國立交通大學

電子工程學系 電子研究所

碩士論文

通用式圖形處理器上考量快取記憶體行為之多執行緒決定機制

A Cache Behavior Aware Multithreading Degree Decision Scheme on GPGPUs

研 究 生：顏大剛

指導教授：賴伯承 教授

中 華 民 國　一〇三年　三月

通用式圖形處理器上考量快取記憶體行為之多執行緒決定機制

A Cache Behavior Aware Multithreading Degree Decision Scheme on GPGPUs

研究生：顏大剛　　Student：Ta-Kang Yen

指導教授：賴伯承　　Advisor：Bo-Cheng Lai

國 立 交 通 大 學

電子工程學系 電子研究所

碩 士 論 文

A Thesis

Submitted to Department of Electronics Engineering and

Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Electronics Engineering

March 2014

Hsinchu, Taiwan, Republic of China

中華民國 一〇三年　三月

通用式圖形處理器上考量快取記憶體行為之多執行緒決定機制

研究生：顏大剛　　指導教授：賴伯承教授

國立交通大學

電子工程學系 電子研究所

摘 要

通用式繪圖處理器是目前一主流支援大量平行運算的處理器。為了在進行大量平行運算時有更好的效能，快取記憶體與多執行緒並行為目前主流設計理念。然而，快取記憶體大小的限制，使此二機制在需要大量記憶體存取的應用中會發生互相抵消其效能上帶來的好處。針對這議題，本篇論文首先探討此二機制間的關聯性，並依此關聯性建立一多執行緒並行運算量的決定機制，取得記憶體快取機制與多執行緒並行運算的平衡點。在擁有大量記憶體存取的應用中，此機制降低多執行緒並行運算量，並提高快取記憶體效率，因此平均運算效能提升 60%。在非大量記憶體存取的應用中，此機制也能將多執行緒並行運算量設定在較高的值，避免效能降低。

A Cache Behavior Aware Multithreading Degree Decision Scheme on GPGPUs

Student: Ta-Kang Yen Advisor: Bo-Cheng Lai

Department of Electronics Engineering and Institute of Electronics

National Chiao Tung University

ABSTRACT

GPGPUs have emerged as one of the most widely used throughput processors. Deep multithreading and cache hierarchy are the two effective implementations to achieve high throughput computing in modern GPGPUs. However, these are two conflicting design options. Finding a proper design point between the two has become a significant performance factor to GPGPUs. This paper investigates the correlation between caching behavior and multithreading technique. By demonstrating the trade-off issue between the multithreading and cache contention, the proposed decision scheme dynamically adjusts the multithreading degree to achieve superior performance. With the proposed decision scheme, the system performance of memory-intensive workloads can be improved by 60% averagely, and it prevents computation-bound workloads from performance degradation.

致謝

此篇論文得以完成，首先要感謝指導教授賴伯承博士不厭其煩的指導，在這三年中，不論是專業知識的培養，做研究的態度和處理問題的方法，都讓我獲益良多，在遇到瓶頸或是困難時，教授也適時地給予建議和方向，真的非常感謝賴伯承教授。

特別感謝郭玹凱學長，在這三年期間，不斷的與我分享研究心得與技巧，並時常與我討論其他人的研究成果，分析優缺點。學長的指導讓我學習到許多做研究的方法，也因此有了這份研究成果。

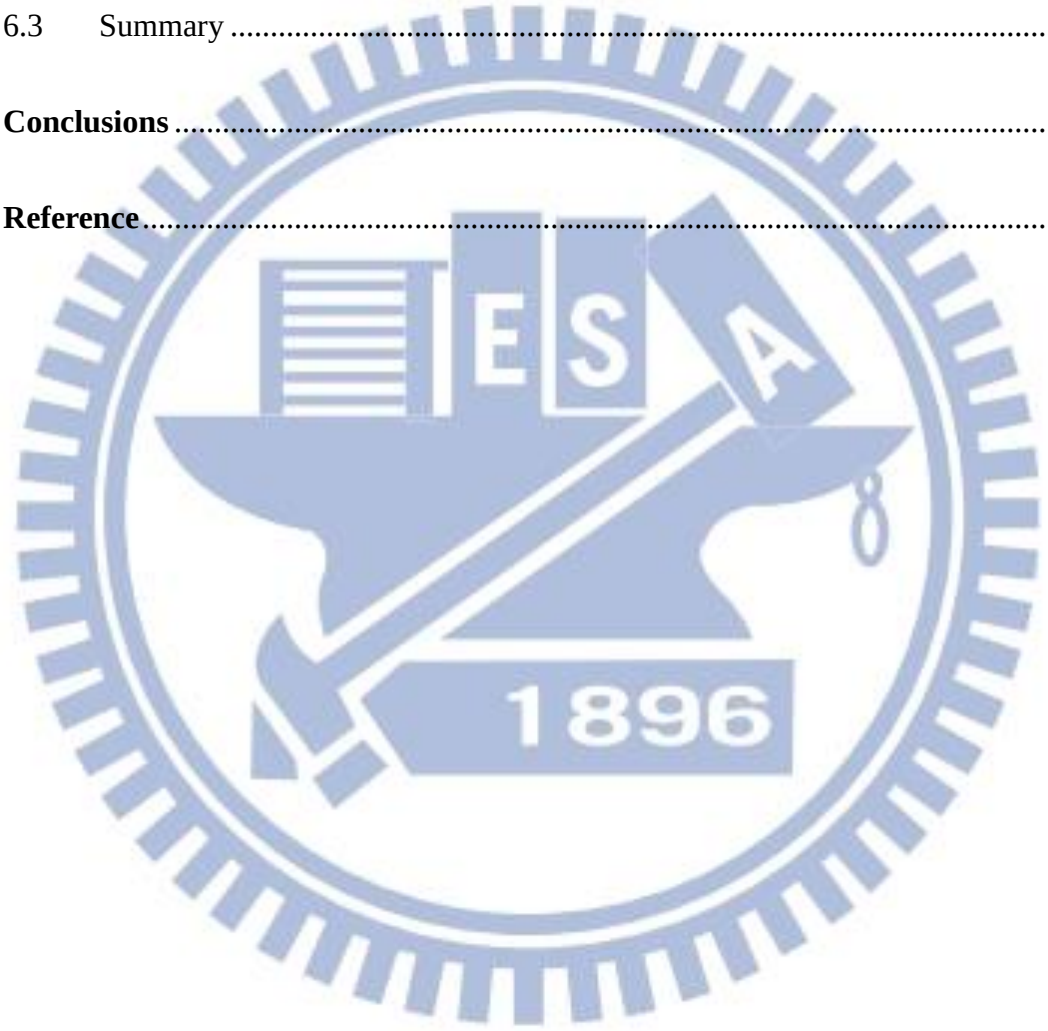
在研究所求學的過程中，最感謝在背後默默支持我的家人，多次容忍我任性的要求，也讓我有一個可以專心求學的環境。

謹以此篇論文獻給所有關心我與我所關心的人們。

中華民國一〇三年三月

研究生顏大剛謹致於交通大學

CONTENTS



LIST OF TABLES

LIST OF FIGURES

1. Introduction

GPGPUs (General Purpose Graphic Processing Units) are becoming widely used many-core throughput processors to achieve great performance in modern computing platforms. By supporting general purpose programming environments, like CUDA [11] and OpenCL [12], GPGPUs can handle massive parallel execution of various types of applications. To expose massive computing parallelism, the stringent data access requirements become a design challenge to achieve high performance on GPGPUs. The enormous concurrent tasks pose high bandwidth data demands with complex data access patterns. These data demands require higher DRAM bandwidth and cache capacity to sustain the memory level parallelism and hide memory request latency. Once the data demand exceeds the limit of cache capacity and DRAM bandwidth, some memory requests that should be processed in parallel would be executed sequentially and result in longer execution time [1]. In addition, the enormous concurrent threads could create serious contention on the on-chip memory resource, and would significantly degrade the overall performance. A thorough understanding of the performance correlation between the multithreading degree and cache utilization has become a critical factor to achieve superior performance.

Exposing massive computation parallelism through multithreading is an essential design factor for a GPGPU to achieve great performance. Figure 1 illustrates an example of the architecture of a modern GPGPU. The concurrent threads in a GPGPU are organized as multiple thread groups, where each group is referred as a Collaborative Thread Array (CTA). A CTA can be further decomposed into many *warps*, where each warp contains 32 threads according to NVIDIA's GPGPU architecture [22]. A centralized CTA scheduler is implemented to dispatch CTAs to the CTA buffer of each *shader core*. Shader cores are the basic functional units that perform concurrent execution on a large number of threads. There is a warp scheduler on each shader core to control the instruction issuing. When the pipeline

is stalled by long latency instructions, the multithreading mechanism allows the warp scheduler to issue warp instructions from other warps in the CTA buffer to improve the pipeline utilization and execution throughput. The number of warps in the CTA buffer is named as *Multithreading Degree* in this work.

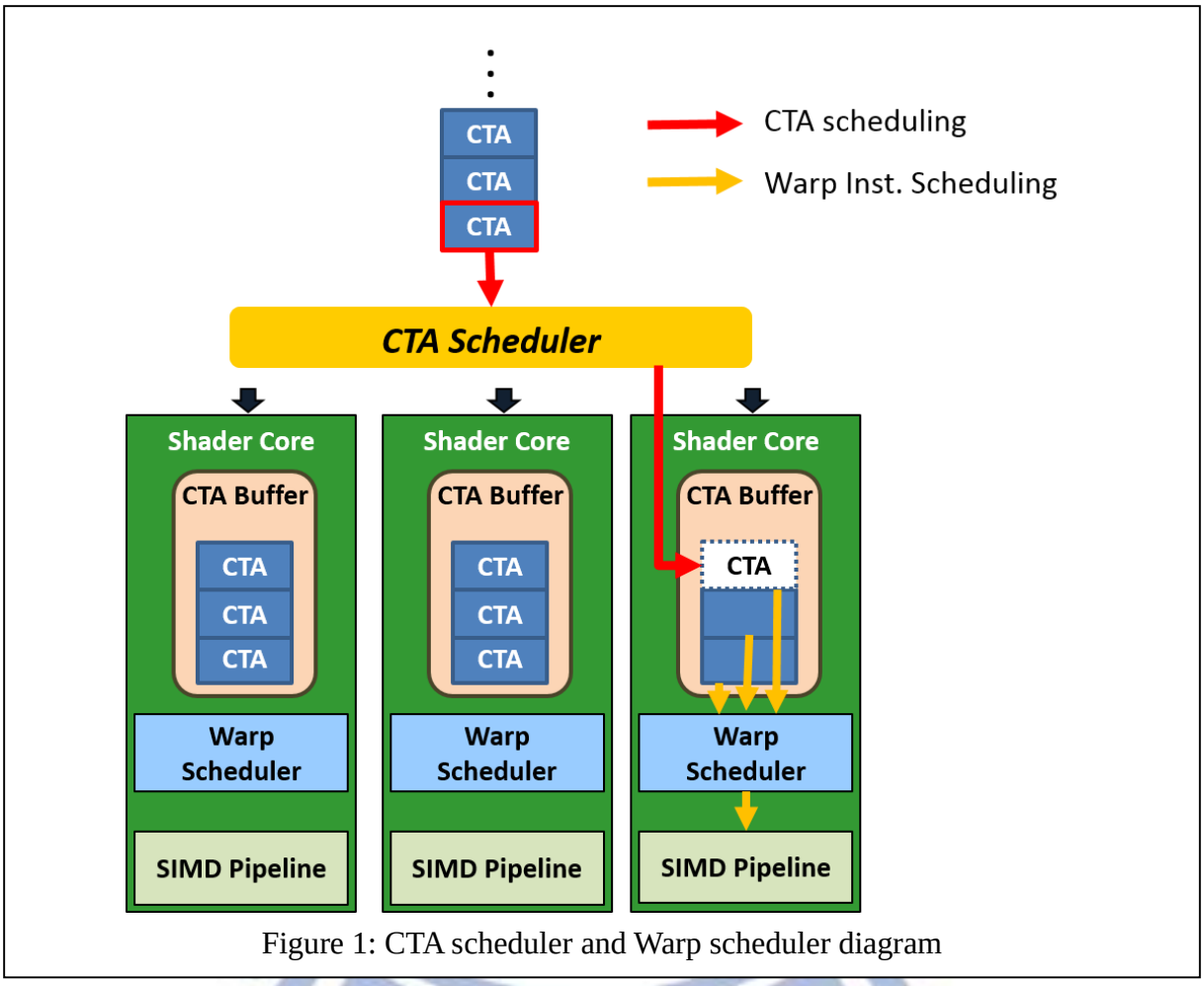


Figure 1: CTA scheduler and Warp scheduler diagram

However, as demonstrated in Figure 2, a higher multithreading degree does not always achieve a better system performance. Figure 2 shows the performance (IPC: Instruction per Cycle) of our in-lab benchmarks [9]. The performance results are simulated by a cycle accurate GPGPU simulator, GPGPU-Sim [13], with 32KB L1 data cache at different multithreading degrees. The benchmarks are selected to evaluate the performance of memory system by featuring memory intensive computing with irregular memory accesses. There are two observations from this experiment. *First*, higher multithreading degrees do not always

lead to better overall performance. Excessive concurrent threads could cause serious contention on the local cache, and hence degrade the overall performance. *Second*, the best multithreading degree is dependent upon the characteristic of each benchmark, such like the working set sizes, data reusing, and the instruction level parallelism.

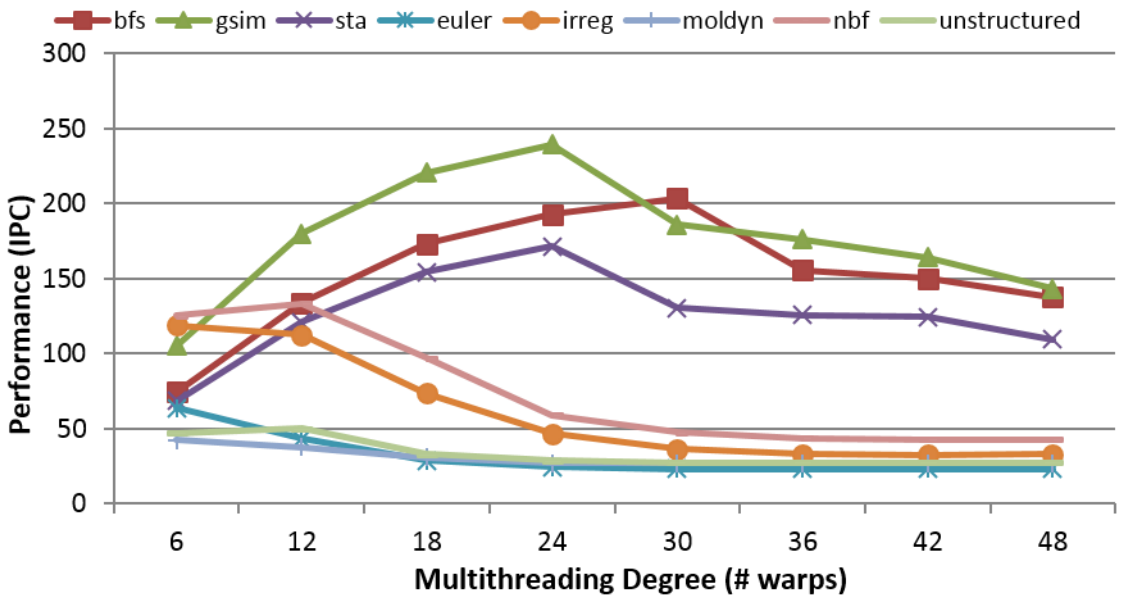


Figure 2: Performance variation of BFS bench with cache sizes and multithreading degree

According to the above observations, the performance of a parallel application on a GPGPU can be optimized by tuning the multithreading degree. This thesis proposes a multithreading decision scheme that observes the memory behavior of CTAs, and predicts the performance at each possible multithreading degree. The proposed scheme will then set an appropriate multithreading degree for better system performance. In summary, this thesis makes the following contributions:

- This thesis demonstrates the performance benefit of the cache hierarchy and multithreading technique when executing the general-purposed workload on GPGPUs. It

also shows the trade-off between multithreading technique and caching behavior when they are all applied.

- This thesis observes that an improper multithreading degree would not only increment cache misses and DRAM reads, but also the execution time of CTAs and the latencies of DRAM accesses. The experiments illustrate the relation between the multithreading degrees and the performance effects. The correlation information is then used to estimate the cache status and overall performance (IPC) at different multithreading degrees based on the status of the single active CTA.
- This estimation would be executed every time workloads are launched, and it will pick a proper multithreading degree that achieves better overall performance. Comparing to the default multithreading degree, the proposed scheme can improve the overall performance of in-lab benchmarks by 60% in average

According to the above observations, the performance of a parallel application on a GPGPU can be optimized by tuning the multithreading degree. This thesis proposes a multithreading decision scheme that observes the memory behavior of CTAs, and predicts the performance at each possible multithreading degree. The proposed scheme will then set an appropriate multithreading degree for better system performance. In summary, this thesis makes the following contributions:

2. Background

2.1 Baseline GPGPU architecture

The baseline GPGPU architecture used in this thesis is shown in Figure 3. It is modeled by a cycle accurate GPGPU simulator, GPGPU-Sim [13]. The configurations of the architecture adopt NVIDIA's Fermi GTX480 [24]. More detail information of this simulator can refer to its manual [14]. The GPGPU model consists of 15 shader cores, a centralized *CTA scheduler*, a unified L2 cache shared by all shader cores, and 6 DRAM partitions. Each shader core has a 32-way pipeline, a private read only L1 cache, a CTA buffer, and a warp scheduler. The default size of the CTA buffer can accommodate 8 CTAs.

The private L1 data cache of each shader core is read-only. All the write requests would be view as write-misses and the target cache line would be evicted from the cache to the lower level memory hierarchy. The unified L2 data cache is writable, and it is divided into 6 memory partitions, which is connected to one controller and one DRAM partition. The cache lines of both L1 and L2 cache can be accessed in a single transaction. This feature is referred as *memory coalescing* while all the requests to the same cache line can be serviced at once. The cache line size is set to 128 bytes, designed for a memory requests from a warp with 32 threads. If some threads in a warp do not access data within the continuous addresses 128 bytes, there would be additional memory accesses until all the requests are fulfilled. These un-coalesced accesses would increase the memory access latency and waste cache space since some data in the cache line may be useless.

To comprehensively understand the performance effects of different multithreading degrees, this work first evaluates the system behavior with no L2 cache, and only enables the private L1 cache. This is mainly because the private L1 cache is easier to control and observe the effects of different multithreading degrees. The unified L2 cache needs to support all the

shader cores, therefore the multithreading from different shader cores together would have more complex and correlated effects. For instance, the inter-block data sharing issue is mainly determined by hardware scheduler at runtime and relatively difficult to predict. If the workloads have no inter-block data locality, then the behavior of L2 cache would be very similar to L1 cache, except for supporting memory requests from all the 15 shader cores by larger cache capacity. Therefore, to simplify the experiments, L2 cache is disabled in this work. A detailed baseline platform configuration is described in Table 8 at section 6. Except for L2 cache, the experimental environment is the same with the default configuration of GTX480 in GPGPU-Sim v.3.2.1.

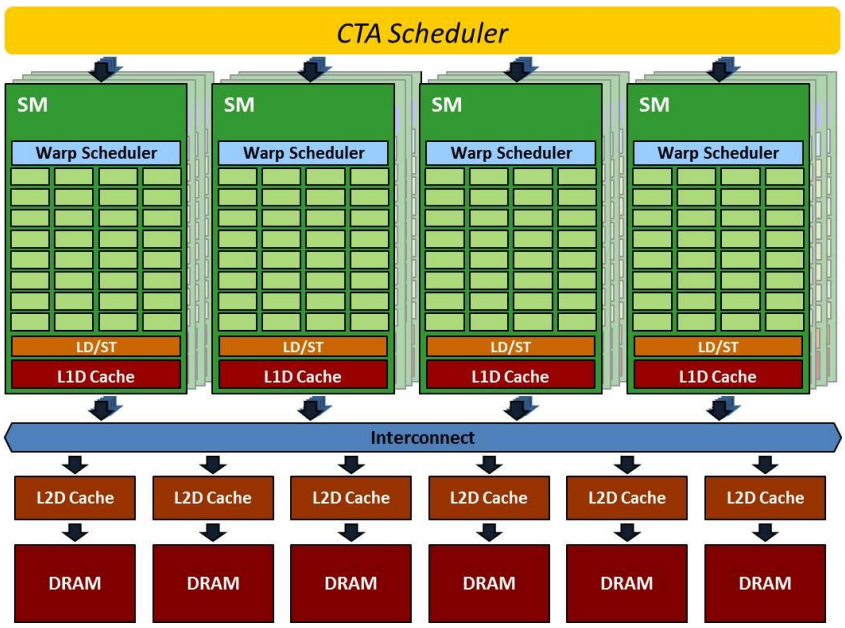


Figure 3: Baseline GPGPU architecture

2.2 CTA Scheduler and Warp Scheduler

The centralized CTA scheduler dispatches the CTA to shader cores. It is simulated as the NVIDIA GigaThread Scheduler in the Fermi architecture. Before CTA dispatching, the scheduler checks the maximum number of CTAs that can be supported by each shader, including the number of registers and the capacity of shared memory. For example, if a CTA needs 20 registers, and there are total 60 registers in one shader core, the maximum number of CTAs is 3.

The warp scheduler of each shader core controls the instruction issuing. It decides the order of warp selection in the CTA buffer, and it checks the status of instructions belonging to the selected warp whether some of them are ready to be issued. One common warp selection order is the round-robin scheme, and other scheduling algorithms are proposed for different purposes. For example, Jog et al. [10] proposed a two-level round-robin scheduling policy to improve the efficiency of hiding instruction latency. Rogers et al. [8] proposed a scheduling scheme to capture the intra-CTA data locality. This work only modifies the CTA scheduler to achieve the goal of controlling multithreading degree of each shader core. The warp scheduling policy is greedy-then-oldest (GTO) scheme, which is implemented by the GPGPU-Sim. The GTO is a widely used scheduling scheme [9] and performs well compared to the simple round-robin scheduling scheme.

2.3 Workloads and Metrics

In this work, some synthetic benchmarks are used to demonstrate the benefits of cache hierarchy design and the multithreading technique. The proposed decision scheme of multithreading degree is evaluated by some GPGPU applications that are implemented by CUDA programming interface, including in-lab benchmarks [19] and Rodinia [15, 18].

The in-lab benchmarks target on massive memory access pattern, and the working set sizes of some benchmarks are bigger than the L1 data cache, inflicting high pressure. The details of this bench suite are shown in Table 1. Another bench suite, Rodinia benchmarks, comes from many fields, including image processing, data mining, scientific simulation, and linear algebra. These benchmarks target on computation intensive workloads with relatively smaller pressure on the memory system. Table 2 shows some details of Rodinia benchmarks. Parboil [16] suite is another famous GPGPU bench suite, but it is not suitable for the performance evaluation in this work because its kernel implementation can only support up to 80 CTAs. The kernel functions of Parboil only execute single iteration of CTAs. This feature does not fit our experiment setup since the proposed approach in this work would evaluate the multithreading degrees in the first iteration, and then decide the best multithreading degree for the following iterations of CTA computation.

This work focuses on the overall performance improvement that is measured by instructions per cycle (IPC). The multithreading degree is defined as the total number of active warps as shown in equation (1)

$$multithreading\_degree = (\text{effective CTA buffer size}) \times (\#\text{warps per CTA}) \quad (1)$$

Table 1. In-Lab Massive Irregular Memory Parallel Applications [9]				
Applications	**Domain**	**Descriptions**	**Source**	**Data Set Sizes**
bfs	Electronic Design Automation	bread first search	Kuo et al.[19]	2.6 MB
sta		Static timing analysis		3.0 MB
gsim		gate-level logic simulation		3.5 MB
nbf	Molecular Dynamics	kernel abstracted from the GROMOS code	Cosmic [20]	6.3 MB
moldyn		force calculation in the CHARMM program		10.2 MB
irreg	Computational Fluid Dynamics	kernel of Partial Differential Equation solver		6.3 MB
euler		finite-difference approximations on mesh	Chaos [21]	8.5 MB
unstructured		fluid dynamics with unstructured mesh		10.2 MB

Table 2. Rodinia Benchmarks [15, 18]			
Applications	**Domain**	**Descriptions**	**Data Set Sizes**
BFS	Graph algorithm	Graph Traversal	3.7 MB
Gaussian	Linear Algebra	Dense Linear Algebra	192 KB
Hotspot	Physics simulation	Structured Grid	5.1 MB
Needleman-Wunsch	Bioinfomatics	Dynamic Programming	4 MB
Pathfinder	Grid traversal	Dynamic Programming	9.5 MB
Srad_v2	Image Processing	Structured Grid	4 MB

3. Related Work

In this section, some previous researches that focused on the CTA/warp scheduling scheme will be discussed. They targeted on different issues, including branch divergence, hiding memory latency, and cache locality optimization. Only few of them address the trade-off between multithreading and cache contention.

Cheng et al. [2] transformed the CMP applications into the streaming programming model that decouples computation and memory accesses. The approach in [2] extracts tasks from a loop that can be entirely unrolled. The execution flow of a for-loop is decoupled to a memory task and a computation task. This decouple scheme is inspired from the CUDA programming interface, which allows higher controllability of both computation and memory tasks. The memory tasks and the computation tasks are all equally-sized. The computation task and memory task decoupled from the same loop is bound as a dispatching unit. Then, the analytical decision scheme monitors memory bandwidth demand of memory tasks, and it estimates the latency of memory accesses considering different level of memory parallelism. The analytical model adjusts the parallelism degree gradually based on the execution time of memory tasks. In contrast to the work in [2], the work in this thesis focuses on the correlation between the multithreading degree, system performance, and cache behavior. This work uses the correlation to predict the performance at different multithreading degrees. The approach in this work does not need a progressive adjustment scheme, and therefore it takes much lower overhead for the adjustment progress.

Hong and Kim [3] proposed a performance analytical model for GPGPUs. They discussed the efficiency of memory latency hiding by the multithreading technique. The key component of this approach is to estimate the number of parallel memory requests by considering the number of running threads and memory bandwidth. Based on the degree of

memory parallelism, their model evaluates the system runtime considering the multithreading effect, coalescing of memory accesses, and synchronization overhead. However, their work did not consider the possible side-effect of multithreading, such like memory bandwidth limit that could lengthen the latencies of memory accesses. In addition, their work did not consider the cache hierarchy, and there is no discussion about the overheads of cache conflicts. This thesis gives a comprehensive discussion on various design concerns and trade-offs between cache performance and multithreading degree. The key issues discussed in this thesis includes the cache conflicts, the increment of memory requests, the latencies of memory requests and the effect of pipelining. These aspects are not addressed in the work of [3].

Fung et al. [4,5], Narasiman et al. [6], and Meng et al. [7] proposed warp scheduling schemes to solve branch divergence issues on SIMD GPGPU computation. Their approaches dynamically reformulate warps from all ready threads with the same PC (Program Counter), which expects warps to fully utilize all pipeline lanes. The work [6] also targeted on the improvement of hiding memory accesses through multithreading. They modified the warp hierarchy as a two-level design, and each warp has different priority to issue instruction. It prevents all warps from reaching the long latency instruction concurrently. In this thesis, the branch divergence is not under consideration, and the built-in warp scheduling policy and branch control on GPGPU-Sim is adapted directly.

Rogers et al. [8] proposed a warp scheduling scheme considering the data reusing on data caches, named as *Cache Conscious Wavefront Scheduling* (CCWS). It uses additional on-chip storage to record the reusing status of data on each cache line, and scores all warps in the CTA buffer for prioritization. They concerned the cache contention issue since the data locality can be destroyed by multithreading when cache capacity is not enough. They implemented an effective approach to control the multithreading scheme. Even though they demonstrated the effect of multithreading on their system performance, there is no further

discussion about the trade-off between multithreading and cache contention. This thesis does not modify the warp scheduler and the data locality improving scheme, but controls the multithreading degrees through CTA scheduler. Besides, this thesis demonstrates various trade-offs between mulithtreading degrees and the contention in memory system.

Kuo et al. [9] proposed a cache contention aware locality optimization scheme on GPGPU by software. They formulate the cache conflict issue of the unified last level cache, and proposed methods to pack CTAs, whose total working set size is smaller than a bound related to the size of last level cache, as a scheduling step. Scheduling the CTAs in a scheduling step would not harm the cache locality, and it blocks the CTA dispatching from other scheduling steps unless the current scheduling step is finished. To implement this approach, it takes off-line profiling of the CTA working set sizes, and the bound of the scheduling steps needs to be adjusted by designers. Unlike the static approach in [9], this thesis proposes an approach to dynamically adjust the multithreading degree based on a decision model that considers the relation between system performance and multithreading degrees.

Jog et al. [10] proposed a work that discusses several performance issues on warp scheduling scheme of GPGPUs, including CTA-aware two-level warp scheduling, locality aware warp scheduling, DRAM bank level scheduling, and the prefetching scheme on memory side. The proposed two-level warp scheduling policy bound a number of CTAs as a group. The key idea of their idea is to set different priorities for warps in different groups. A higher issue priority is set on warps in the same group. Unless all warps in the current group are idled, the scheduler only issue instructions from warps in the current active group. However, this approach is not effective since the total number of candidate warps is far from enough to hide the memory access latencies, which are usually around hundreds of cycles. In our work, a direct limit of multithreading degree is set in CTA scheduling policy based on a

prior estimation, and it demonstrates significant improvement in system performance of memory intensive benchmarks.

4. Analysis of Performance Effect from Multithreading

In this section, simulation results are used to demonstrate the multithreading effects. The detail simulator configurations are shown in Table 8, and some special settings are specified in each section. To simplify the discussion, this paper uses "*MD*" as the abbreviation of "Multithreading Degree".

4.1 The Performance Impact of Multithreading

This section first uses the simulation results of a linear algebra, vector addition, to demonstrate the benefit of multithreading on performance. The kernel code of the vector addition is shown in Table 3.

Table 3. Kernel Function of Vector Addition

```
1: int A [20480];
2: int B [20480];
3: int C [20480];
4: kernel_vec_add ( int *C, int *A, int *B ) {              // CUDA kernel function
5:       id = blockDim.x * blockIdx.x + threadIdx.x;       // per thread id
6:       C[id] = A[id] + B[id];
7: }
```

This workload is a memory-intensive benchmark because the computation time is much less than the latency of memory requests. The computation takes only a small part of the overall execution time in this benchmark, including an integer addition of 6 cycles, and an integer multiplication of 6 cycles with issue period of 2 cycles. Therefore, it is assumed that most execution cycles are consumed by memory tasks, and the multithreading enhances the performance by overlapping memory instructions. The performance at each multithreading degree is shown in Table 4. The performance improvement is almost proportional to the multithreading degree. Based on this runtime profiling, Figure 4 demonstrates the multithreading behavior on GPGPU-Sim. If the multithreading degree (*MD*) is 1 CTA, containing 2 warps, then these two warps can be executed almost concurrently. However, due

to the implementation limit, other CTAs need to wait until the active CTA is finished. In this example, the pipeline takes a long idle time due to the latency of memory accesses, but no other computation tasks can be executed at this moment. An improvement with MD = 4 (a CTA contains four warps) is demonstrated. It shows that four warps in a CTA can be executed concurrently, bring about 2x performance improvement. With MD = 6 warps, the performance is improved by about 3x compared to MD = 2. All these cases are ideal cases without considering memory bandwidth limit and pipeline resource competition.

Table 4. Vector Addition Simulation Results

MD	2 warps	4 warps	6 warps	8 warps	10 warps	12 warps	14 warps
Execution Cycles	197157	100867	67506	50857	40823	34682	29757
Norm. Cycles	1	0.51	0.34	0.26	0.21	0.18	0.15
# DRAM_Reads	1280	1280	1280	1280	1280	1280	1280
DRAM Access Latency	264	269	267	269	268	271	270
* 1 CTA = 2 warps							

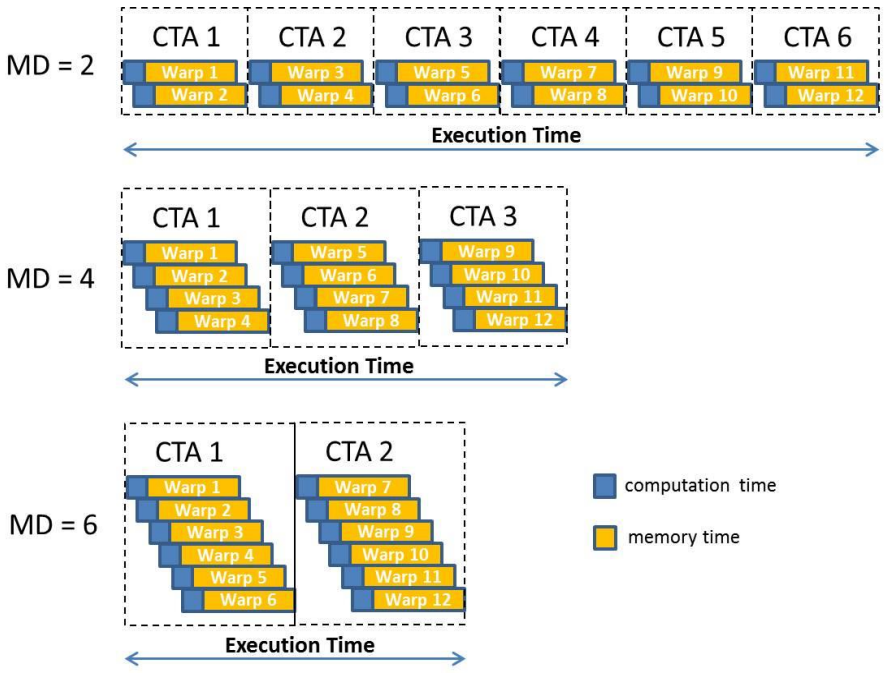


Figure 4: Timing of warp multithreading

Figure 5 demonstrates the multithreading effect on the overall performance of the in-lab benchmarks. To extract the essential impact of multithreading, L1 and L2 data caches are disabled to remove the effect from caches. The performance improvement of multithreading in this bench suite is not obvious. The benches, *bfs*, *gsim*, *and sta*, have smaller working set sizes, and therefore the multithreading enhances the performance when MD is smaller than 18 warps. There shows no performance enhancement when applying higher MDs. Moreover, the multithreading does not attain performance enhancement for benches with bigger working set sizes. The reason is that the limits of memory bandwidth and interconnection still cause the long data access latencies. The prolonged memory latency compromises the benefit of multithreading, and the performance does not have further improvement. Note that some experimental results of benchmarks are not shown since the benchmark reaches the multithreading bound due to insufficient registers.

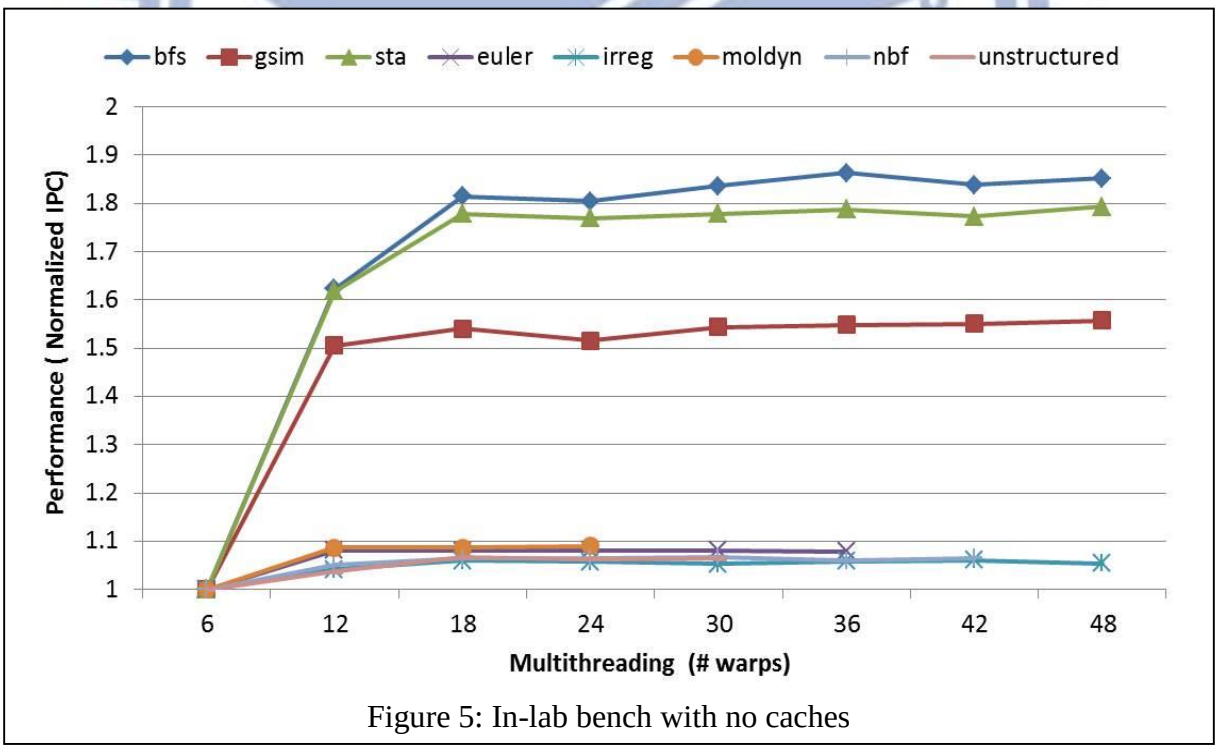


Figure 5: In-lab bench with no caches

4.2 The Effect on Cache Misses and DRAM Reads

The overheads of cache misses and DRAM reads are the main concern of the proposed decision scheme in this work. It is the most intuitive overhead of multithreading, and it is easy to be observed. More clearly, the decision scheme uses the number of DRAM reads to make decisions, since it represents the actual DRAM accesses, which take long latencies to fetch data into the cache.

The increment of DRAM reads is due to the conflicts in caches. For example, some in-cache data might be evicted when the new data is fetched due to multithreading. If the evicted data is still useful for the later execution, extra DRAM reads are needed.

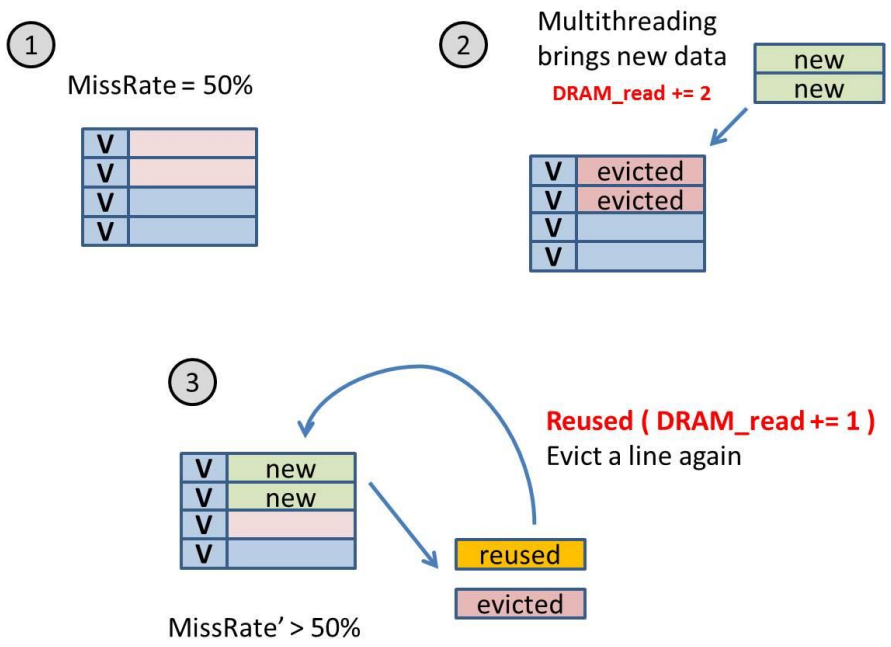


Figure 6: Multithreading adverse effect on cache misses and DRAM reads

Figure 6 demonstrates a possible situation that increases the DRAM reads when cache misses occur. In step 1, there is a cache with 4 lines and miss rate 50%, and 2 cache lines will be evicted since new data is fetched. In step 2, two new data are fetched, DRAM reads are increased by 2. The two corresponding cache lines have been evicted to accommodate the new data. In step 3, one of the evicted data will be reused based on the hit rate approaching to

50%, and it is fetched again from DRAM. The DRAM reads is increased by 1 again.

Cache line evictions do not always harm the performance except for the evicted cache line will be reused. Data reusing status is useful to evaluate the probability of data refetching. This work uses *cache_miss_rate* as an indicator of data reusing. Based on the *cache_miss_rate*, it is assumed that each cache line has the same probability, (*1-cache_miss_rate*), to be accessed again. It infers that if one cache line is evicted now, the cache line has the probability of (*1-cache_miss_rate*) to be reused again. In this case, a DRAM fetching is needed in the neat future. This is the key reason why DRAM_read is increasing. Step 2 in Figure 6 shows other factors, including cache capacity and additional working set sizes with higher multithreading degrees. When additional data is fetched, a cache needs to empty some lines to store new data. Therefore, a bigger working set could cause a cache to evict more reusable lines. Table 5 shows some profiling data that are used as the key indicators in the prediction of DRAM accesses. This work develops the prediction model with a segmented curve fitted to the observed data.

Table 5. Results of the increment of DRAM reads

# CTAs	*bfs*			*nbf*			*unstructured*		
	miss rate	*dram_r*	dr_inc_rate	miss rate	*dram_r*	dr_inc_rate	miss rate	*dram_r*	dr_inc_rate
1	0.41	72.94	(72.94)	0.13	169.39	(169.4)	0.22	1557.52	(1557.5)
2	0.40	71.12	0.98	0.22	439.13	2.59	0.44	3406.75	2.19
3	0.40	71.16	1.00	0.33	763.62	1.74	0.56	4384.41	1.29
4	0.40	71.64	1.01	0.45	1125.74	1.47	0.63	5007.40	1.14
5	0.41	73.03	1.02	0.54	1419.73	1.26	0.67	5341.11	1.07
6	0.43	77.90	1.07	0.58	1545.29	1.09	0.67	5341.11	1.00
7	0.46	83.01	1.07	0.59	1570.32	1.02	0.67	5341.11	1.00
8	0.49	89.29	1.08	0.59	1570.32	1.00	0.67	5341.11	1.00

* 1 CTA = 6 warps

** dr_inc_rate = dram_r'/dram_r

4.3 The Effects of DRAM Access Latencies

According to baseline configurations of GPGPU-Sim (Table 8 in section 6), the initial memory access latency is the sum of the following latencies: *rop_latency* + *dram_latency* + *dram_chip_access_latency*. These three types of latencies accumulate a total of 250 cycles. However, the real memory access latency when executing real applications would vary depending on the characteristics of applications. For our in-lab memory intensive benchmarks, the memory access latencies are around 350 cycles when the multithreading degree is set to 6 warps.

There are two possible reasons causing longer memory access latency. The first one, as demonstrated in section 4.1, is the memory bandwidth limit. This issue (memory bandwidth limit) involves many factors, including interconnection, memory bank-level parallelism, request queue size, and etc. The second reason is the cache allocation fail. The cache allocation fail occurs when all cache lines are reserved to wait for the arrival of newly fetched data back. In this case, the new-issued memory request to DRAM would be blocked until some cache lines become non-reserved. This situation prevents the possibility of overlapping DRAM accesses. The DRAM latencies cannot be hidden and the overall latency would be considerably prolonged since the new memory instruction needs to wait for the finish of a pervious DRAM fetch.

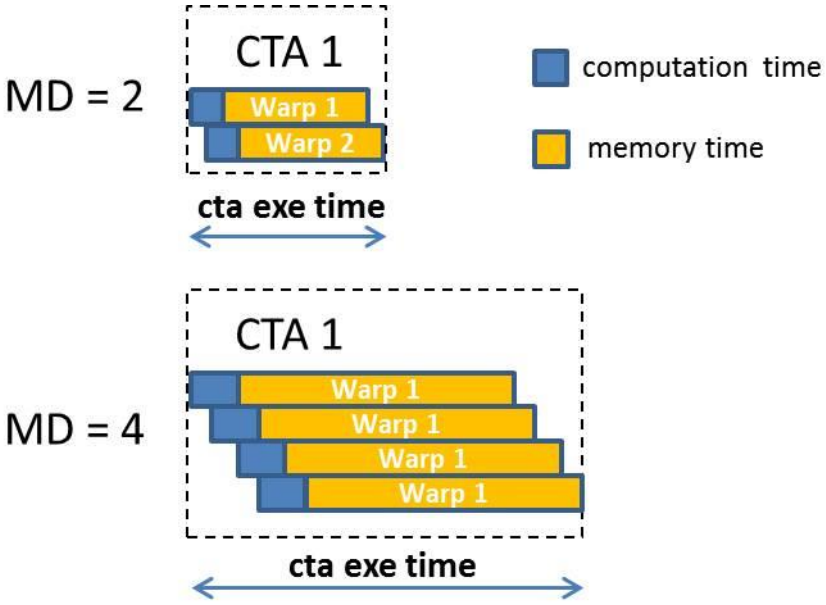


Figure 7: Memory time is extended

Table 6 shows the profiling of execution time and the increment of DRAM reads of a CTA in three benches, *bfs*, *nbf*, and *unstructured*. In most cases, the average execution time of a CTA becomes longer when the multithreading degree is higher, due to resources sharing by more CTAs. So, it assumes that this adverse effect of multithreading is due to the increment of DRAM reads. However, the profiled result shows that the adverse effect of more DRAM reads is not enough, which means the increasing rates of CTA execution times are bigger than DRAM reads growing.

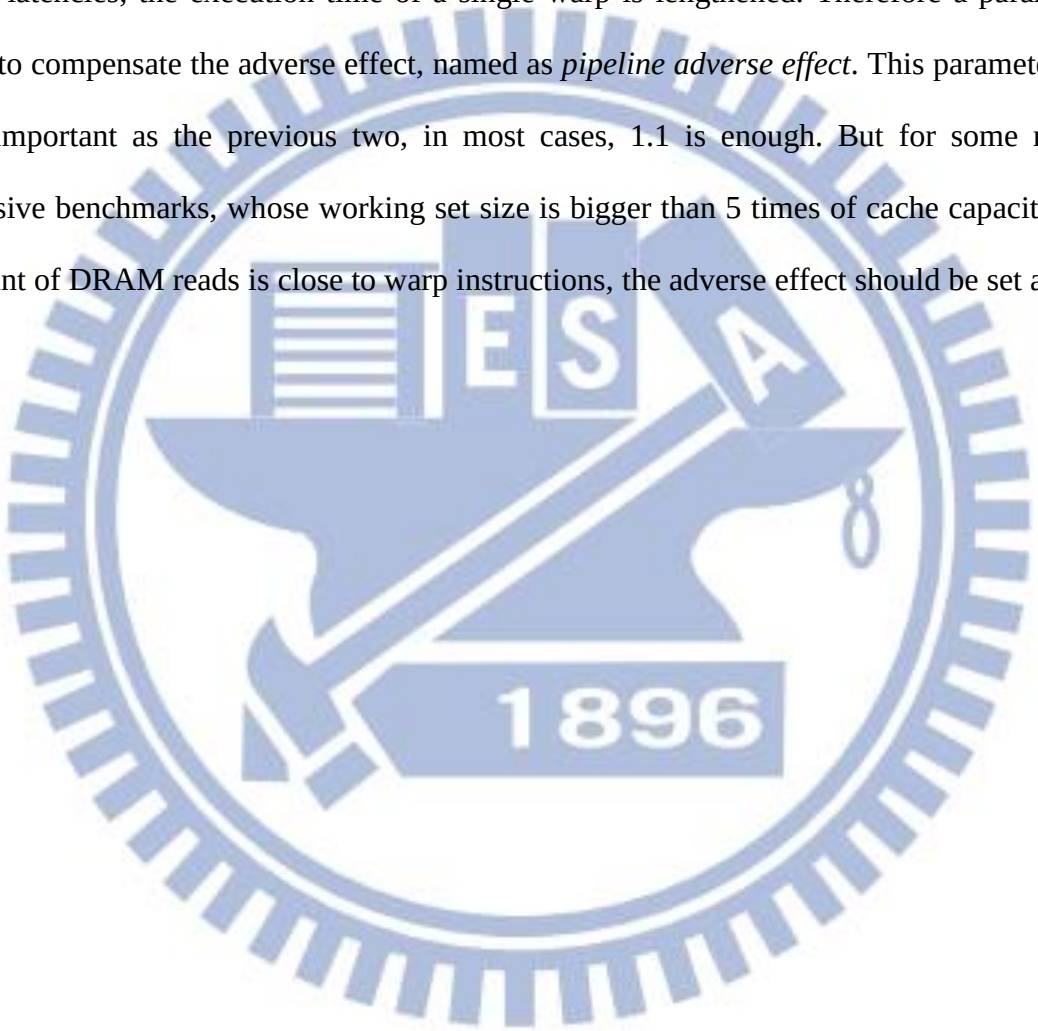
Table 6. DRAM Reads vs Execution Time									
#CTAs	*bfs*			*nbf*			*unstructured*		
	cta exe time	ratio	dr_inc_rate	cta exe time	ratio	dr_inc_rate	cta exe time	ratio	dr_inc_rate
1	2582	(2582)	(72.94)	9161	(91671)	(169.4)	87292	(87292)	(1557.5)
2	2897	1.12	0.98	17261	1.88	2.59	164058	1.88	2.19
3	3351	1.16	1.00	35438	2.05	1.74	377487	2.30	1.29
4	3875	1.16	1.01	73959	2.09	1.47	566328	1.50	1.14
5	4601	1.19	1.02	121446	1.64	1.26	760938	1.34	1.07
6	7097	1.54	1.07	159845	1.32	1.09	915126	1.20	1.00
7	7831	1.10	1.07	190175	1.19	1.02	1065425	1.16	1.00
8	9412	1.20	1.08	217308	1.14	1.00	1217622	1.14	1.00
* 1 CTA = 6 warps, (*val*): the initial *value*									

The following results in Table 7 have demonstrated the adverse effect of longer memory latencies caused by improperly excessive multithreading. One of the main issues is that the *dr_inc_rate* * *mf_latency_rate* is greater than the *cta_exe_rate* in some cases, but smaller in many other cases especially when the growing ratio of *cta_exe_rate* is smaller than 2.

Table 7. DRAM Reads vs Execution Time									
#CTAs	*bfs*			*nbf*			*unstructured*		
	cta exe rate	dr_inc_rate	mf_latency	cta exe rate	dr_inc_rate	mf_latency	cta exe rate	dr_inc_rate	mf_latency
1	(2582)	(72.94)	(329)	(9161)	(169.4)	(374)	(87292)	(1557.5)	(302)
2	1.12	0.98	1.09	1.88	2.59	1.06	1.88	2.19	1.46
3	1.16	1.00	1.08	2.05	1.74	1.46	2.30	1.29	1.74
4	1.16	1.01	1.06	2.09	1.47	1.49	1.50	1.14	1.12
5	1.19	1.02	1.07	1.64	1.26	1.08	1.34	1.07	1.04
6	1.54	1.07	1.43	1.32	1.09	1.03	1.20	1.00	1.00
7	1.10	1.07	1.02	1.19	1.02	1.01	1.16	1.00	1.00
8	1.20	1.08	1.04	1.14	1.00	1.00	1.14	1.00	1.00
* 1 CTA = 6 warps, (*val*): the initial *value*									

4.4 The Effect of Pipeline Architecture

The product of adverse effects of DRAM reads and memory latencies is not enough for the adverse effect of CTA execution time as shown in Table 7. One supposition is that the latencies of most types of instructions are about 6 to 10 cycles. To cover these latencies, 10 warps for multithreading is enough. If the multithreading degree is higher than the pipeline ALU latencies, the execution time of a single warp is lengthened. Therefore a parameter is used to compensate the adverse effect, named as *pipeline adverse effect*. This parameter is not that important as the previous two, in most cases, 1.1 is enough. But for some memory intensive benchmarks, whose working set size is bigger than 5 times of cache capacity or the amount of DRAM reads is close to warp instructions, the adverse effect should be set as 1.2.



5. A Decision Scheme of Multithreading Degree

5.1 Decision Scheme of Multithreading Degree

The proposed decision scheme is based on three performance parameters, including DRAM reads, memory access latency, and effects of pipeline architecture. The flow diagram of multithreading decision scheme is shown in Figure 8. All these predictions are formulated as segmented curve-fitting issues. The predictions on these three parameters are performed by a hardware predictor. The detailed hardware support of the GPGPU architecture will be introduced in section 5.2.

As shown in Figure 8, the first step of this decision scheme is DRAM reads prediction. This is the one of the main factor of this decision scheme. It predicts the DRAM reads of each CTA at different multithreading degrees from 1 CTA to 8 CTAs. Cache miss rate, cache capacity, and the working set size of each CTA will affect the competition of cache. The cache miss rate is set to DRAM reads over total cache accesses. Here, a measurement, increasing rate of DRAM reads, is defined as equation (2), and it is a key indicator to represent the increment of DRAM reads.

$$\text{Increasing Rate of DRAM reads} = \frac{\#\ \text{DRAM reads at MD=N}}{\#\ \text{DRAM reads at MD=(N-1)}} \quad (2)$$

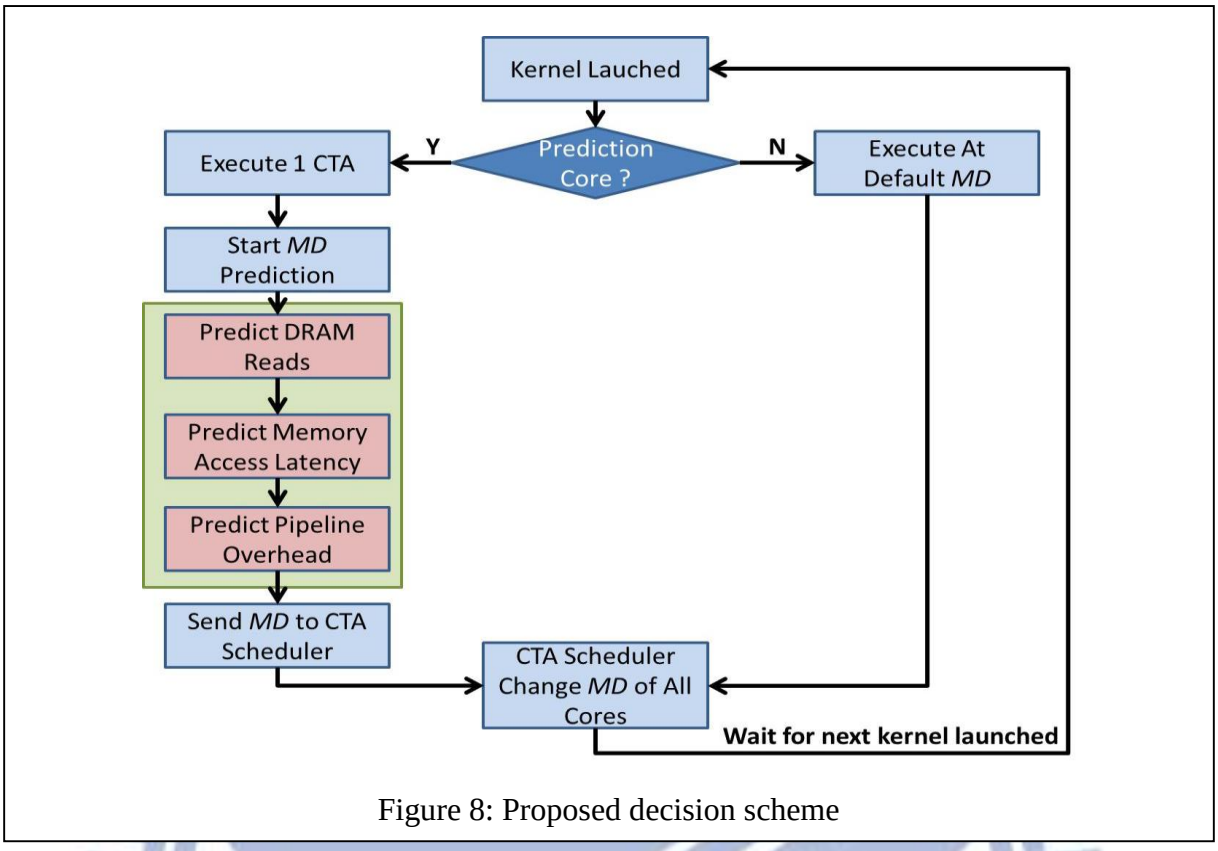


Figure 8: Proposed decision scheme

Figure 9 shows the increasing rate of DRAM reads by different cache miss rate with similar working set size and 8KB L1 cache. The working set size of a *bfs* CTA is about 70 cache lines, and the working set size of an *nbf* CTA is about 65 cache lines; 8KB L1 cache contains 64 cache lines. The data of benchmark, *bfs*, is shown in red color, and the blue data comes from *nbf* benchmark. The most important information in this figure is at *MD* = 1 CTA and *MD* = 2 CTAs. At *MD* = 1, the numbers of DRAM reads per CTA of both benchmarks are similar, but the cache miss rate of *nbf* is much lower than *bfs*. When *MD* increase to 2 CTAs, the total working set size of both benchmarks are doubled. However, the additional working set size comes from another CTA leading to different effect, a lower miss rate making the amount of DRAM reads increase more than two times, reaching about 4 times more.

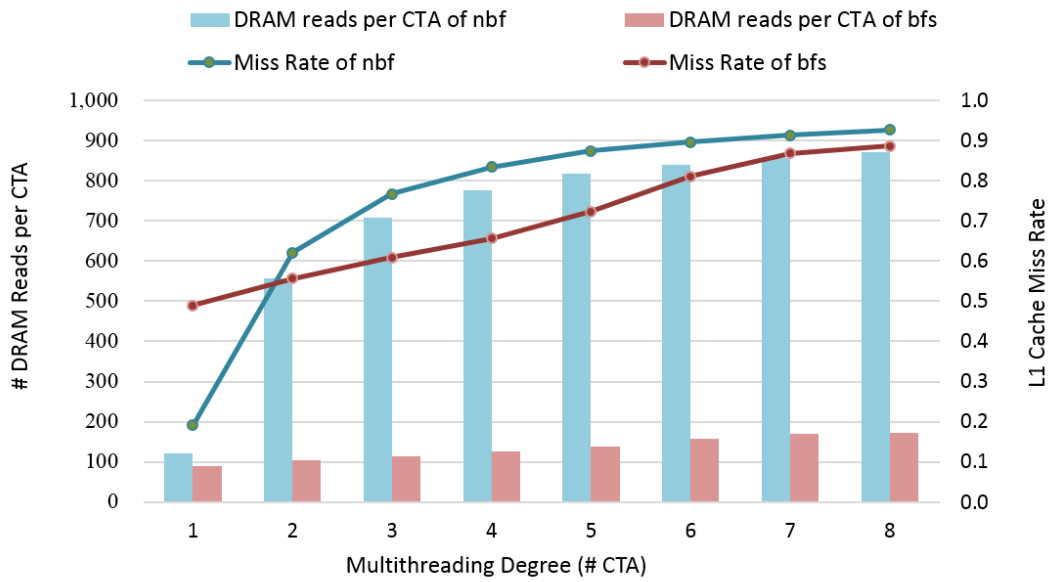


Figure 9: Variation on cache miss rate and DRAM reads as multithreading degree changes

Figure 10 shows the effect of cache capacity. The data is profiled from *bfs* benchmark executed at different L1 cache size. If the same workload executed on caches with different size, the increasing rate of both DRAM reads and cache misses would be lower with a bigger cache capacity, especially when the working set size is bigger than the cache capacity.

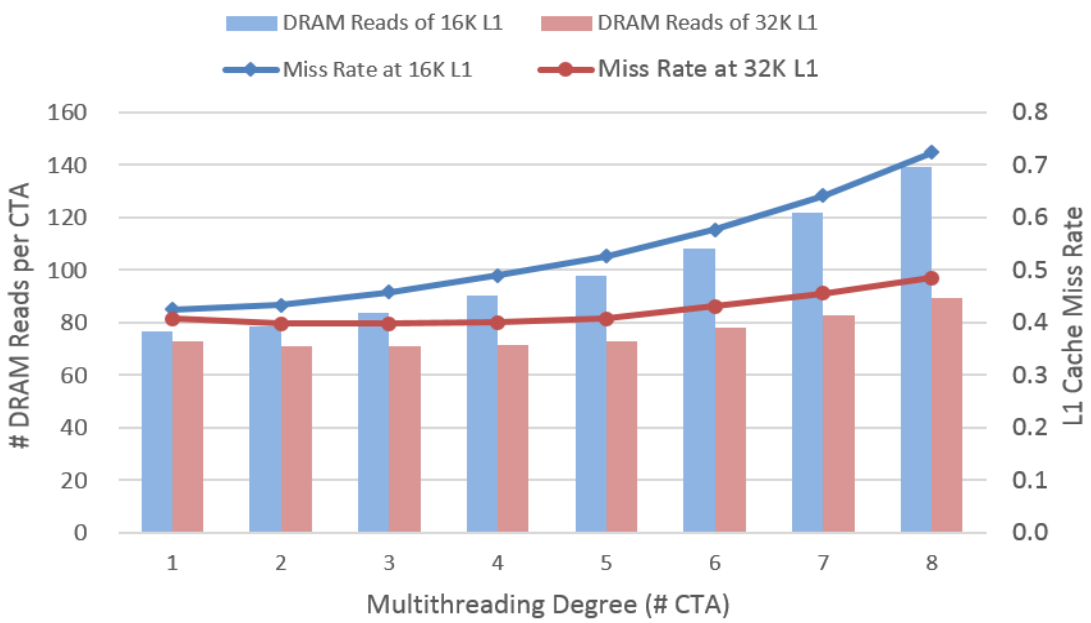


Figure 10: The effect of cache capacity on DRAM reads and cache miss rate when multithreading degree increase

Figure 11 demonstrates the effect of working set size increasing along with multithreading degree rising. The data is profiled from *bfs* benchmark but with different size of CTAs, one is 3 warps, another is 6 warps. In the case of 6 warps per CTA, the increasing rate of either cache miss rate or DRAM reads are higher than the case with smaller working set size.

Based on these three factors, a basic prediction scheme for DRAM reads can be built. However, to fit for more general purpose applications, more runtime data is needed for off-line model building.

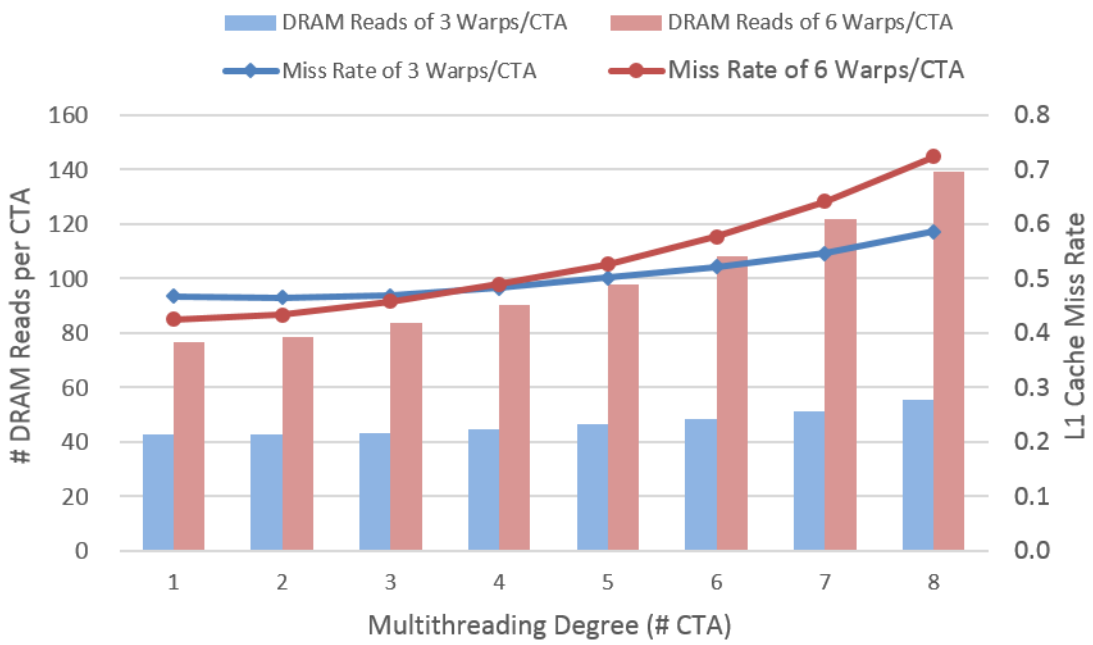


Figure 11: The effect of cache capacity on DRAM reads and cache miss rate when multithreading degree increase

The second prediction stage, latency prediction, takes the predicted miss rate, DRAM reads, and cache capacity as the decision criterions. Before start the decision flow, it needs to decide whether the workload is computation bound or memory bound based on the ratio of (# DRAM reads / # warp inst). From the profiling results, the ratio is set to 0.05. For memory intensive benchmarks, the decision scheme can start directly. If the miss rate is lower than 0.15 or higher than 0.85, the adverse effect of growing latency is small. In this situation, the growing rate is set as 1.03. If the miss rate is lower than 0.3 and higher than 0.15, the growing rate is set as 1.1. For other cases, if the working set of CTA is larger than 5 times of cache capacity and the delta of miss rate is larger than 0.1, the growing rate is set to 1.9. If the working set size is larger than 15 times of cache capacity and the delta miss rate is larger than 0.04, the growing rate is set to 1.75. For other cases, the growing rate is set as 1.1. These settings are attained from observations of extensive experiments.

The third stage is to predict the effect of pipeline architecture. It checks whether the working set size at the target multithreading degree is large than the cache capacity. If the working set size is lower than the cache capacity, the effect ratio is 1.05. For other cases, if the DRAM reads of each CTA is larger than 5 times of cache capacity, the ratio is set to 1.2, or otherwise 1.1.

5.2 Hardware Support on GPGPU Architecture

There require two additional components on GPGPUs to achieve dynamic tuning on multithreading degrees when each kernel launched. First, there needs one predictor on one of the shader cores. The multithreading degree of the shader core with predictor is set as one CTA with 6warps. Other shader cores apply the default multithreading degree, which is set to 8 CTAs. When the core with the predictor finishes the execution of the assigned CTA, it passes the collected statistics of cache accesses, DRAM reads, and the average memory access latency to the predictor. The predictor tries to attain a better multithreading degree by evaluating the memory access latency, increment of DRAM reads, and pipeline overhead. The second component is the *Multithreading Degree Controller* on the CTA scheduler in the CTA scheduler. When the evaluation is done, the decision will be sent to the *Multithreading Degree controller*. The CTA scheduler will control the number of CTAs dispatched to each shader core. The modifications are shown in Figure 12.

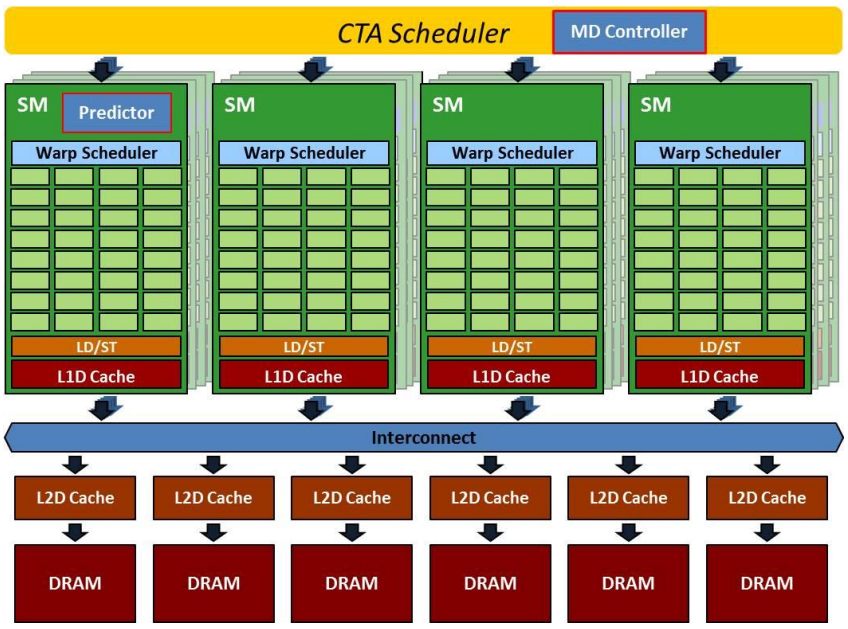


Figure 12: Proposed GPGPU architecture

6. Experimental Results

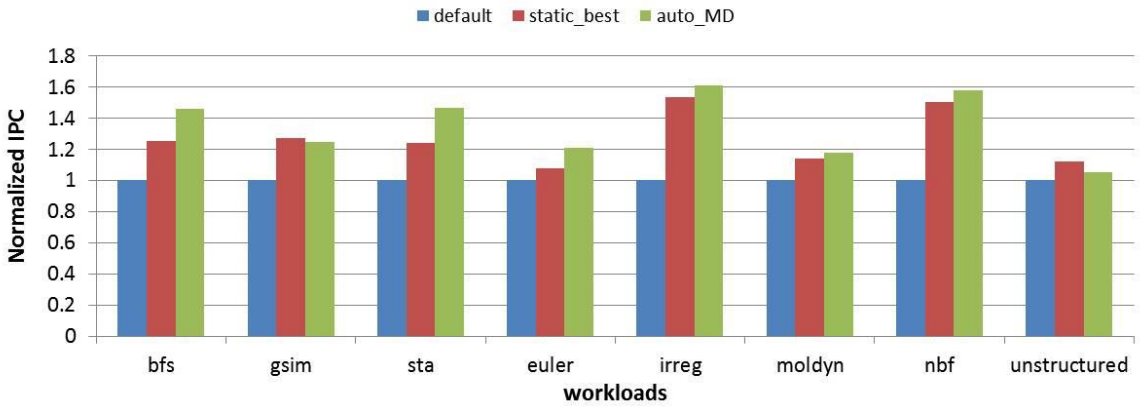
6.1 Experimental Setup

This work runs all the experiments on a cycle-accurate GPGPU simulator, GPGPU-Sim [13]. It uses the default configuration for NVIDIA's GTX480 GPGPU. Note that the L2 data cache is disabled to better observe the correlation between multithreading degrees and data access patters. Some key parameters are shown in Table 8. The benchmarks for performance evaluation include in-lab memory intensive benchmarks [19], Rodinia [15, 18], and Parboil [16].

Table 8. Baseline architecture configuration *	
Shader Core Configuration	1400MHz, SIMT width = 32 threads, total 15 shaders
Resources / Core	max 1536 threads, max 8 CTAs, 32768 registers
Caches / Core	4-way L1 data cache, 128B line size
L2 Cache	disable
Warp Scheduling	greedy-then-oldest (GTO)
CTA Scheduling	load balance scheduling / customization
Interconnection	700MHz, one stage fly topology (15 cores + 12 memory partitions),
DRAM Model	16 banks/partition, 4B partition bus width, FR-FCFS request scheduling (max 132 requests/mem controller)
DRAM Timing	924MHz, t_{CD}=2, t_{RRD}= 6, t_{RCD}=12, t_{RAS}=28, t_{RP}=12, t_{RC}=40, t_{CL}=12, t_{WL}=4, t_{CDLR}=5, t_{WR}=12,
Core to L2 latency	120 cycles
Core to DRAM latency	220 + DRAM chip access time
* use the default GTX480 configuration in GPGPU-Sim 3.2.1	

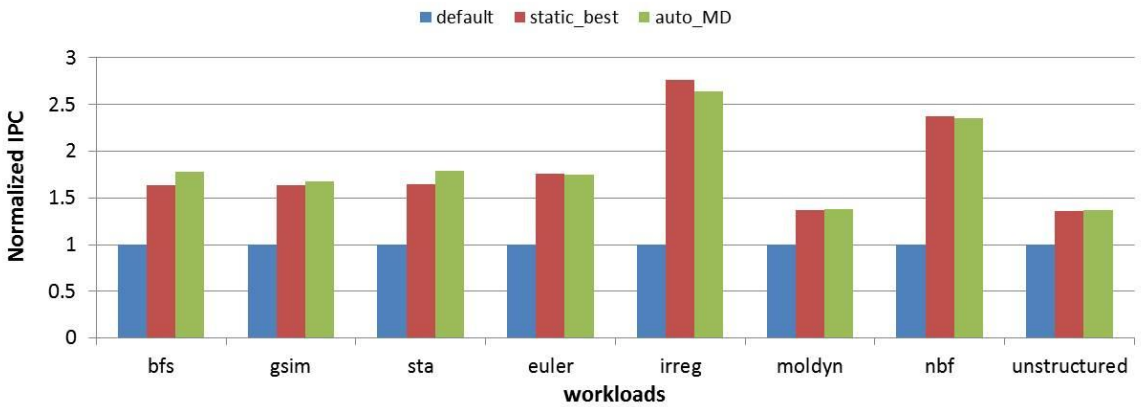
6.2 Experimental Results

Figure 13 to Figure 17 show the experimental results of the performance improvement and the predicted multithreading degrees of our in-lab benchmarks with different L1 data cache sizes, from 8KB to 128KB respectively.



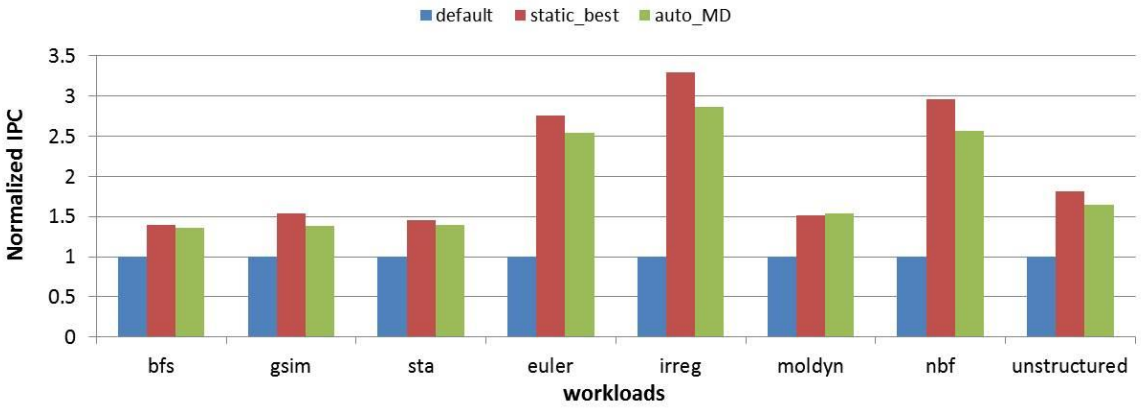
MD	bfs	gsim	sta	euler	irreg	moldyn	nbf	unstr
default	8	8	8	6	8	4	7	5
static best	4	3	4	1	1	1	1	2
auto_MD	4	4	4	1	1	1	1	1

Figure 13: Performance with 8KB L1 data cache



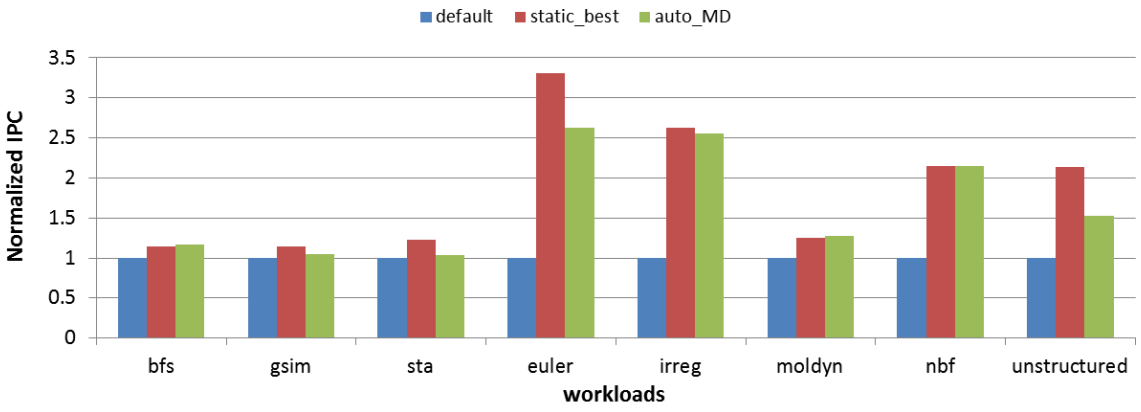
MD	bfs	gsim	sta	euler	irreg	moldyn	nbf	unstr
default	8	8	8	6	8	4	7	5
static best	4	3	4	1	1	1	1	1
auto_MD	4	4	4	1	1	1	1	1

Figure 14: Performance with 16KB L1 data cache



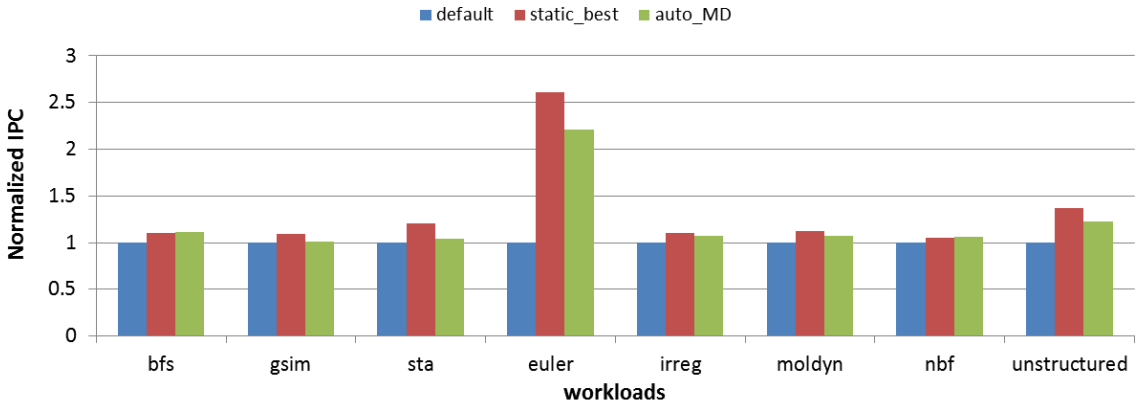
MD	bfs	gsim	sta	euler	irreg	moldyn	nbf	unstr
default	8	8	8	6	8	4	7	5
static best	5	4	4	1	1	1	2	2
auto_MD	4	4	4	1	1	1	1	1

Figure 15: Performance with 32KB L1 data cache



MD	bfs	gsim	sta	euler	irreg	moldyn	nbf	unstr
default	8	8	8	6	8	4	7	5
static best	4	3	4	2	3	2	3	2
auto_MD	4	5	6	1	3	1	3	1

Figure 16: Performance with 64KB L1 data cache



MD	bfs	gsim	sta	euler	irreg	moldyn	nbf	unstr
default	8	8	8	6	8	4	7	5
static best	4	3	4	3	5	2	5	3
auto_MD	4	5	6	4	6	1	6	2

Figure 17: Performance with 128KB L1 data cache

As shown in these figures, the proposed multithreading decision scheme works accurately in cases with smaller cache, i.e. 8KB, 16KB and 32KB; however, the predictions are not as accurate for cache sizes of 64KB and 128KB. The main reason is because the large cache capacity releases the pressure on memory system from memory intensive benchmarks. It makes their characteristics close to memory-computation-balanced workloads, and the current design of the decision scheme cannot handle this indistinct field. Even the accuracy of prediction decreases, the proposed decision scheme still catches a better multithreading degree than the default setting. It shows that the performance improvement reaches 67% when L1 cache is 64KB in Figure 16. With 128KB L1 cache, the performance improvement is about 23%, and the static best solution is about 33% in Figure 17.

The average performance improvement in the in-lab benchmarks is about 60%. The best performance improvement achieves 180% compared to the default multithreading degree. The prediction accuracy becomes lower for computation-bound benchmarks, and therefore

degrade the effectiveness of the proposed decision scheme.

Figures 18 to 22 demonstrate the performance of Rodinia benchmarks. This experiment shows the results of following benchmarks, including *BFS*, *gaussian*, *hotspot*, *nw*, *pathfinder*, and *srad_v2*. The other benchmarks are not used because ……. However, the proposed decision scheme cannot achieve better performance than the default CTA parallelism setting since the data access patterns of these benchmarks in Rodinia are already optimized. The contention on memory systems is minimized and therefore higher multithreading degrees can usually return the better performance.

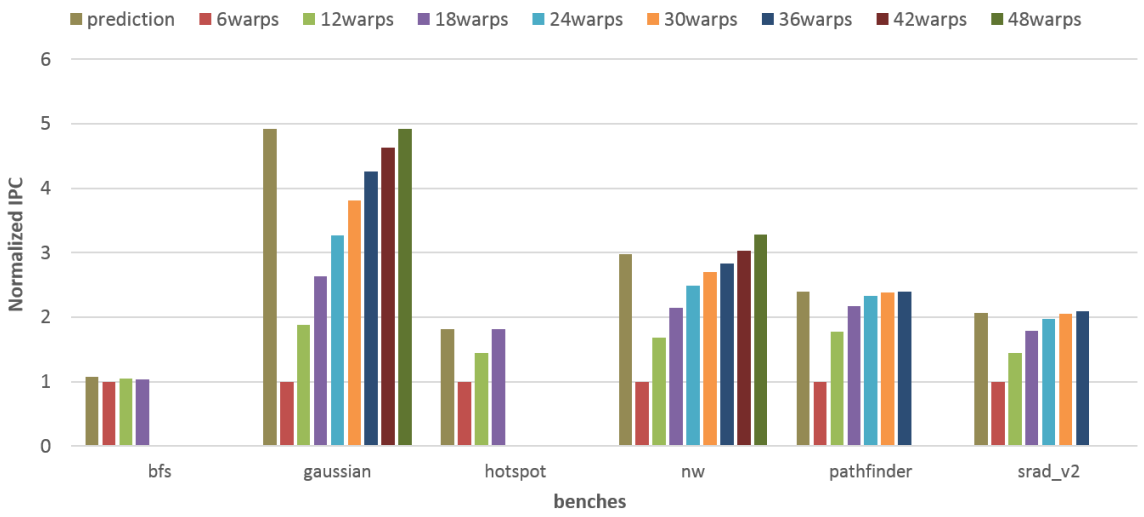


Figure 18: Performance of Rodinia Benchmarks with 8KB L1

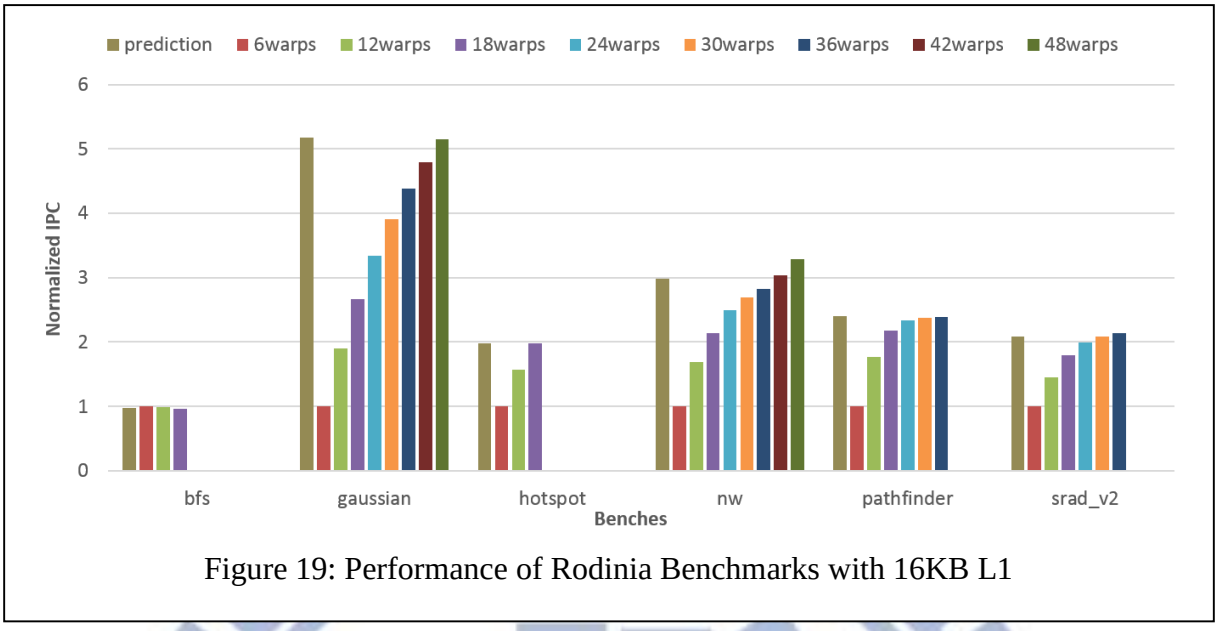


Figure 19: Performance of Rodinia Benchmarks with 16KB L1

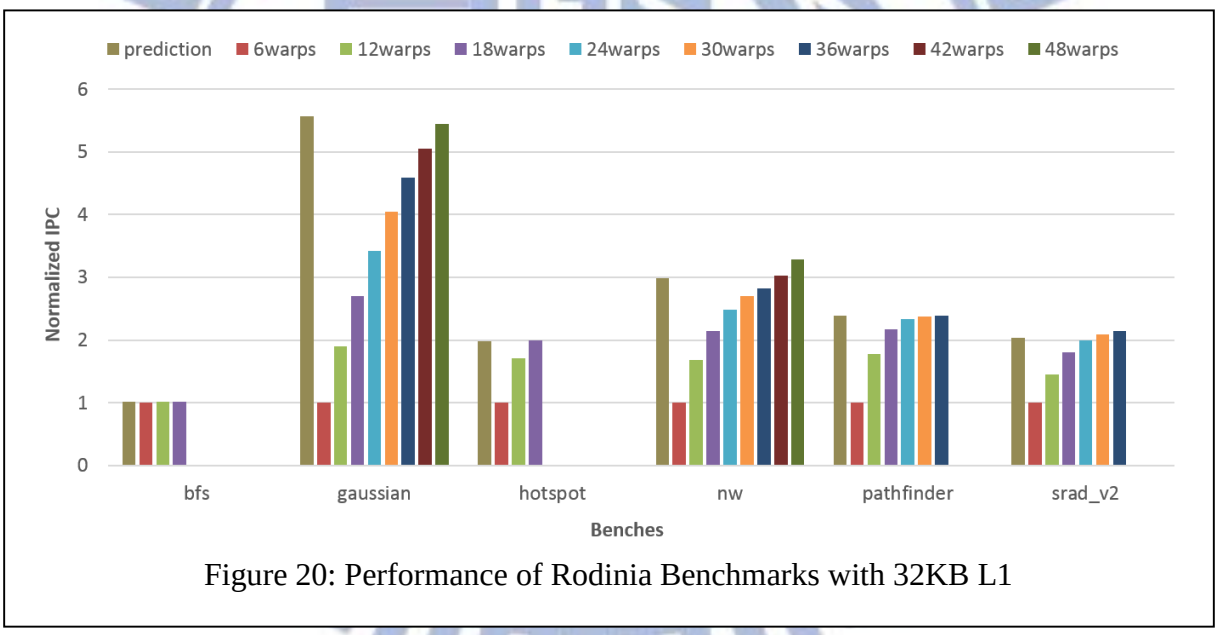


Figure 20: Performance of Rodinia Benchmarks with 32KB L1

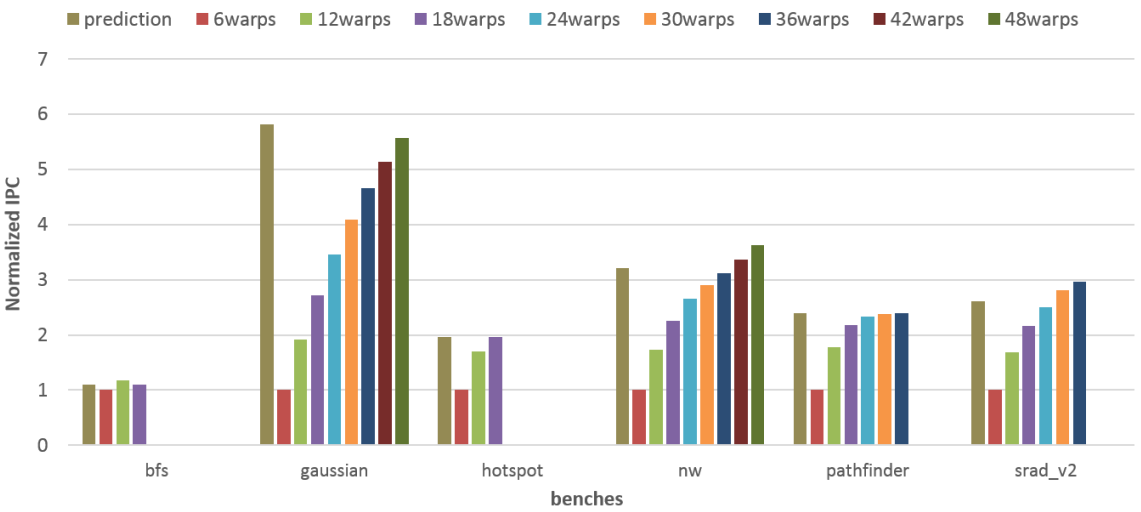


Figure 21: Performance of Rodinia Benchmarks with 64KB L1

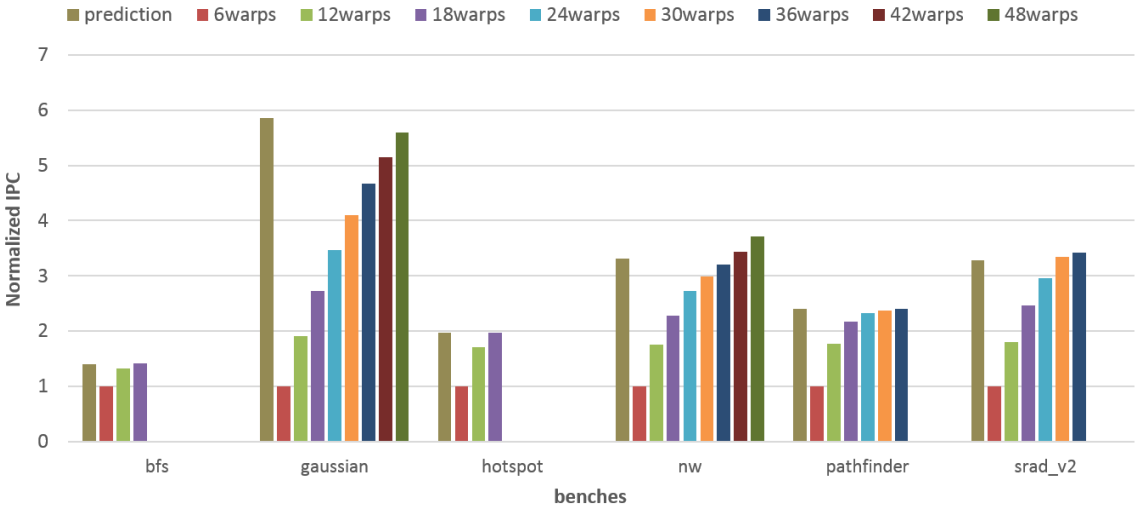


Figure 22: Performance of Rodinia Benchmarks with 128KB L1

6.3 Summary

These experiments demonstrate the effects of multithreading degrees on overall performance. In these benchmark suites (the in-lab suite and Rodinia suite), different multithreading degrees lead to different overall performance, and the selection strategy depends on the benchmark characteristics. For memory intensive benchmarks, a trade-off between multithreading degree and memory behavior is important. The proposed decision scheme can find a better multithreading degree for system performance in most cases, especially when there is high pressure on memory system. The decision scheme can also detect the computation-bound benchmarks, and change the decision scheme tending to higher multithreading degrees. The proposed scheme improves system performance of memory-intensive benchmark 60% averagely. It also tracks computation-bound benchmarks, and keeps them at higher multithreading degrees to efficiently utilize computation resources.

7. Conclusions

This work demonstrates the performance effects of multithreading on GPGPUs. The expriemntal results have demonstrated the performance enhancement by applying the multithreading technique. However, the results also illustrate the non-trivial trade-off between multithreading degrees and cache contention. To achieve the best performance, designers should consider the number and latencies of DRAM accesses, as well as the effects of pipeline architecture. Some hardware and software properties have also impacted the overall performance, including cache miss rate, cache capacity, working set size of each CTA, memory intensive or computation intensive workloads, and the latency of DRAM accesses.

Based on the extensive experiments and observations of these effects, this work proposed a multithreading auto decision scheme to make the trade-off between multithreading benefit and the associated effects. At first, a prediction scheme is needed to estimate the total DRAM reads of every multithreading degree. Second, based on the predicted number of DRAM reads at a multithreading degree, the proposed scheme evaluates the possible latency of DRAM reads. Third, the proposed scheme estimates the effect of pipeline architecture that cause the long execution time of single CTA. Finally, the proposed scheme applies these estimation and changes the multithreading degree to achieve the best system performance.

The experiments are based on two benchmark suites, including an in-lab irregular memory intensive benchmarks and the third-party Rodinia benchmarks. All the experiments are performed on a cycle accurate GPGPU simulator, GPGPU-Sim. The experimental results show that the proposed decision scheme can accurately catch the characteristics of applications and return better performance. The overall performance improvement for in-lab suite is about 60% in average, and up to 180% for the best case. However, it shows limited performance improvement in Rodinia suite since the execution behavior of these benchmarks

are already optimized and there leaves limited room for the proposed decision scheme to attain further performance enhancement.

8. Reference

[1] Rafique, N., Won-Taek Lim, Thottethodi, M., "Effective Management of DRAM Bandwidth in Multicore Processors," *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on* , vol., no., pp.245,258, 15-19 Sept. 2007.

[2] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, Chia-Lin Yang, "Memory Latency Reduction via Thread Throttling," *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* , vol., no., pp.53,64, 4-8 Dec. 2010

[3] Sunpyo Hong, Hyesoon Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," In *Proceedings of the 36th annual international symposium on Computer architecture* (ISCA), 2009.

[4] Wilson W. L. Fung, Sham, I., Yuan, G., Tor M. Aamodt., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), 2007.

[5] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt, "Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware," *ACM Trans. Archit. Code Optim.* 6, 2, Article 7 (July 2009)

[6] 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* (MICRO), 2011.

[7] Jiayuan Meng, David Tarjan, and Kevin Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," In *Proceedings of the 37th annual international symposium on Computer architecture* (ISCA '10), 2010.

[8] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt, "Cache-Conscious Wavefront

Scheduling,” In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-45), 2012.

[9] Hsien-Kai Kuo, Ta-Kan Yen, Lai, B.-C.C., Jing-Yang Jou, "Cache Capacity Aware Thread Scheduling for Irregular Memory Access on many-core GPGPUs," *Design Automation Conference (ASP-DAC), 18th Asia and South Pacific* , vol., no., pp.338,343, 22-25 Jan. 2013

[10] 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 *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (ASPLOS), 2013.

[11] NVIDIA CUDA C Programming Guide v4.2, NVIDIA Corp., 2012.

[12] Khronos Group, “OpenCL,” http://www.khronos.org/opencl/.

[13] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in Proc. Of Int’l Symp. on Performance Analysis of Systems and software (ISPASS 2009), pp. 163–174.

[14] T. M. Aamodt et al., GPGPU-Sim 3.x Manual, http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual, University of British Columbia, 2012.

[15] S. Che et al., “Rodinia: A Benchmark Suite for Heterogeneous Computing,” In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct. 2009.

[16] J. A. Stratton et al., “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” 2012.

[17] NVIDIA CUDA C/C++ SDK code samples, NVIDIA Corp., 2013.

[18] S. Che et al., “A Characterization of the Rodinia Benchmark Suite with Comparison to

Contemporary CMP Workloads,” In Proceedings of the IEEE International Symposium on Workload Characterization, Dec. 2010.

[19] Kuo et al., "Thread affinity mapping for irregular data access on shared Cache GPGPU," ASPDAC, 2012.

[20] Han et al., "Exploiting locality for irregular scientific codes," Parallel and Distributed Systems, IEEE Transactions, 2006.

[21] Das et al., "Communication optimizations for irregular scientific computations on distributed memory architectures," Journal of Parallel and Distributed Computing, 1994.

[22] NVIDIA, “Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Fermi”, NVIDIA Corp., 2009.

[23] NVIDIA, “Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110”, NVIDIA Corp., 2012.

[24] NVIDIA, “NVIDIA GeForce GTX 480/470/465 GPU Datasheet”, NVIDIA Corp., 2010.