit-list (128 Bytes) constitutes around 4% of the total register file size and less than 1% of the register file leakage power. The power overhead of these components is included in our results. The area and power overhead are calculated using HSPICE and CACTI v5.3 [132]. These overheads will reduce when we go for a coarser grained power saving technique at whole register-file, bank or warp granularity. However, since the utilization of these components is much high compared with the individual 128 Bytes register, the corresponding leakage energy gain is likely to reduce.

6.4 Results and Discussion

6.4.1 Methodology

The proposed power management technique for saving leakage energy was evaluated using GPGPU-Sim v3.2.1 [13] based on Fermi-like configuration with 15 SM. Each SM comprises of 1024 registers of 128 Bytes each divided into 4 banks. Each SM has two warp schedulers using two-level warp scheduling policy [99]. In our experiments, 10 benchmarks used were selected from 4 different benchmark suites ISPASS [13], Nvidia CUDA SDK [100], Rodinia [24] and Polybench [116]. We enabled PTXPlus for accurate register file evaluations.

6.4.2 Comparison with Warped Register File [2]

Leakage energy saving of Slumber is 94% and Warped Register File (WRF) is 85% compared to the baseline is shown in Figure 6.11. The power gain of Slumber as compared to WRF come from the following scenarios: (1) During a long latency operation, the output register is in ON state due to the write-conservative policy of WRF. However, in Slumber the output register is power-gated during this time interval. (2) When a warp finishes executing the kernel, the registers associated with the warp and all the other warps in the same thread-block are in drowsy state. However, our technique leverages these power saving opportunities and power gates the registers of the finished warps. (3) The inter-access distance between subsequent accesses to the same register is very high, around 789 clock cycles on an average [2]. In the WRF, the registers are switched to drowsy mode after each access. However, slumber classifies the registers depending on the "next access type". If the next access is a register write (write register), then it is switched to OFF mode in case the

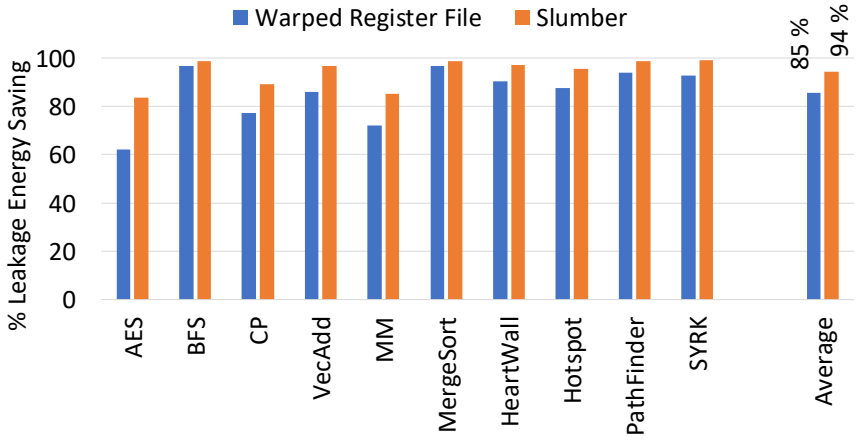

Figure 6.11: Leakage Energy Savings

re-use distance is high. Since the drowsy voltage still consumes a non-negligible portion of the register leakage power, Slumber outperforms WRF in terms of the leakage power savings by switching OFF the "write registers" aggressively.

6.4.3 Leakage energy efficiency improvement

The benchmarks AES, CP, MM and VecAdd have significantly high power saving compared to WRF. The additional power saving for these benchmarks are 23.2%, 14%, 14%, and 10.8%, respectively. The benchmarks with more number of power-gated idle cycles (idle cycle length more than 16 clock cycles) tend to fair better with Slumber. On the contrary, when the frequency of the shorter idle periods in a benchmark is high, it is switched to shallow sleep state (consume more leakage power than deep sleep), thereby reducing the energy gains. This leakage energy saving in register file accounts for about 30% leakage energy saving in streaming multiprocessor and 5% total power saving in the whole GPGPU assuming the dynamic power to leakage power ratio to be 2 to 1.

6.4.4 Performance Penalty

Average performance penalty of Slumber is 1.2% compared to the baseline. In Slumber, when the average number of ready warps in the warp scheduler is less than 3, then the performance loss is experienced as the issued warp has to wait for the register to wake-up from the Shallow sleep state.

6.5 Conclusion

We propose SLUMBER multi-power mode management technique to efficiently reduce the static power consumption of GPU register file unit. The novel approach exploits their under utilization behavior and non-state retentive requirement of the register writes. *SLUMBER* employs three static power reduction modes, each with different static power saving abilities and wake-up overheads in order to reduce power consumption of idle registers effectively. Our experimental results show that by aggressively switching the registers to low-leakage modes, SLUMBER saves static energy by about 94% compared to baseline; it saves 9% more energy than Warped Register File work with minimal performance overhead of about 1.2% compared to baseline. We conclude that SLUMBER provides an effective framework for reducing static power consumption in modern GPU Register Files.

Chapter 7

Conclusion and Future Work

Graphics Processing Units (GPUs) have emerged as an important computational platform for data-intensive applications in a plethora of application domains. They are commonly integrated in computing platforms at all scales, from mobile devices and embedded systems, to high-performance enterprise-level cloud servers. GPUs use a massively multi-threaded architecture that exploits fine-grained switching between executing groups of threads to hide the latency of data accesses. To support the complex memory access patterns of applications, GPGPUs have a multi-level memory hierarchy consisting of huge register file and an L1 data cache private to each SM, a banked shared L2 cache connected through an interconnection network across all SMs and high-bandwidth banked DRAM. With the amount of parallelism GPUs can provide, memory traffic becomes a major bottleneck for present-day GPUs,mostly due to the small amount of private cache that can be allocated for each thread, and the constant demand of data from the GPU's many computation cores. With the ever-increasing data size of GPU applications, and each thread having to process

more data, simply increasing the cache sizes is not a viable option, since the additional area will incur extra cost and overhead. This means that smaller L1 and L2 caches are much more likely to suffer from cache thrashing, i.e. eviction of cache lines which could have been used by other execution units. This results in under-utilization of many SM components like register file, thereby incurring sizable overhead in the GPU power consumption due to wasted static energy of the registers [34, 67, 86].

With the amount of parallelism GPUs can provide, present-day GPUs suffer from the performance and energy challenges due to smaller on-chip caches and under-utilized large register-file. This dissertation offers several novel synergistic compiler/microarchitecture techniques to leverage the data locality information and the cache and registers temporal locality information for enabling high-performance and energy efficient GPUs.

In Chapter 3, we present *PAVER* [139], a priority-aware vertex scheduler, which takes a graph-theoretic approach towards thread scheduling. We analyze the cache locality behavior among thread blocks (TBs) by profiling, and represent the problem using a graph representing the TBs and the locality among them. This graph will then be partitioned to TB groups that display maximum data sharing, which are then assigned to the same SM by the locality-aware TB scheduler. This novel technique simultaneously reduces the leakage and dynamic access power of the L2 caches, while improving the overall performance and energy efficiency of the GPU. Through exhaustive simulation in Fermi, Pascal and Volta architectures using a number of scheduling techniques, we show that our graph theoretic-guided TB scheduler reduces L2 accesses by 43.3%, 48.5%, 40.21% and increases the average performance benefit by 29%, 49.1%, 41.2% for the benchmarks with high inter-TB locality.

However, the profiling approach for extracting the locality graph and then deriving insights from the graph to design efficient TB scheduler can be infeasible for larger workloads and unknown applications.

In Chapter 4, our technique *LocalityGuru* [140] proposes a thread block-centric locality analysis, which identifies the locality among the thread blocks (TBs) in terms of a number of common data references. We seek to employ a detailed just-in-time (JIT) compilation analysis of the static memory accesses in the source code and derive the mapping between the threads and data indices at kernel-launch-time. Our locality analysis technique can be employed at multiple granularities such as threads, warps, and thread blocks in a GPU Kernel. This information can be leveraged to help make smarter decisions for locality-aware data-partition, memory page data placement, cache management, and scheduling in single-GPU and multi-GPU systems. The results of the LocalityGuru PTX analyzer are then validated by comparing with the Locality graph obtained through profiling. Since the entire analysis is carried out by the compiler before the kernel launch time, it does not introduce any timing overhead to the kernel execution time.

In the previous chapters, we gained performance, which eventually translates to static energy saving in the whole GPU, however, we dive into more fine grained microarchitectural granularity and explore the static energy saving of the storage structures like Caches and Register Files by leveraging the reuse distance between the subsequent accesses to the Caches.

In Chapter 5, we propose a static energy saving technique in both L1 and L2 caches which uses a trimodal switch to enable various voltage modes (ON, OFF, Sleep).

The technique incurs insignificant overhead in terms of area and power. The main energy savings come from the fact that the L2 caches are less frequently used than the L1 caches and have longer idle periods. Hence enabling the sleep mode for the caches between the two consecutive accesses helps save significant leakage energy for the GPU storage structures.

Over the past decades, the GPUs have continued to scale up in terms of number of concurrent threads and cores. In order to support the faster context switching among the active threads, the register file size per core has also grown in size. The register file is the largest SRAM structure on the die and hence consumes the most leakage energy compared to L1 and L2 caches. Finally, in Chapter 6, *Slumber* [141], we develop a realistic model for determining the wake-up time of registers from various under-volting and power gating modes. Next, we propose a hybrid energy saving technique where a combination of power-gating and under-volting can be used to save optimum energy depending on the idle period of the registers with a negligible performance penalty. Our results show that *Slumber* can save energy of the register file during the idle period, which significantly reduces the energy consumption. Our simulation shows that the hybrid energy-saving technique results in 94% leakage energy savings in register files on an average when compared with the conventional clock gating technique and 9% higher leakage energy saving compared to the state-of-art technique.

Future work

Some potential future research directions are as follows:

- *Thread to data mapping:* We can employ machine learning to determine the thread-address mapping for the data-dependent workloads with indirect memory accesses like BFS. This information is critical for addressing important problems like data prefetching, inter and intra kernel dependency, data and compute colocation, and data partitioning.

- *Predicting execution time of thread:* Machine learning techniques can be instrumental in determining the execution time of a thread using parameters like cache misses, data size, access pattern, communication between threads, etc. This will help with making important decisions regarding load balancing, and scheduling a latency critical task.

- *Processing in memory*: The thread to data mapping along with the compiler insights can be useful for identifying the computation to be offloaded to the memory and bypassing several stages in the pipeline thereby increasing the accelerator throughput significantly.

Bibliography

[1] `https://github.com/gpgpu-sim/ispass2009-benchmarks`, 2009. Accessed: 2018-04-11.

[2] Mohammad Abdel-Majeed and Murali Annavaram. Warped register file: A power efficient register file for GPGPUs . In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 412–423. IEEE, 2013.

[3] Mohammad Abdel-Majeed, Daniel Wong, and Murali Annavaram. Warped gates: gating aware scheduling and power gating for GPGPUs . In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 111–122. ACM, 2013.

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

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

[6] Jaume Abella, Antonio González, Xavier Vera, and Michael FP O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

[7] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12. ACM, 2000.

[8] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[9] Kanak Agarwal, Kevin Nowka, Harmander Deogun, Dennis Sylvester, and Dennis Sylvester. Power gating with multiple sleep modes. In *Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 633–637. IEEE Computer Society, 2006.

[10] Kunal Agrawal, Charles E Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[11] José L Ayala, Alexander Veidenbaum, and Marisa López-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Programming*, 31(6):451–467, 2003.

[12] Sara S Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *ACM SIGPLAN Notices*, volume 47, pages 23–34. ACM, 2012.

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

[14] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[15] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. Automatic c-to-cuda code generation for affine programs. In *International Conference on Compiler Construction*, pages 244–263. Springer, 2010.

[16] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

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

[18] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. Memory access patterns: The missing piece of the multi-gpu puzzle. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[19] Pavankumar Bikki, Pitchai Karuppanan, et al. Sram cell leakage control techniques for ultra low power application: A survey. *Circuits and systems*, 8(02):23, 2017.

[20] Juan M Cebri'n, Gines D Guerrero, and Jose M Garcia. Energy efficiency analysis of GPUs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1014–1022. IEEE, 2012.

[21] Juan M Cebri'n, Gines D Guerrero, and Jose M Garcia. Energy efficiency analysis of gpus. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1014–1022. IEEE, 2012.

[22] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

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

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

[26] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th annual IEEE/ACM international symposium on microarchitecture*, pages 343–355. IEEE Computer Society, 2014.

[27] Yanhao Chen, Ari B Hayes, Chi Zhang, Timothy Salmon, and Eddy Z Zhang. Locality-aware software throttling for sparse matrix operation on gpus. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 413–426, 2018.

[28] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P Topham. Multiple-banked register file architectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 316–325, 2000.

[29] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

[30] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.

[31] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[32] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. Managing static leakage energy in microprocessor functional units. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 321–332, Nov 2002.

[33] Hodjat Asghari Esfeden, Amirali Abdolrashidi, Shafiur Rahman, Daniel Wong, and Nael Abu-Ghazaleh. Bow: Breathing operand windows to exploit bypassing in gpus.

[34] Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael Abu-Ghazaleh. CORF: Coalescing Operand Register File for GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[35] Thomas L Falch and Anne C Elster. Register caching for stencil computations on gpus. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 479–486. IEEE, 2014.

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

[37] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[38] Naznin Fauzia, Louis-Noël Pouchet, and P Sadayappan. Characterizing and enhancing global memory data coalescing on gpus. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 12–22. IEEE Computer Society, 2015.

[39] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 148–157. IEEE Computer Society, 2002.

[40] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján. Dream: Dynamic re-arrangement of address mapping to improve the performance of drams. In *Proceedings of the Second International Symposium on Memory Systems*, pages 362–373, 2016.

[41] Syed Zohaib Gilani, Nam Sung Kim, and Michael J Schulte. Exploiting GPU peak-power and performance tradeoffs through reduced effective pipeline latency. In

Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pages 74–85. ACM, 2013.

[42] Syed Zohaib Gilani, Nam Sung Kim, and Michael J Schulte. Power-efficient computing for compute-intensive GPGPU applications. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 330–341. IEEE, 2013.

[43] Bhargava Gopireddy, Choungki Song, Josep Torrellas, Nam Sung Kim, Aditya Agrawal, and Asit Mishra. Scalcore: Designing a core for voltage scalability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 681–693. IEEE, 2016.

[44] Xuan Guan and Yunsi Fei. Register file partitioning and recompilation for register file power reduction. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15(3):1–30, 2010.

[45] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, volume 45, pages 341–342. ACM, 2010.

[46] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, (3):243–250, 1977.

[47] Bruce Hendrickson and Robert Leland. The chaco users guide. version 1.0. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1993.

[48] Chih-Chieh Hsiao, Slo-Li Chu, and Chiu-Cheng Hsieh. An adaptive thread scheduling mechanism with low-power register file for mobile gpus. *IEEE transactions on multimedia*, 16(1):60–67, 2013.

[49] Jianping Hu, Tiefeng Xu, and Hong Li. A lower-power register file based on complementary pass-transistor adiabatic logic. *IEICE transactions on information and systems*, 88(7):1479–1485, 2005.

[50] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37, 2004.

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

[52] Ali Jahanshahi, Hadi Zamani Sabzi, Chester Lau, and Daniel Wong. Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers. *IEEE Computer Architecture Letters*, 19(2):139–142, 2020.

[53] Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. The more we share, the more we have: Improving gpu performance through register sharing. *arXiv preprint arXiv:1503.05694*, 2015.

[54] H. Jeon, H. A. Esfeden, N. B. Abu-Ghazaleh, D. Wong, and S. Elango. Locality-aware gpu register file. *IEEE Computer Architecture Letters*, 18(2):153–156, 2019.

[55] Hyeran Jeon and Murali Annavaram. Gpgpu register file management by hardware co-operated register reallocation. *Univ. of Southern California, Tech. Rep. CENG-2014-05*, 2014.

[56] Kwangok Jeong, Andrew B Kahng, Seokhyeong Kang, Tajana S Rosing, and Richard Strong. MAPG: Memory access power gating . In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1054–1059. EDA Consortium, 2012.

[57] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[58] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance. In *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[59] Stephen Jones. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation S*, volume 338, page 2012, 2012.

[60] J Juega, J Gomez, Christian Tenllado, Sven Verdoolaege, Albert Cohen, and Francky Catthoor. Evaluation of state-of-the-art polyhedral tools for automatic code generation on gpus. *XXIII Jornadas de Paralelismo, Univ. Complutense de Madrid*, 2012.

[61] Alek Kaknevicius and A Hoover. Managing inrush current. *Application Report SLVA670A, Texas Instruments*, 2015.

[62] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhnag. Cache topology aware computation mapping for multicores. In *ACM Sigplan Notices*, volume 45, pages 74–85. ACM, 2010.

[63] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[64] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. μC-States: Fine-grained GPU datapath power management. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 17–30. IEEE, 2016.

[65] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G Rogers. Locality-centric data and threadblock management for massive gpus. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1022–1036. IEEE, 2020.

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

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

[68] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. Coda: Enabling co-location of computation and data for multiple gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):1–23, 2018.

[69] Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, and Hwansoo Han. Compiler-assisted gpu thread throttling for reduced cache contention. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[70] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 219–230. IEEE, 2002.

[71] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, 2004.

[72] Suhwan Kim, Stephen V Kosonocky, Daniel R Knebel, and Kevin Stawiasz. Experimental measurement of a novel power gating structure with intermediate power saving mode. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 20–25. ACM, 2004.

[73] John Kloosterman, Jonathan Beaumont, D Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. Regless: Just-in-time operand staging for gpus. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 151–164. IEEE, 2017.

[74] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. Revealing critical loads and hidden data locality in gpgpu applications. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 120–129. IEEE, 2015.

[75] Gunjae Koo, Hyeran Jeon, Zhenhong Liu, Nam Sung Kim, and Murali Annavaram. Cta-aware prefetching and scheduling for gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 137–148. IEEE, 2018.

[76] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[77] Nagesh B Lakshminarayana and Hyesoon Kim. Spare register aware prefetching for graph algorithms on gpus. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 614–625. IEEE, 2014.

[78] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[79] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 245–255. IEEE, 2013.

[80] Jungseob Lee, Vijay Sathisha, Michael Schulte, Katherine Compton, and Nam Sung Kim. Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 111–120. IEEE, 2011.

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

[82] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. Warped-compression: Enabling power efficient gpus through register compression. *ACM SIGARCH Computer Architecture News*, 43(3S):502–514, 2015.

[83] Shin-Ying Lee and Carole-Jean Wu. Caws: criticality-aware warp scheduling for gpgpu workloads. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 175–186. IEEE, 2014.

[84] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Fermi architecture white paper, 2009.

[85] Charles E Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, pages 522–527. ACM, 2009.

[86] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. *SIGARCH Comput. Archit. News 2013.*

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

[88] Chao Li, Yi Yang, Zhen Lin, and Huiyang Zhou. Automatic data placement into gpu on-chip memory resources. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 23–33. IEEE, 2015.

[89] Lingda Li, Robel Geda, Ari B Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Z Zhang, and Mario Szegedy. A simple yet effective balanced edge partition model for parallel computing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):1–21, 2017.

[90] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 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*, pages 469–480. ACM, 2009.

[91] Yun Liang, Xiaolong Xie, Yu Wang, Guangyu Sun, and Tao Wang. Optimizing cache bypassing and warp scheduling for gpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1560–1573, 2017.

[92] Jonathan Lifflander and Sriram Krishnamoorthy. Cache locality optimization for recursive programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–16. ACM, 2017.

[93] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. Hybrid cpu-gpu scheduling and execution of tree traversals. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016.

[94] Wenjing Ma and Gagan Agrawal. An integer programming framework for optimizing shared memory use on gpus. In *2010 International Conference on High Performance Computing*, pages 1–10. IEEE, 2010.

[95] Yan Meng, Timothy Sherwood, and Ryan Kastner. On the limits of leakage power reduction in caches. In *11th International Symposium on High-Performance Computer Architecture*, pages 154–165. IEEE, 2005.

[96] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136. IEEE, 2015.

[97] Shin'ichiro Mutoh, Takakuni Douseki, Yasuyuki Matsuya, Takahiro Aoki, Satoshi Shigematsu, and Junzo Yamada. 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS. *IEEE Journal of Solid-state circuits*, 30(8):847–854, 1995.

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

[99] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving GPU performance via large warps and two-level warp scheduling . In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011.

[100] NVIDIA. CUDA Toolkit. `https://developer.nvidia.com/cuda-toolkit`, 2007.

[101] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[102] NVIDIA. Geforce gtx 1080. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`, 2016.

[103] NVIDIA. Nvidia tesla v100 gpu architecture. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017. Accessed: 2018-11-26.

[104] NVIDIA. Cuda toolkit documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#just-in-time-compilation`, 2020. Accessed: 2020-9-23.

[105] NVIDIA. Nvidia a100 tensor core gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`, 2020. Accessed: 2020-10-08.

[106] CUDA NVIDIA. C programming guide, v4.2, april 2012, 2012.

[107] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. Apres: improving cache efficiency by exploiting load characteristics on gpus. *ACM SIGARCH Computer Architecture News*, 44(3):191–203, 2016.

[108] Mark Oskin and Gabriel H Loh. A software-managed approach to die-stacked dram. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 188–200. IEEE, 2015.

[109] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1):65–109, 2002.

[110] Yunheung Paek and David A Padua. Experimental study of compiler techniques for numa machines. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 187–193. IEEE, 1998.

[111] Ehsan Pakbaznia and Massoud Pedram. Design and application of multimodal power gating structures. In *2009 10th International Symposium on Quality Electronic Design*, pages 120–126. IEEE, 2009.

[112] Ehsan Pakbaznia and Massoud Pedram. Design of a tri-modal multi-threshold CMOS switch with application to data retentive power gating. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(2):380–385, 2011.

[113] Asmita Pal, Aatreyi Bal, Koushik Chakraborty, and Sanghamitra Roy. Split Latency Allocator: Process Variation-Aware Register Access Latency Boost in a Near-Threshold Graphics Processing Unit . *Journal of Low Power Electronics*, 13(3):419–427, 2017.

[114] Saptadeep Pal, Daniel Petrisko, Matthew Tomei, Puneet Gupta, Subramanian S Iyer, and Rakesh Kumar. Architecting waferscale processors-a gpu case study. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 250–263. IEEE, 2019.

[115] Sanghyun Park, Aviral Shrivastava, Nikil Dutt, Alex Nicolau, Yunheung Paek, and Eugene Earlie. Register file power reduction using bypass sensitive compiler. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1155–1159, 2008.

[116] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[117] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, 36(6):1389–1401, 1957.

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

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

[120] Kiran Ranganath, Joshua D. Suetterlein, Joseph B. Manzano, Shuaiwen L. Song, and Daniel Wong. Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021.

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

[122] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 99–110. ACM, 2013.

[123] Soumyaroop Roy, Nagarajan Ranganathan, and Srinivas Katkoori. State-retentive power gating of register files in multicore processors featuring multithreaded in-order cores. *IEEE Transactions on Computers*, 60(11):1547–1560, 2010.

[124] Eri Rubin, Ely Levy, Amnon Barak, and Tal Ben-Nun. Maps: Optimizing massively parallel applications using device-level memory abstraction. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–22, 2014.

[125] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. ITAP: Idle-Time-Aware Power Management for GPU Execution Units. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(1):3, 2019.

[126] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 489–502. ACM, 2018.

[127] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, 40(7):115–126, 2005.

[128] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 73–82. IEEE, 2013.

[129] Du Shen, Milind Chabbi, and Xu Liu. An evaluation of vectorization and cache reuse tradeoffs on modern cpus. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 21–30. ACM, 2018.

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

[131] Jun Shirako, Akihiro Hayashi, and Vivek Sarkar. Optimized two-level parallelization for gpu accelerators using the polyhedral model. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 22–33, 2017.

[132] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. 2001.

[133] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[134] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[135] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[136] I Synopsys. HSPICE User's Manual: Simulation and Analysis. *Synopsys Inc, California*, 2010.

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

[138] Hamed Tabkhi and Gunar Schirner. Application-guided power gating reducing register file static power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2513–2526, 2014.

[139] Devashree Tripathy, Amirali Abdolrashidi, Laxmi Narayan Bhuyan, Liang Zhou, and Daniel Wong. Paver: Locality graph-based thread block scheduling for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(3):1–26, 2021.

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

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

[142] S. Tripathy, D. Sahoo, and M. Satpathy. Multidimensional grid aware address prediction for gpgpu. In *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*, pages 263–268, 2019.

[143] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. Tida: High-level programming

abstractions for data locality management. In *International Conference on High Performance Computing*, pages 116–135. Springer, 2016.

[144] Dominic A Varley. Practical experience of the limitations of gprof. *Software: Practice and Experience*, 23(4):461–463, 1993.

[145] Harry JM Veendrick. Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *IEEE Journal of Solid-State Circuits*, 19(4):468–473, 1984.

[146] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 829–842. IEEE, 2018.

[147] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. Dacache: Memory divergence-aware gpu cache management. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 89–98, 2015.

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

[149] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(3):13, 2011.

[150] Xin Wang and Wei Zhang. Drowsy register files for reducing gpu leakage energy. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 632–639. IEEE, 2017.

[151] Yue Wang, Soumyaroop Roy, and Nagarajan Ranganathan. Run-time power-gating in caches of GPUs for leakage energy savings. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 300–303. IEEE, 2012.

[152] Hao Wen and Wei Zhang. Exploring gpu data cache leakage management techniques. *Journal of Computing Science and Engineering*, 12(3):106–114, 2018.

[153] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010.

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

[155] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on gpus. In *Proceedings of the International Conference on Computer-Aided Design*, pages 516–523. IEEE Press, 2013.

[156] Qiumin Xu and Murali Annavaram. PATS: pattern aware scheduling and power gating for GPGPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 225–236. ACM, 2014.

[157] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 172–187. Springer, 2009.

[158] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. *ACM Sigplan Notices*, 45(6):86–97, 2010.

[159] Richard M Yoo, Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 315–325, 2013.

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

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

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

[163] Hadi Zamani, Devashree Tripathy, Ali Jahanshahi, and Daniel Wong. Icap: Designing inrush current aware power gating switch for gpgpu. *Proceedings of the 15th IEEE/ACM International Conference on Networking, Architecture, and Storage*, 2021.

[164] Yuanrui Zhang, Mahmut Kandemir, and Taylan Yemliha. Studying inter-core data reuse in multicores. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 25–36. ACM, 2011.

[165] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 32–41, 2000.

[166] Liang Zhou, Laxmi N Bhuyan, and KK Ramakrishnan. Goldilocks: Adaptive resource provisioning in containerized data centers. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 666–677. IEEE, 2019.

[167] Intel Developer Zone. Intel vtune amplifier, 2017. *Documentation at the URL: https://software. intel. com/en-us/intel-vtune-amplifier-xe-support/documentation*, 2017.

[168] John H. Zurawski, John E. Murray, and Paul J. Lemmon. The design and verification of the alphastation 600 5-series workstation. *Digital Technical Journal*, 7(1):0, 1995.