



Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations


Graduate School

2016

A REUSED DISTANCE BASED ANALYSIS AND OPTIMIZATION FOR GPU CACHE

Dongwei Wang

Follow this and additional works at: <https://scholarscompass.vcu.edu/etd>

 Part of the [Computer and Systems Architecture Commons](#)

© The Author

Downloaded from

<https://scholarscompass.vcu.edu/etd/4840>

This Thesis is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

©Dongwei Wang, December 2016

All Rights Reserved.

A REUSED DISTANCE BASED ANALYSIS AND OPTIMIZATION FOR GPU
CACHE

A Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at Virginia Commonwealth University.

by

DONGWEI WANG

September 2014 to December 2016

Director: Weijun Xiao,

Assistant Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

December, 2016

Acknowledgements

First, I would like to thank my parents and family members. Although live in different corners of the world, we still connect by our relationship, they give me strong support and encourage to pursue my degree. They also give me the warmth of a family, let me full of energy to face my daily life.

Second, I want to thank my thesis advisor, Dr. Weijun Xiao, from Electrical and Computer Engineering. We discussed many solutions in this research area and he always likes to help me and give a guide for my future work. Also, I would express my appreciation for my committee members: Dr. Preetam Ghosh and Dr. Zhifang Wang. As successful researchers in varied areas, they share the experience to do good research and give me valuable instructions to improve the Writing quality, their insightful opinions make my thesis better.

TABLE OF CONTENTS

Chapter	Page
Acknowledgements	ii
Table of Contents	iii
List of Tables	iv
List of Figures	v
Abstract	viii
1 Introduction	1
1.1 GPGPU computing	1
1.2 Cache in GPUs	2
1.3 Reuse distance in cache	3
2 Literature Review	7
2.1 GPU architecture	7
2.2 Scheduling policies	8
2.3 GPU cache evaluations	10
2.4 GPU cache improvements	11
3 Experiment methodology	13
3.1 Reuse distance calculation	13
3.2 Reuse distance classification	14
3.3 Experiment design	15
3.4 Simulation environment	16
3.5 Benchmarks	17
4 Experiment results and analysis	19
4.1 Hit rate vs. IPC	19
4.2 Reuse distance distributions	21
4.3 Capacity of L1 data cache	22
4.4 Associativity of L1 data cache	25
4.5 Summary	27

5	Cache Bypassing Optimization	28
5.1	Background introduction	28
5.2	Motivation	30
5.2.1	Memory access bypassing	30
5.2.2	Cache line prefetching	31
5.3	Benchmarks	32
5.4	Diagram of Bypassing Aware cache	33
5.4.1	Bypassing Aware cache work flow and bypassing decision . .	34
5.4.2	Replacement Policy	36
5.4.3	Hardware Cost	36
5.5	Experiment Results	37
6	Conclusion and future work	39
	Appendix A Abbreviations	40
	References	41

LIST OF TABLES

Table		Page
1	Classification of reuse distance	14
2	GPGPU-Sim Configurations	17
3	Benchmarks List	18
4	Benchmarks List for Bypassing	32

LIST OF FIGURES

Figure	Page
1 The hardware architecture of CPU and GPU	3
2 The miss rates of L1 data cache	5
3 GPU architecture and core diagram	7
4 The ratio of missed accesses for three types of cache	20
5 The IPC trends of BFS, SPMV and KM	21
6 Reuse distance distributions in a fully associative cache	21
7 The ratio of reuse distance distributions for all benchmarks	23
8 Hit rate trends with the increase of cache capacity	24
9 Reuse distance distributions of MUM, BCK, BT and LUD with different associativities	25
10 Reuse distance distributions of NW and FFT with different associativities	26
11 Hit rate trends with the increase of associativity	27
12 The work flow of bypassed memory access in GPUs	29
13 Reuse distance trends for BFS, AES, CP, and TPACF	31
14 Memory address difference	32
15 Profile for selected 8 benchmarks	33
16 Extra two bits for Bypass Aware cache	34
17 BA cache work flow	35
18 Normalized IPC with traditional bypassed cache design and BA cache . .	37

19	Normalized Cache Miss Rate between BA cache and traditional cache . . .	38
----	---	----

Abstract

A REUSED DISTANCE BASED ANALYSIS AND OPTIMIZATION FOR GPU CACHE

By Dongwei Wang

A Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2016.

Director: Weijun Xiao,

Assistant Professor, Department of Electrical and Computer Engineering

As a throughput-oriented device, Graphics Processing Unit(GPU) has already integrated with cache, which is similar to CPU cores. However, the applications in GPGPU computing exhibit distinct memory access patterns. Normally, the cache, in GPU cores, suffers from threads contention and resources over-utilization, whereas few detailed works excavate the root of this phenomenon. In this work, we adequately analyze the memory accesses from twenty benchmarks based on reuse distance theory and quantify their patterns. Additionally, we discuss the optimization suggestions, and implement a Bypassing Aware(BA) Cache which could intellectually bypass the thrashing-prone candidates.

BA cache is a cost efficient cache design with two extra bits in each line, they are flags to make the bypassing decision and find the victim cache line. Experimental results show that BA cache can improve the system performance around 20% and reduce the cache miss rate around 11% compared with traditional design.

CHAPTER 1

INTRODUCTION

1.1 GPGPU computing

Initially, GPUs are designed to do graphic processing. Multiple pipelines can simultaneously calculate the information of a pixel and update it on the screen, such as position, lighting, and depth etc.. With the development of architecture and hardware, GPUs have been widely employed to do General-Purpose computing on Graphics Processing Units(GPGPU).

GPGPU computing is a widespread research topic in academia and industry with the emerging architecture. Compute Unified Device Architecture, which is known as CUDA¹[1], is introduced by NVIDIA in November 2006. It is a general purpose parallel computing platform and programming model to accelerate the traditional computing efficiency by utilizing the powerful computing capability in NVIDIA GPUs[1]. Programmers could implement a CUDA program by programming with a C extension language, and launch the kernel function in GPU cores to speedup the computing. Many generations of CUDA-supported GPUs have been released to the market. Maxwell is the latest GPU architecture, integrated with 2048 CUDA cores[2], which is four times more than Fermi architecture(512 CUDA cores)[3]. Open Computing Language, known as OpenCL[4], is a similar programming platform with CUDA to run parallel program in heterogeneous platforms which are comprised by CPUs and GPUs. It is maintained by Khronos Group which is a non-profit technology consor-

¹We follow CUDA terminology in this paper

tium.

CUDA and OpenCL provide more flexibilities for programmers to utilize the tremendous computing power of GPUs. They are the key points for the popularity of high performance computing in academic and industry.

1.2 Cache in GPUs

As predicted by Moore's law[5], the performance gap between CPU and main memory is increasingly enlarged. Cache, as a bridge, is introduced according to the locality of programs with the purpose of narrowing this gap. With the improvements of hardware performance, GPUs can provide enormous computing power to accelerate GPGPU computing. Early generations of NVIDIA products, such as G80 and GT200, do not include L1 cache, only 16KB shared memory is accessible by threads in the same thread block. Fermi is the first GPU architecture which includes a 64KB L1 cache in each Streaming Multiprocessor(SM)[3] which is configurable with 48KB/16KB shared memory and 16KB/48KB L1 data cache. It is more flexible in Kepler to split the cache into 32KB/32KB[6]. The latest Maxwell has a dedicated 96KB shared memory, and L1 data cache has to share with texture memory[2].

Cache plays an important role in CPU-based computing. Normally, the hit rate of cache is high enough to gain performance improvements. Many prior works[7, 8, 9, 10, 11, 12, 13] have proposed a lot of improvements for CPU cache to make it work more efficiently. These improvements include flexible associativity, effective insertion, selective bypassing, and smart replacement. They aim to reduce the latency, which is essential for the performance of multi-core systems.

Yet, due to the distinct memory access patterns, this is another story in GPGPU computing model. Figure 1 compares the difference of architecture organizations between CPUs and GPUs. CPU, as a conventional processor, has less computing

units but large cache capacity and more control units. GPU, as a throughput-oriented device, has more computing cores to exploit the tremendous computing capability.

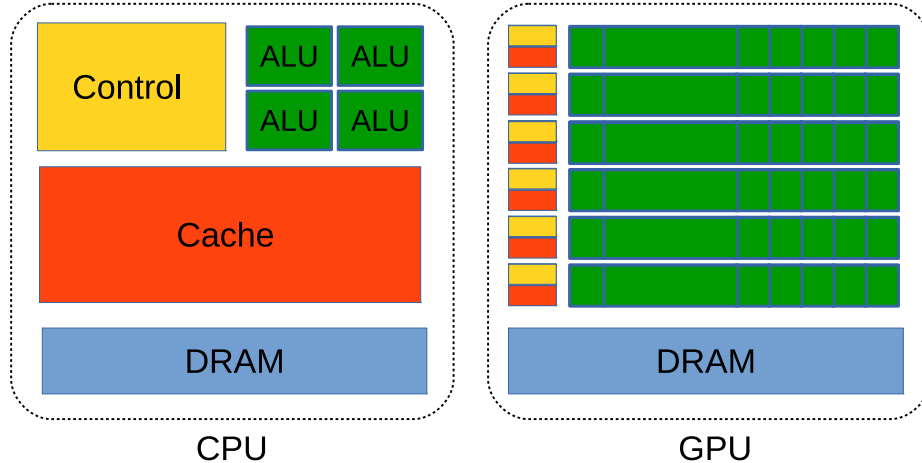


Fig. 1. The hardware architecture of CPU and GPU

SMs execute the same kernel in a fashion of Single Instruction, Multiple Data(SIMD). They deliver an extraordinary computing performance by launching thousands of parallel threads(work items in OpenCL terminology). Massive threads, which are lightweight units, issue memory accesses that are small in size but large in amount. These threads from different warps execute in an independent pattern. The inefficiency of cache mainly comes from threads contention and resource congestion[14]. In this work, we will explain the reason for this phenomenon and quantify the patterns of memory accesses.

1.3 Reuse distance in cache

Our work is based on the reuse distance theory. It can be used as a metric to evaluate the cache performance. Beyls et al. use a stack to represent the reuse distance(stack distance). When an address is referenced, it will move to the top of stack. The value of reuse distance is the depth before this address moves to top[15].

Reuse distance also can help to make an efficient replacement decision. Keramidas et al.[16] calculate the reuse distance and select the cache line with longest remaining distance as a victim via instruction-based(PC) predictions. Duong et al.[17] propose the protection distance to protect a line to stay in cache for a certain time span and dynamically calculate the reuse distance. All these works focus on CPU cache. With the booming developments of GPUs in recent years, some researchers evaluate the cache performance with reuse distance in GPU cores. Tang et al.[18] analyze GPU programs by building a model to predict the cache miss rate. They focus on two subproblems: single thread blocks and multiple thread blocks. The former subset is based on the analysis of stack distance profiling. Nugteren et al.[19] add five extensions in cache to apply the reuse distance theory in parallel computing model of GPUs.

Currently, the cache design in GPUs is similar to the philosophy of CPU's. Because of the distinct access patterns, the hit rates of GPUs cache are far lower than CPUs. Figure 2 shows the miss rates of all benchmarks we choose in this paper 3.5. It is a 16KB, 8-way set associative L1 data cache in this experiment. The size of each cache line is 128 bytes in this paper. These statistics are the summations of all memory accesses and cache misses. AES and NQU miss all of the cache requests. HST, NW, PF and KM give near 100% miss rates. Although the miss rates of NN and TPACF are close to zero, most benchmarks are around 40%-80%. The rightmost bar gives the average value of miss rate for all 20 benchmarks, which is 60%.

Obviously, a high miss rate fails to take the advantage of data locality and aggravates the cache contention. Sometimes, thrashing[20] makes the performance worse if too many cache lines frequently switch in and out. Thousands of threads lay more burden on L1 data cache and thus exacerbates the resource utilization as missed cache requests have to go through the interconnect to access the L2 cache.

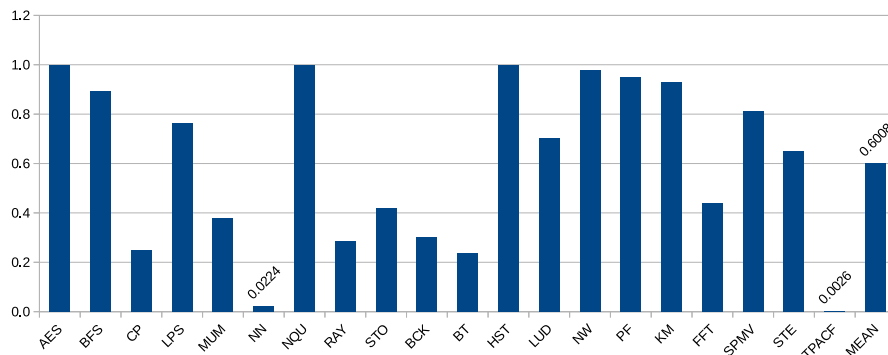


Fig. 2. The miss rates of L1 data cache

Bandwidth is the key point to impact the performance[21, 22]. It will degrade if too many accesses severely consume the bandwidth due to a high miss rate of L1 data cache.

Based on above considerations, we present a detailed analysis on the microarchitecture of L1 data cache and the memory access behavior of all benchmarks. We observe the impacts of cache capacity, set associativity for L1 data cache, then plot the reuse distance distributions and explain why GPU cache exhibits these characteristics. Following the observations, we suggest some optimization techniques to improve the IPC of GPGPU benchmarks, and implement a Bypassing Aware Cache(BA cache) which could improve the performance around 20% with negligible hardware cost.

The most insightful observations and contributions in this paper include:

- The capacity of L1 data cache does not cause obvious variations of cache performance. 17 out of 20 benchmarks maintain a stable hit rate when the cache capacity increases(section 5). Although there is a high mean value of miss rate, most of them do not suffer from the capacity miss. 64KB of a cache is as large as enough to resolve the majority of all capacity misses for most benchmarks.

- Only NW and FFT suffer from the conflict miss, they benefit from a large associativity. Although MUM, BCK, BT, and LUD have relative evenly reuse distance distributions in a fully associative cache, they do not really suffer from the conflict miss after we test them in a 8-way set associative cache. Eight cache lines for each set are adequate for most benchmarks to exploit the data locality.
- The inefficiency of GPU cache not only comes from thread contention and resource over-utilization, but also relates to the memory access pattern of each benchmark. Streaming and short reuse distance accesses take the most part of entire accesses. The former one causes the compulsory miss. The latter one takes the advantage of current cache design because of the temporal locality. Both of them do not benefit from large capacity and associativity.
- We discuss the optimization suggestions for L1 data cache which could help to improve the cache performance, and implement a cost economic cache design, BA cache, only has two more bits than traditional cache line. BA cache achieves around 20% IPC improvement, around 11% cache miss rate reduction on selected 8 benchmarks.

The rest of this paper is organized as follows: Chapter 2 presents the literature review. Chapter 3 illustrates the approach to calculate the reuse distance and how to plot the distributions. Chapter 4 explains our experimental methodology and presents the experiment results and observations. We detailed introduce the BA cache in chapter 5, and get our conclusion at chapter 6.

CHAPTER 2

LITERATURE REVIEW

2.1 GPU architecture

SM is the basic unit to execute the kernel in GPUs. It includes all hardwares to implement the SIMD computing model. All SMs are grouped into clusters. In the left part of Figure 3, the clusters connect to a unified L2 cache via interconnect and the L2 cache directly connects with DRAM.

The right part of Figure 3 exhibits the internal organizations of a SM. All instructions are fetched from instruction cache. Warp schedulers select warps to saturate the pipeline to deliver tremendous system throughput and hide memory access latency. GPU core pipelines execute arithmetic and transcendental calculations and L1 cache processes cache requests.

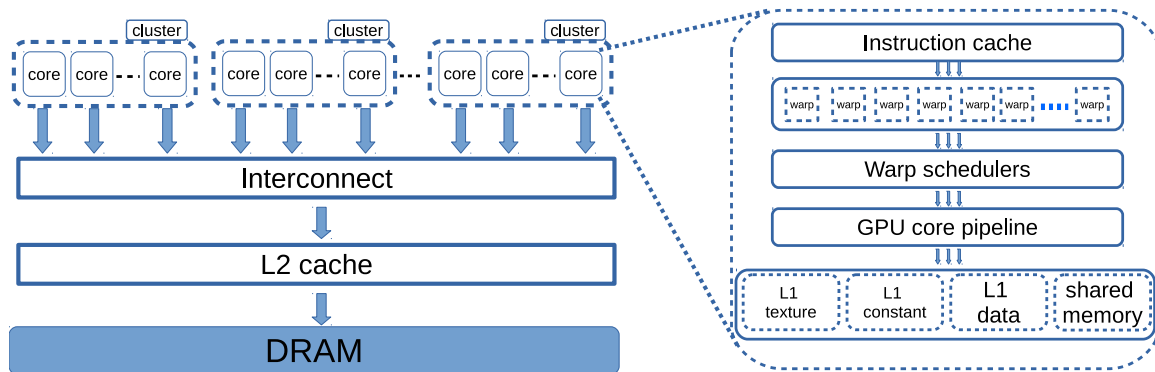


Fig. 3. GPU architecture and core diagram

All memory accesses are coalesced before accessing L1 data cache. At most, 16 memory accesses form a bigger one if they access successive addresses. If a memory access misses in L1 data cache, it will be inserted into the Miss Status Holding

Registers(MSHRs). MSHRs merge the duplicated requests if a same request already exists, otherwise, a new entry will be allocated to the missed request. If no entry is available in MSHRs, the memory pipeline will stall. The L1 cache is comprised by three parts: texture and constant cache (which are read-only), and L1 data cache. Shared memory is a per-core scratch memory which is managed by software, it is accessible by programmers. In this work, we focus on L1 data cache which processes the global and local memory accesses, including read and write.

2.2 Scheduling policies

There are two level scheduling in GPUs. The first level is Cooperative Thread Array(CTA) scheduling[23, 24, 25], CTA scheduler assigns thread blocks to GPU cores according to the availability of resources. Second level is warp scheduling[26, 27, 28, 29, 30, 31], which selects the ready warps to execute in GPU core pipelines.

Kayiran et al.[24] give a dynamic CTA scheduling policy by the utilization of DRAM. Since memory contention is the bottleneck for performance, assigning maximum CTAs to each core can not always yield best performance, especially for memory intensive workloads, in contrast, it degrades the performance. Based on this observation, they initially assign half of maximum number CTAs to each core, increasing and decreasing the number of CTAs at run time according to the memory utilization.

Jog et al.[23] propose four schemes from the perspective of CTA to improve the performance of GPGPU benchmarks and named them by OWL. (1) CTA-aware scheduler is a two-level scheduler which prioritizes part of CTAs in a core, and schedule them first until all of them stall due to memory access. (2) CTA-aware-locality scheduler schedules the warps stalled by long memory latency if their data come back from DRAM. This reduces the number of CTAs accessing L1 data cache and make L1 data cache available for less CTAs. (3) CTA-aware-locality-BLP exploits the

bank level parallelism(BLP), it prioritizes non-consecutive CTAs to schedule them first; however, it reduces the row locality of a bank. (4) Opportunistic prefetching improves the row locality by adding an opportunistic prefetching mechanism to OWL. The prefetcher loads an already opened row into L2 cache for further memory access.

Lee et al.[25] contribute two thread blocks scheduling policies: 1) lazy CTA scheduling(LCS) gets the optimal number of CTAs for each core by a greedy scheduling policy and 2) block CTA scheduling(BCS) dispatches consecutive thread blocks to the same core. LCS finds the optimal number of CTAs for each core in order to maximize the performance. BCS assigns sequential CTA blocks to the same core to exploit inter-CTAs locality. In particular, they exploit the interaction between the thread block scheduler and the warp scheduler to maximize resource utilization.

Fung et al.[27] illustrate a dynamic warp formation to handle severe branch divergences. Typically, GPUs launch massive threads in SIMD pipelines to achieve high throughputs, the performance will degrade if there are too many control flows in a kernel function. They reorganize the threads formation at run time to group the active threads in a new warp to saturate the pipelines. Meanwhile, they also establish that the immediate post-dominator of branch divergence is the optimal options.

Narasiman et al.[26] introduce two main contributions to improve GPU performance. First, they propose a large warp microarchitecture, it can improve the occupancy of resources when there are badly branch divergences in a kernel. The main idea is to organize the threads into a larger group, so that the scheduler can have more active thread candidates when branch occurs. Second, they present a two-level warp scheduling policy to hide the memory access latency. Compared with all threads in a block executing in the same time line, they divide all threads into different fetch groups with varied priorities. The fetch groups with lower priority start to execute only if the fetch groups with higher priority complete or stall for long memory

reference cycles, so that massive threads can hide the memory reference latency.

Lee et al.[31] illustrate an instruction-issue pattern-based adaptive warp scheduling(iPAWS). IPAWS dynamically selects a warp scheduling policy from Round Robin(RR) and Greedy Then Oldest(GTO), which favors current kernel based on the instruction-issue pattern. RR and GTO are two popular warp scheduling policies; however, a benchmarks only favors one of them. They sample the number of issued instructions until there is one warp finishes execution. Based on the shape of sampled number to dynamically make the decision which scheduler is better.

2.3 GPU cache evaluations

GPU, initially used only as a graphics processing device, has already been widely applied to all kinds of computation intensive fields. As an emerging architecture, we are lacking of a profound understanding about how to maximize the computing capability. Intuitions from CPU fail to make GPU work as we expected. Some researchers have noticed this insufficiency and exerted to convey a comprehensive knowledge for GPU[22, 32, 33, 34].

Jia et al.[22] present the taxonomy of three types of locality for GPU: within-warp locality, within-block locality, and cross-instruction reuse. The difference between the first two characterizations is the origin of the reused requests. For cross-instruction reuse, it is hard to exploit the locality in cache. They also argue that the bandwidth limitation, rather than the latency, often acts as an obstacle for system performance. Compared with their work, we focus on memory access patterns which directly work on cache. It is a finer-grain cache evaluation work. Our optimization suggestions also consider more about the bandwidth with the purpose to improve the hit rate and reduce the resource consumptions.

As an comparative evaluation work, Hestness et al.[32] analyze the effectiveness

of memory hierarchy(L1, L2, and DRAM) in heterogeneous system. This work pays more attention to the memory hierarchy of the entire system by comparing the difference of locality and coalescing between CPU and GPU. They also demonstrate that massive threads can hide the long latency and the bandwidth is sensitive to off-chip burst requests. However, our work concentrates on L1 data cache in GPU by classifying the memory accesses depending on reuse distance theory. We provide more detailed statistics for L1 data cache and give the insightful observations.

2.4 GPU cache improvements

The inefficiency of cache in GPU cores has already drawn universal attentions. A lot of advanced techniques have already contributed to improve the cache performance, generally including profiling, reordering, bypassing and throttling.

Profiling[35, 36, 37] is a technique to get the characteristic of a benchmark by analyzing a small part of input. Normally, it takes marginal resources and negligible time. The profiling results support us to make an optimization decision. The disadvantage is the profiling results only represent partial behavior of a benchmark. Reordering[38] aims to reorganize the sequence of memory accesses for cache. Ideally, the threads from a warp will access successive memory addresses. Issuing these accesses sequentially could exclude the interferences and utilize the locality. Bypassing[14, 35, 38] means to bypass some memory accesses from cache to lower level memory hierarchy. The significant issue about bypassing is the candidates selection from the entire work set. Throttling[39, 24, 14] is employed to control the number of active warps according to the status of memory. The purpose is to alleviate cache contention. These techniques generally work together to improve the cache performance.

Xie et al.[35] introduce a static and dynamic approach of cache bypassing based on profiling. They classify cache requests into three categories at compilation. By-

passed memory accesses are intelligently selected at static and dynamic time. Chen et al.[14] propose a cache management and a warp allocation scheme which is named Coordinated Bypassing and Warp Throttling(CBWT). They take a similar protection policy with Duong et al.[17] which keep a cache line from eviction until it's protection distance is 0. They also observe the status of memory system to make the decision of warp throttling. Jia et al.[38] illustrate Memory Request Prioritization Buffer(MRPB). They suggest to reorder the sequence of memory accesses by buffering them in a queue. Bypassing is triggered when cache contention is detected.

CHAPTER 3

EXPERIMENT METHODOLOGY

3.1 Reuse distance calculation

Reuse distance can be used to classify three 'C's of cache misses[15]. Since two accesses to a same cache set are randomly mapped to cache lines, and two accesses to different sets do not interact, the calculations are processed in the unit of a cache set. Each set is independent to response cache requests, and we call it one work set. We calculate the reuse distance by the following principle: *the number of unique memory accesses between two accesses in a cache set which have the same tag part*. If a memory address is referenced the first time, the value of reuse distance is $+\infty$ [15]. For a given configuration, assuming a n-way set associative cache, a memory request will only map to one cache set, but possibly any cache line in this set. In GPGPU computing, a cache request can not cross two cache lines, so we only compare the tag and index part of each request rather the memory address. If the access result of a cache request is reservation failed or hit pending, it does not change the status of any cache line, so we exclude it from the work set. The value of a reuse distance is greater than or equal to 0.

Suppose we have a 4-way set associative cache which employs LRU replacement policy, the reuse distance of a cache request is d . If $d \geq 4$, it will miss, the previous request for this address has already be replaced out from cache. If $0 \leq d < 4$, it will hit, the last request for this address still resides in cache for data reuse. As a metric to measure the performance, the reuse distance for a memory access varied from different cache configurations.

3.2 Reuse distance classification

After calculating the reuse distance of all memory accesses, we divide them into different categories by their value. Suppose there is a n -way set associative cache with LRU replacement policy, we simply divide all memory accesses into three categories. Although the reuse distance is calculated in each cache set, we sum all cache sets up since they are configured the same associativity. Still, assuming d is the reuse distance for a memory access, all three categories and the principles about how to classify a memory access are listed in Table 1.

Table 1. Classification of reuse distance

category	hit/miss	range of d
$rd0$	hit	$[0, n)$
$rd1$	miss	$[n, +\infty)$
$rd2$	miss	$+\infty$

The above classifications depend on the value of d and n . All memory accesses in $rd0$ hit in cache. They have a short reuse distance which is less than the set associativity to take the advantage of data locality. The $rd1$ and $rd2$ are cache misses. However, the reasons, why they miss, are varied. The $rd1$ is the collection of those memory accesses which have a reuse distance greater than or equal to set associativity, they have accessed cache before, but subsequent requests replace them due to the limitation of the number of cache lines. They can be attributed to the conflict miss in a set associative cache and the capacity miss in a fully associative cache. The $rd2$ stands for the streaming accesses, they access cache only once, so the result of a cache request is a miss.

3.3 Experiment design

With the purpose of analyzing the relationship between the memory access patterns of GPGPU computing and the inefficiency of GPU cache, we conduct our experiments in the following three steps.

First, we collect the access patterns of all benchmarks at a 16KB fully associative cache and calculate the reuse distance of each memory access then plot the distributions. The reason to test a fully associative cache is that all memory accesses map to the only one available cache set. It includes entire memory accesses. The value of reuse distance is maximum in this scenario. If it is a set associative cache, all memory accesses are mapped to different cache sets by the index part of their addresses. Each cache set is a subset of the total amount, so the reuse distance is less than or equal to the value at a fully associative cache. These statistics are plotted in Figure 6; however, it is a finer granularity classification than Table 1, and all memory accesses are divided into seven intervals. Based on these distributions, we conduct the following analysis for all benchmarks.

Second, the cache capacity and associativity are two significant parameters in cache design. We evaluate the cache performance from these two perspectives. In previous experiment, there are 128 cache lines in total. Cache can not hold the memory accesses whose reuse distance is greater than or equal to 128(except $+\infty$). The ratio of this distribution is significant to evaluate whether current benchmark suffers from the capacity miss. After this, we test these benchmarks by increasing the cache capacity to verify our observation. For those benchmarks whose reuse distance is evenly distributed in $[0, 128)$, they could suffer from the conflict miss. We simulate their memory access patterns in a set associative cache to verify this assumption. All these patterns are quantified into reuse distance distributions and we discover the

insightful observations.

Third, based on these experiments, we discover the memory access patterns of GPGPU computing and discuss some optimizations which could improve the cache performance. We pick some typical benchmarks and plot their memory access patterns with time fly(part of entire picture due to the huge memory access number). We draw reuse distance trends to support that bypass can improve the performance for the workloads which suffer from capacity miss. Then, memory address difference trends suggest that we still can optimize their performance by prefetching even the majority memory accesses missed, like AES and HST.

3.4 Simulation environment

In our implementation, we collect experiment results from GPGPU-Sim 3.2.2[21]. GPGPU-Sim is a cycle-level GPU performance simulator which simulates the kernel execution on GPU cores. It collects the data of cache behavior, so that we can analyze these statistics to evaluate the cache performance.

The baseline architecture of the simulator in our experiments is Fermi. The upper bound number of threads in each GPU core is 1536, meaning the maximum number of warps is 48. GPGPU-Sim analyzes the PTX[40] code and schedules the warp to execute the instructions in GPU pipeline. The warp scheduling policy is Greedy Then Oldest(GTO). We set 16KB cache as our baseline and increase the cache size to test whether larger cache size is helpful to improve the system performance. In GPUs, L2 cache is a unified last level cache, the cache requests, missing in L1 cache, will access L2 cache through the interconnect. Table 2 lists the detailed configurations of the simulator.

Table 2. GPGPU-Sim Configurations

# of clusters	14
# of cores each cluster	1
# of threads, # of warps	1536, 48
Warp scheduling policy	GTO
Capacity of L1 data cache	16KB /32KB/64KB/128KB/256KB
set associative (# of set, # of cache line)	16KB:(16,8) , (8,16), (4,32), (2,64) 32KB:(32,8) 64KB:(64,8) 128KB:(128,8) 256KB:(256,8)
Replacement policy	LRU
Size of shared memory	48KB
Size of L2 cache	786KB

3.5 Benchmarks

In our experiments, we choose 20 paradigmatic benchmarks from GPGPU-Sim package[21], Rodinia[41] and Parboil[42], which are three popular benchmark suites. They have been applied to all types of research and engineering fields, including image processing, safety communication, and mathematical solution. Every benchmark shows a distinct pattern of memory access. All of them deliver a comprehensive test on GPGPU computing model. Table 3 lists the basic information of these benchmarks, and some of them execute multiple kernels.

Table 3. Benchmarks List

Name	Abbr	Source
Aes Cryptography	AES	Sim[21]
Breadth First Search	BFS	Sim[21]
Coulombic Potential	CP	Sim[21]
3D Laplace Solver	LPS	Sim[21]
Mumer GPU	MUM	Sim[21]
Neural Network	NN	Sim[21]
N-Queens Solver	NQU	Sim[21]
Ray Tracing	RAY	Sim[21]
StoreGPU	STO	Sim[21]
Back Propagation	BCK	Rodinia[41]
B+tree	BT	Rodinia[41]
Hotspot	HST	Rodinia[41]
Lu Decomposition	LUD	Rodinia[41]
Needleman-Wunsch	NW	Rodinia[41]
Pathfinder	PF	Rodinia[41]
Kmeans	KM	Rodinia[41]
Fast Fourier Transform	FFT	Parboil[42]
Sparse Matrix-Dense Vector Multiplication	SPMV	Parboil[42]
Stencil	STE	Parboil[42]
Two-Point Angular Correlation Function	TPACF	Parboil[42]

CHAPTER 4

EXPERIMENT RESULTS AND ANALYSIS

In this chapter, we present the experiment results and the explanations based on the reuse distance distribution analysis. Every distribution is the sum of all cores and kernels. All the cache performance and hit rate are referred to the L1 data cache in the following analysis.

4.1 Hit rate vs. IPC

Unlike CPU, GPUs are designed to deliver a tremendous throughput by launching massive threads in parallel. However, many cache requests will access the L2 cache through the interconnect if they miss in L1 cache. Interconnect, as a bottleneck resource, is crucial for the performance of entire system[21, 22]. L1 cache can filter memory accesses and protect the interconnect from overwhelming. A good cache design can help GPU to balance the allocation of resources.

Jia et al.[22] mention that memory traffic is more commonly used to describe the cache effectiveness. There are three types of memory accesses could miss in L1 cache: data cache, constant cache and texture cache. We do not consider shared memory in our work. The severe contention also probably comes from missed constant and texture cache requests. We list the ratio of missed memory accesses of each benchmark for these three types of cache in Figure 4.

The missed L1 data cache requests dominate the missed memory accesses in 17 of 20 benchmarks. It is a high chance that missed memory accesses severely consume the bandwidth if the hit rate of L1 data cache is low, this exacerbates the intensive

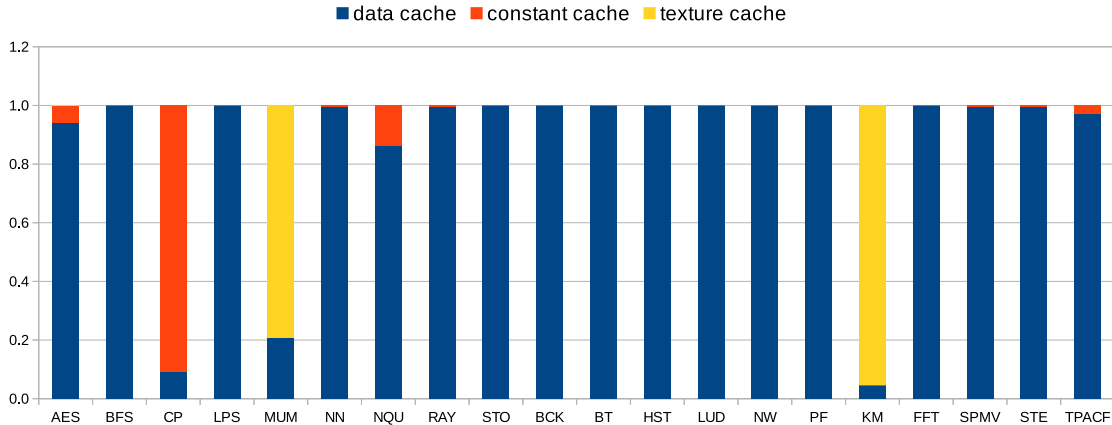


Fig. 4. The ratio of missed accesses for three types of cache

congestion of the interconnect. High hit rate helps these 17 benchmarks to efficiently utilize the bandwidth if there is no other bottleneck exists in entire system, such as BFS and SPMV. In Figure 4, the missed L1 data cache requests occupy 99.9% and 99.6% in all missed L1 cache requests. They consume the majority of bandwidth resources. In Figure 5, their IPCs benefit from a better cache performance which is already plotted in Figure 8. We will give the detailed analysis about how to improve the cache hit rate in next part of this section.

For the remaining 3 benchmarks(CP, MUM and KM), the missed constant or texture cache requests dominate the missed memory accesses. For example KM, the constant and texture accesses take the majority part of all L1 cache requests which is as high as 98.5%. In Figure 4, the missed accesses for L1 data cache only take 0.045% of all missed L1 cache requests. Although the hit rate increases to 82.78% when L1 data cache is enlarged to 256KB, it is limited imparts to alleviate the bandwidth congestion. A better cache performance for these three benchmarks helps to increase the hit rate, however, it is a marginal improvement for IPC in Figure 5.

To sum up, the hit rate of L1 data cache is still significant for the entire system.

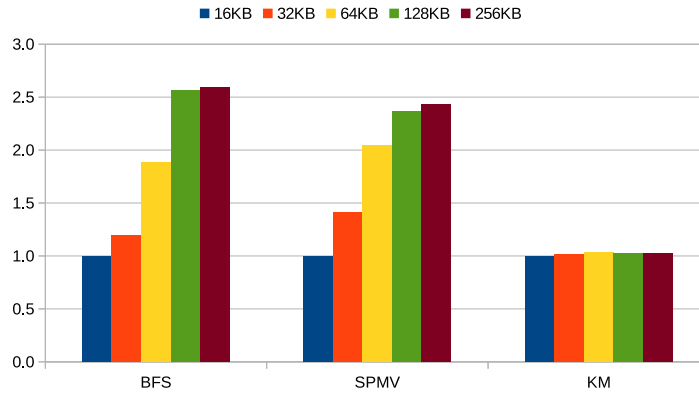


Fig. 5. The IPC trends of BFS, SPMV and KM

Most GPGPU computing benchmarks frequently access the L1 data cache. A low hit rate not only fails to exploit the data locality but also intensifies the congestion for interconnect. We will use hit rate as the metric to evaluate the cache performance in our analysis.

4.2 Reuse distance distributions

First, we plot the reuse distance distributions in a 16KB fully associative cache in Figure 6. It represents memory access behavior for us to conduct the following analysis.

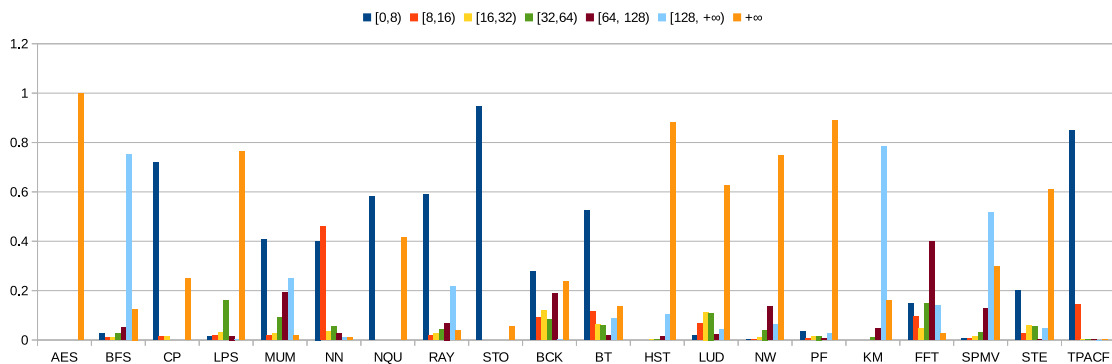


Fig. 6. Reuse distance distributions in a fully associative cache

All memory accesses are classified into seven intervals by the value of reuse distance, which are $[0,8)$, $[8,16)$, $[16,32)$, $[32,64)$, $[64,128)$, $[128,+\infty)$, $+\infty$. They are named from the interval 0 to 6 by the order of the growth of reuse distance. As a unique feature, the reuse distributions from a fully associative cache lead us to an optimization suggestion which favors this benchmark. After we follow the optimization suggestions, we test these solutions in a set associative cache which is more practical in current GPU cache design and classify the distributions by the methodology in chapter 3 and graph them in Figure 7. We evaluate the optimization results by cache hit rate in Figure 8 and 11. All memory accesses of AES and NQU miss in cache, the hit rate is 0.

4.3 Capacity of L1 data cache

We classify all the benchmarks into three categories according to the proportion of interval 5 in Figure 6. The first category includes BFS, KM and SPMV. The second contains MUM, RAY, BT, HST, LUD, NW, PF, FFT and STE. The remaining benchmarks, AES, CP, LPS, NQU, STO, BCK, NN and TPACF belong to the third one.

A large part of BFS, KM and SPMV distributes in interval 5($[128, +\infty)$) while a relatively small portion distributes in other intervals. This indicates that they suffer from the capacity miss. It is better to enlarge the cache capacity to increase hit rate. As few part lies in interval 0-4, a 8-way set associative cache is favorable for this optimization, only the number of cache set increases with the growth of cache capacity. We configure the cache with the first option of each size in Table 2 and analyze the reuse distance distributions which are divided by the principles in Table 1.

In Figure 7(a), 7(b) and 7(c), all of them obviously deliver a varied trend with

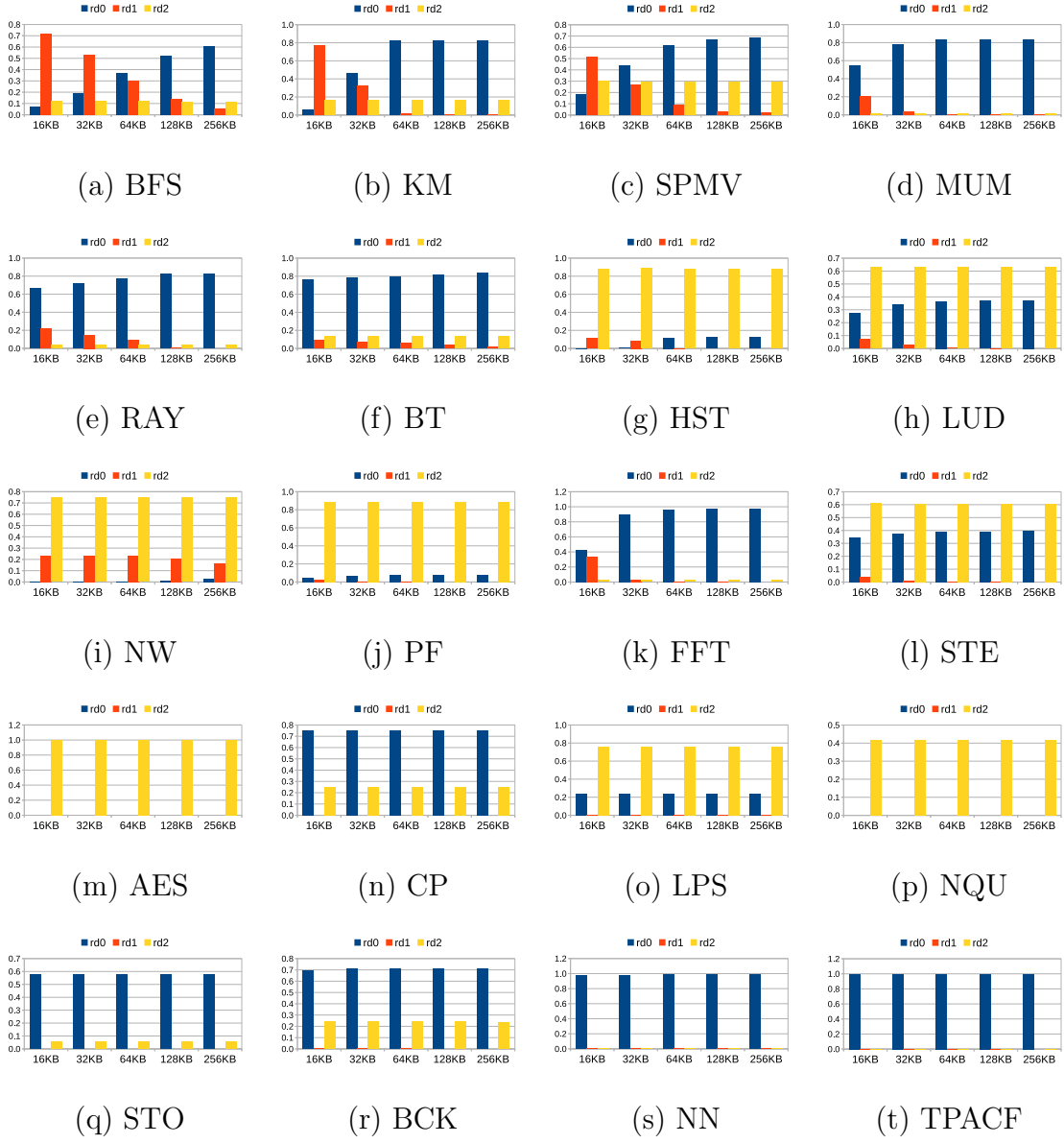


Fig. 7. The ratio of reuse distance distributions for all benchmarks

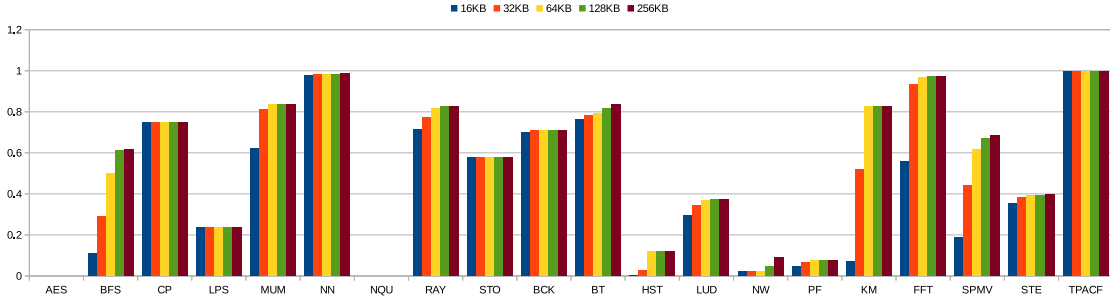


Fig. 8. Hit rate trends with the increase of cache capacity

the increase of cache capacity. We calculate the reuse distance and list the ratio of each distribution. Mostly changed parts are $rd0$ and $rd1$, however, the $rd2$ keeps unchanged. Some of memory accesses belonging to $rd1$ convert to $rd0$ when cache capacity increases.

In Figure 8, the change of hit rate follows the trend of $rd0$. BFS and SPMV benefit slightly from 128KB to 256KB, KM benefits dramatically from 16KB to 64KB and keeps stable at 128KB, 256KB. For these benchmarks, a large cache capacity helps to improve the cache performance if there is no other bottleneck in the system.

Contrast to the first category, MUM, RAY, BT, HST, LUD, NW, PF, FFT and STE distribute a limited interval 5 which is no more than 25% as shown in Figure 6. The conversions from $rd1$ to $rd0$ are relatively small which can be seen from Figure 7(d) to 7(l). Most of them only take the advantage of a cache as large as 64KB then the $rd1$ becomes pretty low, most memory accesses either lies in $rd0$ or $rd2$ which is resemble with the third category. Further growth of capacity does not help the cache performance. The hit rate trends of these nine benchmarks are plotted in Figure 8.

AES, CP, LPS, NQU, STO, BCK, NN and TPACF belong to the third category. They exhibit stable reuse distance distributions and their memory access patterns barely change with the increase of cache capacity. In Figure 6, these eight benchmarks

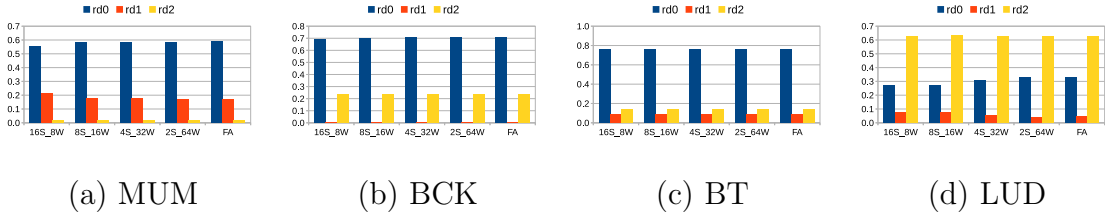


Fig. 9. Reuse distance distributions of MUM, BCK, BT and LUD with different associativities

hardly have memory accesses locate in $[128, +\infty)$. But they have either a large part of memory accesses whose reuse distance is $+\infty$ (AES, LPS, NQU) or most reuse distances are less than 8 (CP, STO, BCK, NN, TPACF). The former situation is a streaming access pattern which exhibits spatial locality only once so that they can not benefit from current cache design. The latter one displays an expected reuse pattern which behaves efficiently to exploit all data locality.

In this part, only the benchmarks in first category suffer from the capacity miss, they benefit from a large cache capacity. However, other benchmarks maintain a stable cache performance. Most of them issue cache requests in two manners: streaming access or short reuse distance access. No long reuse distance ($rd1$) can be converted from miss to hit with the increase of cache capacity. In general, simply increasing the cache capacity barely help to improve cache performance in GPGPU computing. A 64KB L1 data cache is sufficient for all benchmarks to resolve capacity miss.

4.4 Associativity of L1 data cache

The number of cache lines in a set is associativity. It correlates with the conflict miss. In Figure 6, MUM, BCK, BT, LUD, NW and FFT distribute relative evenly in interval 0 to 4. After we measure the experiment results of these six benchmarks in a set associative cache, all memory accesses will be divided into different cache sets by the index part of their addresses. The reuse distance of each access could possibly

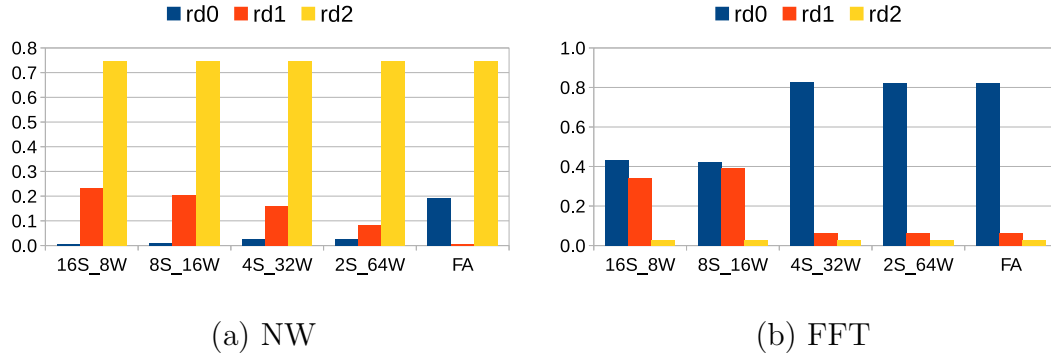


Fig. 10. Reuse distance distributions of NW and FFT with different associativities

change depending on if the memory access pattern is balanced. The $rd1$ in Figure 7 means the cache misses whose reuse distance is greater than the associativity, it is similar to interval 5($[128, +\infty)$) in Figure 6. The ratio of $rd1$ over interval 5 in MUM, BCK, BT and LUD maintain relatively unchanged, the maximum difference of these two values is approximately 0.036. However, it increases to $3.8\times$ in NW and $2.43\times$ in FFT. After the set associativity decreases, more accesses are out of the range to reuse. Eight cache lines can not tolerate the long reuse distances. NW and FFT suffer from the conflict miss while MUM, BCK, BT and LUD do not, original interval 0-4 in Figure 6 of these four benchmarks evenly remap to different sets when it is a set associative cache. Other 14 benchmarks take a big part at the left and right intervals. Like what we have analyzed in previous part of this section, these benchmarks exhibit stable behaviors.

Figure 9 and 10 plot the simulation results of the set associative cache. In this experiment, we simulate these six benchmarks at a 16KB cache with the different values of associativity. With the growth of set associativity, the number of sets decreases. MUM, BCK, BT and LUD show the same distribution trend in Figure 9, they do not really suffer from the conflict miss. However, in Figure 10, the trends of NW and FFT change with different associativities. The hit rates of these benchmarks

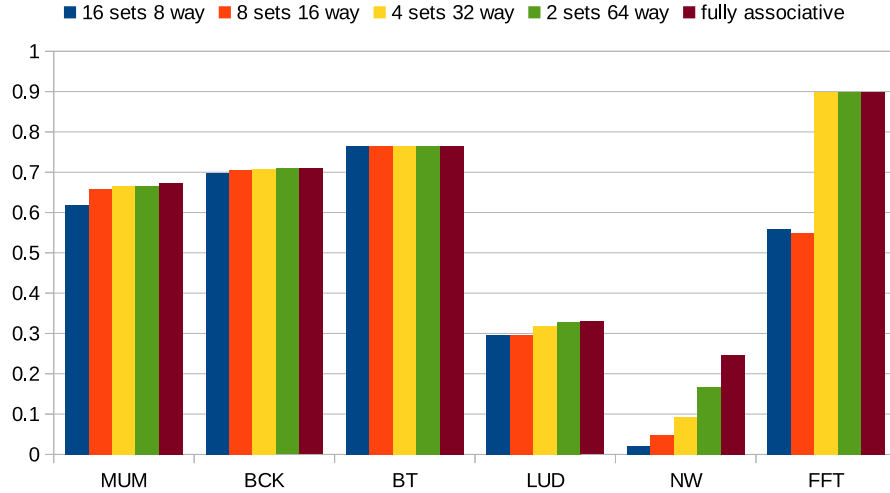


Fig. 11. Hit rate trends with the increase of associativity

in Figure 11 also follow this distributions. The ratio of *rd1* (Figure 10) in NW keep decreasing with the number of associativity increases, the cache performance also increases. For FFT(Figure 10), a 32-way set associative cache is sufficient to deliver a high hit rate, then all parts of the distributions become stable. The hit rate of NW keeps increasing from 0.02 to 0.24 and FFT jumps from 0.55 to 0.89 when it is a 32-way cache.

4.5 Summary

In this section, we analyze the experiment statistics based on the reuse distance. We quantify the memory access patterns and demonstrate that cache capacity and associativity are not the key points to improve the cache performance. The reuse distances of most memory accesses are either less than 8 or $+\infty$. Streaming accesses can not take the advantage of current cache design to exploit the data locality.

CHAPTER 5

CACHE BYPASSING OPTIMIZATION

From the above analysis for all simulations, we illustrate our research on L1 data cache of GPGPU computing. As a throughput-oriented device, GPU is expected to deliver a high performance. We have quantified the inefficiency of current cache design, this drives us to improve the architecture design to increase the hit rate and reduce the cache contention. According to the classifications in Table 1, we want to decrease the ratio of *rd1* and *rd2* in entire work set by bypassing unrelated cache requests.

5.1 Background introduction

Bypassing is a technique which has been extensively studied at CPU cache, especially for the last level cache. In modern CPU design, it normally includes three level caches, one memory access will visit last level cache only if it missed all of previous caches. It is a very low cache hit rate in last level cache since most cache localities have already been filtered out by previous level caches. It is a good idea to bypass last level cache requests to lower level memory hierarchy if there is few cache localities in order to reduce cache contention[12, 43].

Modern GPUs achieve high throughput by launching thousands of threads, the range of memory access is normally very wide. According to the warp scheduling policy, cache requests will burst in short cycles. Based on these reasons, cache is not as significant as in CPU cores. Less cache capacity can not widely exploit the locality of massive memory accesses in GPGPU computing, especially GPGPU com-

puting benchmarks deliver more broad memory address and different memory access patterns(most memory accesses are compulsory misses).

There are at most 32 and at least 2 memory accesses when a warp is issued. GPU will merge the adjacent the accesses if they call for sequential addresses, leading to most memory localities have already filtered out from cache[32]. The scheduling policies, such as GTO and LRR, not always favor for the locality of current memory access pattern, this fails to exploit the inter-warp or intra-warp locality[31]. Filtered locality and improper scheduling policy not only intensify cache contention but also increase the reuse distance of two same memory accesses. Based on the analysis of last chapter, memory accesses with long reuse distance dominate cache requests. Our purpose is to reduce the cache contention and avoid cache pollution. The solution is to bypass this kind of accesses from cache to lower level memory hierarchy.

The work flow of memory accesses in a bypass enabled cache is as in Figure 5.1. Hit and missed cache requests access cache directly layer by layer, meanwhile, bypassed requests go into memory, and detour cache to registers when they get data back.

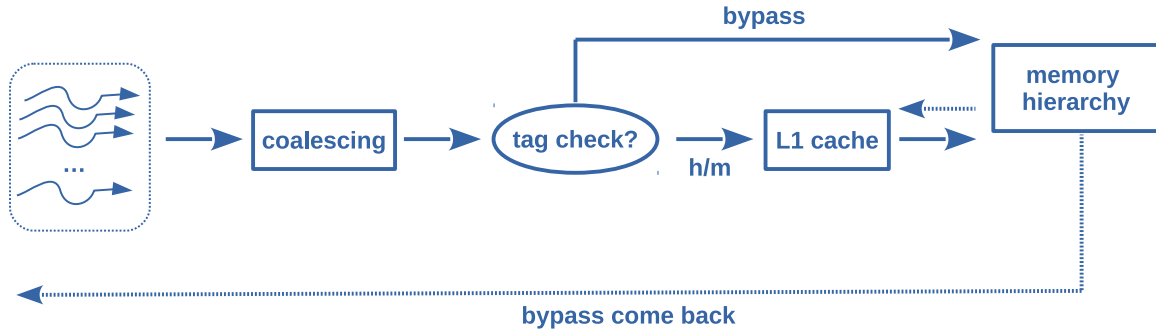


Fig. 12. The work flow of bypassed memory access in GPUs

5.2 Motivation

5.2.1 Memory access bypassing

Bypassing can alleviate the cache contention or protect the hot lines in cache[14]. The key point is to pick the right candidates. It is an optimization solution for long reuse distance accesses. This kind accesses can not get reused in cache before the line is replaced out, and they are thrashing-prone due to frequent switching in and out. For instance, BFS, KM, and SPMV in Figure 7, they could benefit from cache bypassing as they have large part of long reuse distance accesses($rd1$), a smart cache bypassing mechanism can filter unnecessary memory accesses, decrease the reuse distance, and improve cache hit ratio.

However, bypassing hardly helps AES, CP, LPS, NQU, STO, BCK, NN, and TPACF. They have one common feature that $rd1$ takes few part of all memory accesses. Simply bypassing the streaming accesses for these benchmarks can not increase the number of cache hits, like missed cache requests, bypassed requests also go through interconnect, leaving the bandwidth still be the bottleneck for the performance.

Figure 13 displays four typical examples of memory access trends. The x axis indicates at which time of cache request issued, y axis stands for the reuse distance for corresponding cache request. The value of 10000 in y axis means the reuse distance is ∞ . As shown in Figure 13, only BFS could benefit from bypass, since it contains a large part of long reuse distance memory access around 2000, which is far greater than the number of cache lines(128). Meanwhile the reuse distance of most memory access in TPACF is lower than 10, it already performs very well in current cache design. For AES, all memory accesses are streaming pattern, it does not matter bypass the cache request or not. The best optimization solution for AES is prefetching since it will periodically access sequential memory addresses. CP contains two extremes: either

very short reuse distance like TPACF or ∞ like AES, it also will not benefit from bypassing.

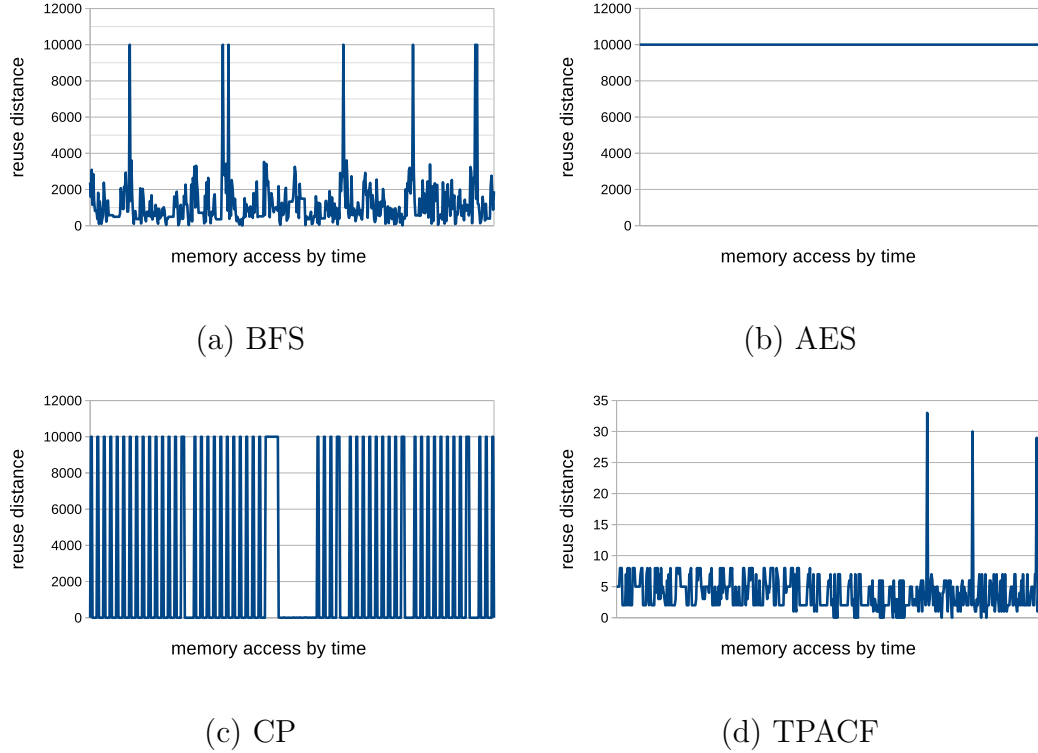


Fig. 13. Reuse distance trends for BFS, AES, CP, and TPACF

5.2.2 Cache line prefetching

Prefetching could improve the cache performance for AES, HST, NW, PF, and NQU. Figure 14 shows the memory address difference of AES and HST, x axis also means cache request at each time, y axis is the memory address difference between current cache request and previous one, it could be negative if they go back for lower address. AES, HST periodically access a sequential address, they display a strong spatial locality, and hardly with temporal locality. The subsequent requests will utilize the spatial locality and hit in cache if we can prematurely load the continuous addresses in cache.

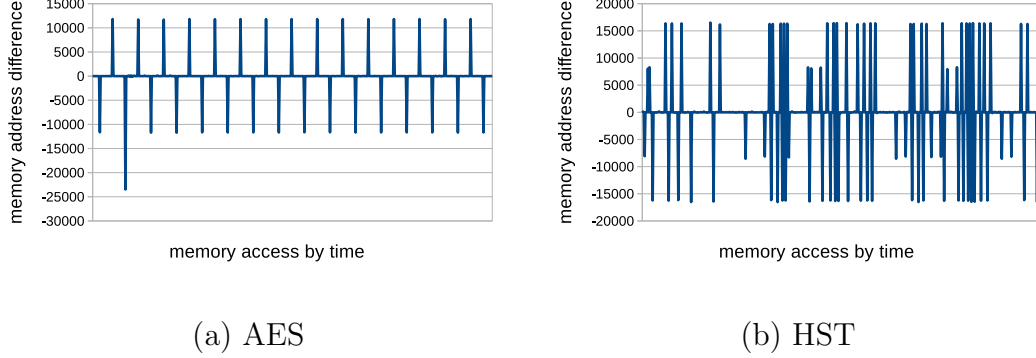


Fig. 14. Memory address difference

5.3 Benchmarks

We choose 8 benchmarks from Rodinia[41], Parboil[42], Polybench[44], and Mars[45], which are four popular GPGPU computing benchmarks have been widely used to test the performance of parallel computing architecture. All of the eight benchmarks are listed in Table 4.

Table 4. Benchmarks List for Bypassing

Name	Abbr	Source
Breadth First Search	BFS	Rodinia[41]
Kmeans	KM	Rodinia[41]
Sparse Matrix-Dense Vector Multiplication	SPMV	Parboil[42]
Matrix Transpose and Vector Multiplication	ATAX	Polybench[44]
BiCG Sub Kernel of BiCGStab Linear Solver	BICG	Polybench[44]
Matrix Vector Product and Transpose	MVT	Polybench[44]
Similarity Score	SS	Mars[45]
String Match	SM	Mars[45]

Most of them show an intensive memory access behavior, for example, SS sends

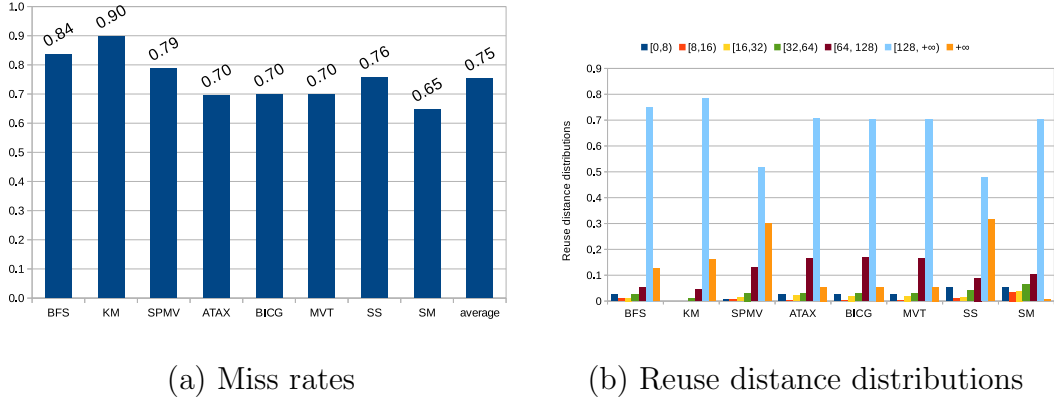


Fig. 15. Profile for selected 8 benchmarks

almost 3 millions cache requests, however, the cache miss rate is very high. Figure 15(a) shows the miss rate of these selected benchmarks in a 16KB L1 cache. Only SM is lower than 0.7(0.65). Other benchmarks are at least 0.7(ATAX, BICG, and MVT), even worse, KM missed 0.9 cache requests. The right most bar gives the average miss rate of these 8 benchmarks(0.75). These high cache miss rates obviously hurt the performance. As in 4.2, we profile the reuse distance distributions with the same cache configuration. In Figure 15(b), all 8 benchmarks have a majority part in $rd1$ which can be optimized by bypassing unrelated memory accesses and decrease the reuse distance of some memory accesses who still access cache. Based on these observations, we present a **Bypass Aware Cache** with small trade-off which could effectively bypass thrashing-prone memory accesses and improve the system performance.

5.4 Diagram of Bypassing Aware cache

In BA cache, the bypass decisions are made after tag comparison stage. In order to distinguish the memory accesses with long reuse distance, we add two bits to the tag part of each cache line: one is reuse bit, another is bypass bit as in Figure 16. The

reuse bit suggests whether current cache line has been reused before. The bypass bit suggests the cache request bypassed the cache or not. When a memory request hits in cache, the reuse bit of corresponding cache line is set to 1, that means this cache line already reused and it could possible be reused again in near future.

The reuse bit are also employed to make replacement policy fair. It naturally splits all cache lines into two group, one group is reused lines another group is never reused lines. We will choose the victim line from these two group alternatively.

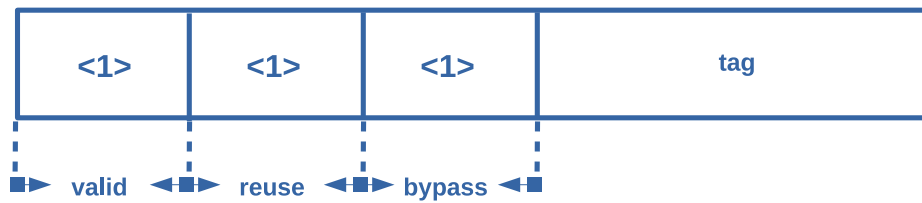


Fig. 16. Extra two bits for Bypass Aware cache

When a cache request bypassed cache, BA cache replaces the victim cache line and update the tag part of corresponding cache line, and set the bypass bit to 1, but the data part of this line does not corresponds to this cache request since BA cache only update the tag part, not send the data request. BA cache stores the information of tag part which helps to make a decision for subsequent cache requests. If a cache request bypass cache, the reuse bit is 0, that means it is first consideration to be the victim when cache miss.

5.4.1 Bypassing Aware cache work flow and bypassing decision

With the purpose of bypassing long reuse distance memory accesses, we add two bits in cache architecture of BA cache. This means the work flow of BA cache is different with traditional cache. Figure 17 illustrates the entire work flow of BA cache.

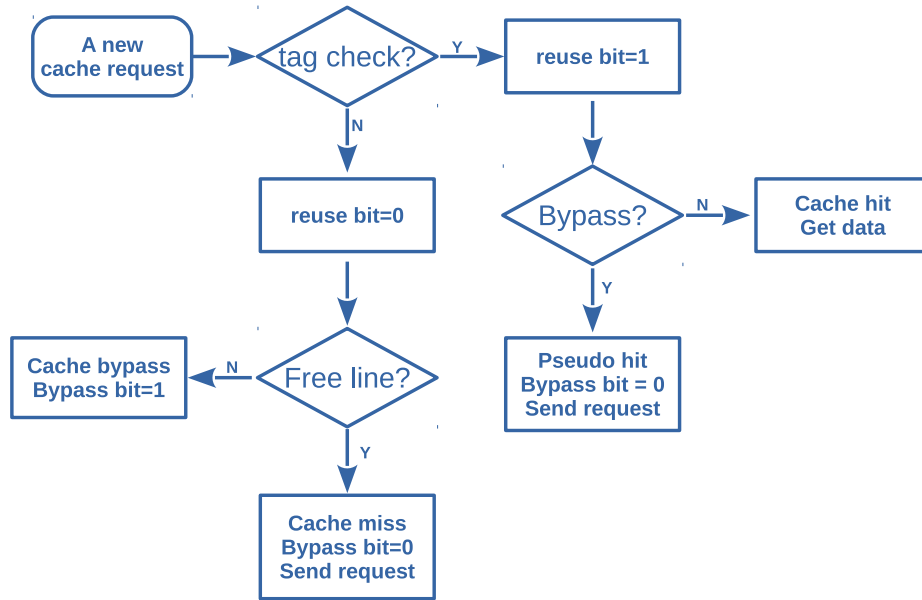


Fig. 17. BA cache work flow

In the conventional design, after a cache request is issued, we calculate which cache set current request locates by the value of index part, then compare the tag part to make a decision for cache hit or miss. If the request hits in cache, it will send to cache directly and get the data back. If this request misses, a victim cache line is selected by replacement policy to move space for new request and send the request to lower level memory to ask for data.

However, for BA cache, it works differently from traditional cache since it will make the decision to bypass or not. Bypassing decisions are made after tag check stage. BA cache will also check the tag part first. If the comparison failed, that means current cache request does not exist in cache, and reuse bit is set to 0. Next, BA cache will find the victim cache line to be replaced, it is a cache miss if there is free line to put this request in cache, or bypass this request if all lines are valid. Bypass bit is set to 0 when cache miss and set to 1 when cache bypassed. Meanwhile, the tag part of victim line will update to corresponding part of this request for the

processing of subsequent cache requests.

If tag check hit, that means the cache line has been referred before (either hit or bypassed), reuse bit is set to 1 and BA cache continue to check the bypass bit. It is a pseudo hit if bypass bit is 1, last request bypassed the cache and only the tag part updated, however data request has not send to lower level memory. BA cache will ask data from memory and disable the bypass bit. While if the bypass bit is 0, that means requested data already load into cache, BA cache can get the data directly.

5.4.2 Replacement Policy

When the tag check of a cache request failed, we need find a cache line to be replaced. Most popular cache replacement policies are Least Recently Used(LRU), replacing the cache line which has not been accessed the longest time and First In, First Out(FIFO), replacing the cache line which is the most earliest one get into cache.

Since all cache lines in a set can be divided into two groups after cache warm up, one group includes the cache line has revisited before, another one does not. In BA cache, we employ a relative balanced replacement policy to find the victim. First, we find the victim from the lines whose reuse bit is 0, when the number of this kind of lines is less than a threshold, then we change to find the victim from the lines whose reuse bit is 1 until this number is equal to the threshold.

5.4.3 Hardware Cost

The hardware cost for BA cache is negligible, we add two bits to each cache line. For a 16KB cache, there are 128 cache lines, the total overhead is $2 \times 128 = 256$ bits.

5.5 Experiment Results

We implement BA cache in GPGPU-Sim[21] with the same architecture configuration with of Table 2. We compare with traditional cache design which fully utilize L1 cache and simple bypassing which bypass all cache requests. The results is shown in Figure 18. Simple bypassing can improve the performance for 6 out of 8 benchmarks with two exceptions: SPMV and SM, whose performance hurts from simple bypassing. Simply bypassing all memory accesses lay more burden on the interconnect which is a race resources between all cores. It enlarges the negative impact of the bottleneck. However, BA cache can improve the performance, the improvement for SPMV is up to 11.06% and for SM, it is up to 77.2%. BA cache can pick the right bypass candidates as they can not take the advantage of cache.

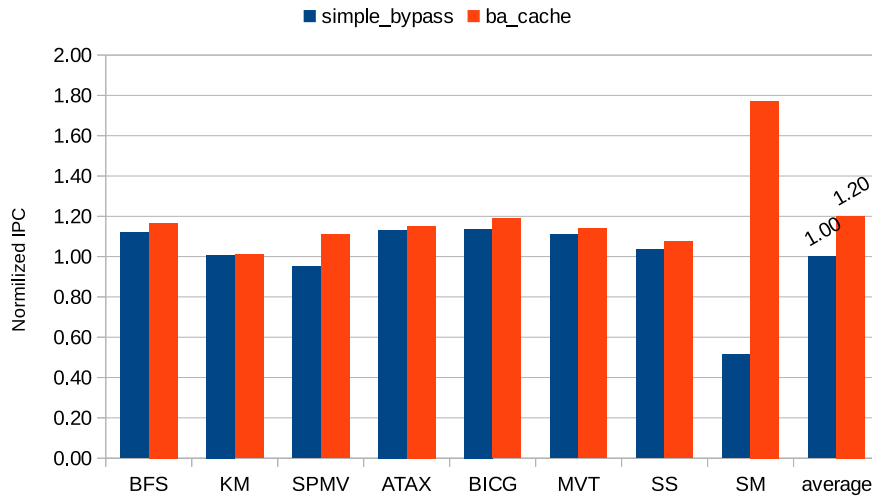


Fig. 18. Normalized IPC with traditional bypassed cache design and BA cache

For BFS, ATAX, BICG, MVT, and SS, all of them can benefit from both approaches, however, BA cache always outperform simple bypassing. The miss rate of these five benchmarks also abide this trend, BA cache not only improves the IPC, but also drops the miss rate.

BA cache improves the IPC for all benchmarks except KM, we have explained in previous part since the L1 data cache only take few part compared with constant and texture memory accesses. As shown in Figure 19, although the miss rate of KM drops around 18.0%, the performance almost keep the same.

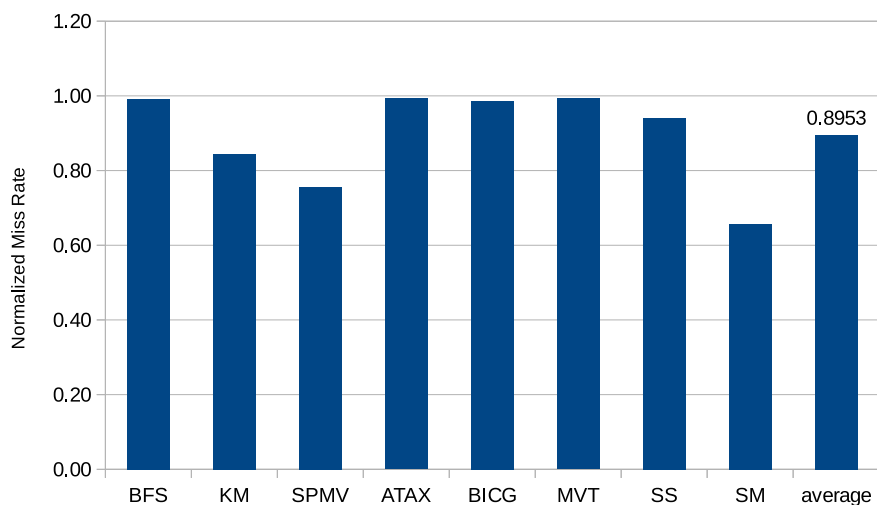


Fig. 19. Normalized Cache Miss Rate between BA cache and traditional cache

As a cost efficient cache design, BA cache is easy to implement and effectively increases the cache performance and decreases the cache miss rate. The average IPC improvement is more than 20%, the average reduction of cache miss rate is more than 10%.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this work, we analyze the memory access patterns of 20 benchmarks in GPGPU computing based on reuse distance theory and give the optimization suggestions depending on their features.

First, we calculate the reuse distance distributions in a fully associative cache. Based on these distributions, we illustrate the impacts of cache capacity, 17 of 20 benchmarks do not suffer from the capacity miss. BFS, KM and SPMV benefit from the growth of cache capacity, other benchmarks keep almost stable because few memory accesses are converted from miss to hit. Furthermore, associativity also does no affect the cache performance for 18 benchmarks, only NW and FFT get profits from a large associativity. Most cache requests in GPGPU computing are either streaming accesses or short reuse distance accesses.

After the analysis, we provide some possible techniques to optimize the GPU cache. In this thesis, we implement a Bypass Aware Cache, which could improve the performance around 20%. For the benchmarks like AES, HPS, prefetching could help to improve their IPC, we leave this as our future work.

Appendix A

ABBREVIATIONS

VCU	Virginia Commonwealth University
RVA	Richmond Virginia
CPU	Central Processing Unit
GPU	Graphic Processing Unit
BA cache	Bypass Aware cache
GPGPU	General-Purpose computing on Graphics Processing Unit
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
SM	Streaming Multiprocessor
SIMD	Simple Instruction, Multiple DAta
SP	Streaming Processor
SFU	Special Function Unit
MSHR	Miss Status Holding Register
CTA	Cooperative Thread Array
PTX	Parallel Thread Execution

REFERENCES

- [1] NVIDIA. *CUDA C Programming Guide*. 2013.
- [2] NVIDIA. *NVIDIA GeForce GTX 980*. 2014.
- [3] NVIDIA. *NVIDIAs Next Generation CUDA Compute Architecture:Fermi*. 2009.
- [4] AMD. *Introduction to OpenCL Programming*. 2010.
- [5] I PRESENT. “Cramming more components onto integrated circuits”. In: *Readings in computer architecture* 56 (2000).
- [6] NVIDIA. *NVIDIAs Next Generation CUDA Compute Architecture:Kepler GK110*. 2012.
- [7] Moinuddin K Qureshi, David Thompson, and Yale N Patt. “The V-Way cache: demand-based associativity via global replacement”. In: *Computer Architecture, 2005. ISCA ’05. Proceedings. 32nd International Symposium on*. IEEE. 2005, pp. 544–555.
- [8] Moinuddin K Qureshi et al. “A case for MLP-aware cache replacement”. In: *ACM SIGARCH Computer Architecture News* 34.2 (2006), pp. 167–178.
- [9] Moinuddin K Qureshi and Yale N Patt. “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2006, pp. 423–432.
- [10] Moinuddin K Qureshi et al. “Adaptive insertion policies for high performance caching”. In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 381–391.

- [11] Aamer Jaleel et al. “High performance cache replacement using re-reference interval prediction (RRIP)”. In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 60–71.
- [12] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. “Bypass and insertion algorithms for exclusive last-level caches”. In: *ACM SIGARCH Computer Architecture News* 39.3 (2011), pp. 81–92.
- [13] Carole-Jean Wu et al. “SHiP: Signature-based hit predictor for high performance caching”. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 430–441.
- [14] Xuhao Chen et al. “Adaptive cache management for energy-efficient gpu computing”. In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2014, pp. 343–355.
- [15] Kristof Beyls and Erik DHollander. “Reuse distance as a metric for cache behavior”. In: *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*. Vol. 14. 2001, pp. 350–360.
- [16] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. “Cache replacement based on reuse-distance prediction”. In: *Computer Design, 2007. ICCD 2007. 25th International Conference on*. IEEE. 2007, pp. 245–250.
- [17] Nam Duong et al. “Improving cache management policies using dynamic reuse distances”. In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2012, pp. 389–400.
- [18] Tao Tang, Xuejun Yang, and Yisong Lin. “Cache miss analysis for gpu programs based on stack distance profile”. In: *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*. IEEE. 2011, pp. 623–634.

- [19] Cedric Nugteren et al. “A detailed GPU cache model based on reuse distance theory”. In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 37–48.
- [20] Peter J Denning. “Thrashing: Its causes and prevention”. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM. 1968, pp. 915–922.
- [21] Ali Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 163–174.
- [22] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. “Characterizing and improving the use of demand-fetched caches in GPUs”. In: *Proceedings of the 26th ACM international conference on Supercomputing*. ACM. 2012, pp. 15–24.
- [23] Adwait Jog et al. “OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance”. In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 395–406.
- [24] Onur Kayıran et al. “Neither more nor less: optimizing thread-level parallelism for GPGPUs”. In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press. 2013, pp. 157–166.
- [25] Minseok Lee et al. “Improving GPGPU resource utilization through alternative thread block scheduling”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 260–271.
- [26] Veynu Narasiman et al. “Improving GPU performance via large warps and two-level warp scheduling”. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 308–317.

- [27] Wilson WL Fung et al. “Dynamic warp formation and scheduling for efficient GPU control flow”. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2007, pp. 407–420.
- [28] Wilson WL Fung and Tor M Aamodt. “Thread block compaction for efficient SIMT control flow”. In: *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE. 2011, pp. 25–36.
- [29] Adwait Jog et al. “Orchestrated scheduling and prefetching for GPGPUs”. In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 332–343.
- [30] Yulong Yu et al. “A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs”. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM. 2015, pp. 15–24.
- [31] Minseok Lee et al. “iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 370–381.
- [32] Joel Hestness, Stephen W Keckler, David Wood, et al. “A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior”. In: *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 150–160.
- [33] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. “A quantitative study of irregular programs on GPUs”. In: *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 141–151.

- [34] Mahmoud Khairy, Mohamed Zahran, and Amr G Wassal. “Efficient utilization of GPGPU cache hierarchy”. In: *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*. ACM. 2015, pp. 36–47.
- [35] Xiaolong Xie et al. “Coordinated static and dynamic cache bypassing for GPUs”. In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE. 2015, pp. 76–88.
- [36] Rachata Ausavarungnirun et al. “Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance”. In: ().
- [37] Mark Stephenson et al. “Flexible software profiling of GPU architectures”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM. 2015, pp. 185–197.
- [38] Wenhao Jia, Kelly Shaw, Margaret Martonosi, et al. “MRPB: Memory request prioritization for massively parallel processors”. In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 272–283.
- [39] Timothy G Rogers, Mike O’Connor, and Tor M Aamodt. “Cache-conscious wavefront scheduling”. In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2012, pp. 72–83.
- [40] NVIDIA. *PARALLEL THREAD EXECUTION ISA*. 2016.
- [41] Shuai Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE. 2009, pp. 44–54.

- [42] John A Stratton et al. “Parboil: A revised benchmark suite for scientific and commercial throughput computing”. In: *Center for Reliable and High-Performance Computing* (2012).
- [43] Saurabh Gupta, Hongliang Gao, and Huiyang Zhou. “Adaptive cache bypassing for inclusive last level caches”. In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 1243–1253.
- [44] Scott Grauer-Gray et al. “Auto-tuning a high-level language targeted to GPU codes”. In: *Innovative Parallel Computing (InPar), 2012*. IEEE. 2012, pp. 1–10.
- [45] Bingsheng He et al. “Mars: a MapReduce framework on graphics processors”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 260–269.