

國立臺灣大學電機資訊學院資訊工程學系

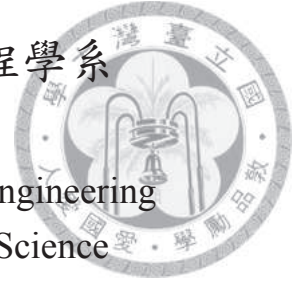
碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



透過位址預先計算以提升圖形處理器記憶體效能

Improving GPU Memory Performance via Address

Pre-computation

陳立展

Li-Jhan Chen

指導教授：楊佳玲 博士

Advisor: Chia-Lin Yang, Ph.D.

中華民國 105 年 8 月

August, 2016



致謝

碩士論文能夠順利成功完成，主要感謝的是我的指導教授楊佳玲老師，在一開始做研究，懵懂無知的時候，老師總是在研究上給了我很多正確的意見與方向，讓我能夠在研究的道路上繼續前進，除了研究上的細心指導之外，老師還經常教導我們做人處事的道理與做事情的方法，這兩年下來，讓我受益良多，在此對老師獻上最誠摯的感謝。此外要感謝口試委員，陳依蓉陳教授、呂仁碩呂教授以及陳坤志陳教授，於口試跟碩論方面提出了很多可以改進的部分，讓此篇論文更加完善。

在研究所生涯中，最特別感謝的是王柏翰學長，學長總是當我遇到研究上的瓶頸時，陪我一起討論，並給出很多寶貴的意見，引領我到正確的方向。此外感謝實驗室戰友老賴、立立跟老薛，大家總是一起討論、一起做事、一起打嘴砲、一起上廁所，讓我在研究所生涯中充滿歡樂的回憶。還有感謝實驗室學弟們，尤其是益昌，陪我一起討論關於機制的部分。最後，感謝家人們的支持與陪伴，在我飢腸轆轆的時候，煮飯給我吃，在我晚回家的時候，等我回家，在我情緒低落的時候，給了打氣加油，讓我有繼續奮鬥的勇氣，願此時此刻的喜悅能與你們一同共享!

陳立展謹識

于台大資訊工程所

中華民國一百零五年八月



摘要

高效能的通用圖形處理器主要源自於大量執行緒的平行化以及資源的充分使用。在通用圖形處理器的執行模型下，數以千計的執行緒用來達到高程度並行。然而，每個核心上卻只有一個很小的第一級快取來減少存取記憶體延遲時間。在大量的執行緒彼此競爭一個很小的第一級快取資源的情況下導致很差的快取效能並因此限制了系統的效能。

本碩論中，我們探索現有圖形處理器上的區塊排程器以及 warp 排程器的設計，我們發現現有的區塊排程器傾向於將使用到相同快取列分散至不同的核心上，而減少了快取列重複使用的機會。因此，我們設計區域感知區塊及 warp 排程器來提升快取效能。透過軟體和硬體的合作方式，讓區塊排程器能夠事先得知每個區塊會用到哪些快取列並將使用到相同快取列的區塊放在同一個核心上以增加快取列的重複使用的機會。此外，並藉由區域感知 warp 排程器細粒度的控制 warp 的執行順序來捕捉快取列重複使用的機會。實驗結果顯示，我們提出的區域感知排程器可以有效地提升快取效能，並比最先進的排程設計多提升約 10% 的整體效能。

關鍵字 — 通用圖形處理器、區塊排程器、Warp 排程器、第一級快取、高效能運算



Abstract

High performance computing on GPGPU is relied on maximizing thread-level parallelism and fully resource utilization. In GPU's execution model, thousands of threads are employed to achieve high level parallelism. However, only a small L1 cache resource is provided in each SM (streaming multiprocessor) to reduce memory access latency. Massive threads competing a small L1 cache resource causes poor cache performance and limits system performance.

In the thesis, we explore the current design of thread block and warp scheduler. We find current thread block scheduler tends to allocate thread blocks that use the same cache line to different SMs and reduce cache reuse opportunities. Therefore, we design Locality-Aware scheduler for improving GPU cache performance. Based on our proposed software and hardware cooperative method, cache line touched in a block can be known a priori so that thread block scheduler could put thread block with sharing cache line to the same SM for increasing cache line reuse opportunities. Besides that, locality-aware warp scheduler fine-grained controls execution order of warp to capture the cache locality. The result shows our Locality-Aware Scheduler could effectively improve cache performance and achieve 10% performance on average over the state-of-the-art scheduling policies.

Keywords — GPGPU, Block (CTA) Scheduler, Warp (Wavefront) Scheduler, L1 Data Cache, High Performance Computing



Contents

致謝	i
摘要	ii
Abstract	iii
1 Introduction	1
2 Background & Motivation	5
2.1 Background	5
2.1.1 GPGPU Architecture	5
2.1.2 Thread Block Scheduling	6
2.1.3 Warp Scheduling	7
2.2 Motivation	8
3 Locality-Aware Scheduler	11
3.1 Overview of locality-aware scheduler	11
3.2 Compiler Support	12
3.3 Locality-Aware Thread Block Scheduler	14
3.3.1 Thread-Block-Level Access Range Calculation	14
3.3.2 Thread-Block-Dispatching Decision	16
3.4 Locality-Aware Warp Scheduler	17
3.4.1 Warp-Level Access Range Calculation	17
3.4.2 Two-level Warp Scheduler	18

4	Experimental Methodology	22
5	Evaluation	24
5.1	Effect of thread block scheduler	25
5.2	Effect of warp scheduler	26
5.3	Pipeline stall reduction	27
5.4	Hardware Overhead	29
6	Related Works	30
6.1	Block Scheduling for improving cache locality	30
6.2	Warp Scheduling for improving cache locality	30
6.3	Improving resource utilization on GPUs	31
6.4	Improving thread-level parallel on GPUs	32
7	Conclusion	33
	Bibliography	34

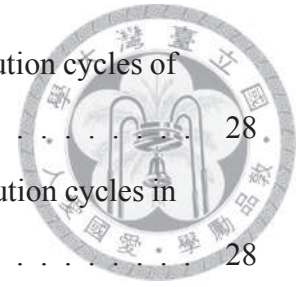




List of Figures

2.1	GPGPU Architecture	6
2.2	Data locality in workloads with different access behavior	9
2.3	Comparison of different thread block scheduling policies for row-major and column-major applications	9
3.1	Overview of our locality-aware scheduler	12
3.2	Address calculation code extraction	13
3.3	Simple address calculation code (Array-based data structures)	13
3.4	Example of memory access region in a thread block	14
3.5	Extended block queue entries	15
3.6	Byte address to line address transformation	15
3.7	Mapping address range of the block to the coordinate of cache line address	15
3.8	Flow of the thread-block dispatching decision algorithm	16
3.9	Locality estimation between two blocks	17
3.10	hierarchical warp encoding	18
3.11	Locality aware warp scheduler	19
3.12	Flowchart of the two-level warp dispatching decision	20
3.13	Example of the Locality degree table	21
3.14	Example of inter-warp locality computation	21
5.1	Comparison of performance for different policies, normalize to baseline architecture using GTO warp scheduler	25
5.2	L1D cache miss comparisons	26

5.3	Breakdown pipeline stall cycles normalize to baseline execution cycles of Type I applications	28
5.4	Breakdown pipeline stall cycles normalize to baseline execution cycles in type-II and type-III	28





List of Tables

2.1	GPU and CPU per thread cache resource	8
4.1	GPGPU-Sim Simulation Configuration	22
4.2	Workloads	23
5.1	Workload categories	24
5.2	Hardware overhead	29



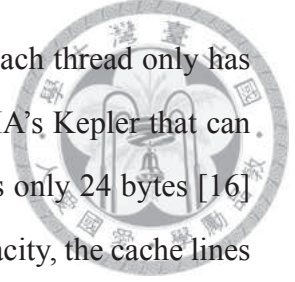
Chapter 1

Introduction

General Purpose Graphic Processing Units (GPGPUs) are becoming more and more popular and widely used in different areas such as image processing, physical based simulation, and cloud computing due to their significant computing capability [14]. Modern GPGPUs allow thousands of threads to be executed in parallel. In order to manage the massive threads on a GPGPU core and simplified hardware design, threads are typically grouped into thread blocks and each thread block are divided into a small group of threads, called warp. Threads within a warp are executed in lock-step, which means that they share the fetch and decode unit in the pipeline and are all executed on the same instruction at the same time. With multiple warps residing in a GPU core, the memory access latency can be hidden by the fast context-switch between different warps. Therefore, GPGPUs can offer better performance and power efficiency than CPUs.

In typical GPGPU execution model, a kernel, which is the minimum task unit on GPUs, is composed of multiple thread blocks. The thread block scheduler distributes those thread blocks to the Streaming Multiprocessors (SMs). In each SM, the thread blocks are further divided into a small scheduling unit—warp, the warp scheduler issues warp to the SIMD(single instructions, multiple data) lane for execution. Besides, each SM contains large amounts of memory resources such as register files, l1 data cache, shared memory resource etc to support the large number of threads for execution.

Despite the highly achievable thread-level parallelism, the GPU cores usually suffer from the serious cache contention. In each core of modern GPGPU architecture, thou-

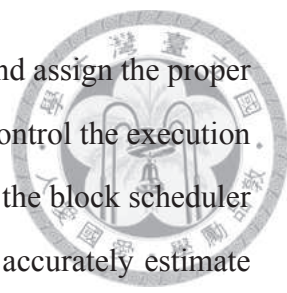


sands of threads share a small L1 cache resource, which means that each thread only has an extremely small L1 cache resource quota. For example, in NVIDIA's Kepler that can support up to 2048 threads in each core, per-thread cache resource is only 24 bytes [16] which is far less than the CPU thread has. With the limited cache capacity, the cache lines brought by one thread can easily be evicted by other threads, leading to serious performance degradation.

Prior studies have been proposed different warp schedulers to mitigate cache contention. G. Rogers et al. [19] propose cache-conscious warp scheduling which uses additional hardware to monitor cache thrashing behavior and then use warp throttling technique to reduce inter-warp interference for preserving intra-warp cache locality. DAWS [20] further preserves more intra-warp cache locality by using a predictor combined with profile-based and online detection information. Most of them try to throttle the number of warps that can access the cache resource to preserve the intra-warp locality. However, very few studies try to mitigate cache contention by putting the warps with cache locality on the same core. In addition to warp scheduling, recent work [11] also shows that thread block scheduling is another important factor to improve cache performance. It schedules two consecutive thread blocks to the same core because they observe that the two consecutive thread blocks usually access the data in shared cache lines. However, their work only focus on a specific data access behavior in various GPGPU workloads. For GPGPU applications with different data access behavior, a comprehensive approach is needed to exploit not only the cache locality between the two consecutive thread blocks but also the cache locality existing in different thread block combinations.

In this thesis, we propose the software and hardware cooperative method to improve cache performance. The main idea is to estimate the thread locality through address pre-computation. With the locality information, the thread block scheduler and warp scheduler within a SM (Streaming Multiprocessor) can then make smart decision for block/warp dispatches with the goal to optimize L1 cache performance.

To achieve the goal, compiler helps to extract the address information from the GPU program during compilation. By utilizing the address information, the thread block sched-



uler can estimate the cache locality between different thread blocks and assign the proper thread block to the SM. Moreover, warp scheduler can fine-grained control the execution order of warps to efficiently exploit the cache locality provided from the block scheduler based on the estimated warp locality. The key challenge is how to accurately estimate the locality between thread blocks and warps at runtime. First, we would need a metric to quantify cache access locality between thread blocks and warps. Second, determine what information is necessary for our metrics and how to get the required information. Third, how to design the block scheduler policy and warp scheduler policy to capture performance improve opportunities by exploiting the cache access locality.

The thesis offers the following contributions:

- We propose the first softwarehardware cooperative mechanism to estimate the cache locality among different thread blocks/warps at run-time. Our mechanism can capture the cache access locality in a broad range of regular GPGPU applications with diverse memory access behavior.
- We develop a locality-aware thread block scheduler and warp scheduler. The thread block scheduler is able to increase the amount of shared cache lines and data reuse in each SM and the warp scheduler is able to maximize cache reuse opportunities in a SM based on the estimated cache locality.
- The Evaluations show that our mechanism can dramatically reduce 11 cache miss rate and provide up to 59% (average 26%) and 42% (average 10%) performance improvement over the conventional round-robin scheduler and the state-of-the-art approach.

The rest of the thesis is organized as follows. In Chapter 2, we will introduce the current GPGPU architecture and then describe our motivation for this thesis. In Chapter 3, we will introduce our proposed software and hardware cooperative method to estimate cache access locality and the proposed locality-aware scheduler policies and the details of architecture. In Chapter 4, we will describe our methodology and workloads. In Chapter 5 we will evaluate our proposed scheduler with the state-of-the-art scheduling policies and

show our experimental results. In Chapter 6, we will summarize related works. Finally, we will conclude the thesis in Chapter 7.





Chapter 2

Background & Motivation

In this chapter, we first introduce GPGPU architecture and then provide background information of thread block/warp scheduler. Next, we would briefly describe our motivation.

2.1 Background

With emerging of high performance computing, many applications are using GPU to accelerate their performance. For GPGPU programming, CUDA(Computing Unified Device Architecture) [17] and OpenCL(Open Computing Language) [22] are the most popular language. In CUDA and OpenCL programming model, a kernel is defined as a minimum unit launched on GPUs. A kernel is consist of thousands of threads and these threads are organized as two-level thread hierarchy – thread block(or called CTA) and warp(or called wavefront). A collection of threads are grouped to warp and one or more warps form a thread block.

2.1.1 GPGPU Architecture

Fig. 2.1 shows a typical GPGPU architecture. It consists of thread block scheduler, multiple streaming multiprocessors(SMs), a L2 cache shared among all SMs and a off-chip DRAM. Thread block scheduler is responsible to distribute thread blocks to SMs. Each SM has dual warp scheduler, register files, L1 cache, texture cache and scratchpad memory. During kernel execution, thread block scheduler dispatches the whole threads within

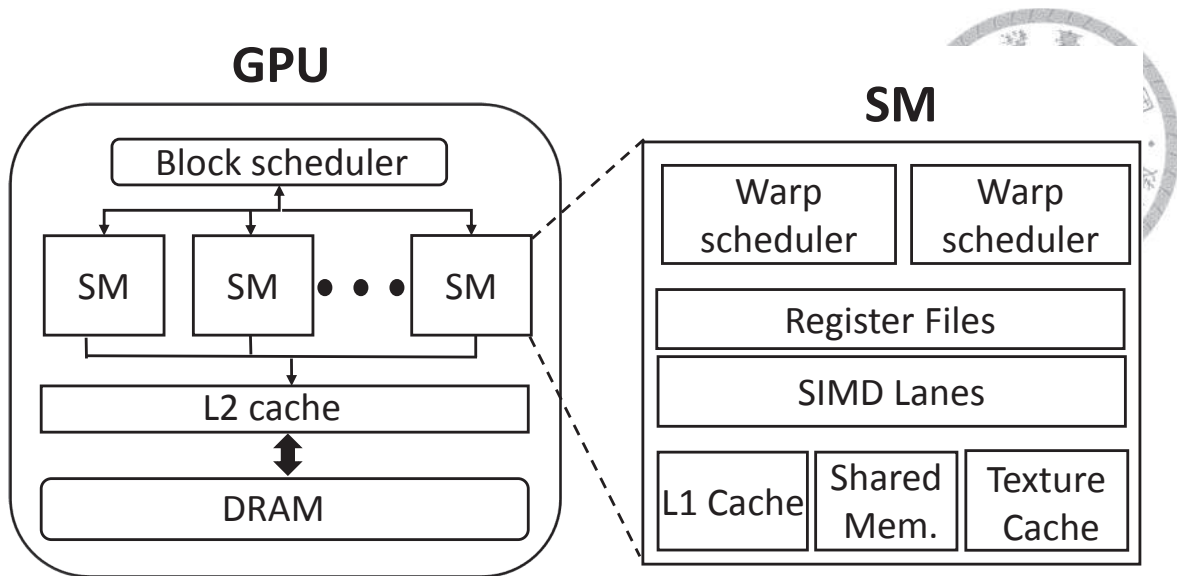


Figure 2.1: GPGPU Architecture

the thread block to SM and fixed number of threads(usually 32, 64) in the thread block are automatically grouped as warp by warp scheduler. Then, in each cycle, warp scheduler issues warp to SIMD(single instruction, multiple data) pipeline. The private L1 cache is used to cache data in the SM. However, scratchpad memory which is usually called shared memory in CUDA programming is used as user-managed cache. The data stored in the shared memory is only visible among warps within the same thread block.

2.1.2 Thread Block Scheduling

Thread block scheduler, which is called gigathread engine in NVIDIA GPUs, is responsible to assign thread blocks to the SMs when SMs have enough hardware resource. There are four hardware source limiting the number of thread blocks running on the SM: register files usage/per thread block, scratchpad memory usage/per thread block, number of threads, and the number of blocks, respectively. Once SM has enough hardware resource to support a new thread block, thread block scheduler would assign a new thread block to the SM. However, there is very few information on the scheduling policy used thread block scheduler. In this work, we use Round-Robin(RR) thread block scheduler as our baseline [1].



2.1.3 Warp Scheduling

In NVIDIA GPUs, 32 threads within a thread block are grouped as a single warp. Multiple warps in the SM can be used to hide memory access latency by fine-grained multithreading. For instance, when one warp suffers a stall, warp scheduler would continually issue other available warps to hide stall latency for achieving high performance. Today, various warp scheduling policy has been proposed to achieve high performance. Here, we will introduce three basic warp scheduler as following.

Round-Robin warp scheduler

In Round-Robin warp scheduler, each warp has an equal priority and warp scheduler issues warp based on the warp priority. This naive warp scheduler policy tends to make all warps have the same progress which potentially makes all warps stalled on the same long latency operation(i.e. off-chip memory access) in the same time and degrades the system performance.

GTO warp scheduler

GTO(greedy-then-oldest)[19] warp scheduler has been proposed to improve the drawbacks of RR warp scheduler. The main idea is to make different warps have different progress such that they might not be stalled on the same time. The detailed policy is explained as following, first, GTO scheduler would greedily select the same warp for execution until it suffers a stall and then choose the oldest warp for execution. Each warp's age is determined by the time it enters the SM. Note that although all warps within a thread block enter the SM at the same time but different warps still have different age. The warp with smaller thread id has more older age than the warp with bigger thread id. Since the warp with oldest age usually has the highest priority, the oldest warp usually has fastest progress while the youngest warp has the slowest progress. The different progress among warps means that they might not be stalled on the same time. That is, once a warp is stalled, other available warps can be selected for execution to make the core busy and improve the system performance.

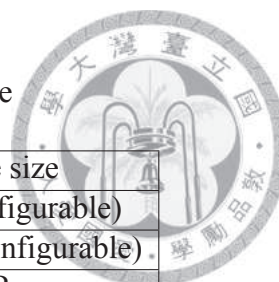


Table 2.1: GPU and CPU per thread cache resource

Device	Maximum threads per core/SM	L1 cache size
Fermi	1536	16/48KB(configurable)
Kepler	2048	16/32/48KB(configurable)
Haswell	2	32KB
Skylake	2	32KB

Two-level warp scheduler

Two-level warp scheduler is also proposed [6, 13] to improve the shortcomings of RR-scheduler. It maintains two warp groups – active group and pending group. Only warps in the active group can be issued by warp scheduler. When any warp in the active group is stalled on the long latency operations, the warp is demoted to the pending group. At the same time, a warp is promoted from pending group to active group. As a result, different warp progress between warps in the active group and the pending group can prevent all warps from suffering stalls in the same time.

2.2 Motivation

In previous section, we have already introduced the GPGPU architecture. Thousands of threads are employed in a SM to achieve high throughput but only a small L1 cache resource is provided to reduce memory access latency. As shown in table 2.1, Kepler-GPU [16] has a 48KB L1 cache and support 2048 threads in a SM while Skylake-CPU has 32KB L1 cache and support 2 threads concurrently running in a core. On average, a gpu thread has only around 24 bytes cache resource which is far less than CPU threads has(around 16KB). With limited per-thread cache capacity, cache block brought by one thread would easily be kicked out by other threads, leading to serious performance loss.

Besides, current thread block scheduler is unaware of cache locality among thread blocks and adopts a Round-Robin fashion policy to dispatch thread blocks. Recent work [11] observes that consecutive thread blocks often access data in shared cache lines, and they proposed a thread block scheduling policy called BCS to improve performance by allocat-

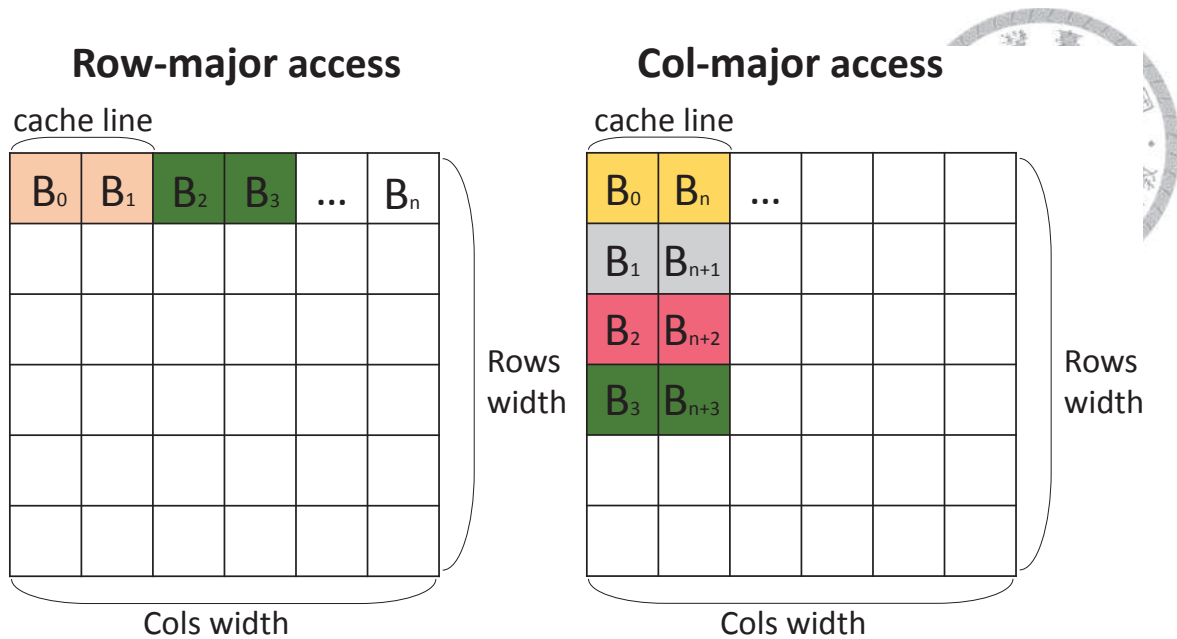
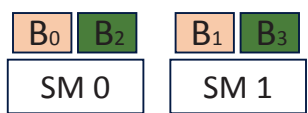
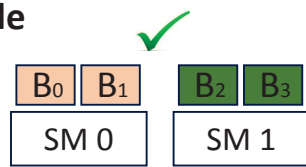


Figure 2.2: Data locality in workloads with different access behavior

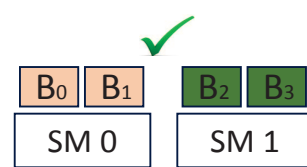
Row-major access example



Block scheduler (RR)

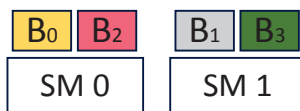


Block scheduler (BCS)

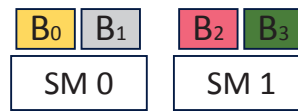


Block scheduler (Ideal)

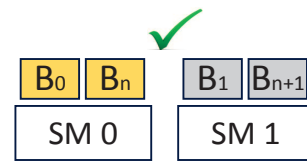
Col-major access example



Block scheduler (RR)



Block scheduler (BCS)

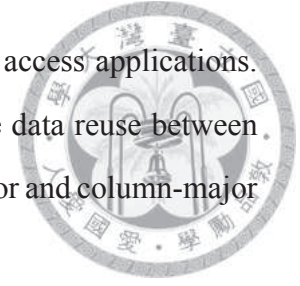


Block scheduler (Ideal)

Figure 2.3: Comparison of different thread block scheduling policies for row-major and column-major applications

ing consecutive thread blocks to the same SM. However, not all workloads benefit from BCS, as the thread block combinations that share cache lines vary in workloads with different memory access behavior. Figure 2.2 shows two common data access behavior in current GPGPU workloads. For row-major access application, consecutive thread blocks usually share the same cache line entries. However, for column-major access application, cache lines are shared between stride thread blocks such as block 1 and block $N+1$. Figure 2.3 illustrates the impact of different thread block scheduling policies on these two type of applications. As we can see, current RR thread block scheduler design tends to allocate thread block using same cache lines to different cores. Prior work(BCS) could

only capture performance improvement opportunities in row-major access applications. Our goal is to design a locality-aware scheduler that can exploit the data reuse between thread blocks in different types of workloads, including both row-major and column-major applications, as presented in the ideal case in figure 2.3.





Chapter 3

Locality-Aware Scheduler

In this chapter, we first describe the main idea of our locality-aware scheduler and then introduce our scheduler policies in block scheduler and warp scheduler, respectively.

3.1 Overview of locality-aware scheduler

To improve cache performance, we design locality-aware scheduler which can enhance the cache reuse opportunities. The goal of our design is to let thread blocks using the same cache line be assigned to the same core and warps using the same cache line entries could be issued nearly at the same time such that cache lines could be reused as many times as possible before they get evicted and therefore raise L1 hit rate. To achieve our goal, we propose software and hardware cooperative mechanism to estimate locality among different thread blocks and warps at runtime.

Figure 3.1 shows the overview of proposed locality-aware scheduling method. The boxes with green color are software side and others are hardware side. The locality-aware scheduling method includes address calculation code extraction and locality-aware block dispatcher. The address calculation code extraction is done by compiler, which extracts the code from a kernel program during compilation and generates a separate address calculation binary. The GPU driver passes the binary to block scheduler at the start of GPU application. The locality-aware block dispatcher is a scheduling algorithm running on a small, in-order processor [18] in block scheduler, which incorporates the address calcula-

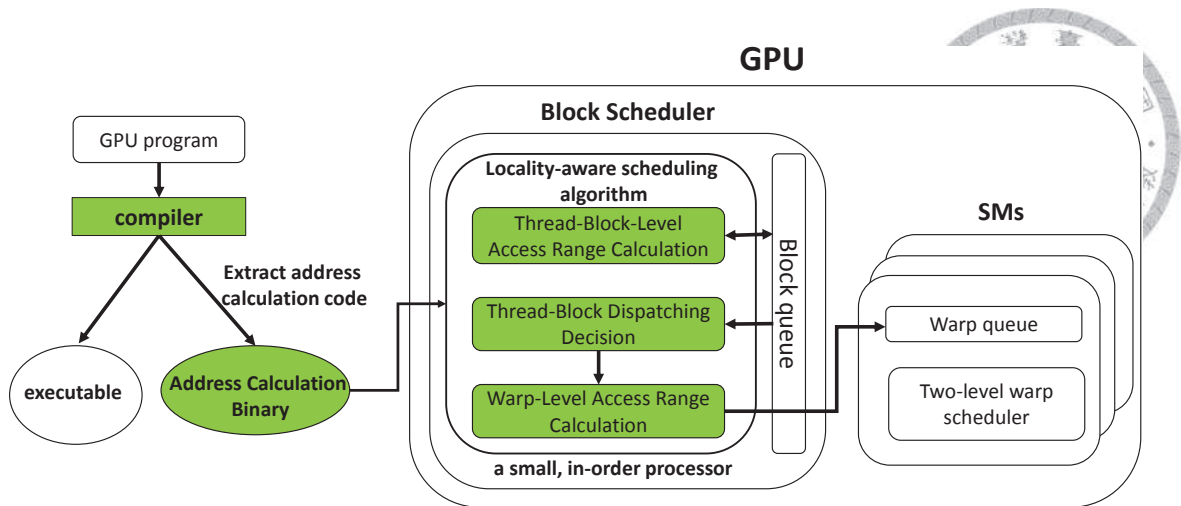


Figure 3.1: Overview of our locality-aware scheduler

tion binary and dispatches thread blocks to SMs maximizing the inter-block locality. After kernel is launched on GPUs, its thread blocks are enqueued in the block queue of the block scheduler. The memory access range of each thread block in the block queue are calculated based on the address calculation binary and stored in the same block queue entry. When any SM has available resources(i.e. a thread block has finished), the thread-block dispatching decision is triggered. The thread-block dispatching decision algorithm would estimate locality among thread blocks and select a thread block, which has the highest locality with all the running blocks in the SM. The access range of warps in the block is then calculated and attached to the corresponding warps during dispatching. Finally, the locality-aware warp scheduler schedules warps based on the access range information generated by the block scheduler to preserve the inter-block locality at warp-level.

3.2 Compiler Support

A GPGPU application is composed of one or more kernels. Each kernel is an array of thread blocks with unique IDs and each thread in the same thread block is given a unique thread id. The programmer usually uses these unique blockIDs and threadIDs to calculate the individual data position of each thread. In regular GPGPU applications, threads often operate on structural data such as one or two dimension data array. So the mapping between thread to data can be computed through simple arithmetics. For example, figure 3.2

```

Kernel Function
1: __global__ void kernel(float *J_cuda, int BLOCK_SIZE, ...)
2: {
3:                                     Constant Value & Data Array Pointer
4:     int blockid = blockIdx.x;
5:     int threadid = threadIdx.x;
6:
7:     int index = blockid * BLOCK_SIZE + threadid;
8:                                     Address Calculation Code
9:     c_cuda_temp[ty][tx] = J_cuda[index];
10:    // ...
11:    // computation
12:    // ...
13: }

```

Figure 3.2: Address calculation code extraction

shows a simplified kernel code. The code in the top box(line 1) shows a typical input kernel parameters including data array(i.e., J_cuda) and constant value(i.e., BLOCK_SIZE). The code in the middle box(line 4-7), which is called the address calculation code, computes the index of the input data array. Finally, the "index" variable is used to access the data array of J_cuda , as shown in the bottom box(line 9). The compiler can easily extract these address calculation code segments from a kernel function and generate the address calculation binary with those code segments and base address of the data array pointers. The block scheduler utilizes the binary, the threadID, and the blockID to compute the memory address on an arbitrary thread in the kernel, as shown in figure 3.3.

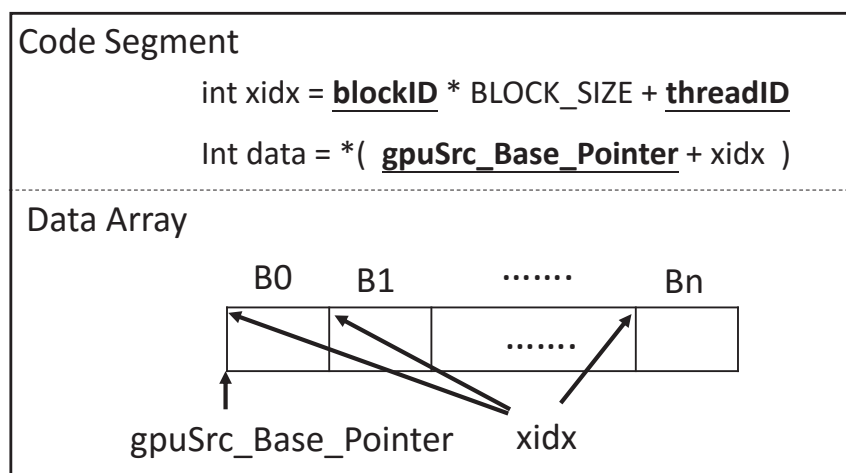


Figure 3.3: Simple address calculation code (Array-based data structures)

Data array

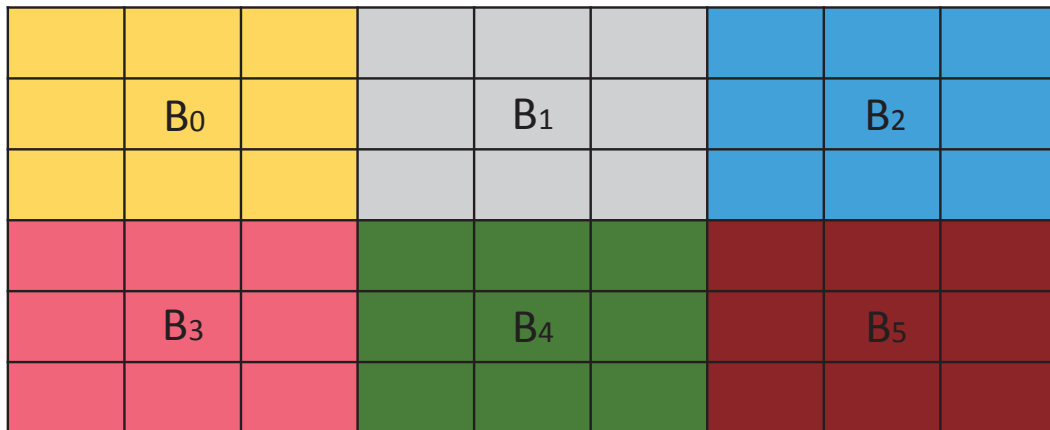


Figure 3.4: Example of memory access region in a thread block

3.3 Locality-Aware Thread Block Scheduler

In this section, we first introduce how to calculate access range of each thread block and then describe our thread block dispatching process including computing inter-block locality by utilizing the access range information and block dispatching criteria.

3.3.1 Thread-Block-Level Access Range Calculation

For regular GPGPU applications, threads often access contiguous memory regions. The memory region usually be linear or 2D data block when threads access data on one dimension or two dimension data array. So the memory access range of each thread block can be viewed as rectangle, as shown in the figure 3.4. Therefore, we could use the start point(i.e., upper-left point), width, and height to represent the rectangle. The start point could be computed by the 1'st thread in the block and the width/height could be computed by the differences of the memory address of the 1'st thread and last thread in the block. We extend the original block queue entries to store the access range rectangles of the block on the data arrays, as shown in figure 3.5.

In order to calculate how many cache lines shared between thread blocks, the memory access range of the thread block could be further represented in the coordinate of cache lines. As shown in figure 3.6, the data array is transformed from byte addresses to the cache line addresses. For example, the memory addresses (0, 0) to (127, 0) are mapped to

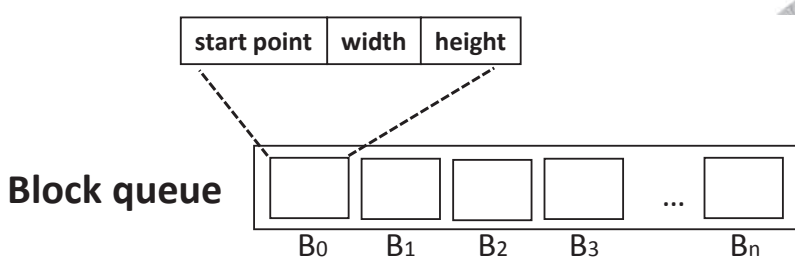


Figure 3.5: Extended block queue entries

the cache line address (0, 0), and the memory addresses (128, 2) to (255, 2) are mapped to the cache line address (1, 2), due to 128 bytes ll cache size in modern GPUs.

The memory access range of the block could be transformed into cache line access range. Therefore, cache lines accessed by the block could be represented in a rectangle with a start point, width, and height, as shown in figure 3.7.

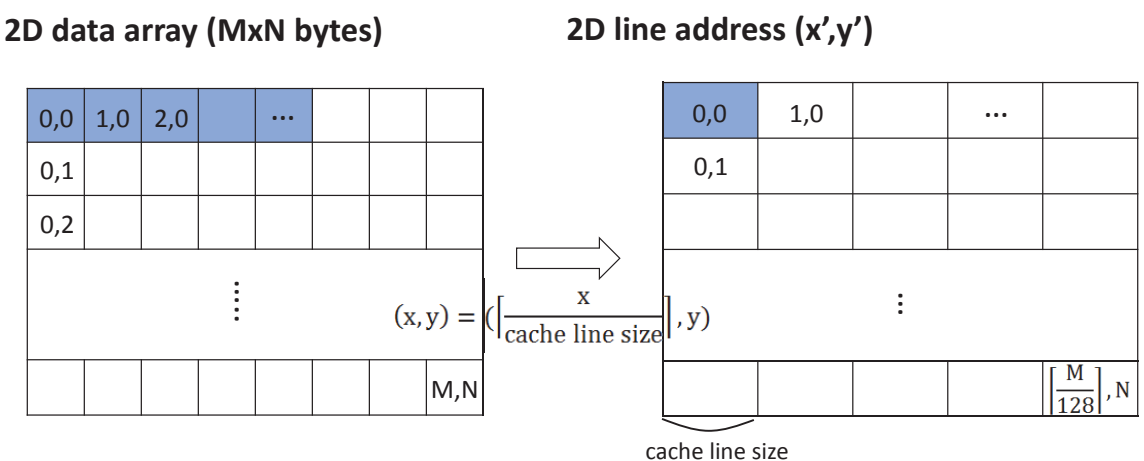


Figure 3.6: Byte address to line address transformation

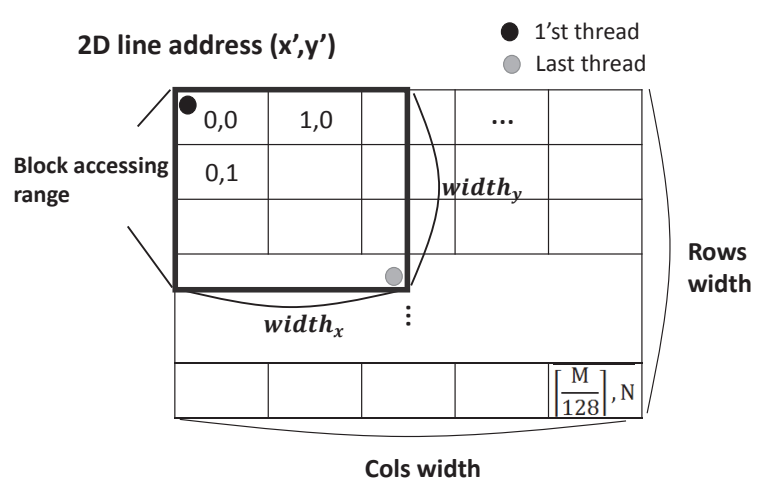


Figure 3.7: Mapping address range of the block to the coordinate of cache line address

3.3.2 Thread-Block-Dispatching Decision

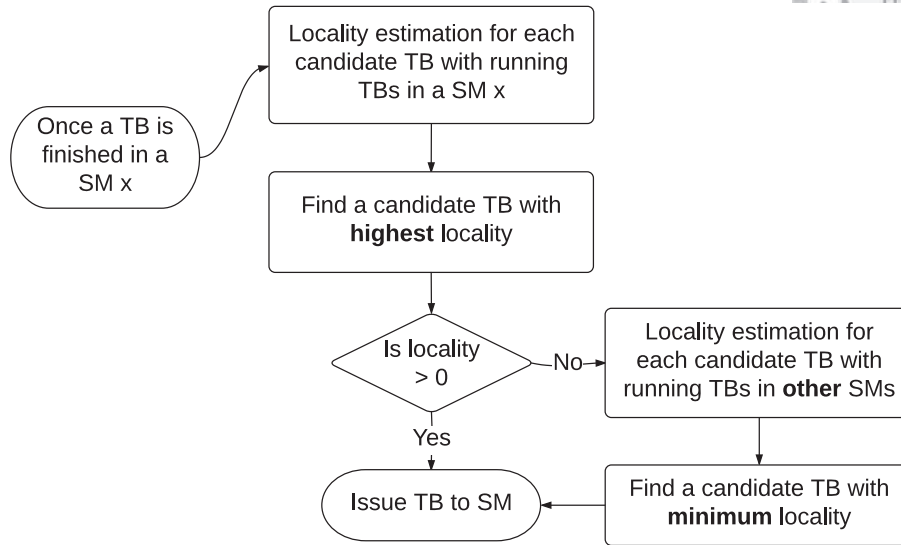


Figure 3.8: Flow of the thread-block dispatching decision algorithm

The flowchart of the thread-block dispatching decision is plotted in figure 3.8. Once a thread block has finished on a particular SM x , the block scheduler would allocate a thread block to the SM based on inter-block locality of the candidate block and all the running blocks on the SM x . The inter-block locality of two blocks is defined as the aggregated overlapped cache lines access range of the two, which is the summation of overlapped area of all data arrays. For each data array, the overlapped area is computed as following:

1. As shown in figure 3.9, the $distance_x$ and $distance_y$ are the difference between two block's upper left position.
2. If $distance_x > width_x$ or $distance_y > width_y$, there is no overlapped area between the two block, which means that there is no locality between these two blocks on this data array.
3. Otherwise, the overlapped area is $(width_x - distance_x) * (width_y - distance_y)$, which is the number of cache lines that shared by the two blocks on this data array.

Finally, the block scheduler would select a thread block that has the highest inter-block locality with the blocks running on the SM x . However, if all the candidate thread blocks have no inter-block locality, i.e., zero overlapped area, with SM x , the thread block, which

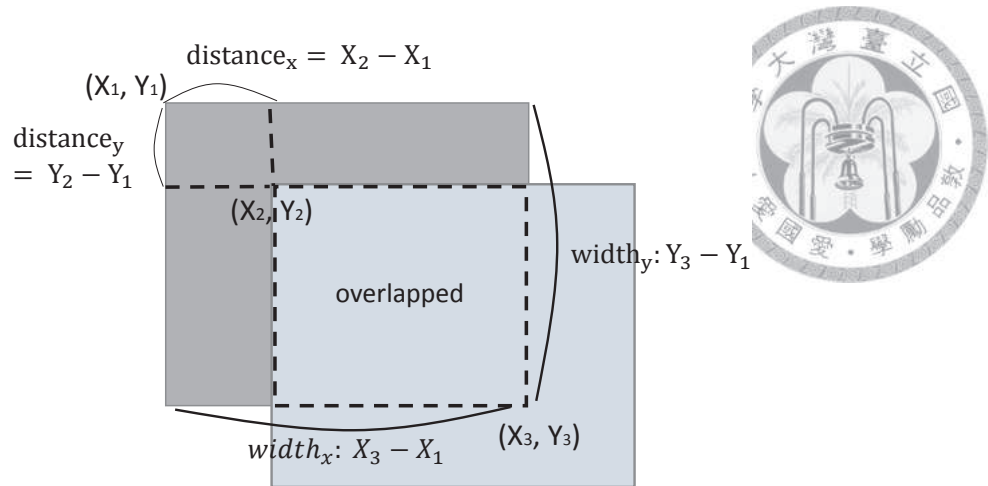


Figure 3.9: Locality estimation between two blocks

has minimal inter-block locality with the blocks running on the other SMs is selected. Since if a thread block, which has cache locality with the running blocks in other SMs, is selected to issue, the cache reuse opportunities in other SMs are reduced.

3.4 Locality-Aware Warp Scheduler

In this section, we first introduce how to calculate access range of each warp and then describe our proposed locality-aware warp scheduler to capture the benefits provided from block scheduler.

3.4.1 Warp-Level Access Range Calculation

Unlike the access range of a block, the access range of a warp usually does not have a fixed shaped, so it can not be represented as the start point, width, and height. Instead, the access range of a warp can be represented as a bit-vector. In the bit-vector, each bit is used to represent the access status of a unique cache line. Bit 0 means that the cache line is not accessed by the warp and bit 1 means that the cache line is accessed by the warp. However, the one-bit-per-line representation is impractical due to the huge working set in the kernel. Hence, we propose the hierarchical encoding method to cut down bit usage, as shown in figure 3.10. The hierarchical encoding method contains the following two steps.

Step 1 : The data array is partitioned into 2^M small regions where each region is repre-

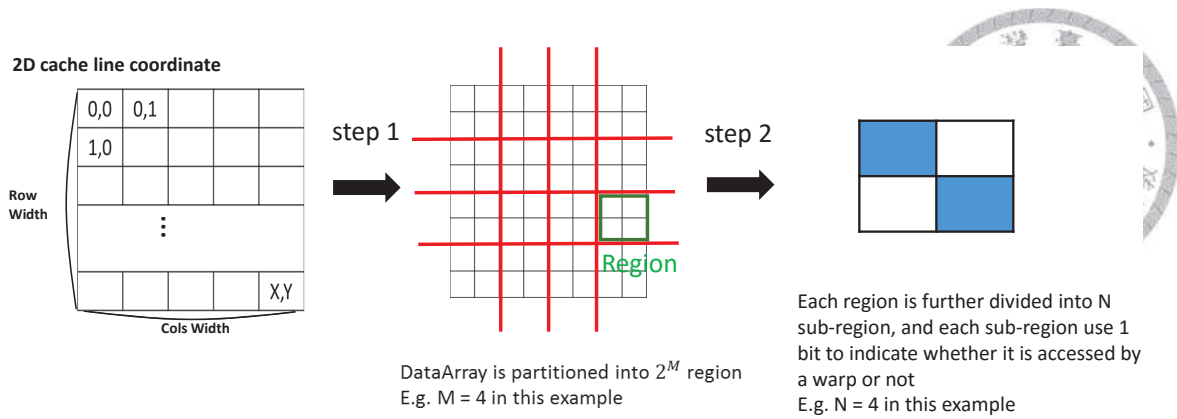


Figure 3.10: hierarchical warp encoding

sented by a region bit-vector with M bits. Then, each thread block could get a M -bit region vector by mapping its memory access range to the data array. As shown in figure 3.10, the data array is partitioned into 2^4 region. If the access range of a block is fallen into the green box (region 11), the 4-bit region vector becomes 1011.

Step 2 : Each region is further partitioned into N sub-region where each sub-region is represented by a subregion bit-vector with N bits. Then, the warp could get a N -bit subregion vector by mapping its memory access range to the subregion. As shown in figure 3.10, each region is partitioned into 4 sub-region. Each subregion uses 1 bit to indicate whether it is accessed by the warp or not. If a warp accesses the subregions with the blue box(the upper-left and lower-right subregions), the 4-bit subregion vector becomes 1001.

Combine the region vector of the block and subregion vector of the warp, the access range of a warp can be represented as a bit-vector with $M(\text{length of region bit-vector}) + N(\text{length of subregion bit-vector})$ bits.

3.4.2 Two-level Warp Scheduler

In order to capture the inter-block locality at warp-level, we should put the warps with inter-warp locality together and then execute those warps roughly at the same time such that shared cache lines could be use as many times as possible before they get evicted.

Based on the above thought, we employ widely used two-level warp scheduler to develop our locality-aware warp scheduler. Figure 3.11 shows our proposed locality-aware

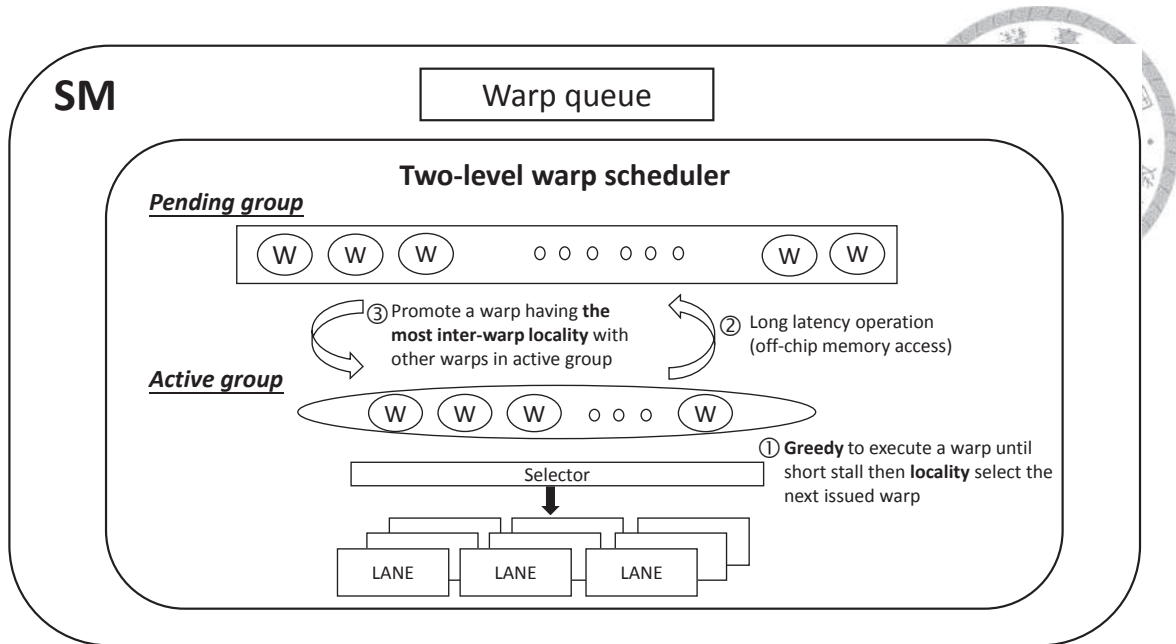


Figure 3.11: Locality aware warp scheduler

warp scheduler. In the SM, an additional warp queue is introduced to store the access range of the running warps on the SM as well as the inter-warp locality between warps. The access range information of warps is updated by the block scheduler and the inter-warp locality between warps is computed by the warp scheduler during execution. The idea of the proposed two-level warp scheduler is to divide all the running warps in the SM into two warp groups, active group and pending group. In each cycle, a warp from the active group is issued to the SIMD lane for execution in the priority order of greedy then locality. Once any warp suffers a long latency stall, i.e., off-chip memory access, the warp is demoted to pending group. At the same time, one of ready warp which has the highest sharing degree with the warps in the active group is promoted from pending group to active group.

Since we always promote warps with locality from pending group to active group, starvation may occur when some warp naturally has no locality with other warps. Once a warp starves, the other warps within the same thread block can not leave SM until the starved warp is finished, leading to performance degradation. To tackle starvation issue, a simple timeout solution is adopted. Each thread block is given an age when it is assigned to the SM. We detect the starvation happened when $Age_{new-dispatched-threadblock} - Age_{current-threadblock} > 2N$, where N is the max number of thread block in the SM. Once

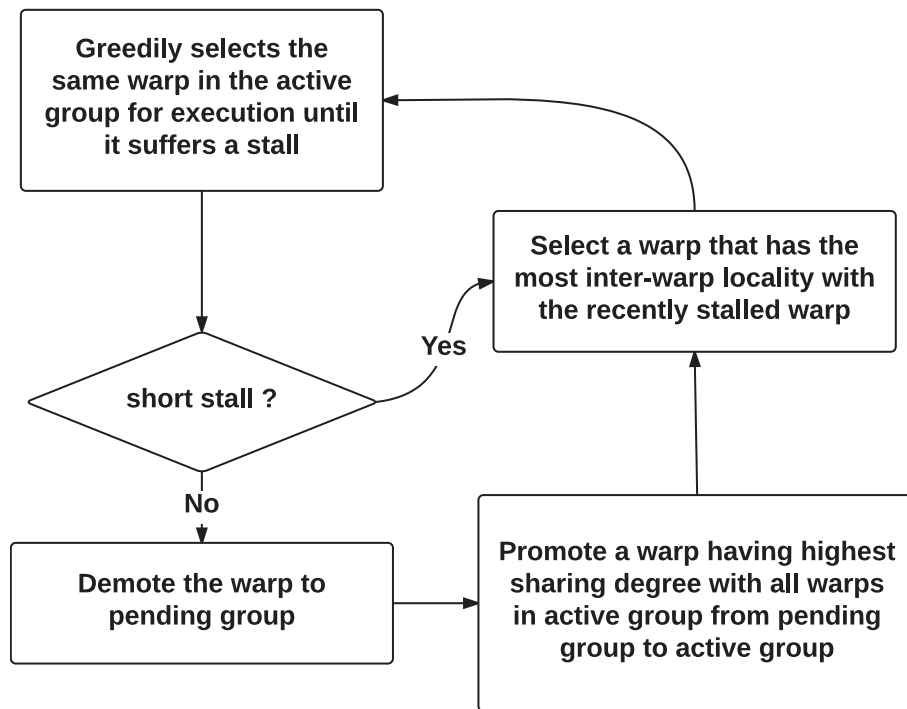


Figure 3.12: Flowchart of the two-level warp dispatching decision

starvation is detected, those starved warps are severed in the first priority.

The flowchart of the two-level warp dispatching decision is shown in figure 3.12. First, warp scheduler would greedily select the same warp in the active group for execution until it suffers a stall. If the stall is a short stall such as pipeline stalls, the warp scheduler would select a warp which has the highest inter-warp locality with last issued warp from the active group for execution. Otherwise, the warp is stalled by a long latency operation and is demoted to the pending group. At the same time, a warp has the highest inter-warp locality with all warps in the active group is promoted from the pending group to the active group.

The inter-warp locality described in the above is kept in a locality degree table, as shown in figure 3.13. Each entry in the locality degree table represents the inter-warp locality of the corresponding two warps. For example, inter-warp locality between warp 0 and warp 1 is stored in the entry (0, 1). The inter-warp locality between the two warps can be computed by comparing their warp access range information as following:

1. Check whether the region-vector between the two warps are the same, if they have different region-vector, there is no inter-warp locality among them. As shown in

Warp access range information

	Warp 0 (001-1011)	Warp 1 (001-0011)	Warp2 (010-1111)	...
Warp 0		2	0	
Warp 1			0	
Warp 2				

Region bit-vector Subregion bit-vector

inter-warp locality between warp 0 and warp 1

Figure 3.13: Example of the Locality degree table

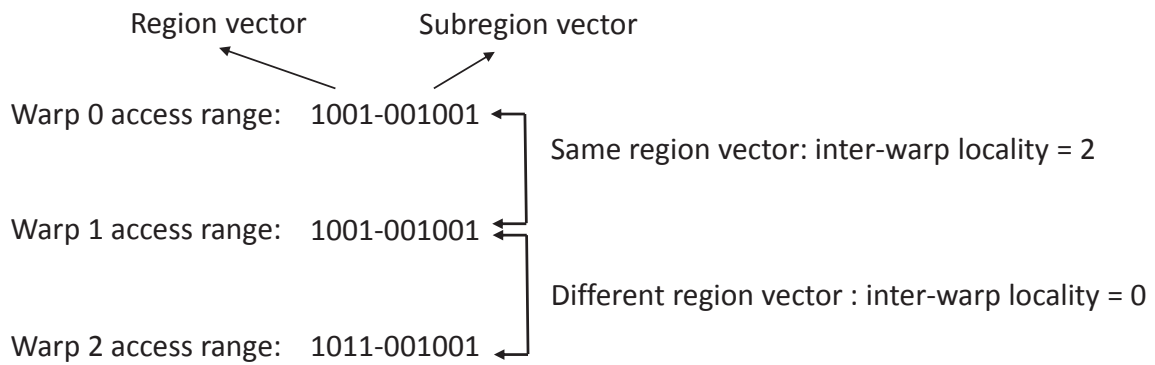


Figure 3.14: Example of inter-warp locality computation

figure 3.14, warp 1 and warp 2 have different region-vector, so there is no inter-warp locality between them.

2. Otherwise, the inter-warp locality is the number of same bit 1 in the subregion-vector. As shown in figure 3.14, warp 0 and warp 1 have the same region-vector, so the inter-warp locality becomes 2 because there are 2 of the same bit 1 in the subregion-vector (001001).



Chapter 4

Experimental Methodology

Table 4.1: GPGPU-Sim Simulation Configuration

Parameters	Value
Cores	15
Warp Size	32
Max number of threads / Core	1536
Max number of thread block / Core	8
Number of registers / Core	32768
Shared Memory	48KB
L1 Data Cache	16KB, 128B line, 4-way
L2 Cache	128B line, 8-way associated, total 768KB
DRAM Model	FR-FCFS, 6MCs, 16-entry request/MC
GDDR5 Timing	$t_{RRD} = 6, t_{RCD} = 12, t_{RAS} = 28$ $t_{RP} = 12, t_{RC} = 40, t_{CL} = 12$

We model locality-aware scheduling mechanism as described in Chapter 3 in GPGPU-Sim [4]. The baseline configuration is a Fermi-like architecture. Each SM has a private 16KB L1 data cache and 48KB shared memory. There are totally 15 SMs sharing a 768KB L2 cache. The global memory is partitioned into 6 DRAM channels and each DRAM channel has a First-Ready First-Come-First-Served (FR-FCFS) memory controller. The other detailed configuration of the simulator is summarized in table 4.1.

To evaluate our mechanism, we use 12 benchmarks from different benchmark suits as listed in table 4.2. In order to evaluate the diverse image processing workloads, we also add two benchmarks – SIFT and Gabor filter, which are the well-known workloads in the

area of computer vision, in our benchmark lists. All benchmarks are fully simulated on the GPGPU-sim.



Table 4.2: Workloads

Name	Abbr.	No. of kernels	No. of CTAs
Hotspot [5]	HS	1	7396
Back Propagation [5]	BP	2	8192
LU Decomposition [5]	LUD	223	140675
Nearest Neighbor [5]	NN	1	5120
Needleman-Wunsch [5]	NW	511	65536
Speckle Reducing Anisotropic Diffusion [5]	SRAD	4	65536
Gaussian Elimination [5]	GAUSSIAN	1022	525308
Separable Convolution filter [5]	CONV	34	52224
Transpose [15]	TRAN	24	98304
Histogram [15]	HIS	8	9860
Gabor filters	Gabor	68	49017
Scale-invariant feature transform	SIFT	2	263164



Chapter 5

Evaluation

In this chapter, we use the simulation methodology described in chapter 4 and evaluate our locality-aware scheduler described in chapter 3 with the state-of-the-art locality-aware thread block scheduler–BCS [11]. The prior work observes the workload behavior and finds that consecutive thread blocks usually have better cache locality. Hence, BCS dispatches two consecutive thread blocks to the SM at the same time. In order to understand the result of BCS and our proposed scheduler, we classify workloads into three categories according to their data access behavior, as shown in table 5.1.

Workloads are classified into Type-I because their data access behavior is dominated by row-major access while applications with a mixture of column-major accesses, random accesses, etc. are classified into Type-II. Other workloads that do not have inter-block locality are classified into Type-III.

The performance result of our locality-aware scheduler is shown in figure 5.1 normalized to the baseline configuration using GTO warp scheduler and L1 miss rate is shown in figure 5.2. For type-I workloads, our locality-aware scheduler improve approximately 3% performance compared to BCS but for type-II workloads, approximately 26% speedup is

Table 5.1: Workload categories

Type	benchmark
Type-I	<i>HS, Srad, Conv, His and BP</i>
Type-II	<i>Lud, Tran, Gaussian, and SIFT</i>
Type-III	<i>NN and NW</i>

further achieved. For type-III workloads, our proposed scheduler has no effect.

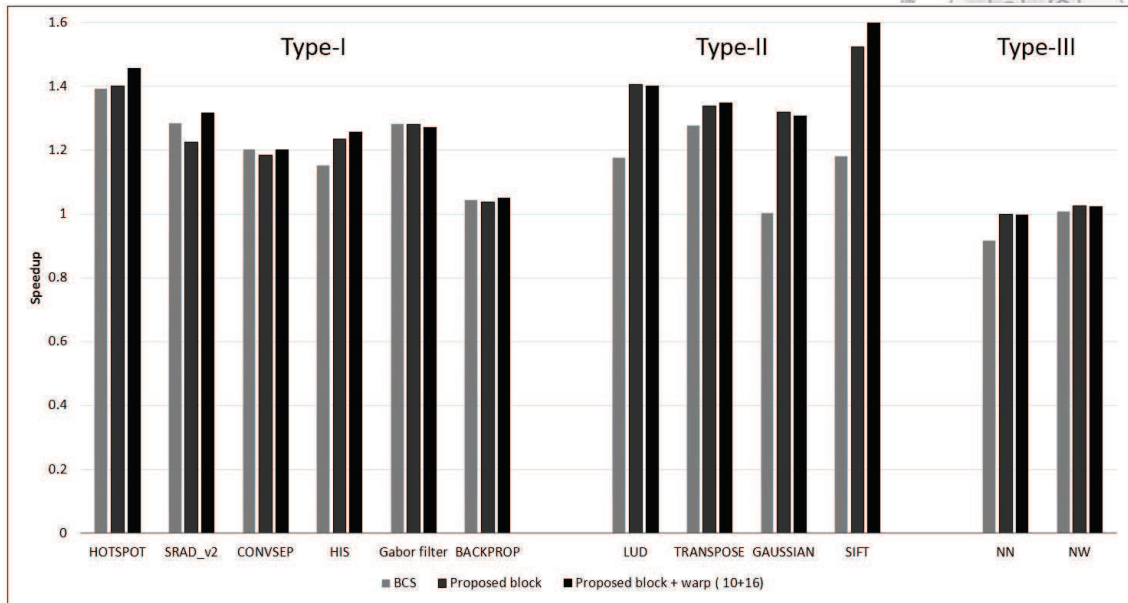


Figure 5.1: Comparison of performance for different policies, normalize to baseline architecture using GTO warp scheduler

5.1 Effect of thread block scheduler

For type-I applications, two consecutive thread blocks usually have better cache locality which is consistent with the observations in [11]. Therefore, in this case, our thread block scheduler achieves roughly the same performance with BCS because our thread block dispatched decision is the same as BCS. However, BCS has two limitations: 1) Due to lack of locality detection in the runtime, it could not dispatch thread block until SM has enough hardware resource for supporting the requirement of **two** thread block. Hence, this kind of delay scheduling causes (1) block-throttle-effect which may relieve resource contention and provides some performance improvement (2) latency-hiding-ability reduction 2) The other limitation is that BCS could only exploit the cache locality among **two** thread blocks. For *SRAD* application, although our dispatched decision is the same as BCS, BCS can perform better than our scheduler due to the effect of block throttling. For *HS* and *His*, cache locality is not only existing in the two consecutive thread blocks but in the *N* consecutive thread blocks (more than 2) and our scheduler can correctly detect locality among thread

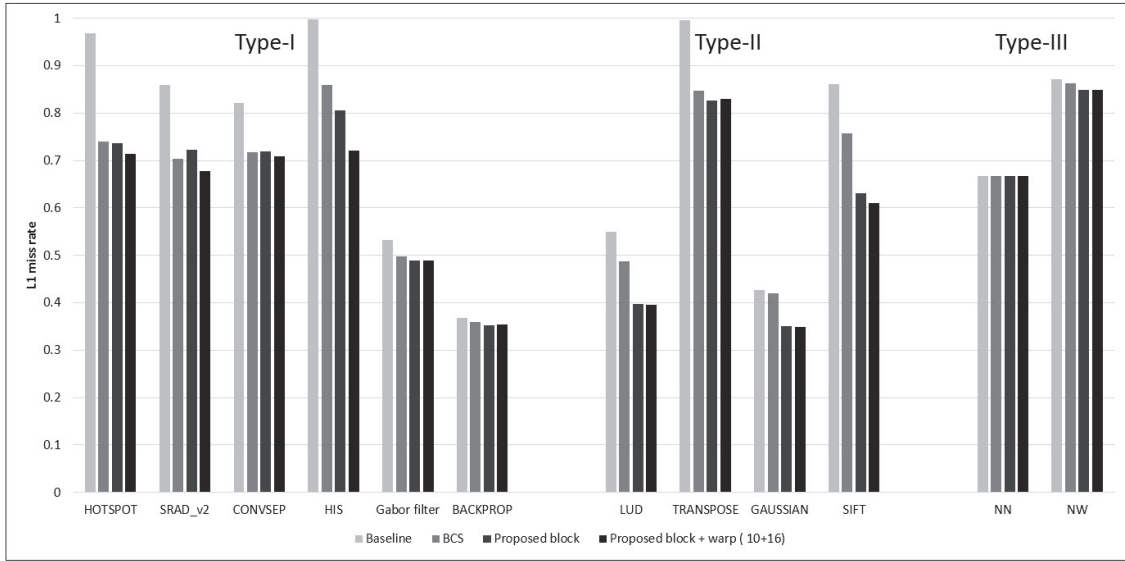


Figure 5.2: L1D cache miss comparisons

blocks and dispatch those N consecutive thread blocks to the same SM as many as possible for exploiting more cache locality.

For type-II applications, since BCS usually can not correctly detect the locality between thread blocks in this type, our proposed thread block scheduler could further achieve 24% speedup and reduce 10% miss rate. For type-III applications like NN and NW , our scheduler has no effect but BCS degrades 8% in NN application due to the effect of delay scheduling. For NN application, there are only 8 warps in the SM; therefore, it is hard to hide memory access latency especially applying delay scheduling policy on block scheduler; Only 6 warps can be used to hide the memory access latency.

5.2 Effect of warp scheduler

In order to exploit more cache locality, we design our locality-aware warp scheduler. As we mentioned in chapter 3, we use a bit-vector to represent its access range by hierarchical encoding method. Here, we choose configuration using at most 10 bits for region-vector and 16 bits for subregion-vector on a data array because its performance and hardware resource is more feasible.

For type-I and type-II applications, after adding our proposed warp scheduler, perfor-

mance could be further improved 3% on average. For *HS*, *Srad*, *His*, and *SIFT*, our warp scheduler can further capture more cache locality provided by thread block scheduler and improve performance from 2%-10%. However, for the other applications, our proposed warp scheduler could not acquire the more benefits because their cache footprints are fitted into L1 cache after allocating thread blocks with inter-block locality to the same SM.

5.3 Pipeline stall reduction

We further breakdown pipeline stall cycles to analysis which parts of stall cycles could be reduced by our locality-aware scheduler. A pipeline stall is caused by three type of hazards: structural hazards, idle or control hazards, and RAW hazards. Structural hazards occur when there is no available hardware resource for warp execution (i.e. lack of miss status holding register(MSHR) resource). When warps do not have a valid instruction for execution, an idle or control hazards occurs. It often happens when a barrier or control branch instruction is executed. The RAW hazards are caused by the two reasons: memory access operations(read after write to register) or data dependences. As we can see in figure 5.3 and 5.4, our locality-aware thread block scheduler can effectively reduce stall cycles which come from RAW and structural hazards. Since our locality-aware scheduler is to put threads with cache locality together, there are two benefits. First, memory requests for acquiring the same cache line are increased and these requests can be coalesced in the MSHR entries which reduce the original structural hazards. Besides, increasing cache performance means that the memory access latency is shorten and the RAW hazards are also reduced. However, after our proposed warp scheduler is adopted, there is a portion of idle or control stall cycles increased in some applications. In these applications, we find programmers tend to use shared memory to optimize the performance. When shared memory is used, it would need barrier instructions to synchronize threads within a thread block to guarantee correctness. However, our locality-aware scheduler may cause more large warp progress variation within a thread block compared to GTO warp scheduler due to capturing inter-warp locality. Hence, our warp scheduler prolongs the average waiting time of the warp at barrier and increases the idle or control hazards.

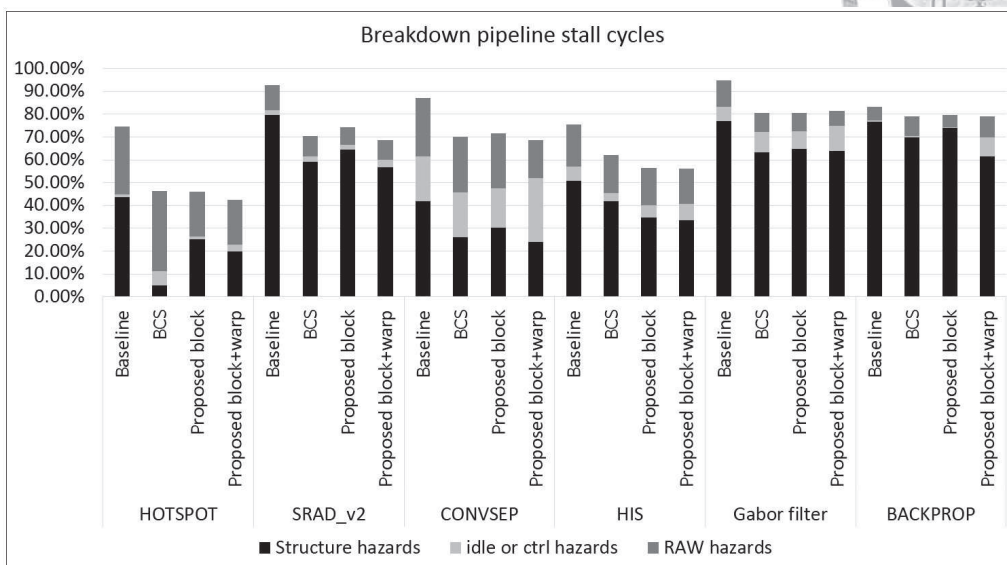


Figure 5.3: Breakdown pipeline stall cycles normalize to baseline execution cycles of Type I applications

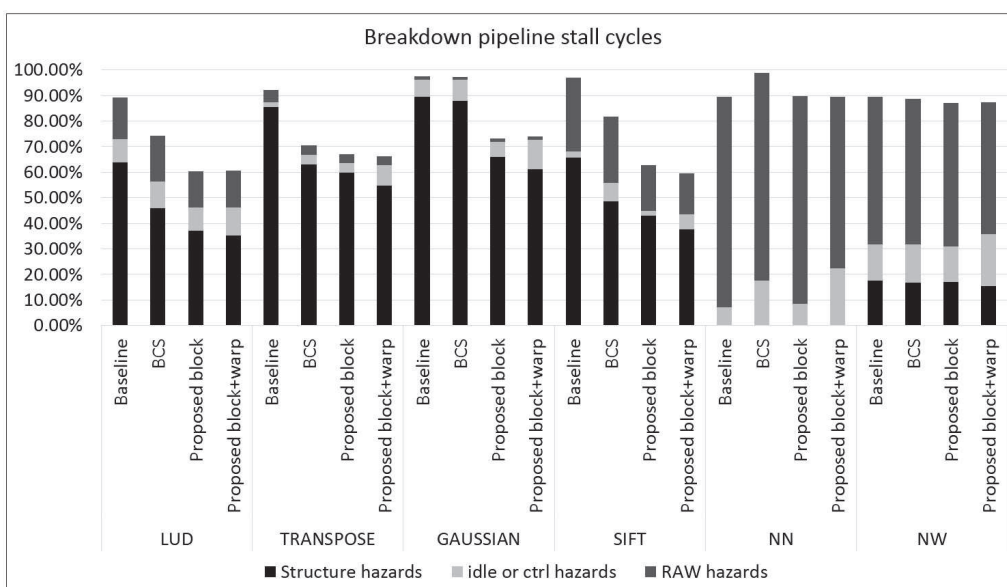


Figure 5.4: Breakdown pipeline stall cycles normalize to baseline execution cycles in type-II and type-III



5.4 Hardware Overhead

The hardware cost includes the storage cost for block queue information and warp queue information. In each block queue entry, additional information of access range rectangles of the block including start points, width and height where each element requires 1 byte. In our experiments, total 15 bytes are required for each block queue entry. For warp queue information, there are two additional information. One is warp access range bit-vector and the other is locality degree table. For each warp access range bit-vector, 16 bytes are required including 6 bytes for region bit-vector and 10 bytes for sub-region bit-vector. For locality degree table, 1 byte-per-entry is required for recording inter-warp locality. Therefore, the total storage cost for warp queue information is around 2.1KB due to current generation GPU supported up to 48 warps. The details of the hardware overhead cost are listed in table 5.2.

The performance overhead in our design is negligible. For block scheduler, the block dispatching process is fast due to the required information stored into the block entries. Besides, the block dispatching process is not on the critical path because there are other running blocks in the SM. For warp scheduler, the inter-warp sharing degree is stored in the locality-degree table. Hence, finding the maximum sharing degree warp requires only linear search time. The consuming time could be hidden by the warp scheduler, which would greedily issues the same warp until it suffers a stall.

Table 5.2: Hardware overhead

Location	Components	Storage overhead
Block scheduler	Block queue	15 bytes / per entry
SM	Warp queue	Locality degree table: $1.1(48 \times 48 / 2 \times 1)$ KB / per SM Warp access range information: $1(48 \times 16)$ KB / per SM



Chapter 6

Related Works

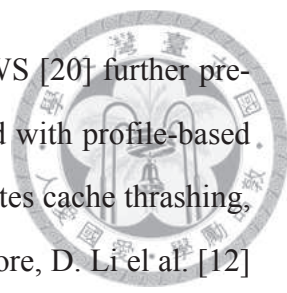
In this chapter, we would summarize the related works. First, we will summarize the works for improve cache locality on block scheduler and warp scheduler. Second, we will introduce other works for improving GPU resource utilization and thread-level parallel.

6.1 Block Scheduling for improving cache locality

Existing researches on thread block scheduler are quite less. M. Lee et al. [11] observe that two consecutive thread blocks usually have spatial cache locality, which means that two consecutive thread blocks often access the data in the shared cache lines. Therefore, they propose BCS scheduling which assigns two consecutive thread blocks to the same SM for exploiting more cache locality. As we discussed in the previous section, their work is suitable for row-major applications while our work is suitable for various applications with different data access behaviors by estimating cache locality at runtime. Moreover, our block scheduler collaborates with warp scheduler to capture as much cache locality as possible.

6.2 Warp Scheduling for improving cache locality

G. Rogers et al. [19] propose cache-conscious warp scheduling which uses additional hardware to monitor cache thrashing behavior and then use warp throttling technique to reduce



inter-warp interference and preserve intra-warp cache locality. DAWS [20] further preserves more intra-warp cache locality by using a predictor combined with profile-based and online detection information. Although throttle technique mitigates cache thrashing, it reduces TLP and make other shared resource under-utilize. Therefore, D. Li et al. [12] propose priority-based cache allocation which limits the number of warps that can allocate cache line entries while other warps bypass cache. X. Xie [24] proposes coordinated static and dynamic cache bypassing mechanism to determine which warp should be allocated the cache resource to improve overall cache locality.

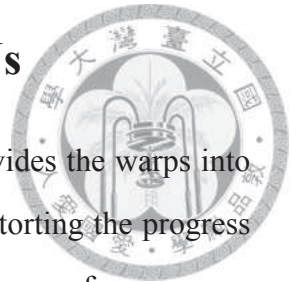
Our work targets on preserving inter-warp locality by putting threads with better cache locality together which is orthogonal to preserving intra-warp locality work likes CCWS, DAWS ...etc. Additionally, our work is unlike other related works which only design from the aspect of warp scheduler. Our locality-aware scheduler design takes one further step from top to bottom, from block scheduler to warp scheduler.

6.3 Improving resource utilization on GPUs

There have been many works on improving GPU resource utilization. Gebhart et al. [7] propose a unified on-chip resource management to manage the on-chip resource, including the register files, scratchpad memory and data cache, for different applications. O. Kayiran et al. [10] reduce cache, network, and memory contention by limiting the number of blocks running in a SM. A. Jog et al. [9] propose a scheduling mechanism that enables GPU data pre-fetch by utilizing a simple predictor and improve bank-level parallelism. R. Ausavarungnirun et al. [3] propose a memory controller design that is able to batch the memory requests with accessing the same row to improve the row buffer locality and therefore improve the DRAM performance. J. T. Adriaens et al. [2] propose a scheme that enables spatial multi-tasking on GPUs, which partitions the SMs for different applications to improve the SM resource utilization and fairness. Z. Wang et al. [23] enable spatial multi-tasking within a SM on GPUs, which supports for multiple kernels in a SM by solving the fragmentation issues and improve the on-chip resource utilization by exploiting heterogeneity of different kernels.

6.4 Improving thread-level parallel on GPUs

Narasiman et al. [13] propose a two-level warp scheduler, which divides the warps into fetch groups to overlap the computation with memory access by distorting the progress between different fetch groups so that some warps within a fetch group perform computation while other warps within other fetch groups execute the memory operations. Jog et al. [8] propose a scheduler called OWL to increase latency hiding ability by reducing cache contention and exploit DRAM bank-level parallelism to improve performance. The OWL aims to mitigate long memory access latency by prioritizing on a selected subset of blocks. A. Sethia et al. [21] propose cache access re-execution system and memory-aware scheduling to improve the system performance. It detects the memory saturation and prioritizes memory requests of one warp to enable the opportunities to overlap compute and memory accesses. Besides, it also enables warp execution even though the memory saturation. Once the warp is failed to execution, the additional hardware queue is used to buffer the warp for future re-execution.





Chapter 7


Conclusion

In this work, we develop a locality aware scheduler to improve the poor cache performance in modern GPGPU applications. We point out that the cache access locality existing in different thread block combinations due to different memory access behavior in various GPGPU workloads, which is not addressed in current studies. Hence, a comprehensive approach is needed to improve performance by exploiting cache access locality in different GPGPU workloads. Therefore, We propose the software and hardware cooperative mechanism to estimate the cache locality in thread-block-level and warp-level at run-time. Based on the estimated locality information, we design the locality-aware block scheduler, which is able to maximize the cache reuse opportunities, and the locality-aware warp scheduler, which is able to capture performance improve opportunities provided from block scheduler. The experimental results show that our locality-aware scheduler can effectively reduce L1 cache miss rate (at most 15%) and totally achieve 10% performance on average compared to the state-of-art scheduling policies.



Bibliography


- [1] K. M. abdalla et al. Scheduling and execution of compute tasks. US Patent US20130185725, 2013.
- [2] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.
- [3] R. Ausavarungnirun, K. K. W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 416–427, June 2012.
- [4] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. 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, April 2009.
- [5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.
- [6] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11*, pages 235–246, New York, NY, USA, 2011. ACM.

- 
- [7] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 96–106, Dec 2012.
- [8] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. *SIGPLAN Not.*, 48(4):395–406, Mar. 2013.
- [9] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13*, pages 332–343, New York, NY, USA, 2013. ACM.
- [10] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, Sept 2013.
- [11] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271, Feb 2014.
- [12] D. Li, M. Rhu, D. R. Johnson, M. O’Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based cache allocation in throughput processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 89–100, Feb 2015.
- [13] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. 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-44, pages 308–317, New York, NY, USA, 2011. ACM.



- [14] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.
- [15] NVIDIA. Cuda c/c++ sdk code samples, 2011.
- [16] NVIDIA. Kepler GK110 whitepaper. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>, 2012.
- [17] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.
- [18] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood. Fine-grain task aggregation and coordination on gpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 181–192, Piscataway, NJ, USA, 2014. IEEE Press.
- [19] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.
- [20] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 99–110, New York, NY, USA, 2013. ACM.
- [21] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185, Feb 2015.
- [22] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

- 
- [23] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369, March 2016.
- [24] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for gpus. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88, Feb 2015.