Inter-core Locality Aware Memory Access Scheduling

by

Dongdong Li

B.E., Beihang University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

April 2015

Abstract

Graphics Processing Units (GPUs) run thousands of parallel threads and achieve high Memory Level Parallelism (MLP). To support high MLP, a structure called a Miss-Status Holding Register (MSHR) handles multiple in-flight miss requests. When multiple cores send requests to the same cache line, the requests are merged into one last level cache MSHR entry and only one memory request is sent to the Dynamic Random-Access Memory (DRAM). We call this inter-core locality. The main reason for inter-core locality is that multiple cores access shared read-only data within the same cache line. By prioritizing memory requests that have high inter-core locality, more threads resume execution. Many memory access scheduling policies have been proposed for general-purpose multi-core processors and GPUs. However, some of these policies do not consider the characteristic of GPUs and others do not utilize inter-core locality information.

In this thesis, we analyze the reasons that inter-core locality exists and show that requests with more inter-core locality have a higher impact performance. To exploit inter-core locality, we enable the GPU DRAM controller to be aware of inter-core locality by using Level 2 (L2) cache MSHR information. We propose a memory scheduling policy to coordinate the last level cache MSHR and the DRAM controller. 1) We introduce a structure to enable the DRAM to be aware of L2 cache MSHR information. 2) We propose a

memory scheduling policy to use L2 cache MSHR information. 3) To prevent starvation, we introduce age information to the scheduling policy.

Our evaluation shows a 28% memory request latency reduction and an 11% performance improvement on the average for high inter-core locality benchmarks.

Preface

This dissertation is based on a project conducted by myself and supervised by Tor M. Aamodt. I was responsible for proposing the solution, creating the timing simulation to validate the solution, analyzing experimental result for all parts of this project. None of the text in the dissertation is directly taken from previously published or collaborative articles.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPU Central Processing Unit

GPU Graphics Processing Unit

FIFO First-In First-Out

FR-FCFS First-Ready First-Come First-Serve

SJF Shortest-Job-First

SIMT Single-Instruction, Multiple-Thread

CUDA Compute Unified Device Architecture

OpenCL Open Computing Language

L1 Level 1

L2 Level 2

MSHR Miss-Status Holding Register

SDK Software Development Kit

IPC Instructions Per Cycle

DRAM Dynamic Random-Access Memory

GPGPU General-Purpose computing on Graphics Processing Unit

MIB Merge Information Buffer

MLP Memory Level Parallelism

BFS Breadth-First-Search

HIL High Inter-core Locality

LIL Low Inter-core Locality

RAW Read After Write

WAW Write After Write

ROB Reorder-Buffer

TB Thread Block

GTO Greedy-then-Oldest

Acknowledgments

I would like to thank my supervisor Professor Tor M. Aamodt for the support and insight he has given during all these years. Tor always motivated me to think independently and gave me guidance at the same time. Without him, this work would not have been possible.

I would also like to thank everyone in computer architecture group for everything I learned from them. They provide useful feedbacks on my work. I am grateful to my friends for helping me with real problems either in research or in life.

Finally I have a special thanks to my family. They always inspire me and provide solid support. Their love, patience and understanding encourage me to overcome every obstacle in my life. I would not be here today without them.

I would like to thank Qualcomm for providing financial support for this work.

Chapter 1

Introduction

Graphics Processing Units (GPUs) enable general-purpose computing via programming interfaces like Open Computing Language (OpenCL) [21] and Compute Unified Device Architecture (CUDA) [1]. The general-purpose computing ability broadens the range of applications on GPUs. For example, machine learning algorithms like the neural network is accelerated by GPU using CUDA [17, 23, 31, 41]. GPU is also used for big data analysis in data science research. Stanford AI Lab accelerated deep learning algorithm using a cluster of GPU servers [11]. Engineering and mathematic tools such like Matlab and Mathematica use GPU to accelerate numerical computation [3]. Bioinformatics and life science use GPU to improve throughput of the DNA sequencing alignment [38]. The extensive use of GPU described above shows the importance of GPUs. In the light of this, researchers are motivated to do research on GPUs to exploit the potential power of GPUs. There are numbers of works on different aspects of GPU architectures.

Memory system is an important part of GPUs. The applications described above require a large volume of data resulting in many memory accesses. Today's GPU architec-

tures typically employ a Single-Instruction, Multiple-Thread (SIMT) [26] architecture that runs thousands of parallel threads. Unlike threads in the Central Processing Units (CPUs), each of which runs different instructions, the GPU has a group of parallel threads, known as a warp in NVIDIA terminology, that run the same instruction in lockstep. On a load instruction, a number of memory requests are issued by these warps. A warp will stall in the pipeline if it reaches an instruction that needs the data that has not yet been returned from a previous load instruction. We show it is important to resolve such pending memory requests quickly to reduce the number of stalled warps and maintain high throughput for the GPU and hence high performance. However, the performance of Dynamic Random-Access Memory (DRAM) is lower than SIMT cores resulting in a memory bandwidth wall. Since thousands of threads are running in parallel, there are a large number of memory requests which aggravate this problem.

Researchers proposed a number of methods to alleviate the memory bandwidth wall from different aspects for both multicore CPU and GPU-like many-core processors. A number of memory controller designs have been proposed to improve DRAM bandwidth utilization based on out-of-order scheduling [27, 32, 34, 36]. To reduce contention of memory requests, Cheng et al. [10] proposed an analytical model to estimate next phase system performance based on the computation to memory ratio. From the computation to memory ratio, the system can decide if it needs to suspend a thread to avoid memory requests congestion. Mutlu and Moscibroda [29, 30] proposed two memory access schedulers to avoid memory requests interference and ensure memory requests fairness between threads for multicore CPUs. Chatterjee et al. [7] proposed a memory access scheduler to balance memory access latency between all memory accesses for GPUs. Yuan et al. [42] proposed modifications to the on-chip network between processors and memory partitions to reserve

row locality of the requests. Using this modified on-chip network, the memory controller can use simple First-In First-Out (FIFO) policy to reduce hardware complexity. Ghose et al. [14] observed that history requests latency in the Reorder-Buffer (ROB) can be used to predict criticality of the next request. They proposed a predictor to prioritize the critical request from the processor side. We will discuss other related works in Chapter 7.

All of the above works tackle the memory bandwidth wall problem. However, there are two problems with these works. First, some of them do not apply to GPUs since they do not scale to thousands of threads. Second, some of these works do not consider criticality of requests. The state-of-art GPUs use a large number of threads to hide memory access latency. While some threads are waiting for the data from the memory, other threads can occupy the computation resources. Thus, memory access latency is not as critical in GPUs. Modern GPUs also exploit several techniques to improve memory bandwidth utilization. Within an SIMT core, a coalescing unit is used to reduce the number of memory requests generated by a warp. On-chip scratch pad and Level 1 (L1) caches are used to reduce long latency memory accesses. Between SIMT cores, a unified last level cache is used to capture locality that is not captured by the L1 caches and/or scratch pads. Memory requests accessing the same cache line issued by a single core are merged into one L1 cache Miss-Status Holding Register (MSHR) entry. Such memory requests exhibit what we call *intra-core locality*. When warps from different cores issue requests that access the same cache line within a short timeframe, they can be merged into one Level 2 (L2) cache MSHR entry and only one request is sent to the DRAM controller. Our experiments show that for a single memory request in the DRAM controller, there are often multiple requests issued from different cores merged together in the L2 cache MSHRs. We call this *inter-core locality*.

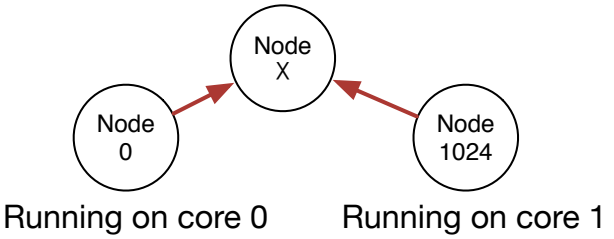
1.1 Motivation

We will describe the reason for inter-core locality in this section. Figure 1.1a shows a CUDA code example for which inter-core locality occurs. The code is Breadth-First-Search (BFS) from the Rodinia benchmark [8, 9]. Each graph node in the BFS is represented by a single thread. Multiple nodes can be assigned to a single core. Assume node 0 is currently running on Core 0, node 1024 is running on Core 1 and they connect to a common node X which is nextNodeID in Figure 1.1a. In line 3, they both want to access `g_graph_visited[X]`. Because they are running on different cores, neither coalescing unit nor L1 cache can help. The memory request to `g_graph_visited[X]` will be merged into one last level MSHR entry and only one memory request is sent to DRAM controller. Inter-core locality occurs in this situation. However, this request may not be prioritized as the DRAM controller is not aware of how many warps are waiting for the request. This problem cannot be avoided by changing threads scheduling policy since common node information cannot be predicted.

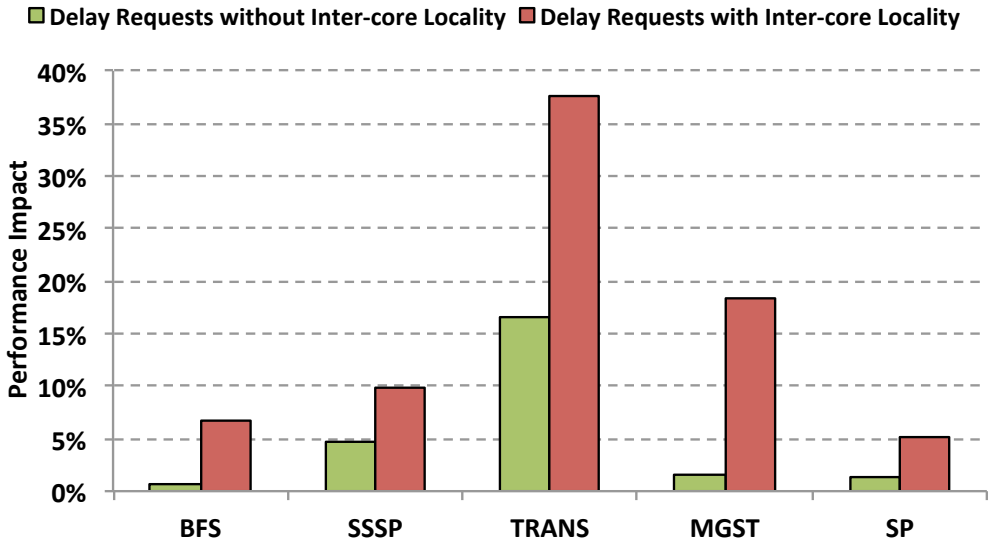
As illustrated by the above example, inter-core locality can occur in General-Purpose computing on Graphics Processing Unit (GPGPU) applications when they access large shared data structures. To help quantifying the potential presented by inter-core locality, we analyze five different kinds of applications by delaying 1000 requests either with or without inter-core locality. Figure 1.1b shows the performance impact, which shows that delaying requests with inter-core locality has more performance impact than delaying requests without inter-core locality. We evaluate the impact when the memory scheduling policy takes inter-core locality into account. Section 3.2 will show more analysis on this result.

Modern GPU memory controllers employ an First-Ready First-Come First-Serve (FR-FCFS)

```
0: int nodeID = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
1: for(i in all edges connected to current nodeID) {
2:   int nextNodeID = g_graph_edges[i];
3:   if(!g_graph_visited[nextNodeID]) {
4:     g_cost[nextNodeID]=g_cost[nodeID]+1;
5:     g_updating_graph_mask[nextNodeID]=true;
6:   }
7: }
```



(a) BFS code in Rodinia benchmark shows the inter-core locality



(b) Delay requests with inter-core locality and without inter-core locality

Figure 1.1: The source of inter-core locality and the effects of delaying inter-core locality requests

[13, 35, 36] scheduling policy to reorder memory requests to achieve maximum DRAM bandwidth utilization. However, a FR-FCFS scheduler may delay a request with high inter-core locality in favor of greedily optimizing for bandwidth. By exploiting inter-core locality, the DRAM system can reduce the overall warp stalling time, causing additional memory requests to be generated and thereby increasing future opportunities for bandwidth optimizations.

In this thesis, we propose an inter-core locality aware memory access scheduling policy and hardware. Inter-core locality can be represented using L2 MSHR merge length which is defined as the number of outstanding requests within one MSHR entry in this thesis. We enable the DRAM to be aware of this information. Our inter-core locality aware scheduler uses L2 MSHR merge information to schedule requests with high inter-core locality first.

1.2 Contributions

This thesis makes following contributions:

- We make a key observation with a concrete example to show that performance can be improved by prioritizing requests with high inter-core locality. We also show that performance is impacted if we delay requests with inter-core locality.

- We quantify inter-core locality across 24 benchmarks from three benchmark suites. We first classify 24 benchmarks into memory insensitive and memory sensitive benchmarks using a perfect DRAM model. We further classify memory sensitive benchmarks into low inter-core locality and high inter-core locality benchmarks.

- We propose an inter-core locality aware scheduler. We show that inter-core locality can be captured by utilizing L2 MSHR merge information. We explored two policies

that consider DRAM row-buffer locality along with age information to reduce starvation. We propose a hardware structure to enable the DRAM controller to be aware of L2 cache MSHR merge information.

In the following sections, all MSHRs are L2 cache MSHRs unless specified otherwise.

1.3 Organization

- Chapter 2 describes the background on baseline GPU architecture, cache and MSHR structure and memory system in this thesis.

- Chapter 3 gives a key observation on inter-core locality using a concrete example. It shows that by utilizing inter-core locality information on memory access scheduling, we can improve performance.

- Chapter 4 describes details about inter-core locality aware memory access scheduling policy. Two types of DRAM row score selection policies are explored in this chapter. A structure is proposed to support proposed policy. Hardware overhead is estimated in this chapter.

- Chapter 5 describes the methodology used in this thesis. A detailed GPU configuration is presented. This chapter also classifies benchmarks into different categories for experiments.

- Chapter 6 describes the experiments in this thesis. An overall performance of our inter-core locality memory access scheduling policy is presented. A detailed analysis of memory requests latency, L2 cache reservation and data dependency stall is given to have a comprehensive understanding of our proposed policy. A sensitivity study

is presented to show the effects of different configurations that affects our scheduling policy.

- Chapter 7 describes related work on memory access scheduling.
- Chapter 8 describes future work.
- Chapter 9 concludes inter-core locality memory access scheduling presented in this thesis.

Chapter 2

Background

In this chapter, we describe our baseline architecture. Section 2.1 describes our baseline GPU architecture. Section 2.2 describes the MSHR structure and how it handles multiple outstanding requests. Section 2.3 describes the DRAM model and DRAM controller in detail.

2.1 Baseline GPU Architecture

In this work, we study a GPU-like many-core architecture. Figure 2.1 shows our baseline architecture. A GPU is composed of a number of SIMT cores (Streaming Multiprocessor in NVIDIA terminology and Computing Unit in AMD terminology) and a number of memory partitions. There is an on-chip interconnection network connecting SIMT cores and memory partitions [12, 19]. Each SIMT core consists of a number of Thread Blocks (TBs). A function runs on the GPU called a kernel. When a kernel is launched, a number of warps are assigned to the TBs. Similar to NVIDIA, each warp consists of 32 threads in our model. Threads in a warp run in lockstep. When a warp executes a load instruction, a number of

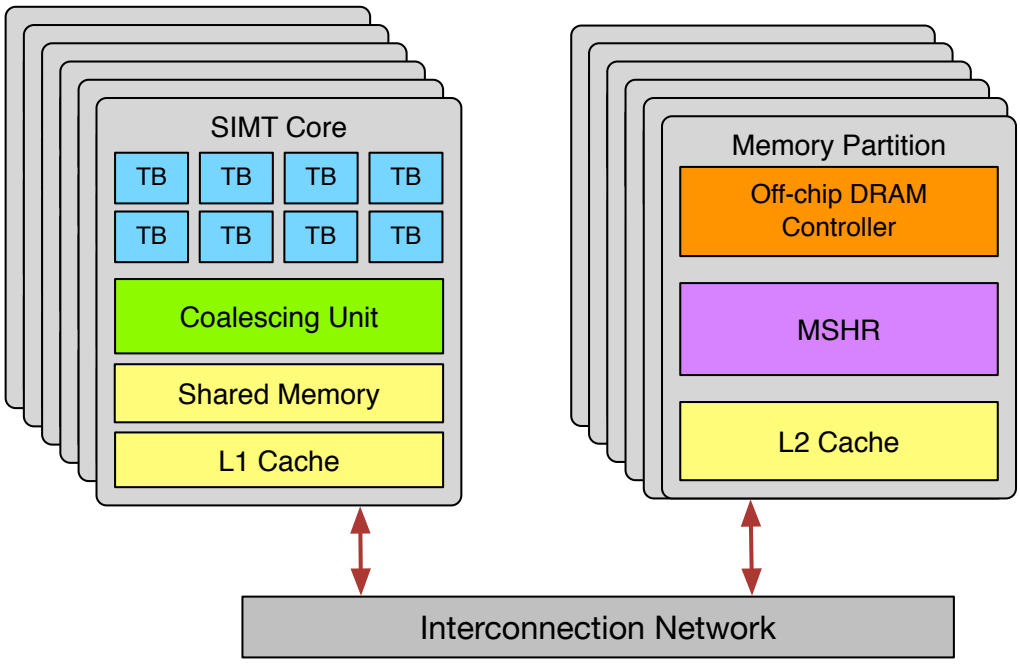


Figure 2.1: Baseline GPU architecture

memory requests are generated from the warp. A coalescing unit is introduced to reduce the number of memory requests sent from a warp. The coalescing unit tries to coalesce requests issued by a warp into as few requests as possible if there are consecutive addresses in these requests. For example, if all threads in a warp access the address within 128 bytes, only a single memory request will be generated. However, if the threads access memory address which has gaps larger than 128 bytes, 32 requests will be generated. This can create significant pressure on the memory system.

In the following sections, we will use cores to represent SIMT cores for simplicity.

2.2 Cache and Miss Status Holding Registers

There are two levels of caches in our baseline GPU architecture. Each SIMT core has its own private L1 cache to capture locality within it. For the locality between SIMT cores, there is a last level cache bank located in the memory partition. When requests miss in the cache, multiple outstanding memory requests are sent to lower memory levels. To keep track of multiple outstanding memory requests, an MSHR is added to the cache [24, 40] .

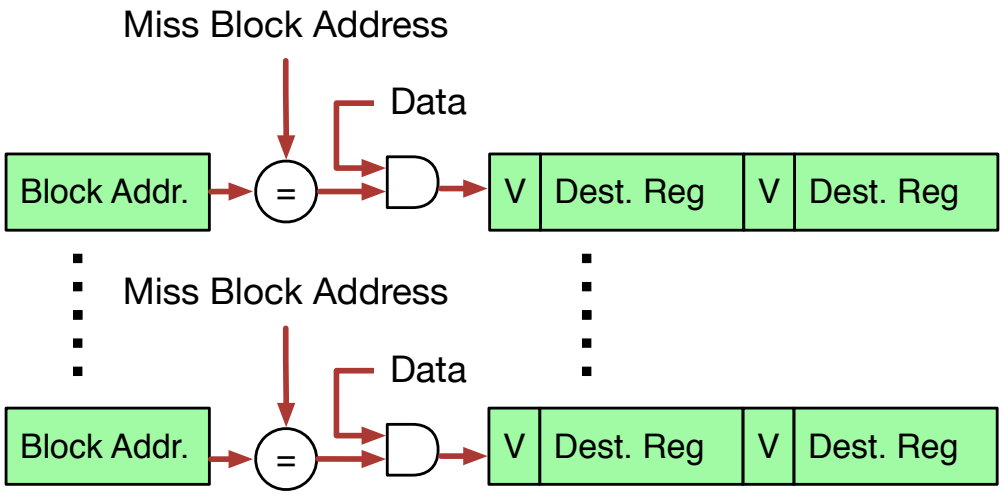


Figure 2.2: MSHR structure

Figure 2.2 shows the structure of the MSHR assumed in this work. When a request misses a cache line, the MSHR compares the address of the request to each entry of its block address. If there is no MSHR entry that matches the block address of the request, the request is stored into a new MSHR entry and a miss request is sent to the next lower level of the memory hierarchy to acquire data for the whole cache line. The MSHR keeps information regarding the waiting requests, such as the destination register to store the data

once it returns from the lower memory level. If the address of the request matches the block address of an MSHR entry, it indicates that there is already an outstanding request for this cache line. This request is merged into the matched MSHR entry, and no additional request needs to be sent.

When multiple memory requests are issued by warps from the same core miss to the same cache line, they are merged into the L1 cache MSHR. Memory requests for the same cache line from different cores are merged into the last level cache MSHR. Thus, the last level cache MSHR captures inter-core locality. In this thesis, we use this information to improve memory access scheduling in Chapter 4.

2.3 DRAM Controller and DRAM

Figure 2.3 shows our baseline DRAM controller and DRAM model. When a request reaches the DRAM controller, it will be buffered in either the *Read Request Queue* or the *Write Request Queue* depending on its type. A scheduler inside the DRAM controller is responsible for the memory access scheduling. The scheduler chooses a request either from the read request queue or the write request queue based on the read-write scheduling policy. After a request is chosen, the scheduler translates the request into DRAM commands (Precharge, Active, Read and Write) and the request is transferred to the *Command Queue*. Each DRAM bank has its own command queue. The command queue is scheduled using a round-robin policy [16]. When a DRAM command is sent, the priority pointer points to the next bank. For each bank, the commands in the command queue are scheduled in-order to preserve scheduling decisions by the request scheduler.

In each DRAM bank, there is a row-buffer. The reason for the row-buffer is to reduce access latency to the same DRAM row due to the timing constrains of the DRAM. When a

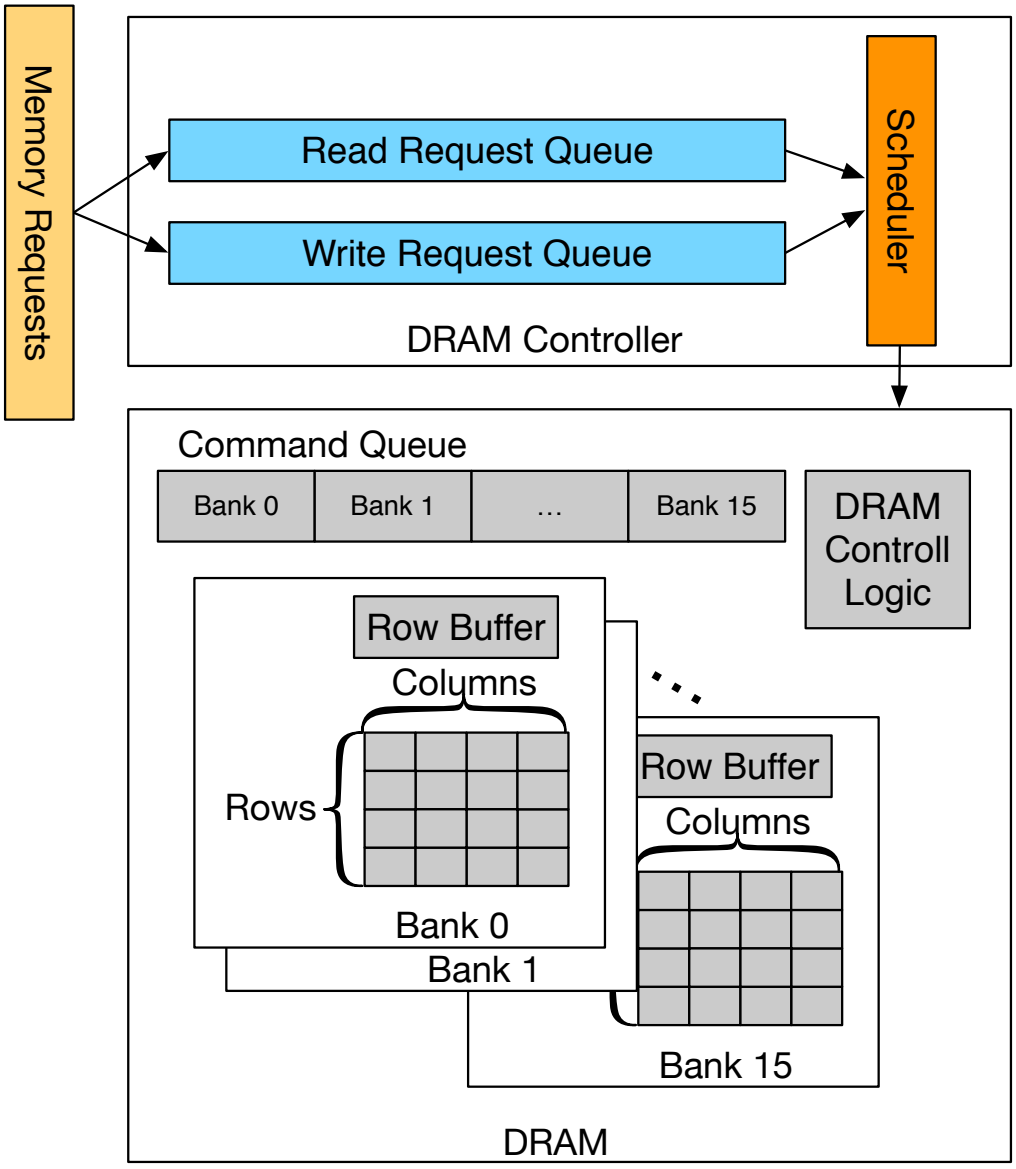
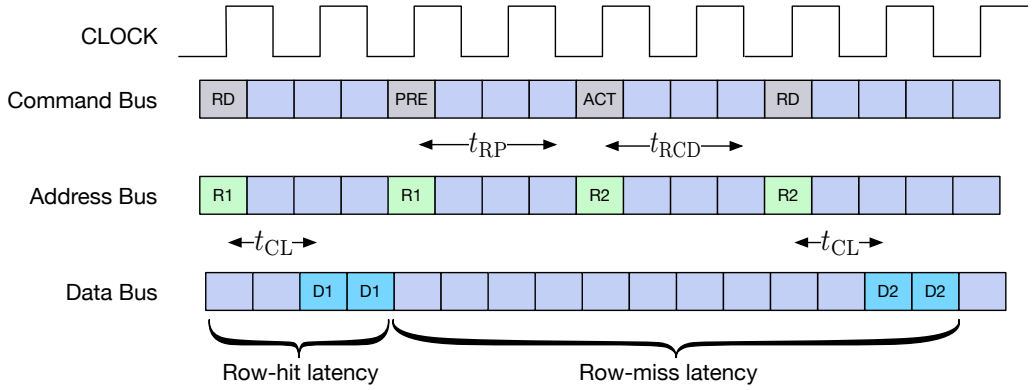
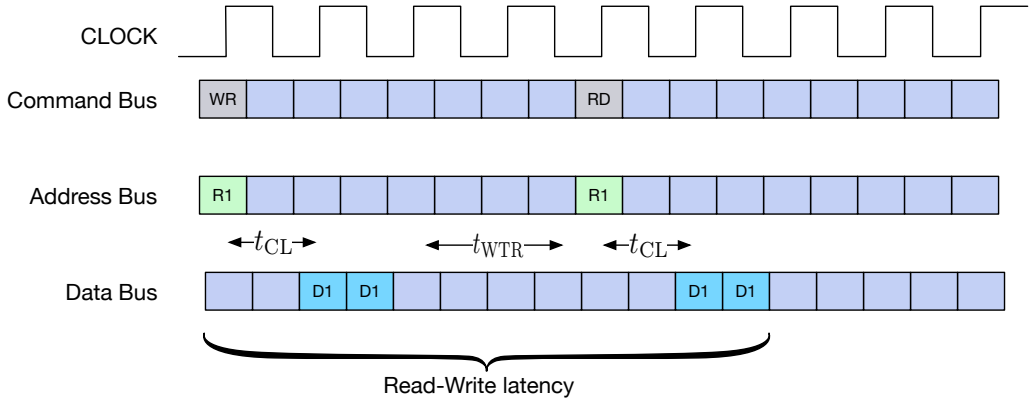


Figure 2.3: Baseline DRAM controller structure

row in the DRAM is activated, data in a DRAM row is transferred to the row-buffer. This row is known as the *open row* for the bank. While a row is buffered in a row-buffer, requests access this row are called a *row-hit*. When a DRAM controller issues a request that accesses a different row, the DRAM needs to close the current open row and activate a new row. This is known as a *row-miss*.



(a) DRAM timing illustration for row-hit and row-miss



(b) DRAM timing illustration for write to read latency

Figure 2.4: DRAM timing constrains, we assume row 1 is the current open-row at the beginning of the clock. (RD = Read, PRE = Precharge, ACT = Activate, WR = Write, R1 = Row 1, R2 = Row 2, D1 = Data for Row 1, D2 = Data for Row 2)

Figure 2.4 shows the DRAM timing constrains. The command bus sends DRAM commands like Read, Write, Precharge and Activate to the DRAM. The address bus indicates which bank, row and column to access. For simplicity, we do not show bank and column address in the Figure 2.4. We assume the read and write commands access the same bank but different columns. After a read or a write command is sent, the data bus output the data after the timing constraint is satisfied.

Figure 2.4a shows the timing constrains of row-hit and row-miss. When there is a row-hit, data will be available at the output port after Column Address Strobe Latency (t_{CL}). When a row-miss occur, two additional latencies are introduced which are Row Precharge Time (t_{RP}) and Row Address to Column Address Delay (t_{RCD}). The row-miss latency (t_{RP}+t_{RCD}+t_{CL}) is much higher than row-hit latency (t_{CL}). To achieve high bandwidth utilization, our baseline *Scheduler* uses an FR-FCFS policy. It schedules requests accessing the current open row first. When there is no memory requests to the current open row, the request scheduler schedules the oldest request to prevent starvation.

Write requests are not critical to performance. Because after a write request is sent, a thread does not block for the write request. There is a minimal delay (t_{WTR}) to switch the bus between read and write. Figure 2.4b shows the timing constrain of the t_{WTR}. To reduce the t_{WTR} penalty, write requests are buffered in write request queue. When the DRAM controller accumulates a high number of write requests in the write request queue, the request scheduler drains the write request queue to prevent the write queue from reaching its maximum capacity. This number is the high-watermark. The write request queue will also be drained when the read request queue is empty. The request scheduler stops draining the write request queue and switch to schedule the read requests until it reaches a low number which is the low-watermark [39].

Chapter 3

Inter-core Locality Benefits Performance

As described above, the coalescing unit is used to increase overall memory bandwidth. However, the coalescing unit can only help in combining requests from one warp. Requests issued by different warps within a single core are combined in the L1 cache MSHR. Memory requests from different cores cannot be combined by the coalescing units or by the L1 cache MSHRs. While memory requests issued by the multiple cores accessing data in the same cache line, they are merged into a single L2 cache MSHR entry. The L2 cache MSHRs keep track of outstanding memory requests sent to DRAM. We show that the number of merges in an L2 cache MSHR entry can be used to represent inter-core locality.

In this thesis, row locality is defined as the average number of requests to access an open DRAM row before it is closed. Row locality is important to DRAM bandwidth utilization. High row locality means a low row-miss rate and high bandwidth utilization. Since row-miss overhead (t_{RP}+t_{RCD}+t_{CL}) is much larger than row-hit overhead (t_{CL}), it is impor-

tant to improve row locality to maximize DRAM bandwidth utilization. Modern DRAM controllers use an FR-FCFS scheduling policy to reorder memory requests to maximize the row access locality. FR-FCFS searches the request queue to find if there is any request accessing the open row. If there is no request accessing the open row, a row-miss occur. In this case, FR-FCFS schedules the oldest request to prevent starvation. This can lead to high DRAM bandwidth utilization but does not always benefit the overall performance. In the next section, we will describe an alternative scheduling policy to utilize inter-core locality to get better performance.

3.1 A Key Observation

As mentioned in Chapter 2, memory requests from different cores accessing the same data are merged into the L2 cache MSHR. Only one memory request is sent the to the DRAM controller. In this case, multiple warps from different cores are virtually waiting for one memory request in the DRAM controller. By servicing the request with the maximum number of waiting warps, the total waiting time of these warps is reduced. These warps can resume execution earlier and increase overall performance. Since memory requests that are issued by different cores to the same cache line are merged into the same L2 MSHR entry, we can use L2 cache MSHR merge length to measure the number of warps waiting for their memory requests. In this thesis, we only focus on memory requests issued by different cores. In the Chapter 8, we will discuss memory requests issued by the same core in future work.

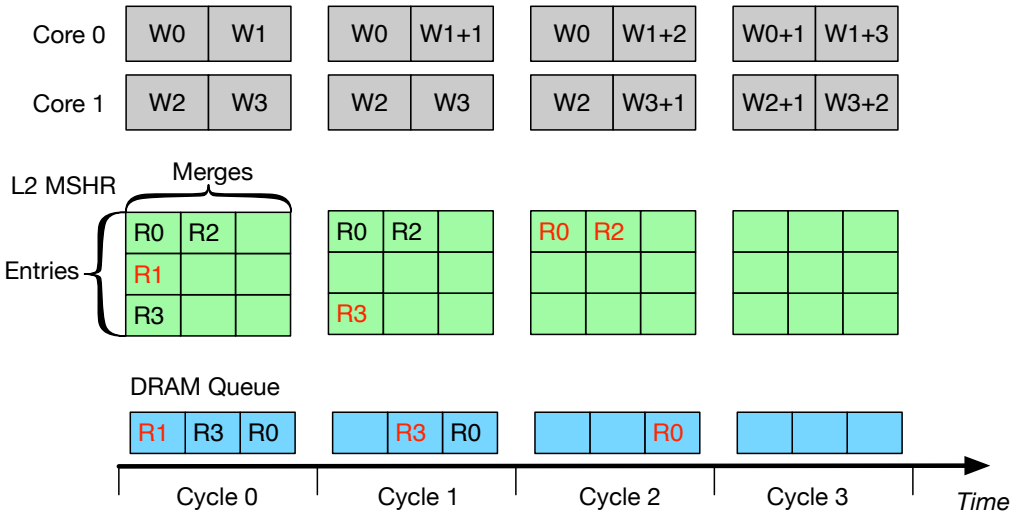
The MSHR merge information can be used to help with the DRAM controller scheduling decision. For a request in the DRAM controller, if the merge length of its corresponding MSHR entry is greater than one, this indicates that by serving this request will benefit

multiple warps from multiple cores. We call this inter-core locality aware memory access scheduling.

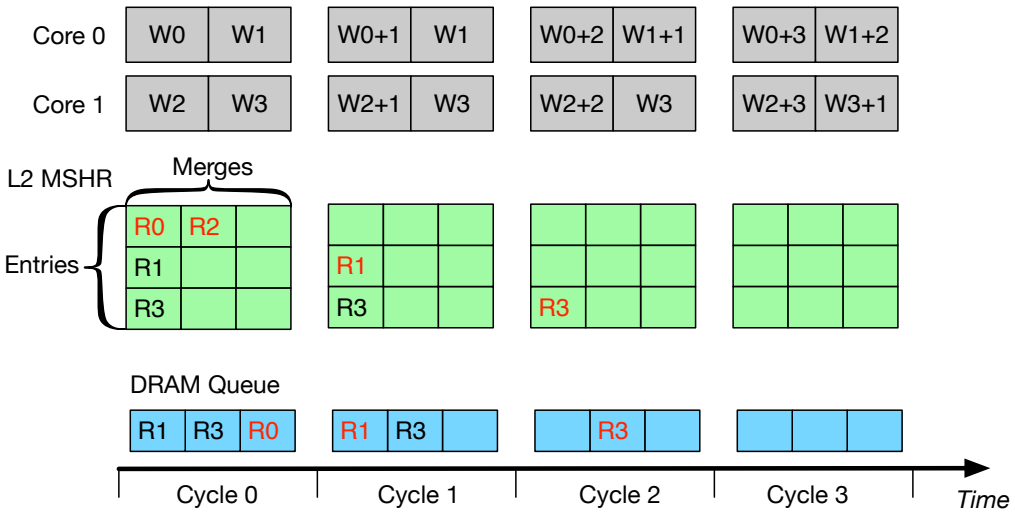
Figure 3.1 shows an example to illustrate how inter-core locality aware memory access scheduling can improve performance. Assume Warp 0 and Warp 1 belong to Core 0 and Warp 2 and Warp 3 belong to Core 2. During Cycle 0, they are all currently stalling on the pending load instructions. Memory requests from Warp 0 and Warp 2 are merged into one MSHR entry, which is the first row of the L2 cache MSHR, as illustrated in Figure 3.1. Four memory requests (R0 to R3) are sent to the DRAM. For simplicity, in this example, we assume that after DRAM services a request, the core can receive this request the next cycle.

Figure 3.1a shows the FR-FCFS scheduling policy described in Section 2.3. Because the FR-FCFS scheduling policy cannot make use of inter-core locality information, it is possible that the DRAM controller schedules requests with high inter-core locality last. We assume the DRAM serves requests in an order R1→R3→R0. After R1 is served, Warp 1 can continue to execute the next independent instruction while other warps are still stalling. This is also the same for R3. After R0 is served, two requests (R0, R2) will be received by Core 0 and Core 1. Thus in the next cycle, Warp 0 and Warp 2 can execute the next instructions concurrently.

After 4 cycles, Warp 1 executed 3 instructions because its request is returned from cycle 0. Warp 3 executes 2 instructions after cycle 1. Warp 0 and Warp 2 execute 1 instruction because their requests returned on the cycle 2. The total number of instructions executed by Warp 0 to Warp 3 is 7 = 1+3+1+2. In this case, the Instructions Per Cycle (IPC) within these 4 cycles is:



(a) Example of FR-FCFS memory scheduling policy



(b) Example of inter-core locality aware memory scheduling policy

Figure 3.1: Example of using inter-core locality aware scheduling generates more ΔIPC. Wx represents warp x, Rx represents the memory requests that are sent by warp x. Wx+y indicates that warp x has executed next y instructions.

$$\Delta IPC_a = \frac{1+3+1+2}{4} = 1.75 \tag{3.1}$$

Figure 3.1b shows the same example with inter-core locality aware scheduling. The memory controller schedules requests with high inter-core locality first. When multiple cores send memory requests to access the same L2 cache line, they will be merged into L2 cache MSHR. In this case, the DRAM serves R0 first because R0 has highest MSHR merge length (length = 2). By serving this request, both Warp 0 and Warp 2 can execute the next instructions from cycle 1. After 4 cycles, the IPC is:

$$\Delta IPC_b = \frac{3+3+2+1}{4} = 2.25 \tag{3.2}$$

This example shows that we can improve performance by scheduling high inter-core locality memory requests first. In the next chapter, we will describe our inter-core locality aware scheduling policy in detail. In Chapter 5, we will show how L2 cache MSHR merge length is distributed across benchmarks.

3.2 Critical Path Analysis

Since we prioritize memory requests with inter-core locality, the request with no inter-core locality will be delayed. In Chapter 1, we have already given a brief overview on the effects of delaying memory requests either with or without inter-core locality. We design two separate experiments by delaying 1000 requests. One is delaying requests with inter-core locality. The other one is delaying requests without inter-core locality. The 1000 requests occupy different percentages of total requests in these applications. The performance impact varies from 1% to 39%. However, delaying requests with inter-core locality always

has 2X to 10X more performance impact than delaying requests without inter-core locality as shown in Figure 1.1b. This data indicates that requests with inter-core locality are more critical than requests without inter-core locality. Prioritizing inter-core locality requests will benefit performance because if the memory controller delays them, the impact on the performance is higher than requests without inter-core locality.

Chapter 4

Inter-core Locality Aware Memory Access Scheduling

This chapter describes our memory scheduling policy in detail. We first start with basic scheduling rules of an inter-core locality aware scheduler. We use the MSHR information to generate scores for each request and each DRAM row. Two types of row score selection policies are proposed. The score is used to help our scheduling policy making decisions. In order to handle starvation of the requests, we added an age information to our scheduling policy.

4.1 Overview of Inter-core Locality Aware Memory Access Scheduling

As describe in Chapter 3, inter-core locality can be represented by the number of merges of an L2 cache MSHR entry. Figure 4.1 shows an overview of inter-core locality aware memory access scheduling. When a request misses an L2 cache line, if there is no MSHR

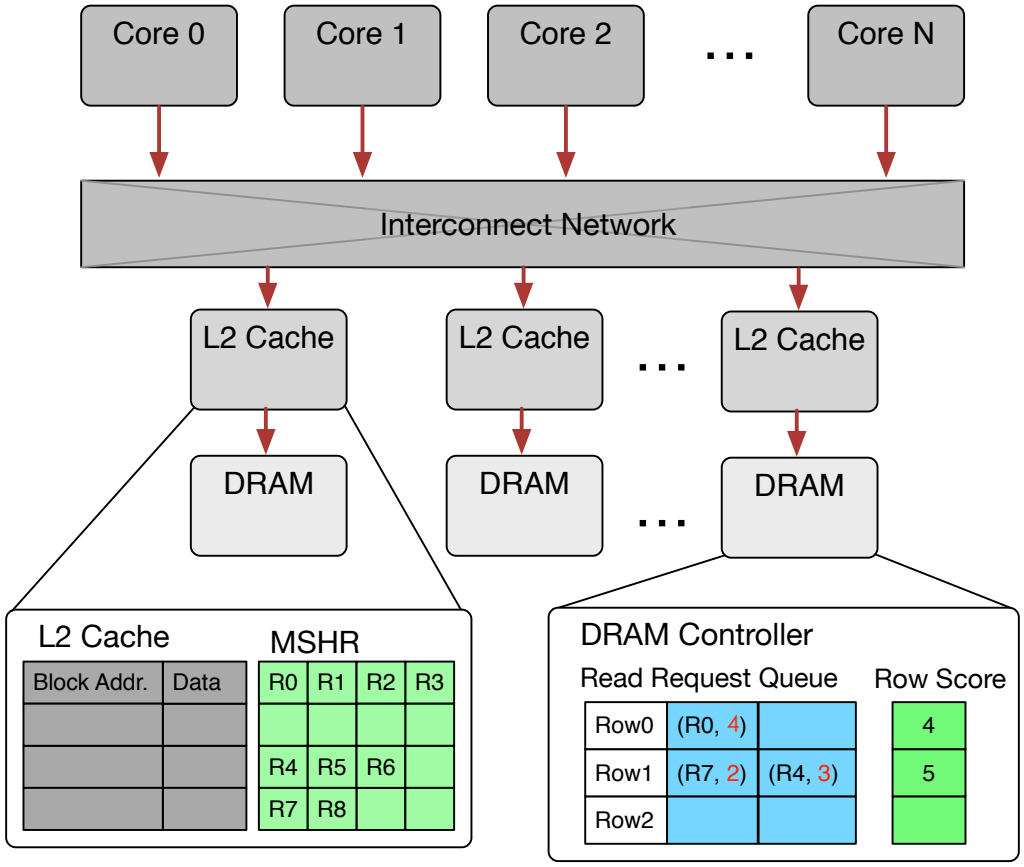


Figure 4.1: Overview of inter-core locality aware memory access scheduling. R0 to R8 represents request 0 to request 8. Red numbers beside requests are request scores

entry for this cache line, this means this is the first request for this cache line. A new MSHR entry will be allocated, and this request will be sent to the DRAM controller to get the data for the whole cache line. Because only one core sends the request to access the cache line, this request has no inter-core locality right now. On the other hand, if there is already an MSHR entry for the cache line, the inter-core locality occurs. There is no need to send this request to DRAM controller since there is already a request getting the data. This request

only needs to merge into the already exist MSHR entry. However, in our inter-core locality aware scheduling policy, we still send a new request called a *dummy request* that only brings the MSHR information. When the dummy request reaches the DRAM controller, we update the score with MSHR information and make scheduling decisions based on the new score.

Rule 1: Scheduling Rules

1. Largest request score for row-hit: When there are requests to access current opened row, schedule the one with largest request score first among all row-hit requests.
2. Largest row score for row-miss: When a row-miss occurs, choose the new row based on the row score. A row with the largest score is chosen. This row is opened, and policy 1 is used to choose a request.

A *request score* is assigned to each request in the DRAM read request queue. For each DRAM row, a *row score* is assigned based on the all request scores for this row. For example in Figure 4.1, There are three requests in the DRAM controller. R0 has a request score as 4 since there are four requests waiting for the same cache line in the L2 cache MSHR. For the same reason, R7 and R4 has a request score as 2 and 4, respectively. For each DRAM row, there is a row score. In the example, we use the summation of all requests score as the row score. For the row 1, the score is 5 which is the summation of request score R7 and R4. The request score is used to select requests and the row score is used to select rows. The scheduler first searches the request queue to see if there are any row-hit requests. Among all row-hit requests, the request with the maximum score is chosen first. If there are no row-hit

requests, the row with the maximum score is selected. Rule 1 shows our scheduling policy.

Write requests do not have a corresponding MSHR entry. The write request queue is drained when it reaches a high-watermark or when the read request queue is empty. The FR-FCFS policy is used on the write queue.

In the following section, we discuss two row score types.

4.2 Row Score Types

We first use the MSHR merge length as the request score. The row score is calculated based on the request score for the row. We explore two types of row score types for a DRAM row:

1. MSHR-M: Row score is defined as the largest request score in this row.
2. MSHR-S: Row score is defined as the summation of all requests' score in this row.

MSHR-M policy always schedules a request with the maximum MSHR merge length. This policy aggressively use make use of inter-core locality. When the DRAM controller selects a row, only one request that with the maximum L2 cache MSHR length is considered. But it ignores other requests in the same row. It is possible that other requests with selected row have low inter-core locality.

We improve MSHR-M policy to consider the inter-core locality of all requests in a DRAM row. We propose MSHR-S policy that augments the row score by using the summation of all requests' score in the same row as the row score. This policy can benefit the most memory requests in the MSHR by opening a row. The overhead of serving row-hit requests is low. The interval between two successive row-hit requests in the same bank is t_{CCDL}, which is three DRAM clock cycles with the bank group enabled in our GDDR5 configuration [15]. This is relatively small compared to the time to open a new row, which

needs tens of cycles. Thus, requests within the same row can be served in a short time. So even though every request in a DRAM row is not the one with highest inter-core locality, MSHR-S can still serve the maximum inter-core locality in the short time by opening a row.

4.3 Reduce Latency by Age Information

A DRAM row with a low score will starve because the proposed scheduler chooses the largest row score which is defined by the MSHR merge length. These requests can hurt performance if they have been waiting in the DRAM controller for a prolonged period of time. In order to prevent starvation, we propose a MSHR-S+A policy. MSHR-S+A augments MSHR-S by including age in addition to merge information from the L2 cache MSHR. Age in this thesis is defined as the life time of a memory request. When a request is generated, the age is 0. The age will increase by one each cycle. There is a latency for a memory request reaching the DRAM controller. When the memory request reaches the L2 cache, the age is the latency from the time it is generated. When a memory request arrives at the MSHR, the age is calculated for this request. By employing the MSHR-S+A policy, the request's score and row score now is defined using age, instead of solely the merge length as described in the previous section. The age of a request is defined as:

$$Age_{Req} = Current\ Time - Request\ Creation\ Timestamp \tag{4.1}$$

Only the first outstanding memory request within an MSHR entry is sent to the DRAM. To represent an age of all memory requests within an MSHR entry, the age of an MSHR entry is the summation of all request ages in this MSHR entry:

$$Age_{MSHR} = \sum Age_{Req} \tag{4.2}$$

When we pass the MSHR information, we pass Age_{MSHR} along with merge length. After the DRAM controller receives the MSHR merge information, which contains merge length and age, the age needs to be updated every cycle to represent the real age. This is because while a request is waiting in the DRAM controller, the age of this request is still increasing as time passes. Notice that a request in the DRAM controller may represent a number of requests in the MSHR. Upon each cycle, the age of each request in MSHR is incremented by one. In total, the age of an MSHR entry is incremented by the MSHR merge length every cycle. So instead of incrementing by one each cycle, the age of a request in the DRAM controller needs to be incremented by the number of MSHR merge requests.

4.4 Implementation Details

Figure 4.2 shows a detailed implementation of the inter-core locality aware memory access scheduler. We introduced a buffer called the Merge Information Buffer (MIB) in the L2 cache and the DRAM controller. The MIB records MSHR block addresses, merge length and age information. There is also a valid field to indicate whether an entry has already been sent to the DRAM controller. When a request misses an L2 cache line, if this request is the first one that misses the cache line, a new MSHR entry is allocated and a new request is pushed into the miss queue. If there is already an outstanding request for the cache line, the request is merged into an MSHR entry. At this time, the MIB is updated with a new merge length, the new age is calculated, and the valid bit is set to one. A multiplexer is introduced after the MIB and the miss queue to determine which request to send in each cycle. If the MIB has valid entries, this indicates there are requests that have not been returned from DRAM yet. It is important to update the DRAM controller with the latest information to prevent it to make a decision based upon the stale information. When the MIB has a valid

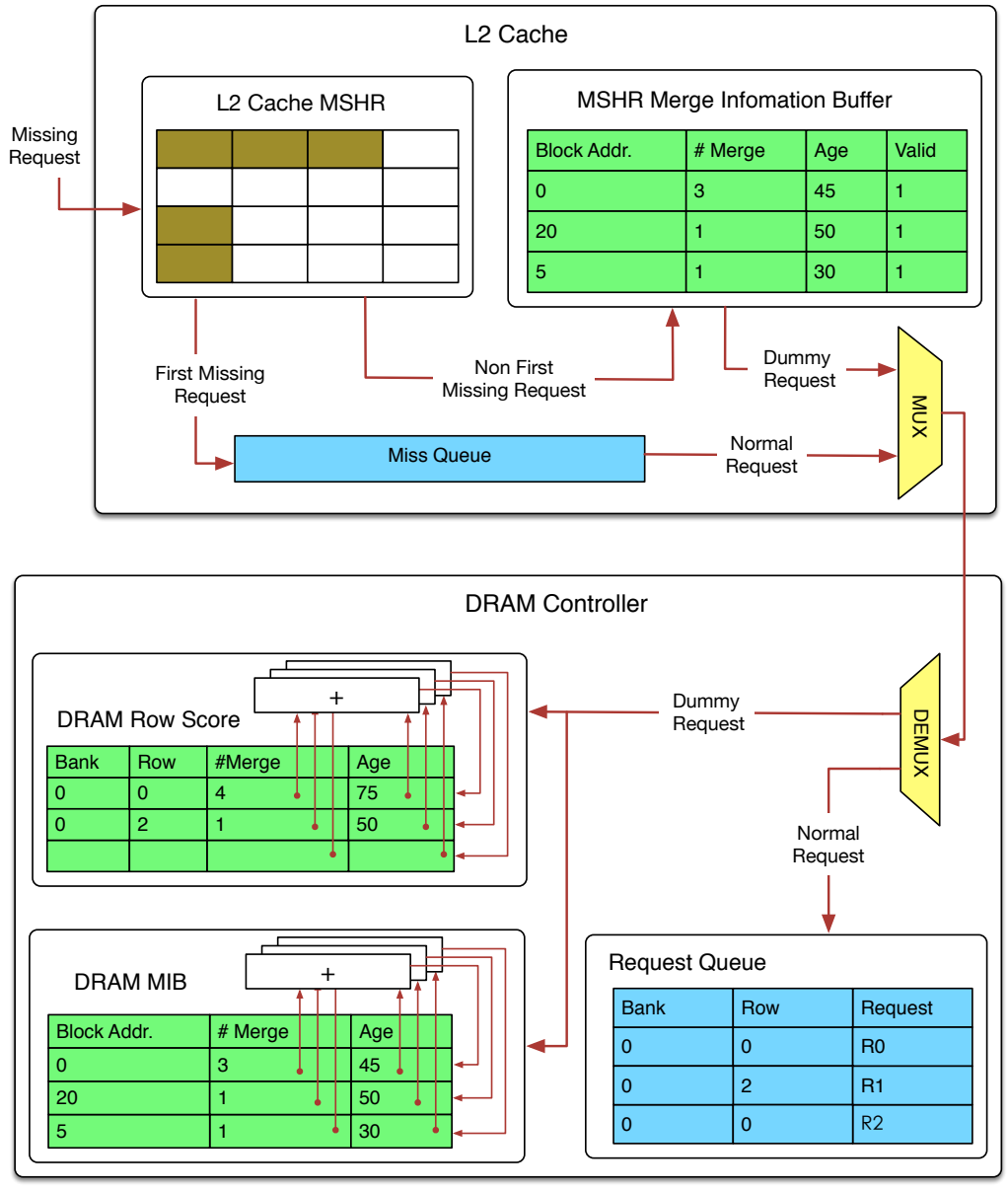


Figure 4.2: Implementation of inter-core locality aware memory access scheduling

entry, the multiplexer will send this entry first to inform the latest information. A dummy request with the maximum request score is generated and sent to the DRAM controller first. The dummy request includes the MSHR block address, merge length, age and a bit to indicate that it is a dummy request. After an MIB entry is sent, the valid bit is set to *zero*. This entry will be set to valid again if there is another outstanding request to the entry. If there is no valid entry in the MIB, requests from the miss queue are sent.

A demultiplexer is introduced to determine which structure a memory request should go to when arriving at the DRAM controller. Normal requests are pushed into the requests queue and dummy requests are used to update the MIB in the DRAM controller. When a dummy request reaches the DRAM controller, the merge length field in the dummy request is used to overwrite the merge length field in the DRAM controller and the age field in the dummy request adds to the age filed in the DRAM controller. Each entry in the MIB requires an adder to update the age each cycle. The adder takes age and merge length as the input and outputs the added result back to the age. Information in the MIB in the DRAM controller is used to help scheduling decisions as described above. When there is no corresponding information in the MIB, it means the merge length is one and the age can be obtained from the current request.

4.5 Hardware Overhead

In this section, we will analyze the hardware overhead of our proposed structure. For each MSHR entry, we need a corresponding MIB entry. In each MIB entry, an additional 20 bits are required to store the MSHR merge information. These 20 bits break down as follows. The merge length requires 4 bits for an MSHR entry with a maximum merge capacity of 16. We use 15 bits for the age field. When all 16 MSHR merges are occupied, each can record

a maximum age of $2^{15}/16 = 2048$ cycles in the memory controller clock domain. There is a chance that age exceeds 2^{15}. In this case, we saturate the counter at 2^{15}. The valid bit only requires a single bit. Each MSHR entry already has a block address field, which we can reuse in the MIB. Given that we have 64 MSHR entries in our baseline configuration (see Table 5.1), the total hardware cost added to an L2 cache is 160 bytes. This is 0.2% of total L2 cache.

The MIB in the DRAM needs to record an MSHR block address so that memory requests in the DRAM request queue can locate its corresponding MSHR entry. For a 32 bit address and 128 byte cache line, the block address requires 24 bits. Merge length and age require the same amount of bits as MIB in MSHR (4 bits and 15 bits, respectively). For each request in the DRAM read request queue, we need an MIB entry for it. The DRAM read request queue contains 64 entries from Table 5.1, requiring a total of 344 bytes. Also, each row in every bank needs to record the row score. In our baseline architecture, we have 16 banks and 4096 rows. To record banks and rows, we need 4 bits and 12 bits, respectively. Also, age and merge length require 19 bits in total. So for each DRAM row, we need 4+12+19=35bits. If each request in the DRAM read queue accesses different rows, the number of rows can be as many as the number of requests in the DRAM read queue. We need the number of row score entries to be the same as the DRAM read queue capacity which is 64. The row score requires a total of 280 bytes. We also need 128 15-bit adders for the DRAM row score and MIB. Given that 1-bit full adder needs 34 transistors, 15-bit adder needs 510 transistors. 1-bit SRAM needs 6 transistors, 510 transistors are 85 bits in storage. 128 15-bit full adders needs 1360 bytes. So the total overhead added to the DRAM is 344 + 280 + 1360 = 1984 bytes.

Chapter 5

Methodology

5.1 Configuration

To evaluate our proposed inter-core locality aware memory access scheduling policy, we use a modified version of GPGPU-Sim 3.2.2 [4]. The DRAM model in GPGPU-Sim 3.2.2 does not support separate read and write queues. There is a bus turnaround latency to switch between read requests and write requests. In order to minimize this latency, we add separate read and write queues in the DRAM controller as stated in Chapter 2. Table 5.1 shows our configuration. We use a GPU model similar to NVIDIA GTX480. To model the DRAM, we use Hynix 1Gb GDDR5 [15] as our DRAM timing model. The warp scheduling policy we use is Greedy-then-Oldest (GTO). This policy will switch to another warp only if the current warp stalls.

We perform the evaluation using a set of benchmarks from Rodinia [8, 9], CUDA Software Development Kit (SDK) [2], GPGPU-Sim [4] and LonestarGPU [5]. Table 5.2 shows the benchmarks used in this study. For LonestarGPU benchmarks, they use CUDA 5 but our simulator only supports CUDA 4.2. We tried to fix the problem and only get two of

Table 5.1: Baseline Configuration

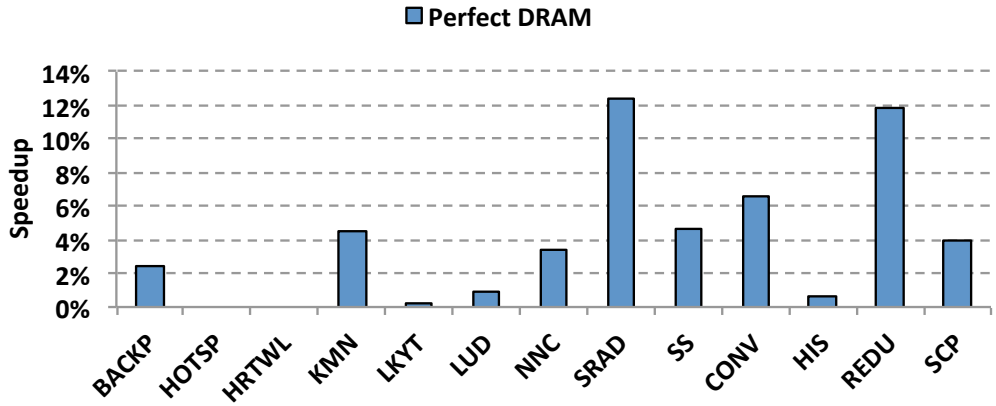
Configuration	Parameter
Number of Cores	15
Warp size	32
Max Threads / Core	1536
Warp Scheduling Policy	Greedy-then-oldest (GTO) [37]
L1 Data Cache / Core	16KB total size, 128B line, 4-way associative
L2 Unified Cache	64KB / Memory Sub-partition 128B line, 16-way associative
L2 MSHR	64 entries / Memory Sub-partition 16 merges / Entry
Number of Memory Partitions	6
Memory Sub-partitions	12, 2 / Memory Partition
L2 to DRAM Latency	20
DRAM Read Queue Capacity	64
DRAM Write Queue Capacity High/Low Watermarks	128 96/80
Core Frequency	1400 MHz
Interconnect Frequency	1400 MHz
DRAM Frequency	924 MHz
Number of DRAM Channels	6
Number of DRAM Banks	16, 4 / Bank Group
Number of DRAM Rows	4096
GDDR5 Memory Timing	Hynix H5GQ1H24AFR tRCD = 12, tRAS = 28, tRP = 12, tRC = 40, tCCDS = 2, tRRD = 6, tCL = 12, tWL = 4, tCDLR = 5, tWR = 12, tCCDL = 3, tRTPL = 2 (Unit in DRAM cycle)

them running. Not all the benchmarks are memory sensitive. We classify our benchmarks in the following section.

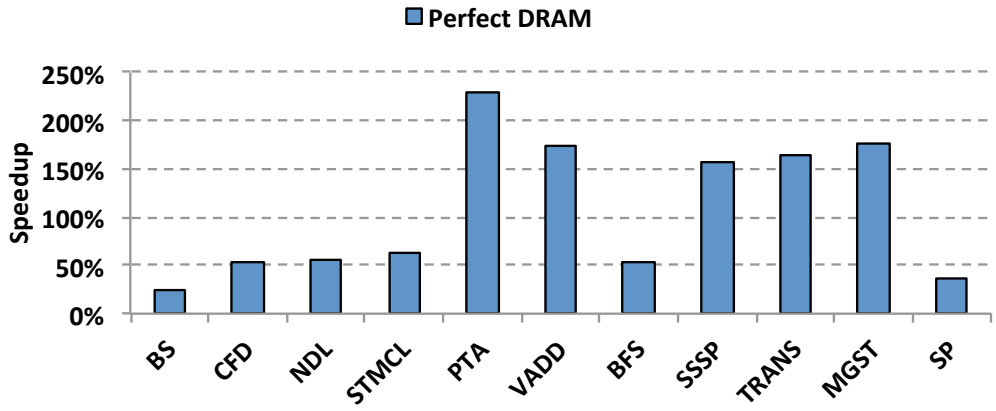
Table 5.2: Benchmarks

Memory Insensitive		
Name	**Abbr.**	**Suite**
Back Propagation	BACKP	Rodinia
HotSpot	HOTSP	Rodinia
Heart Wall	HRTWL	Rodinia
Kmeans	KMN	Rodinia
Leukocyte	LKYT	Rodinia
LU Decomposition	LUD	Rodinia
Nearest Neighbour	NNC	Rodinia
Speckle Reducing Anisotropic Diffusion	SRAD	Rodinia
Similarity Score	SS	Rodinia
convolutionSeparable	CONV	CUDA SDK
histogram	HIS	CUDA SDK
reduction	REDU	CUDA SDK
scalarProd	SCP	CUDA SDK
Memory Sensitive, Low Inter-Core Locality		
Name	**Abbr.**	**Suite**
BlackScholes	BS	CUDA SDK
CFD Solver	CFD	Rodinia
Needleman-Wunsch	NDL	Rodinia
Streamcluster	STMCL	Rodinia
Points-to Analysis	PTA	LonestarGPU
VectorAdd	VADD	CUDA SDK
Memory Sensitive, High Inter-Core Locality		
Name	**Abbr.**	**Suite**
Breadth-First Search	BFS	Rodinia
Single Source Shortest Path	SSSP	GPGPU-Sim
Matrix Transpose	TRANS	CUDA SDK
Merge Sort	MGST	Rodinia
Survey Propagation	SP	LonestarGPU

5.2 Classification of Benchmarks



(a) Memory insensitive applications



(b) Memory sensitive applications

Figure 5.1: Speedup using perfect DRAM

In order to know which applications are memory sensitive applications, we implement a perfect DRAM model which has a *zero-cycle* latency. When a memory request arrives at the memory controller, it is immediately ready at the DRAM return queue. Figure 5.1

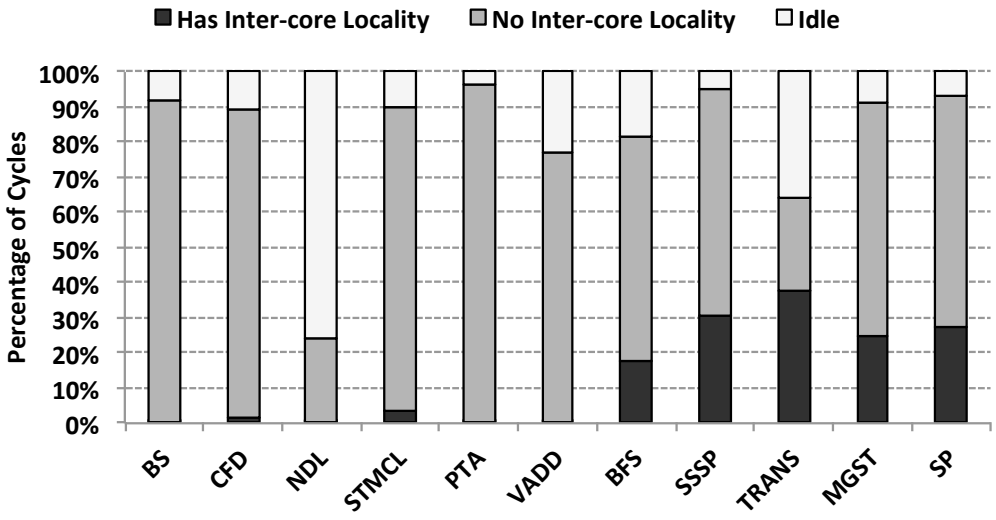


Figure 5.2: Cycle distribution of inter-core locality across benchmarks. We use the L2 MSHR merge length to represent inter-core locality.

shows the speedup when using perfect DRAM.

In Figure 5.1, we classify our benchmarks into two categories: 1) Memory insensitive applications. These benchmarks do not speedup much using perfect memory (speedup < 20%). The main reasons are 1) The application is compute intensive and 2) L1 or L2 cache miss rate is too low. While for some applications, we can increase input data size to get rid of low L1 or L2 cache miss rate. However, this can take a very long time to finish in the simulation environment. From our estimation, applications run 1s on hardware will run 10 days in the simulation environment. Applications in Figure 5.1a fall into this category. 2) Figure 5.1b shows all memory sensitive applications. These applications benefit significantly when using perfect memory. We focus on these applications only.

Our scheduler uses the L2 MSHR merge length to prioritize requests. Figure 5.2 shows a cycle distribution of the L2 MSHR merge length. When all MSHR entries are empty,

the MSHR is idle. The bar at the top indicates the percentage of cycles that MSHR is idle. When there are requests in the MSHR, if all MSHR entries' merge length is 1, this cycle has no inter-core locality. If any of the MSHR entries' merge length is larger than 1, it means there is inter-core locality. The black bar at the bottom indicates the percentages when this happens.

From Figure 5.2, we further classify applications based on the percentage of MSHR merge length. Memory sensitive applications with low inter-core locality (LIL) and memory sensitive application with high inter-core locality (HIL). Applications with MSHR merge length less than 10% are classified as Low Inter-core Locality (LIL). Applications with more than 10% MSHR merge length are classified as High Inter-core Locality (HIL).

Chapter 6

Experimental Results

In this chapter, we present the result of our inter-core locality memory access scheduling policy. Section 6.1 evaluates the performance of the inter-core locality aware DRAM scheduler for all applications. Section 6.2 gives a detail analysis on our scheduler. The analysis on memory request related stalls on L2 cache, memory access latency, data dependency stall, row locality and DRAM bandwidth is given to depth understand the inter-core locality scheduling policy. Section 6.3 gives an analysis on sensitivity of the DRAM request queue size and L2 cache to DRAM latency. We use the harmonic mean when computing average results.

6.1 Performance

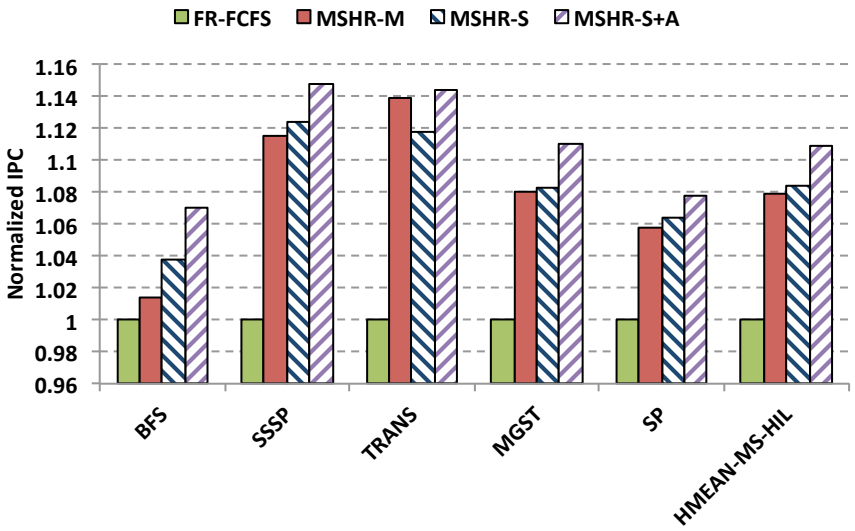


Figure 6.1: IPC for the memory sensitive, high inter-core locality applications

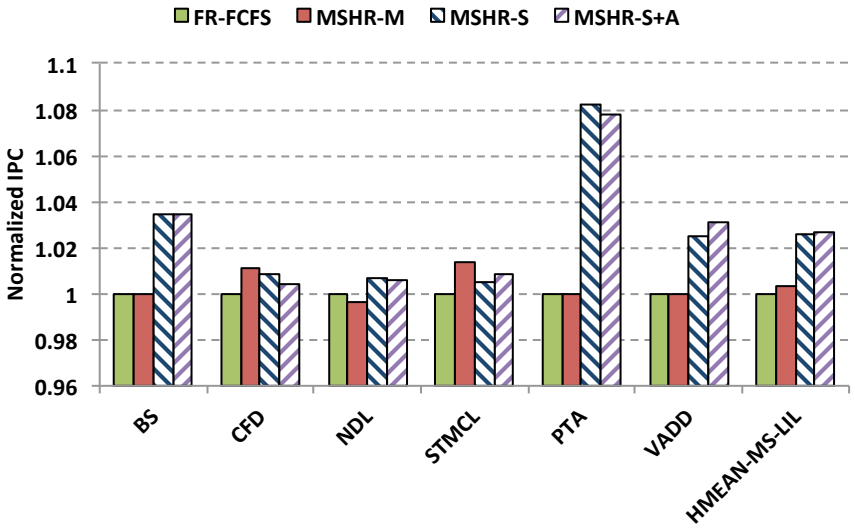


Figure 6.2: IPC for the memory sensitive, low inter-core locality applications

Figure 6.1 illustrates that the MSHR-S+A scheduler achieves an average 10.9% performance improvement over the FR-FCFS scheduler for the memory sensitive benchmarks with high inter-core locality. For the memory sensitive with low inter-core locality benchmarks, Figure 6.2 shows that MSHR-S+A scheduler has the best performance with an average 2.6% performance improvement and no benchmarks show performance degradation. If the inter-core locality is low, MSHR-S still does improve performance. This is because MSHR-S chooses a row with the most pending requests. Rixner et al. [36] use a similar technique called most pending policy. By serving the most pending row, other rows have a chance to wait for more requests thus improve overall row locality. Figure 6.8 shows this in detail.

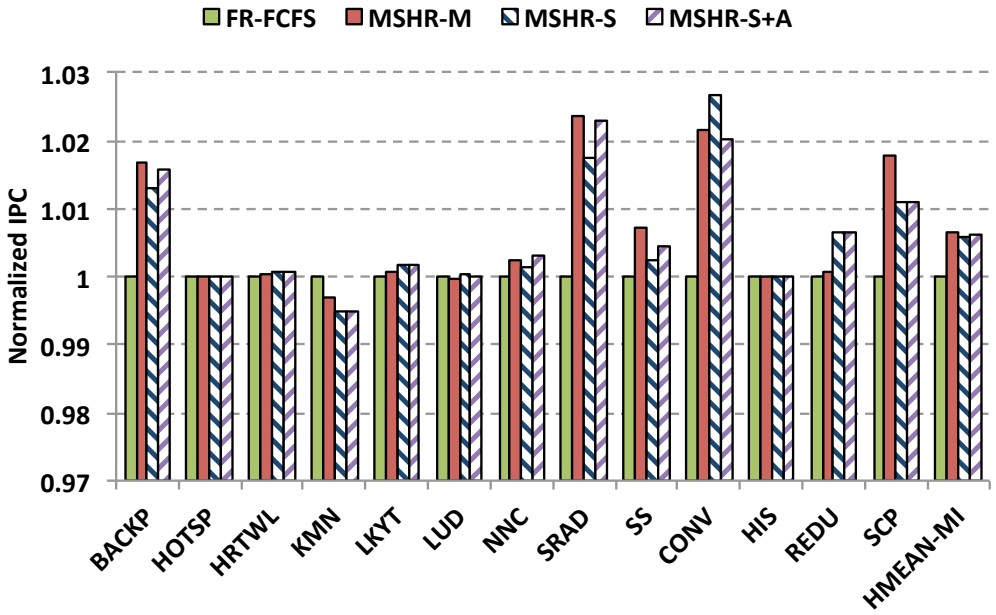


Figure 6.3: IPC for memory insensitive application

In summary, for memory insensitive applications, the average performance of the mem-

ory insensitive applications is improved by 0.7%. Only KMN shows less than 1% performance degradation which is negligible.

The following sections discuss the performance improvements when using the locality aware DRAM scheduler in more detail. In following section, we will only focus on memory sensitive applications.

6.2 Detailed Analysis

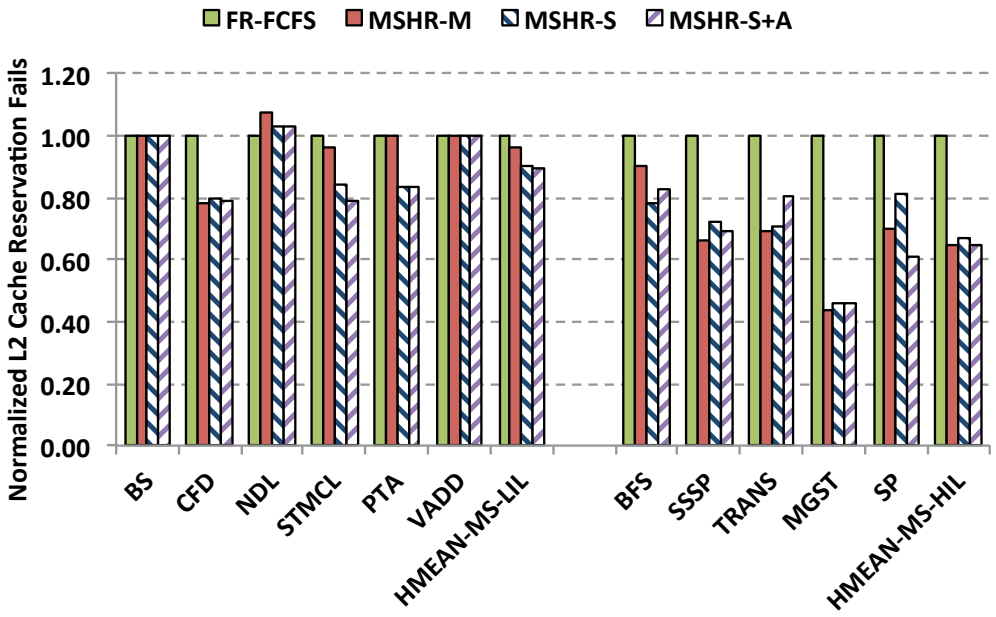


Figure 6.4: L2 reservation fails

Figure 6.4 shows the L2 cache reservation fails reduction normalized to FR-FCFS. The L2 cache reservation fails happens when there is a request trying to require the cache resource, but it cannot. There are three reasons that a missing request fails the L2 cache reservation. 1) The cache line requested by a request has been reserved by another request

but has not been filled. 2) The cache miss queue is full. 3) All MSHR entries are occupied.

Our inter-core locality aware scheduler reduces L2 cache reservation fails because of the third reason. Our scheduler releases MSHR entries to make room for other requests. For LIL and HIL benchmarks, we show 10.5% and 35% L2 reservation fails reduction, respectively.

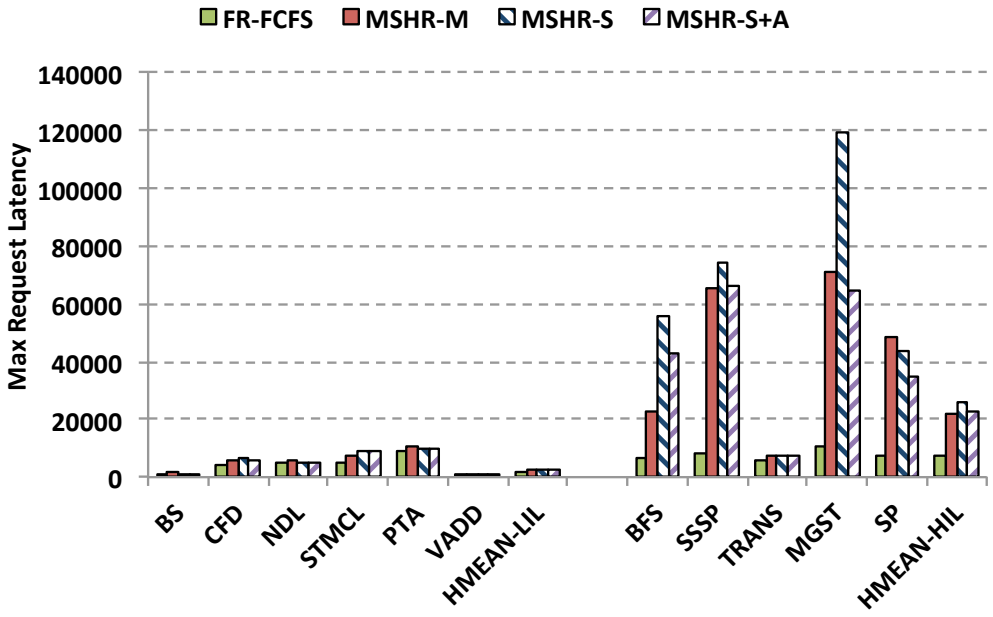


Figure 6.5: Maximum memory request latency

MSHR-M and MSHR-S scheduling policies do not concern about request latency. A DRAM row with low MSRH-S score is stalled for a very long time, which increases the maximum request latency. MSHR-S+A is proposed to reduce maximum memory request latency to prevent starvation. Figure 6.5 illustrate how our three scheduling policies impact maximum memory request latency. Our proposed scheduling policies all increase the maximum memory request latency. But MSHR-S+A has a better maximum memory request

latency by combining age information.

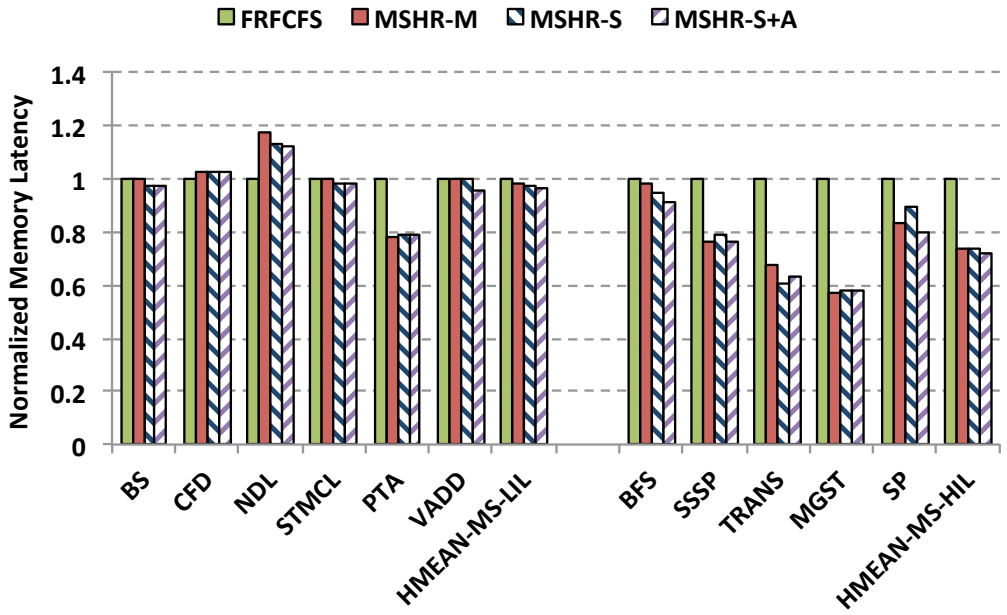


Figure 6.6: Average memory request latency

Figure 6.6 shows the average memory latency reduction of our proposed scheduler normalized to FR-FCFS. Memory latency reduction comes from releasing more memory requests in the L2 cache MSHR. MSHR-S reduces the total waiting time for all requests in the L2 cache MSHR. While MSHR-S+A reduces memory latency further by preventing a memory request from waiting for too long in the DRAM controller, it still serves requests with high L2 cache MSHR merge length. This is because Age_{MSHR} is the summation of all memory requests within an MSHR entry. If an MSHR entry has a very high MSHR merge length, the accumulation feature of Age_{MSHR} will ensure that this request still has high priority.

Scoreboard is a commonly used structure to keep track of data dependency between

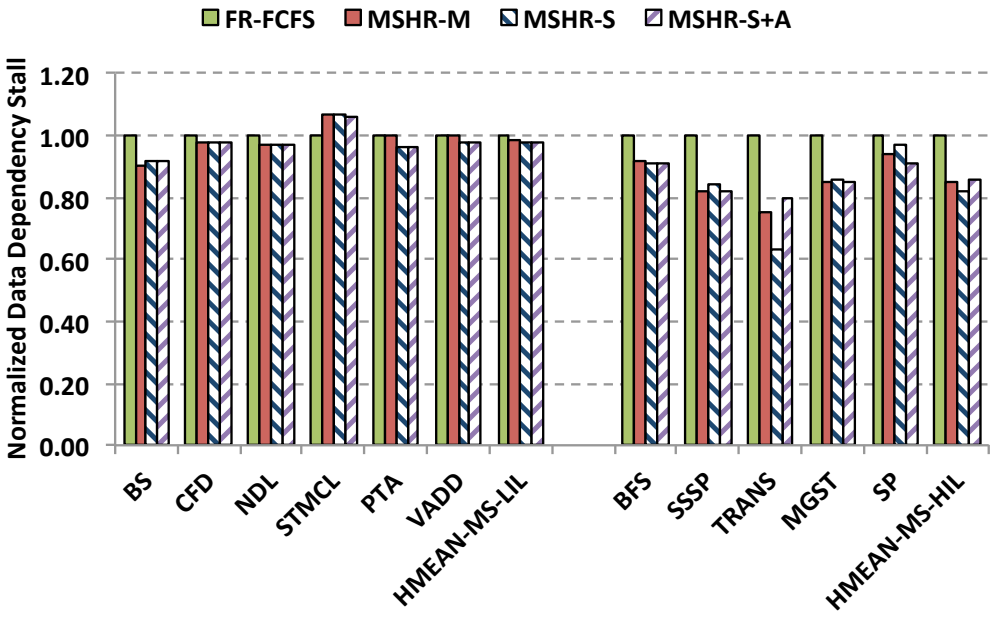


Figure 6.7: Data dependency stall, normalized to FR-FCFS

instructions. Write After Write (WAW) and Read After Write (RAW) hazards are reasons that make the scoreboard stall a warp. When a load instruction is issued, it usually takes tens to hundreds of cycles to get the data from memory. This is one important reason that the scoreboard fails. Our inter-core locality aware scheduler resumes more warps from stalling to reduce data dependency stall. Figure 6.7 illustrates an average reduction of data dependency stalls by 2% for LIL and 18% for HIL respectively, comparing to FR-FCFS. For HIL applications, MSHR-S has the largest impact on reducing scoreboard stalls. This is because MSHR-S reduces memory accesses waiting time in L2 cache MSHR, thus releasing the most warps that are currently waiting for the scoreboard.

Applications with low inter-core locality benefit from the MSHR-S scheduler because it improves row locality by choosing a row with most pending requests. When the MSHR

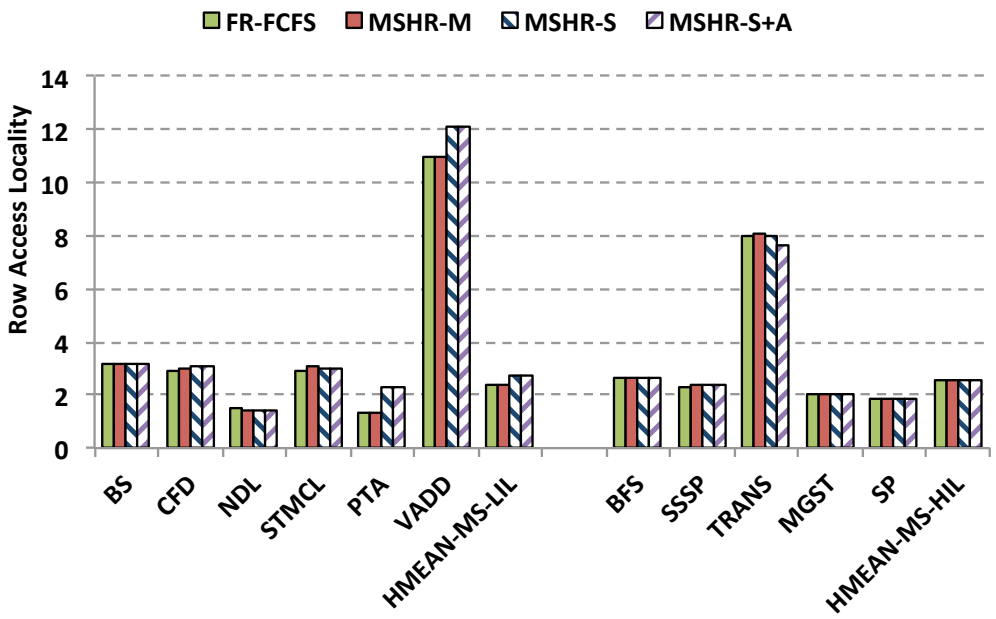


Figure 6.8: Row locality, the y-axis indicates average row-hit of all memory requests

merge length of all entries are one, MSHR-S chooses a row with most-pending requests. While a row with the most-pending requests is being serviced, the DRAM accumulates requests for other rows. Other rows will have a high average row access before the current open row is switching. Figure 6.8 shows row locality improvement. VADD and PTA have a high row locality improvement which turn into performance benefits as discussed in section 6.1. Other applications do not show a big row locality difference because the inter-core locality aware DRAM scheduler still uses row-hit first policy as in baseline FR-FCFS to improve DRAM bandwidth utilization. Figure 6.9 shows the DRAM bandwidth utilization improvement. The DRAM bandwidth is defined as total percentage of DRAM cycles that serves read or write requests. For all applications, our inter-core locality scheduler achieves similar DRAM bandwidth utilization. Because inter-core locality actually focus on proces-

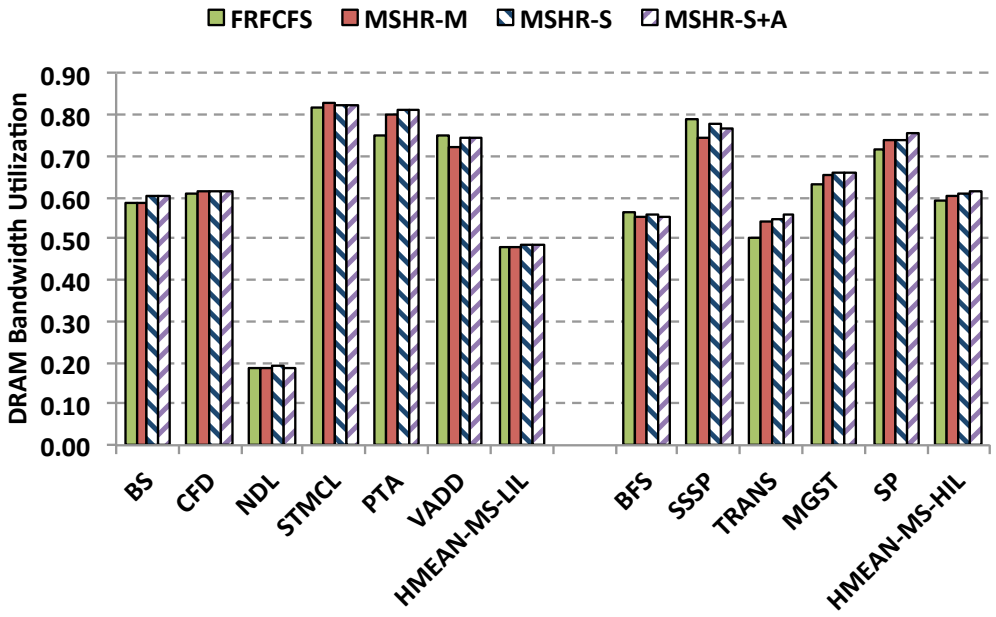


Figure 6.9: DRAM bandwidth utilizaiton

sor side efficiency, even with the small DRAM bandwidth improvement, the performance can have a big improvement.

6.3 Sensitivity Analysis

6.3.1 L2 Cache to DRAM Latency

The inter-core locality aware scheduler needs to send the MSHR information from the on-chip L2 cache MSHR to the DRAM controller. There is a latency between the L2 and the DRAM controller. Figure 6.10 shows different L2 to DRAM latency that affects the inter-core locality aware scheduler. In the baseline, we assume a 20 cycle L2 to DRAM latency. We evaluate the performance from 0 to 100 cycle latency. As latency increases, it takes longer for MSHR information to arrive at DRAM. This causes inaccurate scheduling

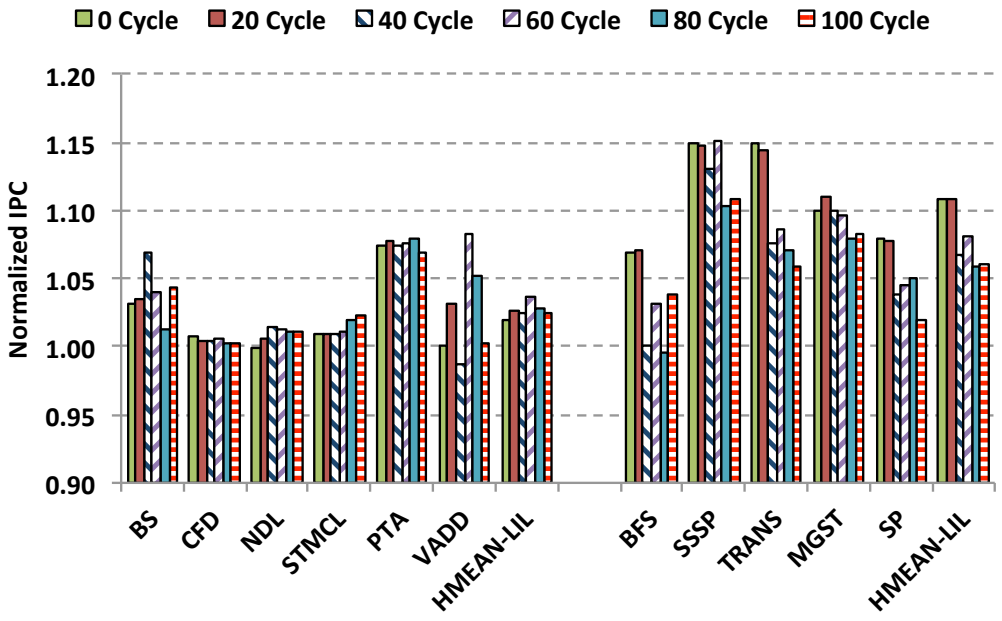


Figure 6.10: IPC normalized to FR-FCFS for different L2 to DRAM latency with MSHR-S+A scheduling policy

decision on inter-core locality.

From the Figure 6.10, the performance of VADD fluctuates as the L2 to DRAM latency increases. Since there is no inter-core locality in VADD, no L2 MSHR information is sent to DRAM. The L2 to DRAM latency does not affect the accuracy of scheduling. The MSHR-S+A scheduling policy now using a row with the oldest age which may benefit performance.

The overall performance improvement with our largest L2 to DRAM latency with MSHR-S+A scheduling policy is 6% for HIL applications and 2% for LIL applications(normalized to FR-FCFS).

6.3.2 DRAM Queue Size

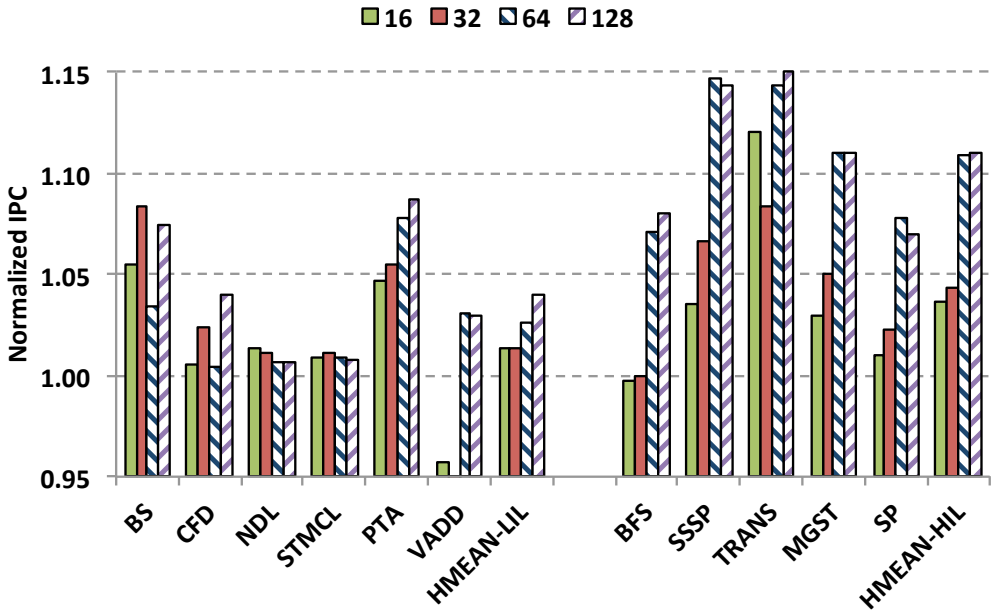


Figure 6.11: IPC normalized to FR-FCFS for different read request queue size with MSHR-S+A scheduling policy

Figure 6.11 shows the performance of our MSHR-S+A scheduler across HIL applications with different DRAM controller request queue sizes. The performance is better with a larger queue size. This is because with a larger queue size, inter-core locality aware scheduler can accumulate more memory requests in DRAM controller while waiting for their MSHR information. Since the DRAM controller has to wait at least for the L2 to DRAM latency to receive the MSHR information, if a request in the DRAM controller queue has not received its MSHR information, the inter-core locality aware scheduler will assume the merge length as 1 and the age as the request's age. This is inaccurate if there is another request in L2 that has been merged into the same MSHR entry but the information of this

MSHR entry has not been received by the DRAM controller. With large DRAM controller queue size, we can buffer the requests while waiting for its MSHR merge information to cover the long L2 to DRAM latency.

With the smallest configuration of the DRAM controller (queue size=16), MSHR-S+A achieves an average improvement in IPC of 4% with HIL applications and 1% with LIL applications, comparing to FR-FCFS. For HIL applications, a queue size of 128 shows a small performance improvement comparing to queue size 64. The reason is these applications have the largest MSHR merge length. A queue size of 64 entries has already accumulated enough requests waiting for its MSHR information. With our largest queue size of 128 entries using the MSHR-S+A scheduling policy, HIL shows 11% performance improvement and LIL shows 4% performance improvement with MSHR-S+A scheduling policy comparing to FR-FCFS.

Chapter 7

Related Work

In this chapter, we are going to present related works on memory access scheduling. We classify all these works into three parts: 1) Early explorations on memory access scheduling, 2) Memory access scheduling policies that focus on memory requests fairness and throughput, 3) Using criticality information to prioritize memory requests, 4) Hardware complexity effective memory access scheduling policies.

7.1 Early Memory Access Scheduling Explorations

Researchers already pay attention to memory access scheduling a long time ago. McKee et al. [27] observe an important problem for memory access in stream applications. Stream applications usually use a loop to access vectors. Within a loop, two memory references on two vectors are accessed. The addresses of the two references have a large gap so that two DRAM rows need to be accessed. Since the latency of row-hit access is much smaller than row-miss access, it is not efficient to serve these two references in a sequence. Instead, they propose a combined compiler and hardware technique to solve this problem. The compiler

is responsible to detecting the streams and unroll the loop so the consecutive references can be sent together. They propose a hardware called stream memory controller between the processor and the DRAM to interleave the references that can use DRAM efficiently.

The above work is for single processor. McKee and Wulf [28] further extend the above technique to multiprocessor. To choose memory requests that are sent by multiprocessor, they add two scheduling policies to schedule between multiprocessors. The first is cyclic scheduling to do round-robin like scheduling. The second is block scheduling that divides the input vector into blocks. The scheduler switch to another processor when current processor finishes the block.

Rixner et al. [36] exploit several memory access scheduling policies. All these memory access scheduling policies consider different aspect of the DRAM characteristic. FR-FCFS is proposed in the paper and is the most popular memory access scheduling policies today. Almost all of the commercial processors use the FR-FCFS today. The idea is to prioritize row-hit requests over row-miss requests to achieve maximum DRAM bandwidth utilization and high performance. We use the FR-FCFS as the baseline scheduler for the comparisons in this work.

7.2 Fairness and Throughput Memory Access Scheduling

There are a number of memory access schedulers proposed for multi-core systems recently [6, 25, 29, 30]. The primary focuses of these works are providing fairness among threads while keeping high bandwidth utilization. Mutlu and Moscibroda propose two memory scheduling policies, Stall-Time Fair Memory-Scheduling (STFM) [29] and Parallelism-Aware Batch Scheduling (PARBS) [30]. STFM provides quality of service to a shared DRAM memory system among all threads. The paper observes that a memory request from

one thread can be slowed down by memory requests from other threads due to interference between their memory requests. STFM balances all memory service time by scheduling memory request with the high slow down first. PARBS maintains quality of service from STFM while improving memory system throughput. PARBS groups memory requests into a batch and schedules them together to reduce memory latency across all threads. These techniques work on the multicore CPU but have not been tested on the GPU with thousands of threads.

Jog et al. [20] proposed a scheme to schedule memory requests between kernels. They use a First-Ready Round-robin First-Come First-Serve (FR-RR-FCFS) policy. The policy considers fairness between kernels using round robin scheduling and overall system throughput using FR-FCFS. This techniques ensures memory access fairness between GPU kernels. This is scheduling policy focuses on kernel level granularity while we focus on warp level granularity.

Lakshminarayana et al. [25] consider three aspects of GPU memory scheduling. They propose a memory scheduling policy to switch between Shortest-Job-First (SJF) and FR-FCFS to balance tolerance, SIMD-execution and row-buffer locality. This policy prioritizes a warp based on smallest memory request count to resume a warp as soon as possible. But it does not consider multiple warps. If multiple warps wait for a memory request, the memory request with smallest count is not benefit these warps. We consider memory requests that can resume largest warps from different cores.

Chatterjee et al. [7] exploit memory latency divergence within a warp. From the experiment they present, after a warp sends multiple memory requests, the latency among all these memory requests has a large divergence because of inter-warp interference. They proposed a memory scheduling policy to balance this latency divergence. They added an intercon-

nection network between all memory controllers for exchanging memory access latency divergency information. By using this information, the memory access latency divergence is reduced. Their work and our work solve the memory access latency problem from different aspects. Their work focuses on one warp that issues divergence memory requests. We focus on reducing the memory accesses latency of all warps form different cores that accessing the same data. Both techniques can improve performance on the GPU.

7.3 Memory Requests Prioritization

Ghose et al. [14] propose a scheme that assigns criticality information to memory requests. Memory request criticality in this scheme is defined as number of consumer instructions and historical stall time when a memory request reaches the head of ROB. This information is sent to the memory controllers to help memory controllers prioritizing threads that have the most criticality. Prieto et al. [33] use a similar technique by using request distance to head of the ROB as criticality information. They reduce a single thread stall time by using ROB which cannot reuse on the GPU architecture. Instead, we consider all threads on the GPU architecture. To do this, we use the L2 MSHR to reduce stall time of all threads.

Jia et al. [18] propose a scheme to reduce memory requests contentions between threads. This scheme focuses on efficiency of the L1 cache. To achieve this goal, they propose a memory prioritization buffer to reorder memory requests and bypass cache when necessary. To reduce cross-warp contention, the scheme reorders memory requests to avoid cache thrashing. To reduce intra-warp contention, the scheme by passes L1 cache and sends memory requests directly to lower level memory system. This work focuses on warp-level and L1 cache optimization to reduce memory access latency. Our work focuses on the last-level cache and the DRAM controller to improve overall memory system bandwidth.

7.4 Complexity Effective Memory Access Scheduling

The FR-FCFS is the most commonly used scheduling policies in GPU as mentioned previously. However, the hardware complexity of the FR-FCFS is high because it needs a large number of full-associative comparisons to exploit row-buffer locality. To reduce complexity design of memory controller, there are works on scheduling memory requests at early stages.

Yuan et al. [42] propose an interconnection network arbitration scheme to reserve row locality to replace complexity circuit design of FR-FCFS DRAM controller. They observe that the interconnection network which is between cores and memory controllers can destroy memory access row-buffer locality. To preserve the memory access row-buffer locality, they use an interconnection arbitration scheme to prioritize memory requests accessing the same row first. Using this scheme, they achieve a performance similar to FR-FCFS only using a simple FIFO memory controller.

Kim et al. [22] consider interconnection network congestion and row-buffer locality. To avoid network congestions, they use a local congestion aware function to control injection rate from each core. To prevent interconnection network interleaving memory requests, they use a technique similar to the scheme in Yuan et al. [42]. They introduce a superpacket that group packets in the interconnect using two configurations. The superpacket is grouped when there are consecutive requests accessing the same DRAM row from the same core or there are any requests accessing the same DRAM row from the same core.

But these scheduling policies do not outperforms FR-FCFS. They concern the hardware complexity rather than the performance. Our memory access scheduling policy outperforms FR-FCFS scheduling policy with little hardware overhead.

Chapter 8

Future Work

In this thesis, we use inter-core locality to schedule memory access. In the future, we plan to explore intra-core locality to improve performance further. Intra-core locality is captured by the L1 cache MSHR that is also a potential metric which can benefit performance. By utilizing the intra-core locality, we can make a warp-level fine-grained memory access scheduling policy. The L1 cache MSHR can be used for intra-core locality. The DRAM controller can get information about number of warps a memory request is representing. There are two problems need to be solved by using intra-core locality. The first problem is to handle large amount of on-chip network traffic because we need to send extra requests to inform the DRAM controller about L1 MSHR information. This will put a heavy pressure on the on-chip network. If we do not deal with it carefully, the on-chip network will congestion result in a bad performance. The second problem is long latency and low accuracy. Because there are a few hundred cycles to send a normal request from the processor side to the DRAM controller. When a request that carries intra-locality information reached the DRAM controller, a hundreds of cycles already passed. It is high chance that the intra-

core locality information in L1 cache has been changed. This result in a low accuracy of intra-core locality in the DRAM controller side. To help with these two problems, a possible solution is to redesign current interconnection networks to handle intra-core locality requests.

Chapter 9

Conclusion

In this thesis, we introduce inter-core locality for GPUs. Inter-core locality can be captured by L2 cache MSHR merge length. We quantify inter-core locality for GPU applications with thousands of threads. To exploit the inter-core locality, we introduce an inter-core locality aware memory scheduling policy by using L2 cache MSHR merge information.

We propose three scheduling policies. MSHR-M schedules a request with largest inter-core locality but does not consider other requests within the same row. MSHR-S uses the summation of request scores as a row score to choose a row that benefits most inter-core locality. MSHR-S+A further improves performance by preventing starvation of requests with low scores. We show a harmonic mean performance improvement of 11% with applications with high inter-core locality and 3% performance improvement with applications with low inter-core locality.

Bibliography

[1] NVIDIA CUDA C Programming Guide v4.2. http://developer.nvidia.com/nvidia-gpu-computing-documentation/, April 2015. → pages 1

[2] NVIDIA CUDA SDK code samples. http://developer.nvidia.com/cuda-downloads , April 2015. → pages 31

[3] Numerical analytics. http://www.nvidia.com/object/numerical-packages.html, February 2015. → pages 1

[4] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, 2009. → pages 31

[5] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '12, pages 141–151, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-4531-6. doi:10.1109/IISWC.2012.6402918. → pages 31

[6] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. Staged reads: Mitigating the impact of dram writes on dram reads. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-0827-4. doi:10.1109/HPCA.2012.6168943. → pages 51

[7] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing dram latency divergence in irregular gpgpu applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 128–139, Piscataway, NJ, USA, 2014. IEEE

Press. ISBN 978-1-4799-5500-8. doi:10.1109/SC.2014.16. URL http://dx.doi.org/10.1109/SC.2014.16. → pages 2, 52

[8] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi:10.1109/IISWC.2009.5306797. → pages 4, 31

[9] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-9297-8. doi:10.1109/IISWC.2010.5650274. → pages 4, 31

[10] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang. Memory latency reduction via thread throttling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 53–64, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi:10.1109/MICRO.2010.39. URL http://dx.doi.org/10.1109/MICRO.2010.39. → pages 2

[11] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, Atlanta, Georgia, USA, 2013. → pages 1

[12] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 0122007514. → pages 9

[13] R. Eckert. Page stream sorter for dram systems, May 20 2008. URL http://www.google.com/patents/US7376803. → pages 6

[14] S. Ghose, H. Lee, and J. F. Martínez. Improving memory scheduling via processor-side load criticality information. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 84–95, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi:10.1145/2485922.2485930. → pages 3, 53

[15] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf, April 2015. → pages 25, 31

[16] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123797519, 9780123797513. → pages 12

[17] H. Jang, A. Park, and K. Jung. Neural network implementation using cuda and openmp. In *Digital Image Computing: Techniques and Applications (DICTA), 2008*, pages 155–161, Dec 2008. doi:10.1109/DICTA.2008.82. → pages 1

[18] W. Jia, K. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 272–283, Feb 2014. doi:10.1109/HPCA.2014.6835938. → pages 53

[19] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 86–96, April 2013. doi:10.1109/ISPASS.2013.6557149. → pages 9

[20] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 1:1–1:8, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2766-4. doi:10.1145/2576779.2576780. URL http://doi.acm.org/10.1145/2576779.2576780. → pages 52

[21] Khronos Group. OpenCL. http://www.khronos.org/opencl/, April 2015. → pages 1

[22] Y. Kim, H. Lee, and J. Kim. An alternative memory access scheduling in manycore accelerators. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 195–196, Oct 2011. doi:10.1109/PACT.2011.37. → pages 54

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf. → pages 1

[24] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press. → pages 11

[25] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. Dram scheduling policy for gpgpu architectures based on a potential function. *IEEE Comput. Archit. Lett.*, 11(2): 33–36, July 2012. ISSN 1556-6056. doi:10.1109/L-CA.2011.32. URL http://dx.doi.org/10.1109/L-CA.2011.32. → pages 51, 52

[26] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008. ISSN 0272-1732. doi:10.1109/MM.2008.31. → pages 2

[27] S. McKee, R. Klenke, A. Schwab, W. Wulf, S. Moyer, J. Aylor, and C. Hitchcock. Experimental implementation of dynamic access ordering. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 1, pages 431–440, Jan 1994. doi:10.1109/HICSS.1994.323142. → pages 2, 50

[28] S. A. McKee and W. A. Wulf. A memory controller for improved performance of streamed computations on symmetric multiprocessors. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 159–165, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7255-2. URL http://dl.acm.org/citation.cfm?id=645606.660865. → pages 51

[29] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. pages 146–160, 2007. → pages 2, 51

[30] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 63–74, 2008. → pages 2, 51

[31] J. Nageswaran, N. Dutt, J. Krichmar, A. Nicolau, and A. Veidenbaum. Efficient simulation of large-scale spiking neural networks using cuda graphics processors. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2145–2152, June 2009. doi:10.1109/IJCNN.2009.5179043. → pages 1

[32] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi:10.1109/MICRO.2006.24. URL http://dx.doi.org/10.1109/MICRO.2006.24. → pages 2

[33] P. Prieto, V. Puente, and J. A. Gregorio. Cmp off-chip bandwidth scheduling guided by instruction criticality. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 379–388, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3.

doi:10.1145/2464996.2465019. URL http://doi.acm.org/10.1145/2464996.2465019. → pages 53

[34] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 245–258, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi:10.1109/PACT.2007.29. URL http://dx.doi.org/10.1109/PACT.2007.29. → pages 2

[35] S. Rixner. Memory controller optimizations for web servers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 355–366, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi:10.1109/MICRO.2004.22. URL http://dx.doi.org/10.1109/MICRO.2004.22. → pages 6

[36] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 128–138. IEEE, 2000. → pages 2, 6, 40, 51

[37] 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. ISBN 978-0-7695-4924-8. doi:10.1109/MICRO.2012.16. → pages 32

[38] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8:474, 2007. doi:10.1186/1471-2105-8-474. → pages 1

[39] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: coordinating DRAM and last-level cache policies. *ACM SIGARCH Computer Architecture News*, 38(3):72–82, June 2010. → pages 15

[40] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 409–422, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi:10.1109/MICRO.2006.44. URL http://dx.doi.org/10.1109/MICRO.2006.44. → pages 11

[41] R. Uetz and S. Behnke. Large-scale object recognition with cuda-accelerated hierarchical neural networks. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 1, pages 536–541, Nov 2009. doi:10.1109/ICICISYS.2009.5357786. → pages 1

[42] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 34–44, New York, New York, USA, Dec. 2009. ACM Request Permissions. → pages 2, 54