2017

IMPROVING THE PERFORMANCE AND TIME-PREDICTABILITY OF GPUs

Yijie Huangfu

IMPROVING THE PERFORMANCE AND TIME-PREDICTABILITY OF GPUS

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

YIJIE HUANGFU, PH.D. VIRGINIA COMMONWEALTH UNIVERSITY, MAY 2017

Director: Wei Zhang, Ph.D.

Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

May, 2017

Acknowledgements

I thank my advisor, Prof. Wei Zhang. I am grateful for the opportunity of working with Prof. Zhang in this Ph.D. program, where I received excellent training and insightful guidance and feedback from him. With his diligence and preciseness, Prof. Zhang sets up a model for young academics to follow to succeed. I also appreciate Prof. Zhang's supportive attitude in encouraging me to explore research ideas and career opportunities.

Dr. Carl Elks, Dr. Preetam Ghosh, Dr. Weijun Xiao and Dr. Qiong Zhang, who also nicely serve on my committee, are appreciated for their highly suggestive feedback and comments to my research plan proposal. I also thank their efforts and patience in accommodating my proposal and dissertation defense dates to their tight schedules.

I thank my friends Elaine and Bob Metcalf, Susie and Bud Whitehouse, Kate and Lex Strickland, Jan and Jim Fiorelli, Geoffrey and Eunice Chan, Kun Tang, Tao Lyu and Qianbin Xia for being a huge and wonderful part in my life at Richmond. It is such a blessing to know them and to have them as friends.

I thank my parents for giving me life and the abilities and educations, with which I can survive and thrive. I thank my parents in law for their unfailing supports, without which I could not finish my Ph.D. program. I thank my daughter Claire for bring joy, hope and noisy peace to the family. Specially, I thank my wife Xiaochen, who is always supportive and encouraging and is forever the source of my energy for pursuing excellence.

TABLE OF CONTENTS

Abstract

IMPROVING THE PERFORMANCE AND TIME-PREDICTABILITY OF GPUs

By Yijie Huangfu, Ph.D.

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2017

Major Director: Wei Zhang, Professor, Electrical and Computer Engineering

Graphic Processing Units (GPUs) are originally mainly designed to accelerate graphic applications. Now the capability of GPUs to accelerate applications that can be parallelized into a massive number of threads makes GPUs the ideal accelerator for boosting the performance of such kind of general-purpose applications. Meanwhile it is also very promising to apply GPUs to embedded and real-time applications as well, where high throughput and intensive computation are also needed.

However, due to the different architecture and programming model of GPUs, how to fully utilize the advanced architectural features of GPUs to boost the performance and how to analyze the worst-case execution time (WCET) of GPU applications are the problems that need to be addressed before exploiting GPUs further in embedded and real-time applications. We propose to apply both architectural modification and static analysis methods to address these problems. First, we propose to study the GPU cache behavior and use bypassing to reduce unnecessary memory traffic and to improve the performance. The results show that the proposed bypassing method

can reduce the global memory traffic by about 22% and improve the performance by about 13% on average. Second, we propose a cache access reordering framework based on both architectural extension and static analysis to improve the predictability of GPU L1 data caches. The evaluation results show that the proposed method can provide good predictability in GPU L1 data caches, while allowing the dynamic warp scheduling for good performance. Third, based on the analysis of the architecture and dynamic behavior of GPUs, we propose a WCET timing model based on a predictable warp scheduling policy to enable the WCET estimation on GPUs. The experimental results show that the proposed WCET analyzer can effectively provide WCET estimations for both soft and hard real-time application purposes. Last, we propose to analyze the shared Last Level Cache (LLC) in integrated CPU-GPU architectures and to integrate the analysis of the shared LLC into the WCET analysis of the GPU kernels in such systems. The results show that the proposed shared data LLC analysis method can improve the accuracy of the shared LLC miss rate estimations, which can further improve the WCET estimations of the GPU kernels.

CHAPTER 1

INTRODUCTION

1.1 Background

In the past decade or so, Graphics Processing Units (GPUs), originally designed to accelerate graphical computation, have rapidly become a popular platform for high-performance parallel computing. Modern GPUs can support massive parallel computing with thousands of cores and extremely high-bandwidth external memory systems. The single-instruction multiple-thread (SIMT) programming model used by GPUs well matches the underlying computing patterns of many high-performance embedded applications, including imaging, audio, video, military, and medical applications [1]. At the same time, GPUs are increasingly used in System-on-Chips (SoCs) for mobile devices, for example ARM's Mali graphics processor [2], the NVIDIA Tegra[3] and the DRIVE PX platform [4].

GPUs can also provide considerable benefits to a variety of real-time applications that demand high throughput and energy efficiency. In particular, GPUs are promising for many computation-intensive hard real-time and safety-critical applications such as medical data processing [5], autonomous auto navigation [6], vision-based aircraft controls [7] and human pose recognition [8]. All these applications need to meet strict deadlines and require high system throughput, making GPUs the ideal potential computing engines for them.

There are efforts made to explore the performance and energy benefits of the heterogeneous CPU-GPU architectures. For instance, the management method in [9] employs a unified Dynamic Voltage Frequency Scaling (DVFS) approach to further

reduce the power consumption for 3D mobile games. Studies have also been done on real-time image processing in different types of applications based on the CPU-GPU architecture[10][11][12]. Besides the real-time image processing field, moreover, the CPU-GPU architecture is more and more used in other real-time applications, e.g. the NVIDIA PX 2 self-driving car computing platform[4] using the Tegra[3] chips. And, with the development of the general purpose GPU programming model and the CPU-GPU architectures, it is expected that such architectures will be widely used in all kinds of different real-time applications, e.g. computer vision, automation control and robotics.

However, many GPU architecture features designed for improving the average-case performance are harmful to the time-predictability feature of the system. Therefore, before exploiting the computing power of GPUs in these applications, the impacts of these advanced GPU architecture features in time-predictability and performance need to be analyzed and studied accordingly. One example is the usage of cache memories. In CPUs, cache memories help to reduce the speed gap between the processor cores and the main memory, by exploiting the spacial and temporal localities. GPU applications, nevertheless, are different in spacial and temporal localities, which leads to the first problem of how to better utilize the cache memory in GPUs. According to the time-predictability of GPUs, the advanced architecture features, such as dynamic scheduling and out-of-order execution, make it very hard, if not impossible, to estimate the WCET of GPU applications, since at run-time there are usually thousands of warps scheduled and executed dynamically. Furthermore, the trend of building and utilizing the integrated CPU-GPU architectures raises the problem of how to model the behavior of the shared resources in such architectures, e.g., the shared Last Level Cache (LLC), so that the behavior of the whole chip in the worst case can be better modeled.

1.2 GPU L1 Data Cache Bypassing

The first topic is about using cache bypassing to study the impact of the GPU L1 data cache on the performance and finding a way to use the GPU L1 data cache more effectively. We comparatively evaluate the GPU performance without and with the cache memory. We find that unlike CPU caches, GPU applications tend to exhibit low temporal and/or spatial locality in the L1 data cache. On average, the GPU with the L1 data cache actually leads to worse performance than the one without the L1 data cache. However, this does not necessarily imply that caches should not be used for real-time GPU computing. By examining the GPU application behavior and architectural features, we propose to use GPU L1 data cache bypassing methods to filter out the GPU cache accesses that are detrimental to performance, so that the cache can be used in a more effective way.[13]

1.3 GPU L1 Data Cache Access Reordering

Secondly, the dynamic warp scheduling operations in GPUs can benefit the average-case performance of general-purpose GPU (GPGPU) applications. But such a kind of dynamic behaviors is hard to be analyzed statically. Therefore, we propose a warp-based load/store reordering framework that is based on collaborative static analysis and architectural extensions in GPUs to improve the predictability of the GPU L1 data caches. The proposed framework supports dynamic warp scheduling while reordering the load/store instructions to enable safe and accurate timing analysis for L1 GPU data caches. As a result, the predictability is improved without putting constraints on the dynamic warp scheduling behaviors, which helps to keep good average-case performance.[14]

1.4 WCET Timing Model for GPU Kernels

The third effort is to build a timing model and static analyzer for the purpose of GPU WCET analysis and estimation. We proposed to employ a predictable pure round-robin scheduling policy, based on which a timing model is built for GPGPU kernels. With this timing model, a static analyzer is built to analyze the assembly codes of the GPGPU kernels and to give their WCET estimations. Building such a kind of low-level timing model of a processor architecture requires detailed documentation of the processor, which is usually unavailable for GPUs. Furthermore, the proposed methods involve changes in the architecture. Therefore, the detailed and configurable GPU simulator *GPGPU-Sim* [15] is used to implement and evaluate the proposed model and analyzer.

1.5 WCET Analysis of Shared Data LLC in integrated CPU-GPU Architecture

The last work is to improve the time-predictability of the integrated CPU-GPU architectures. Specifically, the focus in this work is the shared Last Level Cache (LLC). The method of *Access Interval* regulations is used to improve the time-predictability of the shared data LLC, according to the cache miss rate estimations. The improved miss rate estimations are then integrated into the WCET timing model for better WCET estimations of GPU kernels. The *gem5-gpu*[16] simulator is used to implement the integrated architecture with shared LLC and to evaluate the impact of the shared LLC in such systems.

1.6 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 briefly introduces the background information about the GPU architecture and GPGPU programming model. In Chapter 3, the profiling-based GPU L1 data cache bypassing is present to illustrate how the GPU L1 data caches can be used in a more effective way. Chapter 4 talks about a reordering framework, which is based on both architectural extensions and static analysis, and how this framework can improve the predictability of the GPU L1 data cache. A timing model for WCET analysis of GPU kernels that is based on a predictable warp scheduling policy is introduced in Chapter 5, after which a static WCET analysis technique for the shared data LLC in the integrated CPU-GPU architecture is discussed in Chapter 6. In Chapter 7, the conclusions are made.

CHAPTER 2

GPU ARCHITECTURE AND GPGPU PROGRAMMING MODEL

2.1 GPU Architecture

Fig. 1 shows the basic architecture of a NVIDIA GPU[1], which has a certain number of Streaming Multiprocessors (SMs), e.g., 16 SMs in Fermi architecture[17]. All the SMs share the L2 cache, through which they access the DRAM global memory. Other parts, like the interface to host CPUs, are not included in Fig. 1.

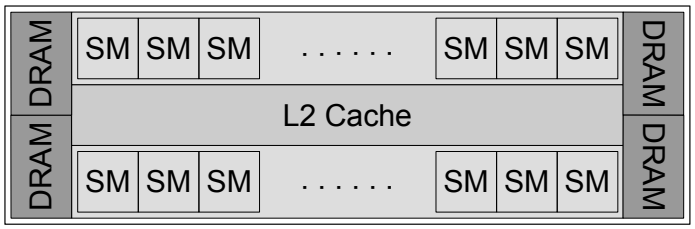


Fig. 1. GPU Architecture[17]

Fig. 2 shows the architecture of an SM, which contains a group of Shader Processors (SPs, also called CUDA processor or CUDA core). Each SP has the pipelined integer arithmetic logic unit and floating point unit, which execute the normal arithmetic instructions, while the Special Function Units (SFUs) execute the transcendental instruction, such as sin, square root, etc. Besides the computing functional units, there are several L1 caches for instruction, data, texture data and constants. The register file contains a huge number of registers shared by all the SPs and SFUs, while the warp scheduler and dispatching unit choose among the active warps and collect

[1]The NVIDIA CUDA GPU terminologies are used in this dissertation.

the operands needed and send the warp to execution.

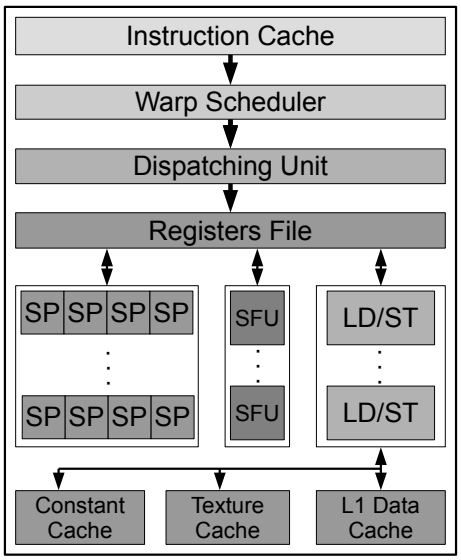


Fig. 2. SM Architecture[17]

2.2 GPGPU Programming Model

With the support of a massive number of cores, GPUs use the SIMT execution model to allow a big number of threads to execute in parallel. A GPGPU program, which is also called a GPU kernel, can be written in either CUDA C[18] or OpenCL[19]. A GPU kernel is configured and launched by a host CPU. Through the configuration of the kernel, the host CPU tells the GPU how many threads there are in the execution of the kernel and what the hierarchy of the threads is like. The hierarchy of a kernel has two levels; the dimensions in kernel grid (how many kernel blocks there are in a kernel grid) and in kernel block (how many threads there are in a kernel block). For example, the kernel in Fig. 3 has *64 (*$2 \times 4 \times 8$*)* kernel blocks in the kernel grid and *512 (*$32 \times 16 \times 1$*)* threads in one kernel block.

The kernel code describes the function and behavior of a single thread, based on the position of this thread in the hierarchy of the kernel, e.g., thread and block IDs.

```
dim3 grdDim( 2, 4, 8);
dim3 blkDim(32,16, 1);
Kernel<<<grdDim, blkDim>>>(...);
```

Fig. 3. GPU Kernel Configuration Example

The most common way is to use the thread and block IDs to calculate the indices, which each thread uses to access a certain array, so that the threads work on different parts of the data in parallel. In the execution of a GPU kernel, a kernel block is assigned to an SM and stays there until finishing its execution. 32 threads in a kernel block are grouped together as the basic scheduling and execution unit, which is called a *warp*. The threads in the same warp execute the same instruction together in the SIMT model. Therefore, a GPU kernel instruction is also called a warp instruction.

CHAPTER 3

PROFILING-BASED GPU L1 DATA CACHE BYPASSING

3.1 Introduction

To exploit the localities in GPGPU applications and boost the average-case performance, both the L1 data cache and the unified L2 cache are included in modern GPUs. Although the cache memory can effectively hide the access latency for data with good temporal and/or spatial locality for both CPUs and GPUs, GPGPU applications may exhibit divergent memory access patterns from traditional CPU applications. Moreover, the recent study shows that GPU caches have counter-intuitive performance trade-offs [20]. Therefore, it is important to explore the techniques to use the on-chip cache memories effectively to boost GPU performance and/or energy efficiency. In particular, for embedded and mobile GPU applications, it is also crucial to develop cost-effective optimization methods for improving performance and/or energy efficiency.

To address this problem, we comparatively evaluate the GPU performance without and with the cache memory. As the first step toward studying time predictability of GPU caches, we focus on the L1 data cache. we find that unlike CPU caches, GPU applications tend to exhibit low temporal and spatial locality in the L1 data cache. On average, the GPU with the L1 data cache actually leads to worse performance than the one without the L1 data cache. However, this does not necessarily imply that caches should not be used for real-time GPU computing.

By examining the GPU application behavior and architectural features, we propose a profiling-based cache bypassing method to filter out the GPU cache accesses

that are detrimental to performance. The evaluation results show that the cache bypassing method improves the performance adequately as compared to the GPU without using the cache, because the rest of GPU memory accesses with good temporal and spatial locality can still efficiently exploit the L1 data cache. Therefore, employing the L1 data cache can still benefit real-time GPU applications in terms of the average-case performance; however, time-predictable architecture or static timing analysis techniques need to be developed to use the GPU caches deterministically for high-performance real-time computing.

3.2 Related Work

Cache bypassing has been extensively studied for CPUs in the past. Some architectures have introduced ISA support for cache bypassing, for example HP PA-RISC and Itanium. Both hardware-based [21][22][23][24][25] and compiler-assisted [26][27] cache bypassing techniques have been proposed to reduce cache pollution and improve performance. However, most CPU cache bypassing approaches use hit rates as performance metrics to guide cache bypassing, which may not be applicable to GPUs due to the distinct architectural characteristics and the non-correlation of GPU performance with data cache hit rates [28].

Mekkat et al. [29] proposed Heterogeneous LLC (Last-Level Cache) Management (HeLM), which can throttle GPU LLC accesses and yield LLC space to cache sensitive CPU applications. The HeLM takes advantage of the GPUs tolerance for long memory access latency to provide an increased share of the LLC to the CPU application for better performance. There are several major differences between HeLM and my work. HeLM targets the shared LLCs in integrated CPU-GPU architectures, while my work focuses on bypassing the L1 data caches in GPUs. Moreover, HeLM is a hardware-based approach that needs additional hardware extension to monitor the thread-level

parallelism (TLP) available in the GPU application. In contrast, my cache bypassing method is a software-based approach that leverages profiling information statically, which is simple and low cost and is particularly useful for embedded and mobile GPUs. Moreover, my method is complementary to the hardware-based HeLM, which can be used in conjunction with HeLM to further improve the GPU performance or energy efficiency in the integrated CPU-GPU architecture.

GPU Cache Bypassing. Jia et al. [28] characterized application performance on GPUs with caches and proposed a compile-time algorithm to determine whether each load should use the cache. Their study first revealed that unlike CPU caches, the L1 cache hit rates for GPUs did not correlate with performance. Recently, Xie et al. [30] studied a compiler-based algorithm to judiciously select global load instructions for cache access or bypass. Both Jia and Xie's approaches can achieve performance improvement through cache bypassing. However, both approaches make cache bypassing decisions based on each global load instruction, which can access a variety of data addresses with diverse temporal and spatial locality. In contrast, our method is based on data addresses, not load instructions. This gives us finer-grained control on which data to be cached or bypassed to further enhance performance and energy efficiency.

3.3 Profiling-Based GPU L1 data Cache Bypassing Method

3.3.1 Global Memory Access Utilization

The 32 threads in a warp access the global memory in a coalesced pattern. Assuming that each thread needs to fetch 4 bytes, if the data needed by each thread are well coalesced, this load operation can be serviced by one 128-byte transaction, as shown in Fig. 4 (a). In this case, all the data in the memory transaction are useful,

thus the utilization rate (or efficiency) of this load, which represents the percentage of bytes transferred from global memory that are actually used by the GPU, is 100% (128/128). However, when the memory access pattern changes a little bit, as shown in Fig. 4 (b) and (c), the address range becomes 96 to 223, which spans across the boundary of 128 bytes. In this case, two 128-byte transactions are needed to transfer the data needed by the threads. Thus the utilization rates of these two transactions are 25% and 75% respectively, resulting in a 50% (128/256) overall utilization rate. This indicates half of the memory traffic, generated by this two load operations, are useless and unnecessary if they are not reused, which may degrade both performance and energy efficiency for GPGPU computing.

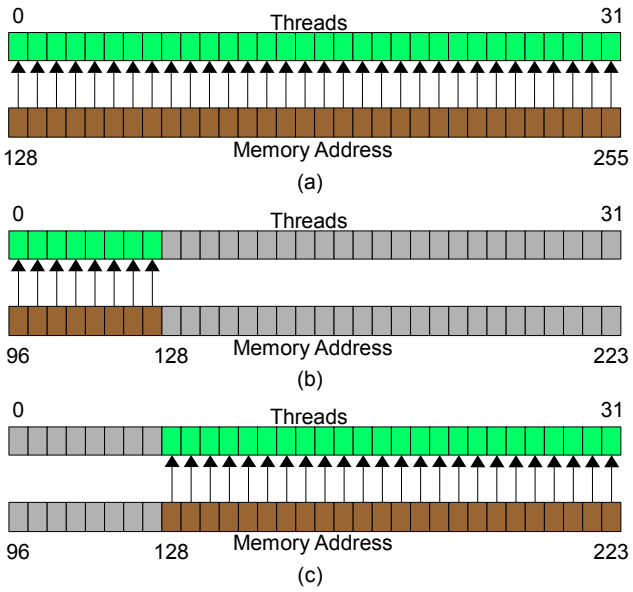


Fig. 4. Examples of different memory access patterns with different utilization rates.[13]

The example of low load utilization rates in Fig. 4 may be caused by improper mapping between threads and memory addresses, which, sometimes but not always, can be avoided through the effort of programmers. However, the divergences in the

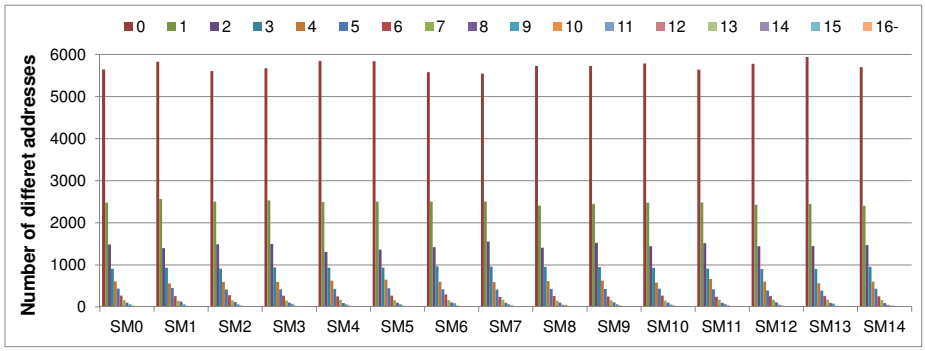
CUDA kernel, which are caused by the algorithms and are generally hard to eliminate, can also lead to such load operations with low utilization rates.

3.3.2 Global Memory Reuse Time

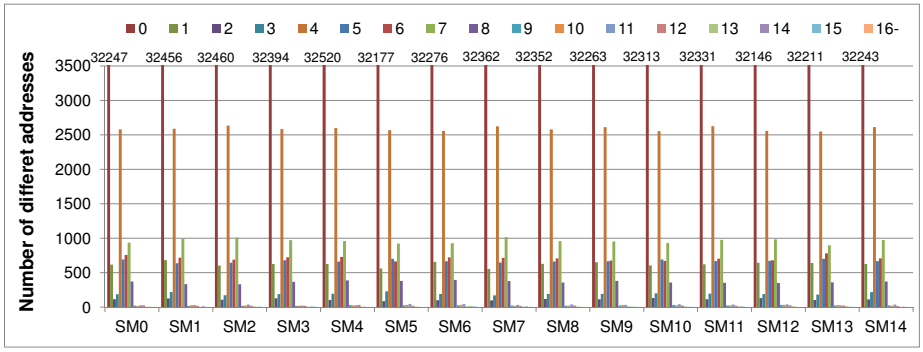
The GPGPU applications usually operate on a massive amount of data. However, the cache line usage among the data with different addresses may differ significantly. This is not only because GPGPU applications can exhibit irregular data access patterns, but also because the effective L1 data cache space per SP is too small. Thus even if some data are reused within a warp, they may have been replaced from the cache by other data from the same warp or from other warps from the same thread block before they can be reused, resulting in cache misses and hence increasing global memory accesses.

Fig. 5 shows the data reuse distribution in the L1 data cache across different SMs for the benchmarks gaussian and srad, both of which are selected from Rodinia benchmark suite [31]. In this figure, each bar indicates the number of different data addresses that are reused in the L1 data cache by a certain number of times, which varies from 0, 1, up to 15, or more. As we can see, the number of different addresses reused in the L1 data cache varies slightly across different SMs because of the GPU SIMD execution model. We also find for both benchmarks a considerable number of data addresses are never reused at all or are only reused for a very small number of times. For example, in gaussian, nearly half of the addresses are used for just once, while in the srad the majority of the addresses are not reused at all. The very low temporal locality from GPGPU applications is quite different from typical CPU applications that tend to have good temporal locality; therefore, we need to explore novel cache management techniques for GPUs.

For data that are never reused at all, loading them into the cache is not helpful



(a) Data usage distribution of gaussian benchmark



(b) Data usage distribution of srad benchmark

Fig. 5. The data usage distribution[13]

to reduce neither latency nor memory bandwidth. On the contrary, bypassing them may reduce cache pollution. Even if the data are reused a few times, loading them into the L1 data cache may increase the global memory traffic if the load utilization rate is low. This may negate the benefit of a small number of cache hits. Therefore, it becomes attractive to bypass those data that are never reused or only reused a few times to reduce the memory bandwidth pressure and cache pollution for GPUs.

3.3.3 Heuristic for GPU Cache Bypassing

We propose to use profiling to identify the L1 data cache accesses that should be bypassed. We focus on bypassing the data accesses that have low load utilization

rates and low reuse times in the L1 data cache, with the objective to minimize the global memory traffic. More specifically, for each data address A that is accessed by a global load, we use profiling to collect its load utilization rate U and the reuse time R. Equation 3.1 is used to determine which data accesses should be bypassed.

$$U \times (1 + R) < 1 \tag{3.1}$$

In the above equation, *(1 + R)* represents the number of times A is accessed from the L1 data cache, including the first time when it is loaded into the cache, i.e., 128 bytes are transferred from the global memory. If U is 1, then this product is at least 1, even if A is not reused at all, indicating A should not be bypassed. On the other hand, if U is less than 1, and if R is 0 or a small integer (e.g. 1, 2, 3) such that the condition in Equation 3.1 holds, then storing A into the L1 data cache will actually increase the global memory traffic as compared to bypassing this access from the L1 data cache. Therefore, in this case, bypassing A can reduce the global memory traffic, potentially leading to better performance or energy efficiency. The reduction of cache pollution will also be a positive side effect of bypassing this data from the L1 data cache. Our cache bypassing method considers both spatial locality (i.e. U) and temporal locality (i.e. R). For example, for the memory access pattern with low load utilization rate as depicted in Fig. 4 (b), i.e., $U = 25\%$, this address must be reused at least 3 times in the L1 data cache (i.e. $R = 3$) to not be bypassed. In contrast, for the memory access pattern with high load utilization rate that is shown in Fig. 4 (c), i.e., $U = 75\%$, if this address is reused at least once from the L1 data cache (i.e., $R = 1$), then it should not be bypassed. To support the profiling-based method, we modify the *GPGPU-Sim* by adding the functions to generate detailed statistics of L1 data cache accesses and enable the L1 data cache model to selectively bypass the identified

data addresses. The detailed statistics results include the information of data reuse time and load utilization rate of each memory access with different addresses, which are automatically analyzed by scripts to generate the list of bypassing addresses for each SM separately. The bypassing addresses are annotated and the benchmarks are simulated again with *GPGPU-Sim* with the bypassing function enabled to implement the profiling-based cache bypassing method.

3.4 Evaluation Results

Fig. 6 compares the performance of the three schemes, which is normalized to the total number of execution cycles of the L1 data cache without bypassing. As we can see, the cache bypassing method improves the performance for all benchmarks. For example, the total number of execution cycles for lud is reduced by more than 40% with cache bypassing, and the average reduction of execution cycles for all benchmarks is 13.8%. Compared to the performance without the L1 data cache, the L1 data cache with bypassing achieves superior performance for all benchmarks except bfs, and on average, the L1 data cache with bypassing improves performance by 8.5% as compared to that without the L1 data cache.

The performance improvement of cache bypassing comes from two factors. The first factor is the reduction of the global memory traffic caused by cache bypassing, which is shown in Fig. 7. The results indicate that cache bypassing reduces the global load memory traffic by 24.7% on average, as compared to the L1 data cache without cache bypassing. Compared to the GPU without using the L1 data cache, cache bypassing reduces the global load memory traffic by 3.1%, leading to better performance.

The second factor for performance improvement is that cache bypassing reduces L1 data cache miss rates as shown in Fig. 8. The cache miss rate is decreased by

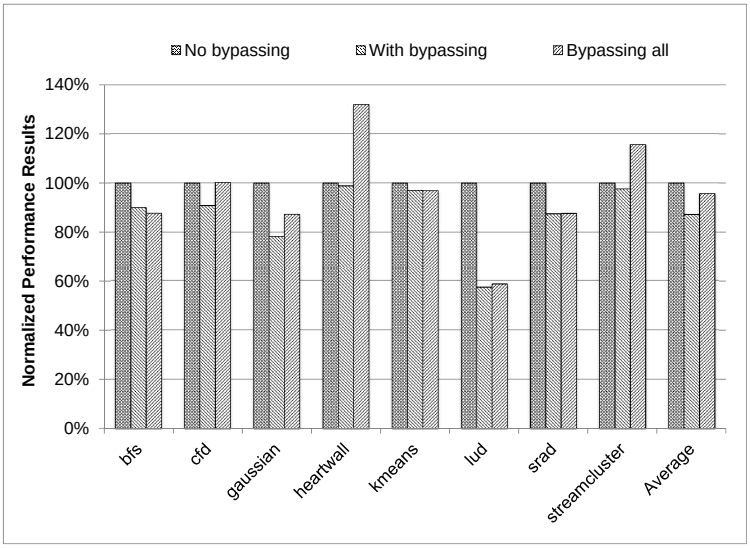


Fig. 6. Normalized total number of execution cycles with the L1 data cache without bypassing and with bypassing, and without the L1 data cache, which are normalized to that with the L1 data cache without cache bypassing.[13]

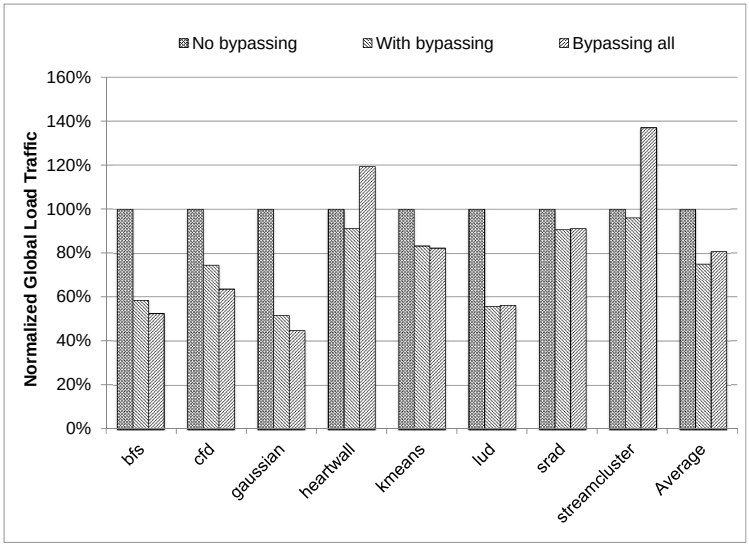


Fig. 7. Normalized global memory traffic with the L1 data cache without bypassing and with bypassing, and without the L1 data cache, which are normalized to that with the L1 data cache without cache bypassing.[13]

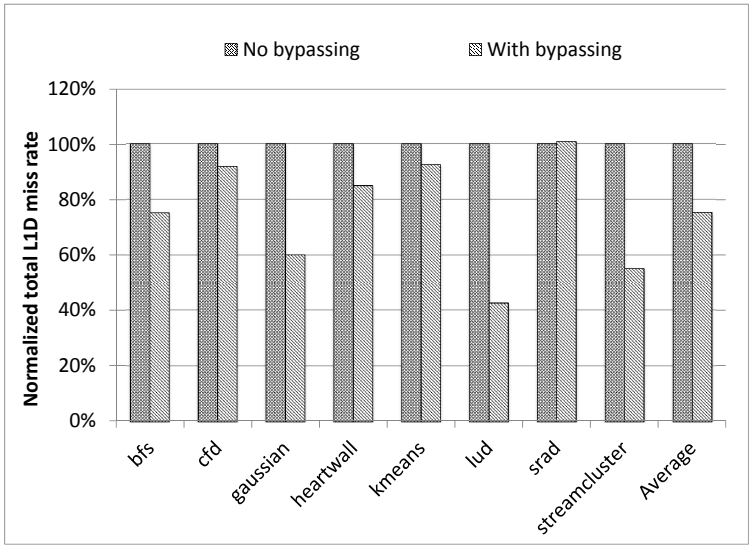


Fig. 8. Normalized L1 data cache miss rates with and without cache bypassing, which are normalized to that without cache bypassing.[13]

up to 57.5% for lud, and the average reduction is 24.6%. Particularly, when cache bypassing reduce both global memory traffic and cache miss rates, the performance is improved dramatically. For example, for both lud and gaussian, both the global memory traffic and cache miss rates are reduced significantly. As a result, the performance of lud and gaussian is improved by 42.7% and 21.8%. In contrast, for some benchmarks such as streamcluster, although cache bypassing reduces its cache miss rate by 44.8%, its global memory traffic is only reduced by 3.8%, leading to small performance improvement of 3.4%. This also indicates that reducing memory traffic may be more important than reducing cache miss rates for GPGPU programs.

It should also be noted the proposed bypassing method does not necessarily reduce the L1 data cache miss rate, for example *srad*, because the total number of accesses to the L1 data cache is also reduced by bypassing. However, on average, the L1 data cache miss rate is reduced by 24.6%, indicating that the proposed cache bypassing method can effectively alleviate cache pollution and improve performance.

CHAPTER 4

WARP-BASED LOAD/STORE REORDERING FOR BETTER TIME-PREDICTABILITY IN GPU L1 DATA CACHE

4.1 Introduction

In hard real-time systems, there are plenty of applications that need to process a massive amount of data, for example, real-time traffic sign and speech recognition, and autonomous navigation. GPUs are a promising platform to accelerate those applications, as long as the execution time is predictable so that the WCET can be computed accurately and efficiently. Unfortunately, many architectural features designed for improving the average-case performance are harmful to time predictability, for example, dynamic scheduling, out-of-order execution, etc. In particular, cache memories are well known to be good for performance but bad for time predictability, because the memory access time is now dependent on whether the access hits in the cache or not, which is often hard to predict statically. For the cache memories used in GPUs, due to the use of many threads and the dynamic warp scheduling, the memory access time is not only dependent on the run-time access history of the cache, but also dependent on the execution orders of the threads, warps and the instructions in each thread. This makes the WCET analysis for GPU caches much more complicated and challenging. Since the Fermi architecture, NVIDIA GPUs have begun to use the L1 data caches and L2 unified cache to improve the average-case performance, and now all kinds of cache memories are increasingly used in various GPUs. Therefore, a cache memory architecture that can offer both time predictability and high performance becomes critical to support hard real-time computing on GPUs.

To improve the predictability of the GPU L1 data cache, we propose a framework that is based on compiler and architectural extensions in GPUs. The proposed framework supports dynamic warp scheduling while reordering the load/store instructions to enable safe and accurate timing analysis for L1 GPU data caches. The experiment results indicate that the worst-case GPU L1 data cache misses can be tightly computed, while the proposed approach achieves better performance than a pure round-robin scheduling.

4.2 Related Work

Studies have been done on real-time scheduling algorithms for GPUs and heterogeneous processors [32] [33][34]. These works basically assume that the WCET of the real-time tasks is already known, which reveals the importance of improving time predictability of GPU architectures to support hard real-time computing.

A large number of research efforts also have been made to improve the time predictability of cache memories for CPUs, among which cache locking is a well-studied method for better predictability [35][36][37]. Some alternative designs to normal cache memories are Scratchpad Memory (SPM) [38] and method cache [39]. There are also a number of studies on WCET analysis of cache memories [40][41][42][43][44], which, however, focus on normal CPU caches rather than GPU caches.

Studies have also been done on regulating the memory accesses to GPU caches to improve the performance. Xie et al. propose a compiler-based framework to bypass the memory access instructions with bad localities for better performance[30]. Jia et al. use reordering and bypassing to get more cache-friendly access orders[20]. But neither of these aims at improving the predictability of GPU caches.

4.3 Dynamic Behaviors in GPU

Dynamic warp scheduling and out-of-order execution of warp instructions are involved when a GPU kernel runs. GPUs rely on these dynamic mechanisms to hide memory and other latencies and to improve the average-case performance and throughput.

4.3.1 Dynamic Warp Scheduling

Whenever a warp is stalled, e.g. the needed resource is unavailable, the warp scheduler dynamically finds a ready warp among the active ones to issue. Therefore, the issuing order of warps does not necessarily follow the order of the warp IDs. For instance, if there are 3 warps *W0, W1, W2*, for the same instruction, issuing order can be any one out of the 6 possible combinations of the 3 warps, e.g. [W1, W2, W0] or [W2, W0, W1].

4.3.2 Out-of-Order Execution

Among the instructions from the same warp, the execution order does not necessarily follow the order of the instructions in the kernel program either. This is because after the instructions are issued, they need to wait until all the operands are ready before execution. Due to data dependencies, a trailing instruction in the kernel program may have all its operands ready earlier than a leading instruction in the kernel program, and thus can be executed earlier. For example, if there are two instructions *I0* and *I1* where *I1* is behind *I0* in the kernel code, the execution of *I1* can be earlier than that of *I0* if its data get ready before that of *I0*.

When the dynamic scheduling and out-of-order execution are combined together, there are many more possible execution orders. An example is shown in Fig. 9 to

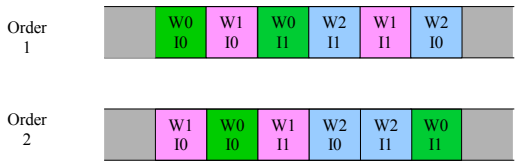


Fig. 9. An Example of Possible Execution Orders of Warp Instructions.

illustrate this. Although due to space limitation, only 2 possible orders are shown, the total number of possible execution orders of 3 warps and 2 instructions can be totally 6! (or 720). As the number of warps and instructions increases, the number of possible execution orders would be prohibitively large for efficient and accurate WCET analysis.

4.3.3 Independent Execution Among Warps

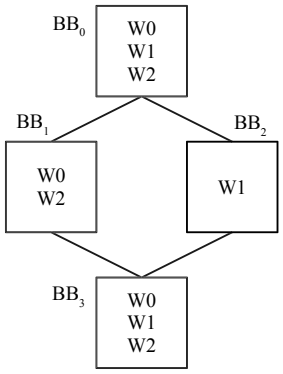


Fig. 10. Example of Warp and Basic Block Relations.

In the CUDA programming model, warps are independent to each other if no special synchronization instruction is used, i.e. there is no synchronization at the boundaries of basic blocks by default. In the example shown in Fig. 10, *W0* and *W2* execute BB_0, BB_1 and BB_3, while *W1* executes BB_0, BB_2 and BB_3. This makes it possible that *W1* executes in BB_3, while *W0* and *W2* are still in BB_1. Consequently,

the warp instructions may not follow the basic block order in the control flow to access the data cache or memory.

4.4 GPU L1 Data Cache Access Reordering Framework

4.4.1 Challenges of GPU Execution on Cache Timing Analysis

The dynamic warp scheduling and out-of-order instruction execution pose great challenges for cache timing analysis. Abstract interpretation is a technique that has been successfully used in cache timing analysis for CPUs. It uses a global abstract cache state to model and predict the cache behavior in the worst case at each boundary of basic blocks [44]. A basic assumption of applying the abstract interpretation in cache timing analysis is that for each basic block, the memory access sequences to the cache can be statically derived from the control flow graph. However, this assumption *cannot* be guaranteed at all in GPUs due to the aforementioned dynamic warp/instruction execution behaviors of GPUs.

In static timing analysis for CPUs, a range of memory space can be used for the data accesses whose addresses are unpredictable. However, this approach becomes unaffordable for GPU kernels, because the huge number of memory accesses a kernel usually has can lead to overly pessimistic or useless WCET estimation results. For example, the maximal number of the relative age of memory blocks that *may* be in the cache can be significantly overestimated due to the massive number of threads and cache accesses that can be executed out-of-order. Therefore, in this framework, it is assumed that the addresses of data accesses to the L1 data cache and the branch conditions are statically known, which actually are not uncommon in GPU kernels that access data and operate based on the thread and block IDs. The proposed method contains 3 software and hardware components, including a compiler-based

GPU L1 data cache access analyzer, a worst-case L1 cache miss rate estimator, and a channel-based architectural extension for predictable L1 data cache access reordering.

4.4.2 Issues of Regulating the Warp Scheduling Orders

In GPUs, it is possible to improve the predictability by reducing the dynamic behaviors, like using a strict round-robin warp scheduling policy, but the performance overhead can be significant. Therefore, our goal in this work is to achieve predictable caching by imposing a minimum constraint on regulating the GPU dynamic execution behavior and minimizing the performance overheads.

Actually, even a pure round-robin warp scheduling policy, in which the warp scheduler issues one instruction for a warp following the warp IDs strictly, still cannot guarantee the order of different warp instructions in different basic blocks, since the execution traces of different warps are usually independent to each other. Therefore, simply regulating the warp execution order does not change the out-of-order execution of instructions in a certain warp, which can still impact the time predictability of GPU data caches.

4.4.3 The Load/Store Reordering Framework

As shown in Fig. 11, the proposed framework consists of 3 major components, including the CUDA kernel analyzer, the worst-case L1 data cache miss rate estimator, and the warp-based load/store reordering architectural extension. The kernel analyzer analyzes the PTX code of a CUDA kernel and generates a reorder configuration to guide the load/store reordering unit in the GPU. The kernel analyzer also outputs the memory address values that will be used by the global memory warp instructions in the kernel, which is also used by the L1 data cache miss rate estimator. The details of these three components are discussed in the following three subsections.

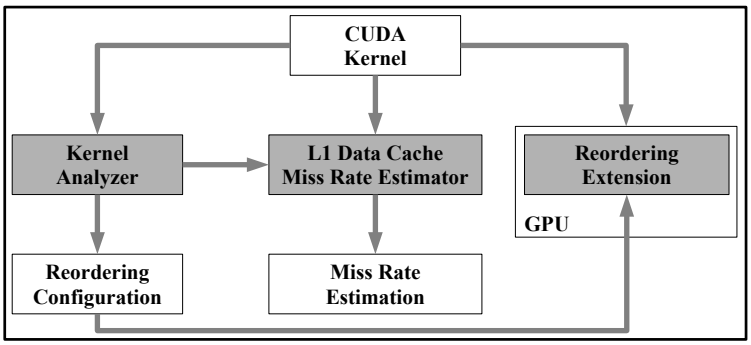


Fig. 11. General Structure of the Load/Store Reordering Framework.[14]

4.4.4 Compiler-Based Kernel Analyzer

The proposed kernel analyzer uses the PTX code and the input values, including parameter values and the kernel hierarchy configuration values, of the CUDA kernel to derive the L1 data cache access pattern and the memory access addresses of both the global load and store instructions of the kernel. Algorithm 1 shows the pseudo code of the analyzer. The kernel analyzer first collects the information about the kernel, including the inputs values, the control flow graph, the number of global load/store instructions and their addresses, from the files of the PTX code and inputs. Based on this information, the analyzer can know the hierarchy of the kernel, such as the number and the size of kernel blocks. For every warp in each kernel block, the analyzer parses the kernel with the information of the warp. The pseudo code is shown in Algorithm 2.

In Algorithm 2, the *KernelParser* takes the information of both the kernel and the warp as inputs and starts with the first instruction in the control flow graph. Each instruction is parsed based on its type. If it is an arithmetic instruction, the value of the target operand is updated based on the calculation type and the value of the source operands (lines 6-8). If it is a global load/store instruction, which accesses

Algorithm 1 GPU L1 Data Cache Access Analyzer[14]

```
Inputs = CollectKernelInputs(FileKernel, FileInput);
CFG = GenerateKernelCFG(FileKernel, FileInput);
LDSTPCList = GenerateLDSTPCs(FileKernel);
BlockAddrInfo = [];
BlockAccInfo = [];
for Each Kernel Block B_i ∈ k do
    for Each Warp W_i ∈ B_i do
        [WarpAddrInfo, WarpAccInfo] = KernelParser(Inputs, CFG, LDSTPCList, B_i, W_i);
    end for
    BlockAddrInfo.append(WarpAddrInfo);
    BlockAccInfo.append(WarpAccInfo);
end for
Return [LDSTPCList, BlockAddrInfo, BlockAccInfo];
```

Algorithm 2 GPU Kernel Parser[14]

```
procedure KernelParser(Inputs, CFG, LDSTPCList, Block, Warp)
    INST = FirstInstruction(CFG);
    WarpAddrInfo = []×length(LDSTPCList);
    WarpAccInfo = []×length(LDSTPCList);
    while INST is not Exit do
        if INST is arithmetic instruction then
            UpdateRegisterValue(INST, Inputs, Block, Warp);
        end if
        if INST is global load/store then
            pc = GetInstPC(INST);
            pcidx = GetPCIndex(INST, LDSTPCList);
            AddrList = AddrListGen(INST, Warp);
            WarpAccInfo[pcidx]=True;
            WarpAddrInfo[pcidx]=AddrList;
        end if
        if INST is a branch or at the end of the current BB then
            INST = FindNextBB(CFG, INST);
        else
            INST = NextInstCurBB();
        end if
    end while
    Return [WarpAddrInfo, WarpAccInfo];
end procedure
```

Algorithm 3 Addresses Generation for Instruction I and Warp W[14]

```
1: procedure AddrListGen(I, W)
2:     AddrList = []
3:     for Each Thread T_i ∈ W do
4:         if CheckAcitve(T_i) then
5:             CurAddr = GetAddr(T_i, I)
6:             Coalesced = False
7:             for Each Address A_j ∈ AddrList do
8:                 if Coalesce(CurAddr, A_j) then
9:                     Coalesced = True
10:                    Break
11:                end if
12:            end for
13:            if Not Coalesced then
14:                AddrList.append(CurAddr)
15:            end if
16:        end if
17:    end for
18:    Return AddrList
19: end procedure
```

the global memory through the L1 data cache and thus is our focus in this paper, all the addresses used by the threads in the warp are coalesced to form a list of addresses (lines 9-15). This information will be used later for the worst-case L1 data cache miss rate estimation as the memory access addresses of this instruction from this warp. The pseudo code of coalescing the addresses is shown in Algorithm 3.

The corresponding value in the *WarpAccInfo* list will be set as true to indicate that this load/store instruction will access the L1 data cache. The parser finds the next basic block based on the control flow graph, if the current instruction is a branch or at the end of the current basic block (lines 16-20). The two lists *WarpAccInfo* and *WarpAddrInfo* are returned by the parser, which contains the access flag and the addresses of each instruction for the warp (line 22).

```
.entry _example(
.param .u64 __cudaparm_input_cuda)
{
.reg .u32 %r<29>;
.reg .u64 %rd<33>;
.reg .f32 %f<20>;
.reg .pred %p<6>;
$Lbegin:
  ld.param.u64    %rd5, [__cudaparm_input_cuda];
```

```
    cvt.s32.u32     %r3, %tid.x;
    mul.wide.s32    %rd3, %r3, 32;
    add.u64         %rd8, %rd5, %rd3;
    cvt.s32.u32     %r1, %ctaid.y;
    mov.s32         %r2, 0;
    setp.eq.s32     %p1, %r1, %r2;
    @!%p1 bra       $L1;
    ld.global.f32   %f1, [%rd8+4];
    bra $L2;
$L1:
    st.global.f32   [%rd8+2048], %f2;
$L2:
    exit;
$Lend:
} // _example$
```

If the configuration of the above kernel is $<<< dim3(1,2,1), dim3(16,4,1) >>>$, for example, and suppose the input value of *__cudaparm_input_cuda* is 0, the output of the kernel analyzer is shown as the follows. As we can see, the kernel has 2 global load/store instructions, and their addresses are 64 and 80 respectively. There are 2 kernel blocks, and each block has 2 warps. The warps in the first kernel block execute the first load/store instruction, and the warps in the second kernel block execute the second load/store instruction. The list of memory access addresses and access types are also shown in the output of the analyzer (i.e. the reorder configuration).

```
-num_pcs 2
-pc_addrs [64, 80]
-grid [1, 2, 1]
-block [16, 4, 1]
Block [0, 0, 0]
Warp0 [1, 0]
Warp0 [[[0, 128, 256, 384], L], None]
Warp1 [1, 0]
Warp1 [[[0, 128, 256, 384], L], None]
Block [0, 1, 0]
Warp0 [0, 1]
Warp0 [None, [[2048, 2176, 2304, 2432], S]]
Warp1 [0, 1]
Warp1 [None, [[2048, 2176, 2304, 2432], S]]
```

4.4.5 Architectural Extension for Warp-Based Load/Store Reordering

We propose to extend the GPU architecture to ensure a predictable load/store order that enables accurate cache timing analysis. Fig. 12 shows the extensions made to the default GPU memory architecture between the load/store unit and the L1 data cache. We propose to add a channel for each active warp[1], and each channel is a buffer to hold requests to the L1 data cache. Besides the head and tail pointers for the buffer, an extra search pointer is used to allow the *Reordering Unit* to search for the expected memory access in the channel, which enables the reordering of memory accesses from the same warp as described below. The *Distributing Unit* accepts the memory accesses from the load/store unit and sends the accesses to different channels according to the warp ID of the access, i.e. memory accesses from warp 0 are sent to channel 0, memory accesses from warp 1 are sent to the channel 1, etc. It should be noted that the warp ID here refers to the dynamic runtime warp ID for a warp when it is executing the kernel. The mapping between a runtime warp ID and the index of a warp in a kernel block can be calculated at runtime when a kernel block is selected to be active.

The load/store reordering happens at two locations in this extended GPU memory architecture. First, load/store instructions within the same warp are reordered in the channel for this warp as aforementioned, because instructions from the same warp can be executed in an out-of-order fashion. For example, the load/store unit can send out memory access request of instruction *I1* before it sends out that of instruction *I0*, even if *I0* is actually before *I1* in the same basic block. This can happen when the two instructions are close to each other and the operands of *I1* become ready earlier

[1]CUDA limits the maximum number of threads per SM, and thus the number of simultaneous active warps is limited.

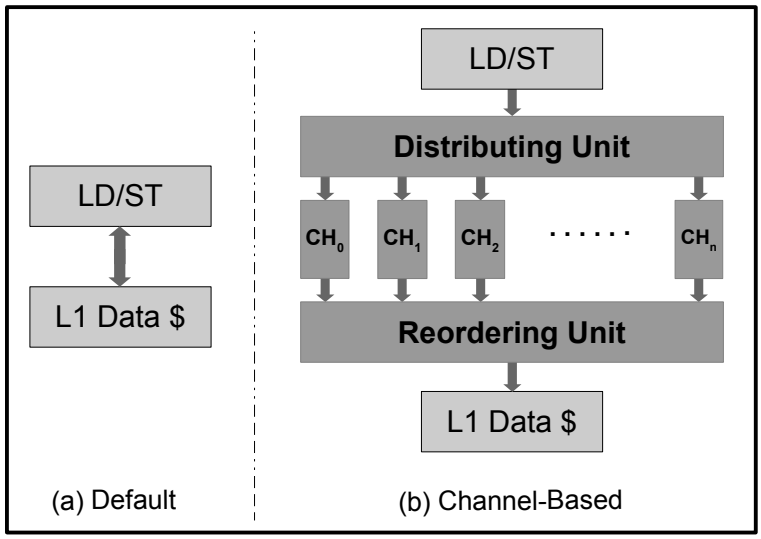


Fig. 12. Warp-Based Load/Store Reordering Architectural Extension.[14]

than those of *I0*. In this case, the *Reordering Unit* uses the aforementioned search pointer to search for *I0* in the corresponding channel, rather than fetching *I1* by the head pointer of the channel. It should be noted that the reordering is only applied to the load/store instructions, which does not affect the dynamic order of other instructions and thus may not affect the overall performance as much as reordering all the instructions such as the pure round-robin scheduling.

The load/store reordering across warps happens in the *Reordering Unit*, which reorders the memory accesses from different channels (i.e. warps). For instance, in the aforementioned PTX code example, there are 2 kernel blocks and 4 warps totally, including *B0W0*, *B0W1*, *B1W0*, and *B1W1*. Assuming the mapping between these warps and the runtime warp IDs is based on the mapping shown in Table 1, Fig. 13 gives 2 out of 24 possible orders of memory requests from the load/store unit.

The analysis results of the GPU L1 data cache analyzer are sent to the reordering unit as the reorder configuration before a kernel is launched, which is used at runtime to decide how to reorder the memory accesses from different channels. The

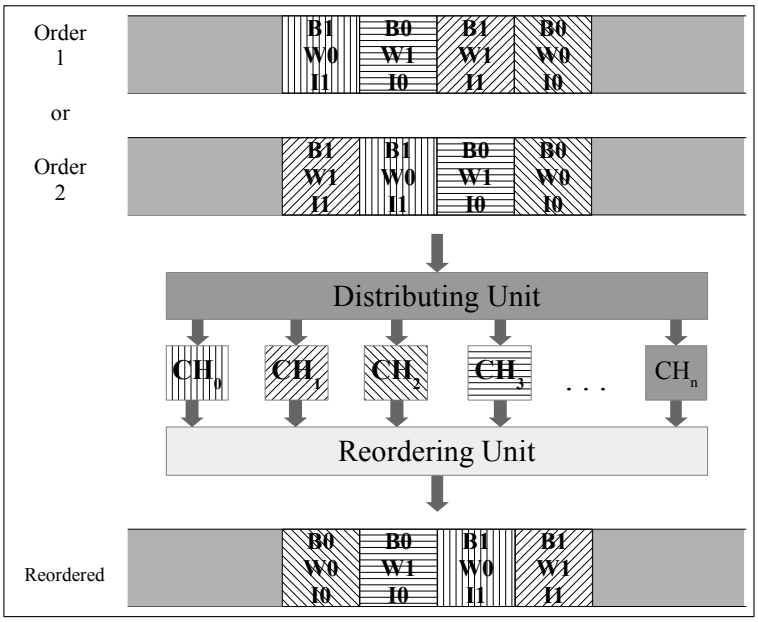


Fig. 13. An Example of Memory Warp Instruction Reordering.[14]

Reordering Unit always searches in the reorder configuration for the warp (channel) with the smallest warp index (in a kernel block rather than runtime warp ID) and the smallest kernel block ID that still has the global load/store instruction with the lowest instruction address to execute. After the *Reordering Unit* gets a memory request from that channel and sends it to the L1 data cache, it updates the reorder configuration so that it can move on and wait for a different channel or a different

Table 1. An Example of Mapping Between Static Block/Warp IDs and Runtime Warp IDs.

Static Block/Warp ID	Runtime Warp ID
B0W0	W2
B0W1	W3
B1W0	W0
B1W1	W1

instruction at the same channel.

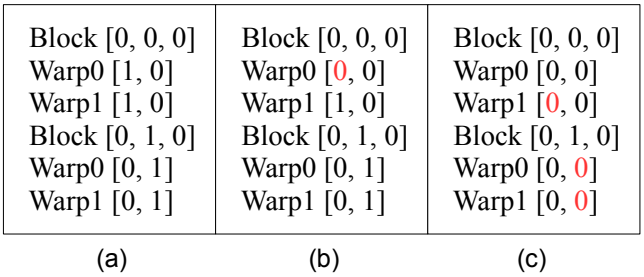


Fig. 14. An Example of Reorder Configurations.

An example is given in Fig. 14 to illustrate the load/store reordering process. The initial reorder configuration is depicted in Fig. 14 (a), based on which the *Reordering Unit* knows it should wait at the channel 2 (i.e. CH_2) for *B0W0*, because *B0W0* is mapped to *Warp 2* according to Table 1. Even if the requests from other warps have entered their channels, the *Reordering Unit* still waits at CH_2 until it receives the memory request from the expected warp instruction and dispatches it to the L1 data cache. After this the reorder configuration is updated to be the one shown in Fig. 14 (b), based on which the *Reordering Unit* knows it should wait at CH_3 for *B0W1* now. The reordering process is continued and eventually the reorder configuration becomes what is shown in Fig. 14 (c) after the reordering unit has dispatched all the memory requests to the L1 data cache in the predictable order.

Fig. 15 shows the different sequences of the memory accesses to the L1 data cache in the above example in three schemes, including the default (i.e. dynamic warp scheduling), the pure round-robin warp scheduling, and the proposed reordering framework. In the default scheme, the access sequence to the L1 data cache can be arbitrary. When the pure round-robin warp scheduling policy is used, the warp scheduling order follows the runtime warp ID. Therefore, in this example the warps in kernel block *B1* are scheduled before those in *B0* according to the mapping between

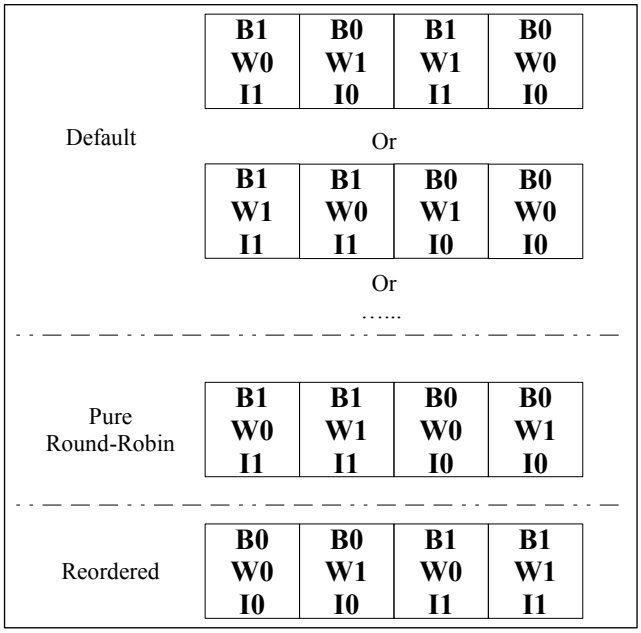


Fig. 15. L1 Data Cache Access Orders of Different Schemes.

the warps in each kernel block and the runtime warp IDs as shown in Table 1. In the reordering framework, the sequence of the accesses is controlled by the reordering configuration as explained above. Therefore, both the reordering framework and the pure round-robin scheduling policy can improve the predictability in the sequence of GPU L1 data cache accesses, compared to the default system.

Fig. 16 uses the same example to demonstrate the performance differences between different schemes. The meanings of the time points Ⓐ to Ⓕ are as shown in the figure. Due to the latency introduced by the reordering extension, the time point for a warp to be ready in the reordering framework may be later than those in the other two schemes, as shown in Fig. 16. Assuming there are 4 warps and the warp scheduler starts with *W0*, with the pure round-robin policy, the next warp is *W1*, which will not be ready for a long period of time as shown in the figure. By comparison, the dynamic warp scheduler, which is used in both the default system

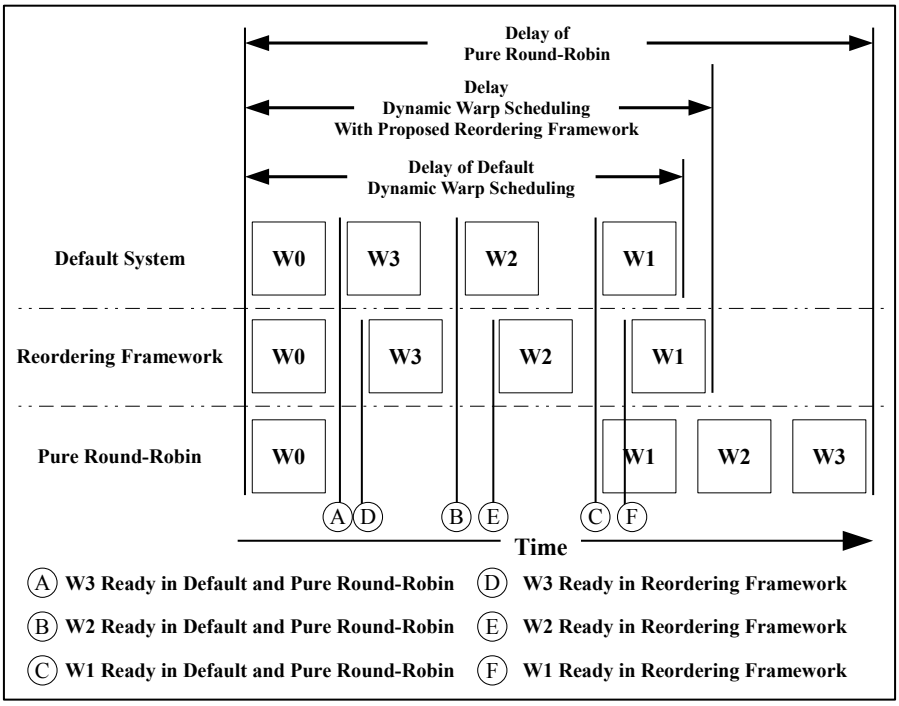


Fig. 16. Delay of Issuing Warp Instructions of Different Schemes.[14]

and the reordering framework, can choose to schedule other ready warps, i.e. *W3* and then *W2* in this example, before *W1*. As a result, the dynamic scheduler can lead to better performance, i.e. shorter delay as shown in the figure. However, due to the latency introduced by the load/store reordering, the delay of the reordering framework in this example is still larger than that of the default system, which also explains the performance overhead of the reordering framework as compared to the default scheme.

4.4.6 GPU L1 Data Cache Miss Rate Estimation

The abstract interpretation method [44] has been successfully used in the static timing analysis of cache memories for CPUs. Abstract Cache State (ACS) is used to analyze the content and behavior of the cache at a certain point in the program. As

shown in Fig. 17, every basic block has an initial ACS_i and an exiting ACS_e at the beginning and the end respectively. The ACS is updated upon each memory reference. Therefore, the differences between the ACS_i and ACS_e of the same basic block depend on the global memory instructions within this basic block. The ACS_i is updated by the memory references in the basic block using a specific cache replacing/updating policy, e.g. LRU.

The ACS_i of the basic blocks with only one predecessor is the ACS_e of its predecessor, e.g. both $ACS_i(BB_1)$ and $ACS_i(BB_2)$ are $ACS_e(BB_0)$. If a basic block has more than one predecessor in the control flow graph, the ACS_e of each of its predecessor is joined together to form the ACS_i of this basic block, e.g. $ACS_i(BB_3) = Join(ACS_e(BB_1), ACS_e(BB_2))$. The *Join* operation can be set intersection or set union depending on whether it is analyzed as an "always hit" or an "always miss".

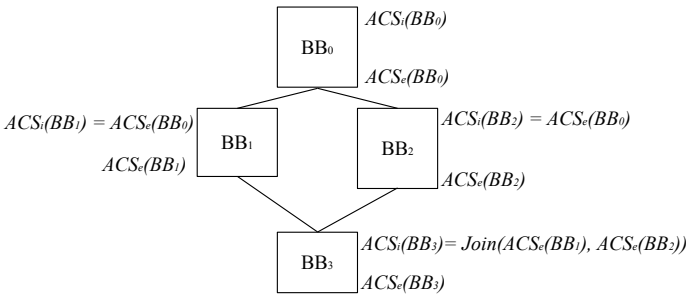


Fig. 17. An Example of Abstract Interpretation Based Static Timing Analysis.

A basic assumption of the abstract interpretation is that the execution of the program only diverges and converges at the boundaries of basic blocks, not in between. It also assumes there is no interference between different program traces. In GPUs, however, these assumptions can only be guaranteed in the same warp, not across warps.

Recalling the example shown in Fig. 10, in abstract interpretation, the $ACS_e(BB_0)$ should be only decided by the content of BB_0. However, due to the independent ex-

ecution of different warps, it is possible that when *W0* and *W2* are still in BB_0, *W1* is in BB_2 already. In this case the $ACS_e(BB_0)$ is affected by the content of BB_2. In another scenario, if *W0* is in BB_1, *W1* is in BB_2 and *W2* is in BB_3, both $ACS_e(BB_1)$ and $ACS_e(BB_2)$ will be affected by the content in BB_3. Therefore, the abstract interpretation cannot be applied to GPU caches directly, because the boundaries between basic blocks are destroyed by the independent and dynamic execution of different warps. In the reordering framework, the boundaries are restored for global memory and data cache accesses despite the dynamic and independent execution of warps. For instance, after the reordering, no global memory accesses from BB_1 or BB_2 can access the L1 data cache until all the accesses from BB_0 for all the active warps are done. Similarly, accesses from BB_1 and BB_2 need to be finished before accesses from BB_3 can retrieve the L1 data cache.

In the proposed reordering framework, the worst-case GPU L1 data cache miss rate estimator uses the information of memory access addresses from the analyzer introduced in Section 4.4.4 and the reordering scheme introduced in Section 4.4.5 to generate the sequence of global memory access addresses for a CUDA kernel. This address sequence is used by the estimator to update the cache models with different configurations to estimate the worst cache miss rate of the GPU L1 data cache for this kernel. Since the sequence of L1 data cache accesses is predictable under the control of the warp-based load/store reordering framework, the miss rate estimator can statically analyze this sequence and generate accurate worst-case cache miss estimation.

4.4.6.1 Limitation of the GPU L1 Data Cache Timing Analyzer

Although the proposed framework can achieve accurate GPU L1 data cache miss rate with low performance overhead as our experiments indicate, not all types of

GPU kernels can be analyzed by our worst-case data cache miss analyzer currently. For example, GPU kernels with input-dependent branches and input-dependent data references cannot be analyzed. Also, the proposed framework requires knowing the loop upper bound statically, which is typical for WCET analysis.

4.5 Evaluation Results

4.5.1 Performance Results

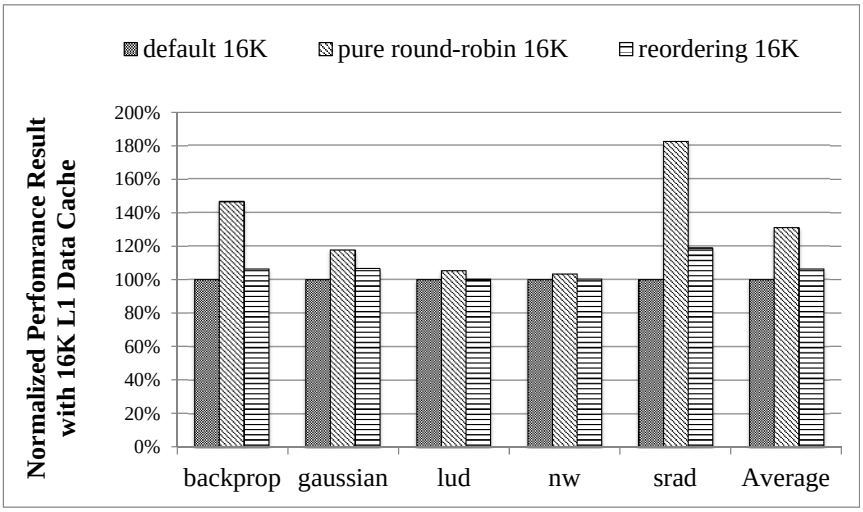


Fig. 18. Normalized Performance Results with 16KB L1 Data Cache.[14]

Figure 18 shows the normalized performance results (in terms of the total number of execution cycles) of 3 different GPU configurations with a 16KB L1 data cache, which are normalized to the performance of the default configuration (i.e. dynamic warp scheduling without load/store reordering). Our experimental results show that the default warp scheduling has the best (average-case) performance as expected. However, the warp-based load/store reordering has much less performance overheads than the pure round-robin scheduling. This is because in the warp-based load/store reordering, instructions other than loads/stores are still scheduled by the

dynamic scheduling, which can dispatch a ready warp into execution faster than the pure round-robin scheduling. An average, the proposed reordering framework can achieve performance 24.4% better than that of the pure round-robin scheduling while achieving time predictability for the GPU data cache.

Figure 19 shows the normalized performance results of different cache sizes. The results are normalized to the performance results of the default configuration with the 16KB L1 data cache. As shown in the results, the reordering scheme has much better performance than the pure round-robin scheme and has only a small performance overhead compared to the default scheme without reordering for all these three different cache sizes.

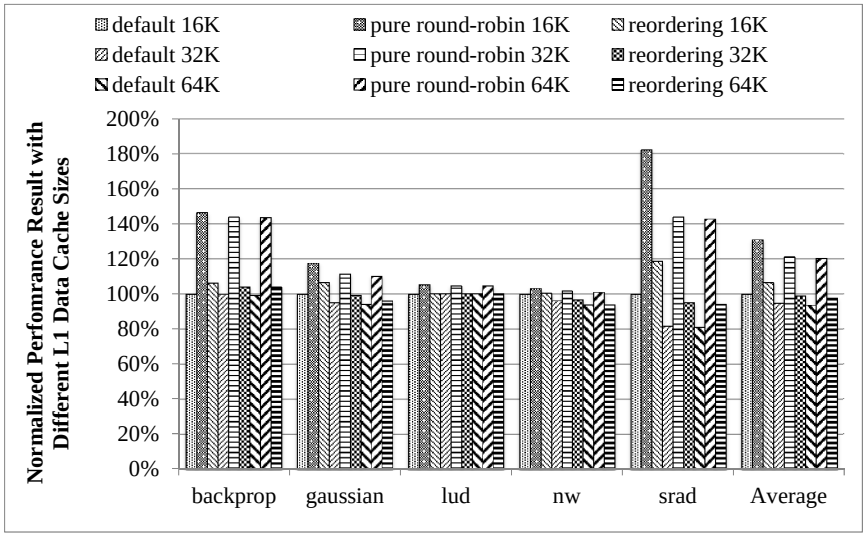


Fig. 19. Normalized Performance Results with Different L1 Data Cache Sizes.[14]

4.5.2 GPU L1 Data Cache Miss Rate Estimation Results

Figure 20 shows the simulated and estimated GPU L1 data cache miss rate results, which are normalized to the simulated miss rate, with a 16KB L1 data cache in each SM. The results show that the proposed estimator, together with the channel-

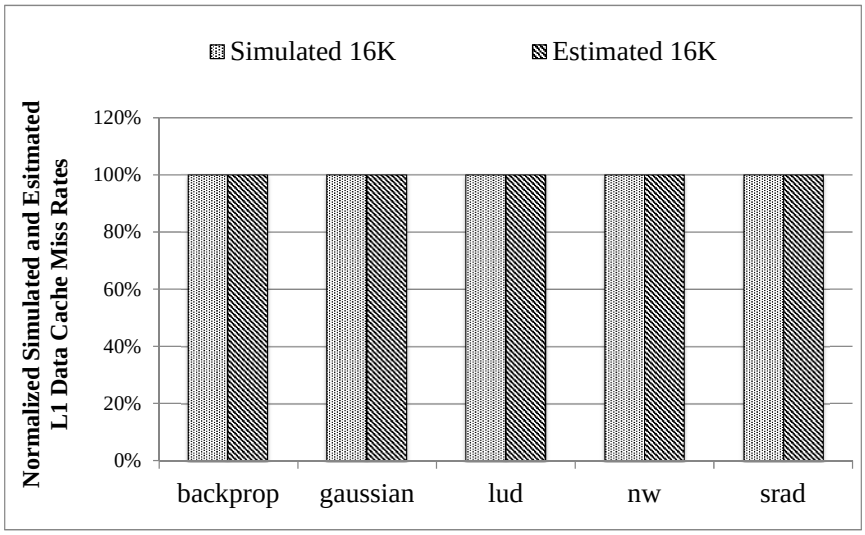


Fig. 20. Miss Rate Estimation Results with 16KB L1 Data Cache.[14]

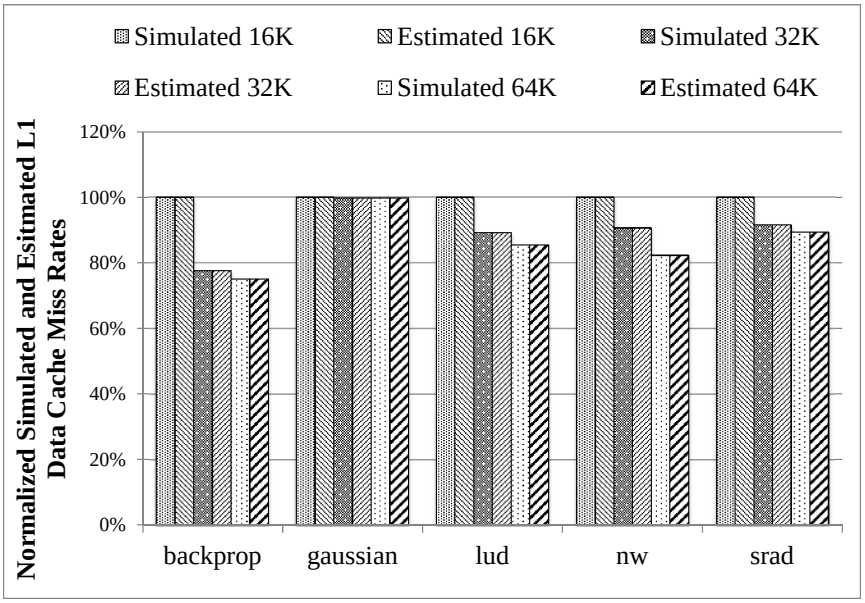


Fig. 21. Miss Rate Estimation Results with Different L1 Data Cache Sizes.[14]

based reordering framework, can have very accurate estimation of the GPU L1 data cache miss rate. Figure 21 shows the simulated and estimated GPU L1 data cache miss rate results of 3 different cache sizes. The results show that the miss rate estimator can provide accurate miss rate estimations in different cache sizes.

CHAPTER 5

TIMING MODEL FOR STATIC WCET ANALYSIS OF GPU KERNELS

5.1 Introduction

To achieve high average-case performance and throughput, modern GPUs maintains a massive number of active threads at the same time and uses a large number of on-chip cores to schedule and execute among these threads. The scheduling among the active threads is a dynamic behavior, which is very hard to analyze statically and harms the predictability. Moreover, the dynamic scheduling among different threads executing the same program code breaks the orders and relations of the instructions and basic blocks. Therefore, the traditional static analysis methods cannot be applied to GPUs directly. Furthermore, the computing cores on a GPU chip are divided into groups, which are connected to the memory partitions through interconnection networks. The dynamic behavior of cores in competing for the memory resources is also hard to predict statically.

Therefore, before applying GPUs to real-time applications, the time predictability of the GPU architecture needs to be improved and analyzed. To address this problem, we proposed to employ the pure round-robin scheduling, which has predictable behaviors, as the scheduling policy, based on which we propose a worst-case timing model for GPGPU programs. With the proposed timing model, a static analyzer, which can analyze the assembly codes of the GPGPU programs and give the WCET estimations, is also built. The evaluation results show that the proposed timing model and static analyzer can provide safe WCET estimations for GPGPU

applications.

5.2 Related Work

The studies on performance analysis of GPU architecture and GPGPU applications [45][46][47][48] focus on building the performance and/or energy models. There are also studies focusing on building and analyzing the performance model of a specific algorithm on GPU or heterogeneous platforms[49][50]. These studies mainly focus on the models of average-case performance and/or using the model to identify the performance bottleneck, while the performance model in this work focuses on the WCET estimation.

There are also studies on the GPU warp scheduling policies[51][52] to improve the efficiency in utilizing the computational resources and to access the memory in a more friendly way, so that the performance is improved. However, the proposed scheduling policy in this work focuses on improving the predictability of the GPU architecture. The memory access reordering method we proposed in [53] regulates the order of memory accesses to the GPU L1 data cache to improve the time predictability of the GPU L1 data cache, while the proposed scheduling policy and analyzer in this work focus on the timing model of the whole GPU system rather than just the L1 data cache.

There are some studies on GPU WCET analysis[54][55] using measurement-based methods, while the proposed WCET analysis method in this work is based on detailed analysis of the GPU architecture and can give safe WCET estimations for GPU kernels.

5.3 GPU WCET Analysis with Predictable Warp Scheduling

5.3.1 Pure Round-Robin Scheduler Timing Model

The method we propose is to use the pure round-robin warp scheduling policy, so that a timing model can be built for the execution of the warps in a GPU kernel. In this scheduling policy, the scheduler must issue one instruction one warp, before moving to the next warp according to the order of warp IDs, as long as the current warp is not waiting at a synchronization barrier. Therefore, based on the dependencies between instructions, the PTX code of a GPU kernel can be divided into segments, each of which has one instruction and are called *Code Segment*. The dependencies between these code segments are decided by the contents of adjacent segments. The instruction in one code segment cannot be issued until the source operand with the longest latency is ready. Therefore, the latency between code segments can be estimated by the dependencies among the instructions and the execution latencies of each instruction in a warp.

$$
\begin{aligned}
&T_{00} \Leftarrow 0 \\
&T_{i0} \Leftarrow T_{i'0} + LI_{i'0} (i > 0) \\
&i' = (i - 1)
\end{aligned}
\tag{5.1}
$$

$$
\begin{aligned}
&T_{ij} \Leftarrow MAX(T_{i'k} + LI_{i'k}, T_{ij'} + LI_{ij'} + LE_{ij'}) \\
&k = (i == 0) \ ? \ (j - 1) \ : \ j \\
&i' = (i == 0) \ ? \ (N - 1) \ : \ (i - 1) \\
&j' = j - 1 \\
&N : Number\ of\ Warps
\end{aligned}
\tag{5.2}
$$

$$
\begin{aligned}
T_{i(end)} &= T_{ij\_last} + LI_{ij\_last} + LE_{ij\_last} \\
WCET &= MAX(T_{0(end)}, T_{1(end)}, ..., T_{N-1(end)})
\end{aligned}
\tag{5.3}
$$

$$
\begin{aligned}
LI_{inst_{Arithmetic}} &= (N <= C_{pipeline})\ ?\ 0 : LI_{Stall_{Arithmetic}} \\
LI_{inst_{Memory}} &= (N <= C_{pipeline})\ ?\ 0 : LI_{Stall_{Memory}}
\end{aligned}
\tag{5.4}
$$

$$
\begin{aligned}
LI_{Stall_{Arithmetic}} &= L_{initiation} \\
LI_{Stall_{Memory}} &= N_{coal} + N_{coal} \times N_{CompetingSM}
\end{aligned}
\tag{5.5}
$$

$$
\begin{aligned}
LE_{inst_{Arithmetic}} &= Length_{pipeline} + L_{initiation} + L_{execution} \\
LE_{inst_{Memory}} &= L_{base} + \\
&\quad Length_{pipeline} \times (N_{coal} + N_{coal} \times N_{CompetingSM})
\end{aligned}
\tag{5.6}
$$

$$
\begin{aligned}
LI_{ij} &= 1 + LI_{inst_{ij}} \\
LE_{ij} &= LE_{inst_{ij}}
\end{aligned}
\tag{5.7}
$$

Fig. 22 shows the scheduling of N warps (W_0 to W_{N-1}), each of which has a certain number of code segments, with the pure round-robin scheduling policy. T_{ij} represents the time point when the GPU can start to issue the code segment j of warp i. LI_{ij} is the latency of issuing the code segment j of warp i, while LE_{ij} represents the execution latency of between segments. After initializing the starting issuing time point of each warp by Equation 5.1, then the rest of the time points in the scheduling can be calculated using Equation 5.2, which basically means that the time point when one code segment in a warp can start to issue depends on the maximal latency

between the latency of the execution the previous code segments in the same warp and the latency of issuing the segments in other warps before the scheduler gets back to this warp. For instance, the time point for the second second segment in W_0 to start to issue could be T'_{01} in the figure, if LE_{00} is less than LI_{10} + LI_{20} + ... + $LI_{(N-1)0}$. Based on this timing model, the estimated WCET is the time point when all the warps finish the execution, as shown in Equation 5.3.

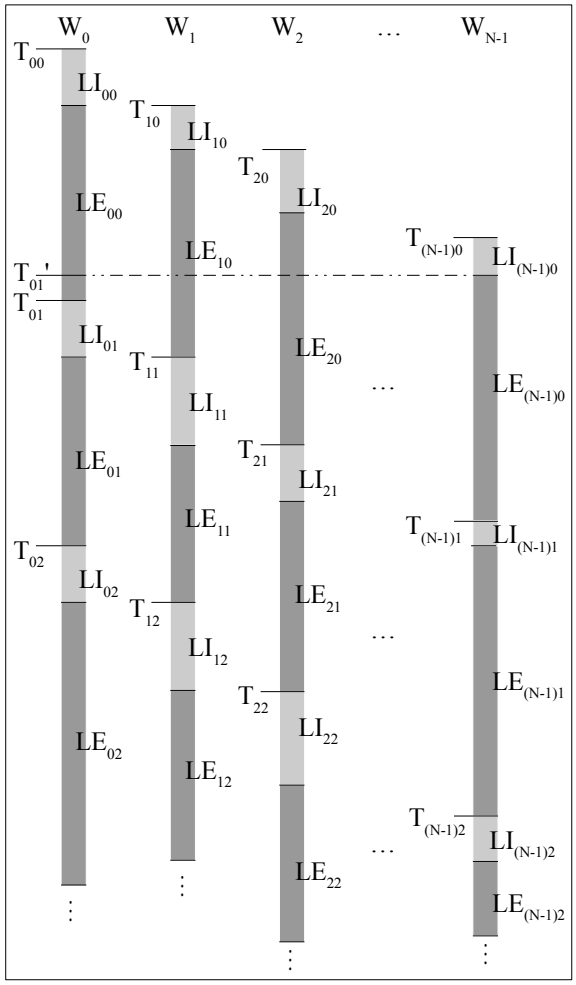


Fig. 22. Timing Model of Pure Round-Robin Scheduling Policy

Fig. 23 shows an example of the WCET calculation based on this timing model and Equation 5.1 to 5.3. Among the four warps, W_0 has three code segments while

the other warps have two. The latency values, i.e., LI and LE, of each warp are as shown in the figure. Based on these latency values, the values of T_{ij} of each code segment in each warp can be calculated using Equation 5.1 and 5.2, as shown in the figure. Finally, the $T_{i(end)}$ of each warp and the WCET of this kernel are calculated using Equation 5.3. As shown in the figure, since W_1 finishes its execution the last at cycle 31, the estimated WCET is 31 for this example.

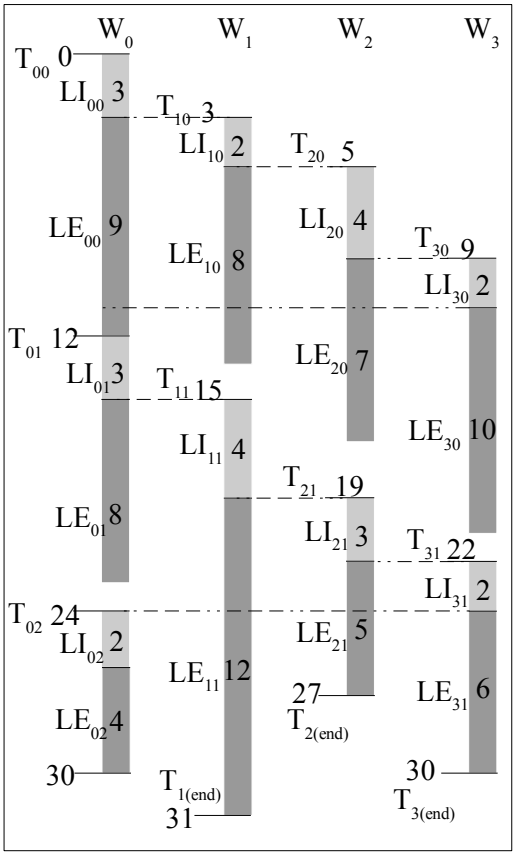


Fig. 23. WCET Calculation Example

5.3.2 Code Segment Issuing and Execution Latency Timing Models

To use the timing model, the latencies of issuing and executing code segments in a GPU kernel need to be estimated statically. Generally, the time needed to issue a code

segment is related to the length of the code segment, if there is no stall in issuing any instruction. But stalls can happen when the number of active warps, which decides how many same type of instructions can be sent to the pipeline in a burst, is larger than the capacity of a pipeline. Also, for the global memory instructions, how many coalesced memory accesses one instruction has and how many SMs will compete to access one memory partition affect both the issuing latency and the execution latency. Therefore, we are interested in three things: the number of active warps, the maximal and average numbers of coalesced memory accesses from one global memory instruction, as well as the maximal and average numbers of competing SMs to access a memory partition.

Number of Active Warps The pipelines in the simulator architecture under analysis can act as buffers for different types of instructions, since a certain number of instructions can stay in the pipeline after being issued. In other words, as long as the pipeline is not full, there will be no extra stalls in issuing. For arithmetic instructions, the configurations of the number of operand collectors and the length of the initiation buffer in function units decide how many instructions the pipeline can hold before the stall happens, while the configuration of initiation latency determines how long the stall is. The kernel analyzer checks whether the number of active warps is larger than the capacity of the pipeline and adds the stall latencies to the code segment issuing period according to the instruction types, as shown in Equation 5.4 and 5.5. For the global memory instructions the following two values play more important roles in the latency estimations.

Number of Coalesced Memory Accesses In a global memory warp instruction, different threads in the warp can access different memory addresses, which are coalesced

together so that addresses belonging to the same 128-Byte memory space are merged together. However, there is no guarantee that all the memory addresses can be coalesced into one. Therefore, there can be as many as 32 memory requests with different addresses from one memory warp instruction. Since these memory requests need to be sent out by the load/store unit one by one at each clock cycle, the number of coalesced memory requests affects not only the issuing latency but also the execution latency of the instruction, as shown in Equation 5.5 and 5.6. The kernel analyzer gives both the maximal and average numbers of coalesced accesses in a GPU kernel. The WCET analyzer can use either one depending on whether it targets hard or soft real-time applications.

Number of Competing SMs Based on the interconnection model in GPGPU-Sim, it is clear that different SMs may compete to access the same memory partition in the memory system. In the simulated architecture, the requests from different SMs are served in a round-robin order. Therefore, if there are M SMs trying to access the same memory partition, the interval for two consecutive requests from the same SM to be served is *M-1* cycles. This latency can happen at every coalesced memory request, as shown in Equation 5.5 and 5.6. Similarly, either the maximal or the average number of competing SMs can be used in the WCET estimation.

Equation 5.5 calculates the possible stall latency in issuing an instruction. For arithmetic instructions, if a stall happens, the latency equals to the initiation latency, whose value is configurable for different types of instructions in the simulator. For global memory instructions, if a stall happens, each coalesced memory access will cause one cycle of stall by itself. Besides, since every coalesced memory access needs to compete to access the global memory, the number of coalesced accesses needs to multiply the possible number of competing SMs. Putting these two parts together,

we have the possible stall latency for global memory instructions.

Equation 5.6 calculates the execution latency of instructions. The execution latency of arithmetic instructions equals to the length of the SP or SFU pipeline plus the summary of the initiation and execution latencies. For global memory instructions, the baseline latency L_{base} is the latency to access the global memory, which every memory access needs to take. The other part in the equation represents the latency for the instructions buffered in the pipeline before the current instruction is sent out to the interconnection network.

Equation 5.7 calculates the LI and LE of a code segment. Adding the size of a code segment $S_{CodeSeg_{ij}}$ and the possible stall latency of the instructions of the code segment, we have the issuing latency, while the execution latency is the maximal execution latency among the instructions in the code segment.

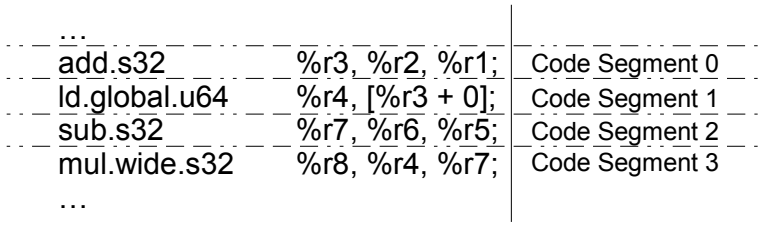


Fig. 24. Example of Timing Model of Code Segments

For example, in Fig. 24 the PTX code is divided into code segments for each instruction, among which we focus on the first two. If in the configuration of the GPU architecture the SP pipeline can buffer 8 instructions, e.g., *add*, *sub*, etc., and the initiation and execution latencies of integer addition operation is 1 and 4 cycles respectively and there are 16 active warps in a GPU kernel, the LI of *code segment 0* in this example is 2 ($S_{CodeSeg}$: 1, $LI_{Stall_{Arithmetic}}$: 1), while the LE of this code segment is 13 ($Length_{pipeline}$: 8, $L_{initiation}$: 1, $L_{execution}$: 4). The LI_{inst} and LE_{inst} of the *sub*

instruction are the same as the ones of *add* instruction in *code segment 0*. For the *ld.global* instruction, assuming the number of coalesced memory accesses (N_{coal}), the number of competing SMs ($N_{CompetingSM}$), the length of the pipeline for this type of instruction ($Length_{pipeline}$) and the baseline memory access latency (L_{base}) are 8, 6, 5 and 200 respectively, the LI_{inst} and LE_{inst} of the *ld.global* instruction are 56 (8 + 8 x 6) and 480 (200 + 5 x (8 + 8 x 6)). Therefore, the LI and LE of *code segment 2*, in this example, are 59 (2 + 1 + 56) and 480 respectively. It should be noted that there is no dependency between *code segment 1* and *code segment 2*, in which case the LE of *code segment 1* is 0.

5.3.3 Static GPU Kernel Analyzer

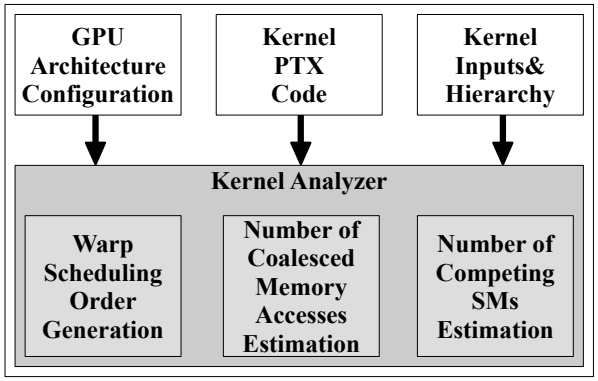


Fig. 25. GPU Kernel Analyzer

The static kernel analyzer parses the PTX code of a GPU kernel to get the estimated value of the metrics in the equations in Section 5.3.2, as well as the scheduling order of each warp, which is used to generate the code segments in the timing model. The analyzer also needs the kernel inputs and the hierarchy configuration of the kernel as inputs for the analysis. Fig. 25 shows the components in the analyzer.

5.3.3.1 Warp Scheduling Order

Algorithm 4 shows how the scheduling order of a warp is generated. The analyzer starts with the first instruction of the first basic block and parses each instruction in the current basic block. The register values are updated with the arithmetic instructions. The analyzer uses the *immediate post-dominator* [56] method to deal with the branch divergence. When the analyzer reaches the last instruction of a basic block, it checks whether it is a branch instruction. If it is a branch instruction and there is branch divergence, then the analyzer finds the immediate post-dominator basic block and pushes it, together with the not taken and then taken basic blocks, to the reconvergence stack. If there is no branch divergence, then either the taken or not taken basic block is pushed to the stack. If this last instruction of the current basic block is not a branch instruction, the analyzer pops the top from the reconvergence stack as the new current basic block. The analyzer appends every new current basic block to the warp scheduling order and returns this trace when it reaches the end of the kernel.

5.3.3.2 Number of Coalesced Memory Accesses

During the analysis process, the analyzer also collects the information of the memory addresses used by each global memory instruction. For these instructions, the memory access addresses of each thread in the warp are calculated based on the values of the registers used by the kernel. All the memory addresses used by the threads in a warp are coalesced together using Algorithm 5. The list of coalesced memory addresses is appended to the result list *AddrCoalAccessList*, which contains lists of coalesced memory addresses of each global memory instruction in the warp. Then the analyzer gets the number (N) of coalesced memory addresses for this instruction

Algorithm 4 Warp Execution Trace Generation

```
1: procedure WarpExeTraceAna(Inputs, CFG, Block, Warp)
2:     WarpExecutionOrder = []
3:     ReconvergenceStack = []
4:     NumCoalAccessList = []
5:     AddrCoalAccessList = []
6:     CurrentBB = FirstBB(CFG)
7:     WarpExecutionOrder.append(CurrentBB)
8:     INST = FirstInstruction(CurrentBB)
9:     while INST is not Exit do
10:        if INST is arithmetic instruction then
11:            UpdateRegisterValue(INST, Inputs, Block, Warp)
12:        end if
13:        if INST is global load/store then
14:            CoalList = CoalescedAddrListGen(INST, Warp)
15:            AddrCoalAccessList.append(CoalList)
16:            N = SizeOf(CoalList)
17:            NumCoalAccessList.append(N)
18:        end if
19:        if INST is last of CurrentBB then
20:            if INST is branch then
21:                if Has Divergence then
22:                    IPD = FindImmediatePostdominator(CFG, CurrentBB)
23:                    ReconvergenceStack.push(IPD)
24:                    ReconvergenceStack.push(NotTakenBB)
25:                    ReconvergenceStack.push(TakenBB)
26:                else
27:                    if Taken then
28:                        ReconvergenceStack.push(TakenBB)
29:                    else
30:                        ReconvergenceStack.push(NotTakenBB)
31:                    end if
32:                end if
33:            end if
34:            CurrentBB = ReconvergenceStack.pop()
35:            WarpExecutionOrder.append(CurrentBB)
36:            INST = FirstInstruction(CurrentBB)
37:        else
38:            INST = NextInstCurBB()
39:        end if
40:    end while
41:    Return WarpExecutionOrder, NumCoalAccessList, AddrCoalAccessList
42: end procedure
```

and appends it to the result list *NumCoalAccessList* of this warp. At the end, the analyzer returns the warp execution trace, the list of numbers that represent the number of coalesced memory accesses, and the list of address lists of each global memory instruction in this warp. The same process is done for every warp and all the results are collected together to calculate the maximal and average numbers of coalesced accesses of the GPU kernel respectively.

Table 2. Example of Number of Coalesced Memory Accesses

	I_0	I_1	I_2	I_3
W_0	16	20	4	6
W_1	8	1	6	8
W_2	12	30	10	2

For instance, if there are 3 warps (W_{0-2}) and each warp has 4 global memory instructions (I_{0-3}) and the number of coalesced memory accesses of each instruction is as shown in Table 2, the maximal and average numbers of coalesced memory accesses are 30 and 11 respectively[1].

5.3.3.3 Number of Competing SMs

Algorithm 6 shows how the kernel analyzer estimates the possible number of competing SMs that may access the same memory partition at the same time. Based on the memory addresses each warp instruction uses, the analyzer builds a vector for every global memory instruction in every SM. This vector represents the distribution of the memory addresses among the memory partitions from a certain instruction on a certain SM. For instance, if there are 3 memory partitions and, from one instruction I on SM s, there are 5 memory addresses used, among which 2 addresses go to partition

[1]The exact average value is 10.25, which is rounded to 11 using ceiling.

Algorithm 5 Coalesced Addresses Generation

```
1: procedure CoalescedAddrListGen(I, W)
2:     CoalAddrList = []
3:     for Each Thread T_i ∈ W do
4:         if CheckAcitve(T_i) then
5:             CurAddr = GetAddr(T_i, I)
6:             Coalesced = False
7:             for Each Address A_j ∈ CoalAddrList do
8:                 if Coalesce(CurAddr, A_j) then
9:                     Coalesced = True
10:                    Break
11:                end if
12:            end for
13:            if Not Coalesced then
14:                CoalAddrList.append(CurAddr)
15:            end if
16:        end if
17:    end for
18:    Return CoalAddrList
19: end procedure
```

0 and 3 addresses go to partition 2, then the distribution vector is *[2,0,3]*. As shown in the algorithm, there is one such vector for every global memory instruction in every SM, i.e., *MemPtnAccVector* is a 2D array of such a kind of vectors. Two metrics are calculated using this vector.

The first metric represents the unevenness of the distribution. The *Distance2Center* function calculates the Euclidean distance between the vector of the address distribution and the vector that represents an even distribution (called *center* in the algorithm). This distance is a metric that indicates how uneven the distribution to different partitions is. The larger the distance is, the more uneven the distribution is and thus the more possibly SMs compete for the same partition. For example, the *center* (the even distribution) vector of 3 memory partitions is *[1, 1, 1]*. Then, for the aforementioned distribution vector *[2, 0, 3]*, the normalized vector is calculated by dividing each element in the vector by the average value of all elements, i.e., *5/3*, resulting *[1.2, 0, 1.8]*. The *Distance2Center* then returns the distance between *[1.2, 0, 1.8]* and *[1, 1, 1]*, which is *1.29*.

Another metric is the Euclidean distance between the distribution vector of one

Table 3. Example of Memory Partition Access Distributions

	I_0	...	...	...
SM_0	[2, 0, 3]	...	...	...
SM_1	[2, 0, 2]	...	...	...
SM_2	[1, 3, 1]	...	...	...

instruction on one SM to the distribution vector of the same instruction on other SMs, named as *D2OtherSM* in the algorithm. The smaller the value of *D2OtherSM* is, the more similar the address distributions from two SMs (s and s') are. If the distance is 0, we have the same distributions and then the number of possibly competing SMs is increased by 1, as show on line 9, where the *MaxDistance* means the maximal distance of two vectors, whose distributions all focus on single but different partitions, e.g., *[5, 0, 0]* and *[0, 0, 4]*. This is a constant value according to the total number of SMs; if there are M SMs, *MaxDistance* is $(M-1)\sqrt{2}$. With the same aforementioned example, if there are 3 SMs and the distribution vectors of the same instruction on the other 2 SMs are *[2, 0, 2]* (SM_1) and *[1, 3, 1]* (SM_2), as shown in Table 3. Then, for the distribution vector *[2, 0, 3]* from SM_0, the *Distance2Vector* function calculates the distance between this vector and those vectors from the other 2 SMs. Similar to the *Distance2Center* function, the *Distance2Vector* function normalizes the vector by the average value and then calculates the Euclidean distance between the two normalized vectors, i.e., the results of *Distance2Vector([2, 0, 3], [2, 0, 2])* and *Distance2Vector([2, 0, 3], [1, 3, 1])* are *0.42* and *2.2* respectively. This indicates that SM_0 and SM_1 have similar distribution in accessing the memory partitions, i.e., it is highly possible that they compete to access the same partition.

Then the number of possible competing SMs (*CompetingSM*) and the distance to the center (*D2Center*) are compared to heuristic thresholds, i.e., $T_{CompetingSM}$

and $T_{D2Center}$, to decide whether the number of possible competing SM of current instruction counts to the final result (line 11). After all the instructions are analyzed for all the SMs, an average value of the number of competing SMs is returned and used in the calculation in Equation 5.5 and 5.6. The maximal value of the number of competing SMs is the number of active SMs minus one. The reason that heuristic thresholds are used is that the behaviors of different SMs are basically independent of each other and, therefore, their interactions are very hard to predict statically. So, we use these heuristic threshold values to estimate the *average degree* of competing among SMs. The heuristic values used in this work are 13 for $T_{CompetingSM}$ and 0.5 for $T_{D2Center}$, for the architecture configuration with 15 SMs and 12 memory partitions. It should be noted that we do not claim the WCET estimation with the *average degree* of competing SMs to be a safe upper bound, while the WCET estimation with the maximal number of possible competing SMs can be considered as the safe upper bound.

Algorithm 6 Average Number of Competing SMs

```
1:  NumCompetingSM = []
2:  for Each I in all load/store instructions do
3:      for Each s in all SMs do
4:          D2Center = Distance2Center(MemPtnAccVector[I][s])
5:          CompetingSM = 0
6:          for Each s' in all the rest SMs do
7:              D2OtherSM = Distance2Vector(MemPtnAccVector[I][s],
8:                                          MemPtnAccVector[I][s'])
9:              CompetingSM += (MaxDistance - D2OtherSM)/MaxDistance
10:         end for
11:         if CompetingSM ¿ T_CompetingSM or D2Center ¿ T_D2Center then
12:             NumCompetingSM.append(CompetingSM)
13:         else
14:             NumCompetingSM.append(0)
15:         end if
16:     end for
17: end for
18: Return average(NumCompetingSM)
```

Table 4. Estimated Average and Maximal Number of Coalesced Accesses and Competing SMs

Benchmark	gsn k1	gsn k2	nw k1	nw k2	cfd k1	cfd k2	lud k1	srad128 k1	srad128 k2	srad512 k1	srad512 k2
Avg. Coalesced Access	22	7	3	2	1	1	2	2	2	2	2
Max. Coalesced Access	32	8	16	16	1	1	2	2	2	2	2
Avg. Competing SMs	0	10	7	5	7	7	10	9	9	13	13
Max. Competing SMs	0	14	14	14	14	14	14	14	14	14	14

5.4 Evaluation Results

Fig. 26 shows the normalized estimated WCET of the simulated GPU architecture with and without the perfect memory configuration. With the perfect memory configuration, every memory request just takes one cycle after it has arrived at the load/store unit and does not go to the memory partitions through the interconnection network. Therefore, with this configuration, no latency in the memory system contributes to the performance or the estimated WCET. The estimated WCET results with the perfect memory configuration is normalized to the simulation performance results with the *same* configuration.

The normalized estimated WCET results with normal memories in Fig. 26 are the estimated WCET results when the simulator and the WCET analyzer use a normal memory system model, in which memory requests further go to different memory partitions through the interconnection network after they arrived at the load/store unit. These estimated WCET results are normalized to the simulation performance results with the *normal* memory system configuration. The results show that generally with a perfect memory model, the estimator has tighter estimations, compared to that with a normal memory model. This is because the predictability within an SM, when no interference from other SMs needs to be considered, is better than the predictability when the interconnection network is included in the model and the interferences from other SMs need to be considered. It should be noted that the

average values of the number of coalesced memory accesses and the average number of competing SMs are used in getting the estimated WCET results in Fig. 26 and the estimated WCET results are normalized to the simulated performance with and without perfect memory respectively. Therefore, the overestimation in the estimated WCET with normal memory can be smaller than the overestimation in the estimated WCET with perfect memory, e.g., in benchmark *gsn k1* and *cfd k2*.

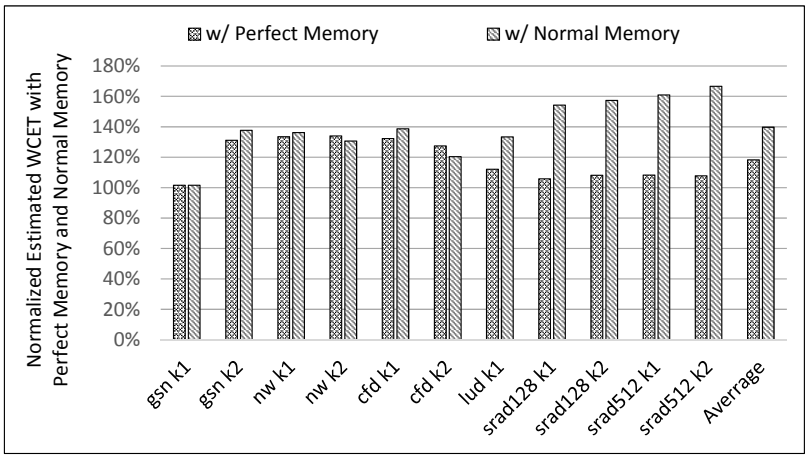


Fig. 26. Normalized Estimated WCET with and without Perfect Memory Model

Fig. 27 shows estimated WCET results of the estimator using different combinations of the estimated number of coalesced memory accesses and the number of competing SMs. Since there are two types of estimated values for each metric, i.e., the average and the maximal, there are four groups of estimated WCET results. In the kernel *k1* of the *gsn* benchmark, only one SM is active in the execution. Therefore, both the average and the maximal number of competing SMs are 0. In all the other kernels all the SMs are active. Therefore, the maximal number of competing SMs is 14, since there are 15 SMs in the configuration. As shown in the results, the estimator has the lowest overestimation when the average values are used in both metrics. The overestimation increases when the maximal estimated value in either or

both of the metrics is used. The estimated average and maximal values of the number of coalesced accesses and competing SMs are shown in Table 4. When the difference between the average and maximal values is small or when they are the same, the increase in the overestimation is small. But, when the difference grows, the overestimation increases. For example, in the *gsn k2* and *nw*, both the number of coalesced access and the number of competing SMs are different in average and maximal values and, as a result, the overestimation is huge when the maximal value is used. For the two hierarchy configurations in the *srad* benchmark, the *srad128* has less estimated average number of competing SMs than *srad512*, since there are less active warps per SM in the *srad128*. Therefore, when the maximal values are used, the overestimation in *srad512* is less then in *srad128*, since the estimated average value is closer to the maximal one. For hard real-time applications, the maximal estimated values of these two metrics should be used, while for soft real-time applications, the average values can be used.

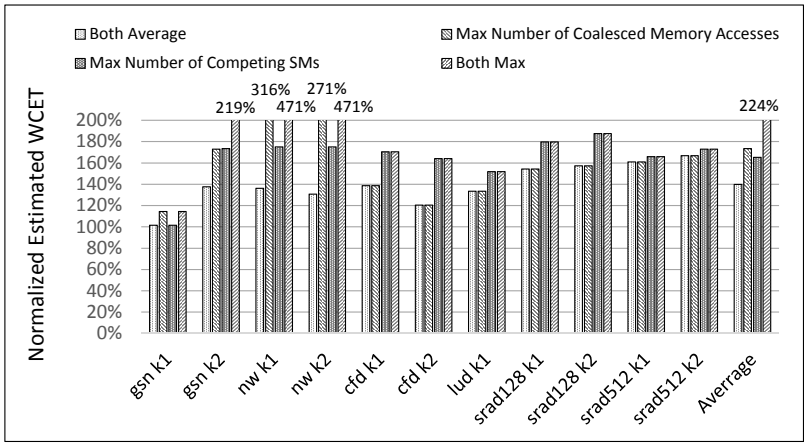


Fig. 27. Normalized Estimated WCET with the Maximal and Average Numbers of Coalesced Accesses and Competing SMs

Fig. 28 shows the normalized average-case performance results of the pure round-robin scheduling policy and the default loose round-robin policy in GPGPU-Sim.

GPUs rely on the warp scheduler to switch the execution among active warps to hide the latency of stalled warps. In the proposed pure round-robin scheduling policy, the scheduler tries to issue one instruction for a warp as long as the warp is not waiting at a synchronization barrier. However, this can introduce performance overhead due to missing some of the opportunities of hiding latency. As shown in Fig. 28, on average the performance overhead is about 50%, which we consider as the trade-off for predictability.

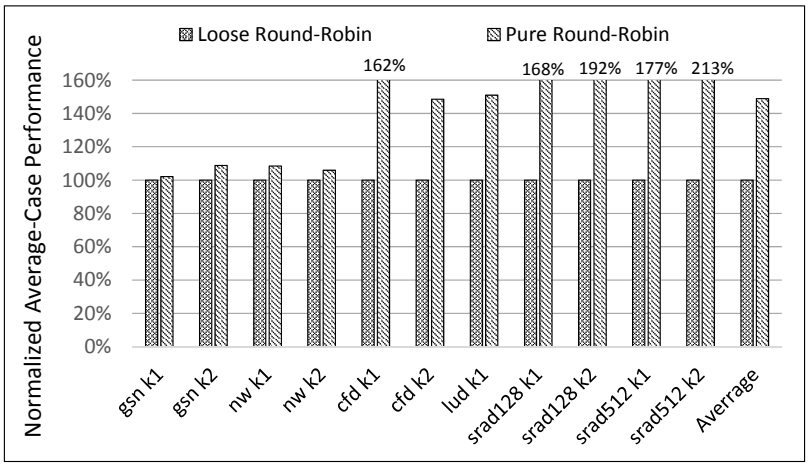


Fig. 28. Normalized Average-Case Performance Results of Loose Round-Robin and Pure Round-Robin Scheduling Policies

CHAPTER 6

STATIC WCET ANALYSIS ON LAST LEVEL DATA CACHE IN INTEGRATED CPU-GPU ARCHITECTURE

6.1 Introduction

While GPUs are used as powerful accelerators, CPUs are also increasingly considered and used as coprocessors, rather than just the host cores that simply launch tasks to GPUs and are not involved in computing. Such integrated CPU-GPU architectures exploit the unique strengths of both types of Processing Units (PUs) as well as the shared resources to further improve the performance, compared to a GPU- or CPU-only system. For instance, seven out of the top ten Green500 supercomputers use both CPUs and GPUs[57], i.e., Heterogeneous Computing Systems.

In a heterogeneous architecture, CPUs and GPUs can have separate memory spaces and can be connected together through a Peripheral Component Interconnect Express (PCIe) bus, which is referred as a *discrete* system. GPUs and CPUs transfer data back and forth through the PCIe bus in such a system, which requires programmers to manage the data needed by both CPUs and GPUs and can introduce performance overheads. As a result, the *integrated* CPU-GPU architecture is proposed and implemented to allow the CPUs and GPUs to share the same memory space and avoid such data transfer, e.g., AMD's Accelerated Processing Units (APUs)[58].

In embedded applications and systems, such a kind of heterogeneous architectures has become popular as well. For instance, the *big.LITTLE* technology [59] combines high-performance cores and energy-efficient cores to achieve power-optimization while

delivering peak-performance capability. Also, by the integration of GPU and CPU architectures, the Tegra [3] processors bring the general-purpose GPU computing power to the embedded systems.

In real-time systems, it is very promising to exploit the computing power of the integrated CPU-GPU architecture as well. However, the issues of the time-predictability in such systems need to be addressed first. One of the problems is the estimation of the behavior of the shared Last Level Cache (LLC) in such an integrated architecture, since it is shared by both the CPU and GPU and can affect the WCETs of both. Therefore, we propose to first explore the WCET analysis of the shared data LLC in the integrated architecture, the analysis results of which are then used in estimating the WCET of the GPU kernels.

6.2 Related Work

For WCET analysis of the multicore architecture, page coloring and locking techniques are studied and used to reduce or remove the conflicts between different cores in the LLC[60][61][62], so that the time-predictability can be improved. Hardware supports are proposed in [63] to guarantee an upper bound delay for hard real-time tasks in multicore systems, while the Time Division Multiple Access shared bus access scheme is proposed in [64] to enable the static shared bus scheduling and shared cache conflict analysis.

Research efforts have been made on partitioning and/or scheduling tasks or specific algorithms on heterogeneous architectures, based on the relative performance of different processing units and/or the characteristics of different subtasks [65][66][67][68]. Some studies focus on the compiler-level methods to automatically generate the programs for heterogeneous systems[69][70], while others propose programming frameworks to utilize the resources[71][72]. Comparisons between the discrete and inte-

grated CPU-GPU architectures show that the integrated architecture can help to reduce the performance and/or energy overheads [73][74][75]. However, few studies focus on the time-predictability issues of the integrated CPU-GPU architectures.

6.3 Reuse Distance

The analysis method proposed in this work is based on the *Reuse Distance* theory. The metric of *Reuse Distance* [76] can be used to analyze the cache behaviors in CPU or GPU programs[77][78]. For set-associative cache memories, the reuse distance of a cache access A can be defined as the number of unique cache accesses that are mapped to the same cache set with A but with different tag values from A since the last access of A. For the very first access to a certain address, the reuse distance is infinity. Assuming the associativity is N, in an LRU cache, a cache access with the reuse distance less than N will be a hit, otherwise it will be a miss.

Table 5. An Example of Reuse Distance.[76]

Access	0	1	2	3	4	5	6	7
Address	A	B	C	D	A	C	E	B
Reuse Distance	∞	∞	∞	∞	3	2	∞	4

For instance, Table 5 shows a sequence of memory accesses with the addresses of A to E, which map to the same cache set but with different tag values. The reuse distance values of each access are as shown in the table. Accesses *0* to *3* with addresses *A, B, C and D* have the reuse distance of infinity, since they are all the very first access of that address. So is the access *6* with address *E*. Accesses *4* with addresses A has the reuse distance of 3, since there are accesses with 3 unique different addresses (*B, C, D*) between access *0* and access *4*. Similarly, access *5 and 7* have the reuse distance values of 2 and 4 respectively.

6.4 Shared LLC Analysis

6.4.1 The Integrated CPU-GPU Architecture Under Analysis

In this work, the gem5-gpu [16] simulator is used as the target architecture under analysis. The default architecture of the gem5-gpu simulator is shown in Fig. 29, where the CPU and GPU both have its own LLC and then connect to the off-chip memory.

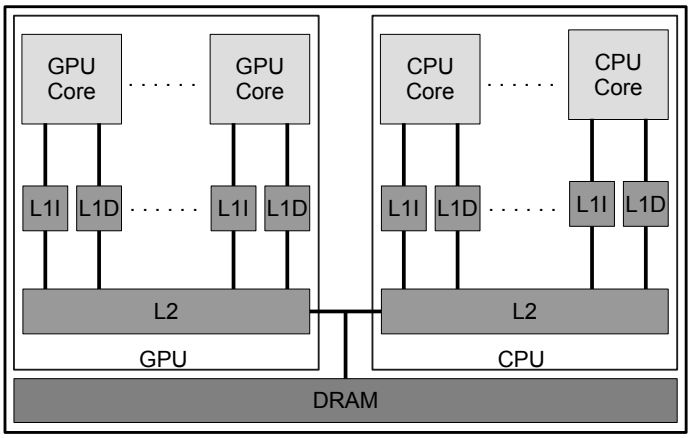


Fig. 29. The Default gem5-gpu Simulator Architecture

To support the shared LLC between GPU and CPU, the memory system in the simulator is modified as shown in Fig. 30, where there are LLCs for instructions and data before going out to the off-chip memory. It should be noted that the LLC is usually used for both instruction and data. However, since the focus of this work is to analyze the shared last level data cache, the LLCs in the target architecture is separated into instruction and data as shown in the figure.

6.4.2 Simple Shared Data LLC Analysis Method

Knowing the order of the memory accesses to the cache is important in using the reuse distance to predict cache hit and miss. In the example of the sequence of

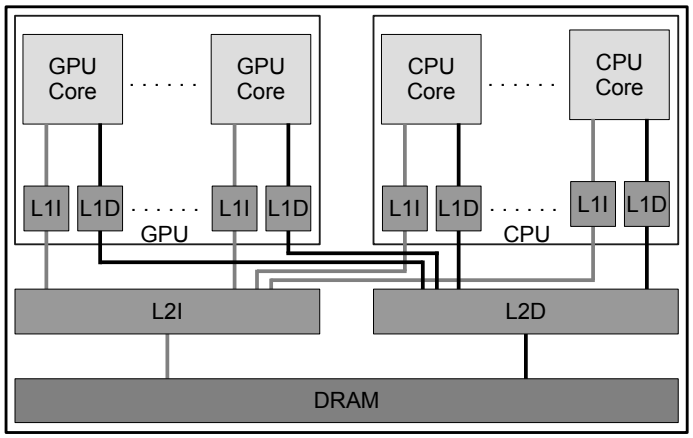


Fig. 30. The Modified gem5-gpu Simulator Architecture With Shared LLC

memory access addresses in Table 5, if the access order between access *3* and *4* is not for sure, i.e. access *4* with address A can possibly be either before or after access *3* with address D, the reuse distance of access *4* with address A then can be either 2 or 3. If there are many accesses whose access order to the cache can not be known for sure, there can be many possibilities in the reuse distance results.

In the worst-case timing analysis for caches, the maximum reuse distance of each access to the cache needs to be estimated, so that it can be compared with the associativity of the cache to predict whether the access is a hit or not. For instance, in Table 5 if the access order of the accesses *4* to *7* is not known, access *4* in the table can become the last access in the sequence, in which case the reuse distance of this access will be 4 rather than 3 of its current position. If the associativity of the cache under analysis is 4, the change of the reuse distance calculation from 3 to 4 will make the prediction of this access from hit to miss. This shows how the uncertainty in the access order can lead to overestimation in cache miss rates.

Unfortunately, for the shared data LLC in the integrated CPU-GPU architecture, the order of accesses from different CPU and GPU cores to the shared LLC is hard

to predict statically. This is because the executions of the GPU kernels and the CPU programs are independent of each other. In other words, while the order of the accesses from the same GPU or CPU core can be analyzed and predicted statically based on the content of the code, the orders of the accesses among different cores are mostly based on the run-time execution and warp/thread scheduling.

	Core 0	Core 1	Core 2
	C0_A	C1_A	C2_E
	C0_B	C1_D	C2_B
T0	**C0_C**	C1_C	C2_D
	C0_D	C1_E	C2_A
	C0_A	C1_A	C2_C
T1	**C0_C**	C1_B	C2_B
	C0_B	C1_D	C2_A
	C0_E	C1_C	C2_D
	.....	.....	.....

Fig. 31. Example of Accesses From Different Cores

Fig. 31 shows an example of how the accesses to shared LLC from different cores can affect the estimation of the reuse distance of one access. There are three cores *0* to *2*, each of which has a sequence of accesses to the shared LLC as shown in the figure. The reuse distance, for instance, of the access $C0\_C$ on *Core 0* at the time point *T1* depends on the accesses that happen between the time point *T0* and *T1*. If there is only one core, then the accesses in the gray area in the column *Core 0* are enough to predict the reuse distance and the hit/miss results. However, there are 2 other cores which access the shared LLC simultaneously and independently. In this case, to find the safe upper bound of the cache miss rate, all the accesses in the gray

area under the three columns need to be considered as the possible accesses to the shared LLC between time point *T0* and *T1*. Obviously, this analysis method simply takes all the accesses from other cores and the accesses in between from the current core to estimate the worst-case reuse distance. Therefore, it is referred as the *Simple* method. It should be noted that that the addresses of memory accesses are calculated statically as described in Section 4.4.4.

Table 6. Shared Data LLC (512KB) Miss Rate Estimations of the *Simple* Method

GPU Kernels	1	2	3	4	5	6	7	8
Actual Number of Misses	304	608	643	2311	521	2084	3179	24816
Estimated Number of Misses	372	737	2330	12499	1641	7610	15514	55524
Total Number of Access	670	1292	3228	12499	2609	11363	15514	55524
Actual Miss Rate	45.4%	47.1%	19.9%	18.5%	20.0%	18.3%	20.5%	44.7%
Estimated Miss Rate	55.5%	57.0%	72.2%	100.0%	62.9%	67.0%	100.0%	100.0%

Table 6 shows the miss rate estimation results using the *Simple* method. These results are from 8 GPU kernel benchmarks running on the gem5-gpu simulator with a shared data LLC of 512 KB. The cache line size is 128B and the associativity is 32. The simulator is configured to have 15 GPU SMs (Streaming Multiprocessors) and 1 CPU core. The results show that, except for the first two GPU kernels, the overestimation in the miss rate is very high. This is because all the accesses from other cores are considered as possible conflicting accesses in estimating the reuse distance. We also find that the first two benchmarks have less overestimation because they have much less total numbers of accesses than the others.

6.4.3 Access Interval Based Shared Data LLC Analysis Method

Although the *Simple* method introduced in Section6.4.2 is straightforward and easy to implement, the overestimation can be very high. The major reason is that too many accesses from other cores are considered as possible conflicts. Based on the comparison between the results of first two kernels and the others, the results indicate

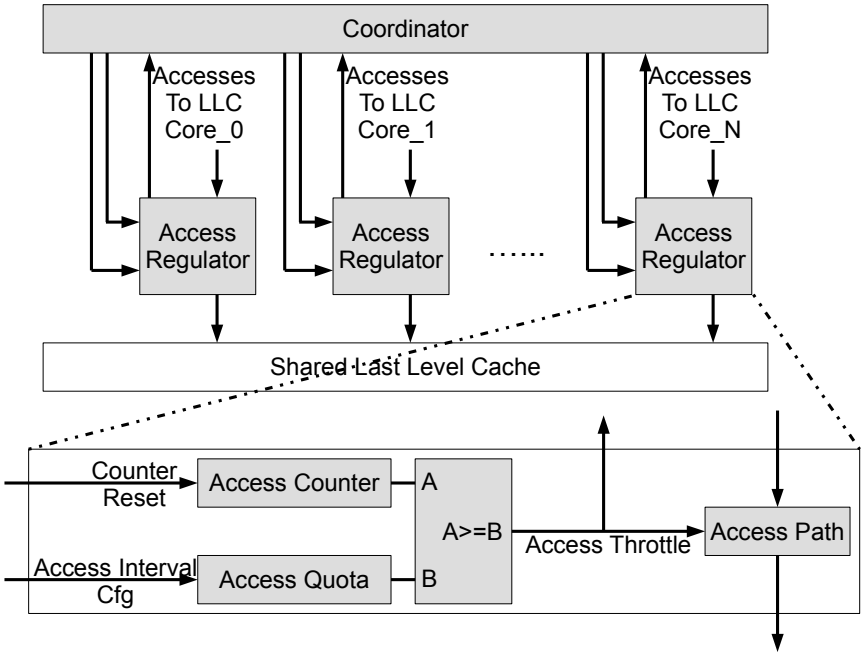


Fig. 32. Architectural Extensions for Access Interval Regulation

that limiting the number of total accesses in reuse distance and hit/miss estimation may help to reduce the overestimation.

Due to the large number of GPU SMs in the integrated architecture, the number of possible conflicting LLC accesses can be significantly overestimated. To address this problem, we propose the *Access Interval* based analysis method to enable tighter WCET analysis of the data LLC in the integrated CPU-GPU without significant impact on the average-case performance. Some architectural extensions are needed in this *Access Interval* based method, as shown in Fig. 32. Specifically, each core in the system will be assigned with a quota of the number of accesses that this core is allowed to send to the shared LLC during each access interval. If the quota is reached, the path of sending accesses to the shared LLC is throttled. When all the active cores have reached the quota, the coordinator resets the access counter and the next interval begins.

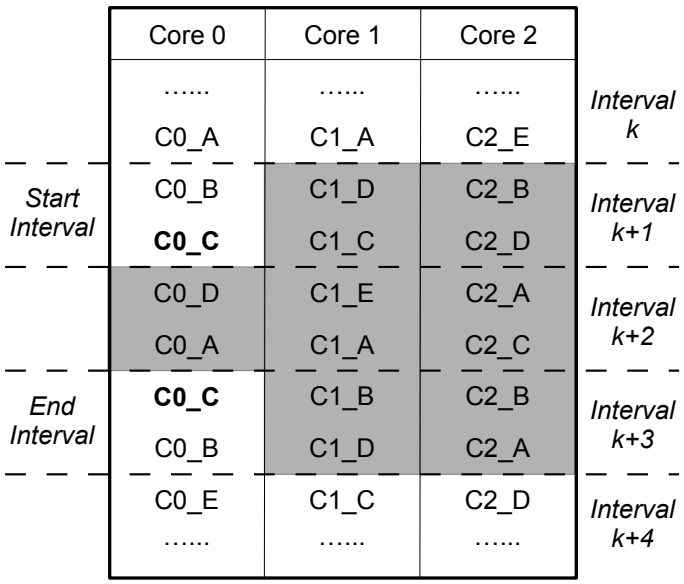


Fig. 33. Example of Access Interval Based Method

Fig. 33 shows a simple example to illustrate the access interval based method. In this example, the quota of each access interval is set to 2 accesses. Then, for the estimation of the access $C0\_C$ in the interval *k+3*, the interval that this access belongs to is set as the *End Interval*. The interval that has the latest previous access to the same cache line is set as the *Start Interval*, e.g. interval *k+1* in this example. Then the possible conflicting accesses are the accesses from the *Start Interval* and *End Interval* from all the cores, except (1) the accesses from the core that has the latest previous access and that are also earlier than the latest previous access in the *Start Interval* and (2) the accesses from the core that has the access under analysis and that are also later than the access under analysis in the *End Interval*. In this example they are the accesses in the gray area.

It should be noted that the latest previous access to the same cache line can be from other cores, and the *Start Interval* should be set accordingly, as shown in Fig. 34. This example assumes that the access $C1\_E$ is the latest previous access of the

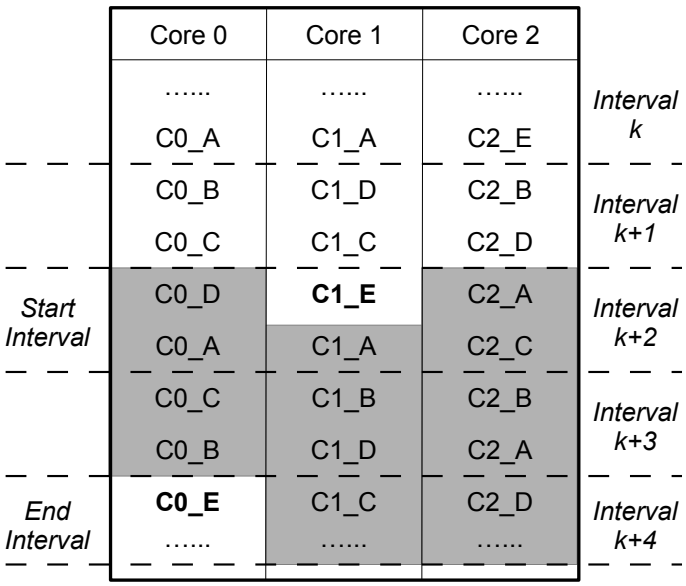


Fig. 34. Example of Access Interval Based Method

access $C0\_E$. Then the possible conflicting accesses are as shown in the gray area in the figure.

The comparison between Fig. 31, 33 and 34 shows that the number of possible conflicting accesses is largely reduced by the *Access Interval* based method. Therefore, this *Access Interval* based method is likely to lead to a much tighter WCET estimation for the data LLC of the integrated CPU-GPU. Also, since different SMs execute the same GPU kernel code and, thus, generally have similar access patterns to the memory system (e.g. when memory access happens along the kernel execution), the overhead introduced by this *Access Interval* based method is expected to be small, i.e. it does not significantly impact the performance of the system, as shown by the evaluation results.

6.5 WCET Analysis of GPU Kernels with Shared Data LLC Estimation Results

With the shared data LLC analysis method based on the *Access Interval* technique that is proposed in Section 6.4, the WCET timing model in Chapter 5 can be improved to analyze a GPU system with L1 and L2 data caches, which is a more realistic system compared to the system assumption without caches in Chapter 5. The $LE_{inst_{Memory}}$ in Equation 5.6 represents the latency to access the memory system. Under the assumption that there is no L1 or L2 data cache, the $LE_{inst_{Memory}}$ needs to cover the latency of accessing the off-chip memory (L_{base}) and the stall latency caused by the interconnection of the network-on-chip (NoC) for every instruction. However, based on the hit/miss prediction results, the $LE_{inst_{Memory}}$ value of each memory instruction can be set with different values, according to whether it is predicted to be a hit or miss. Specifically, in the memory system with L1 and L2 data caches, the value of $LE_{inst_{Memory}}$ can be set to the latency of a L1 hit, a L2 hit or a L2 miss. It should be noted that, besides the latencies of accessing the different levels of caches, there is still some latencies caused by the NoC in the system. However, since the focus of this work is the shared data LLC, it is assumed that the latency of the NoC is known, which will be explained in details in Section 6.6.1. It also should be noted that the other parts of this timing model is not affected by the integration of the cache hit/miss estimations.

6.6 Evaluation Results

6.6.1 Experimental Methodology

6.6.1.1 Simulator

As mentioned in Section 6.4.1, the gem5-gpu [16] simulator is used to implement and evaluate the proposed methods. The gem5-gpu simulator integrates the simulators of GPGPU-Sim [15], which simulates the GPU cores and executes the GPU kernels, and the gem5 [79], which simulates the CPU cores, executes the CPU code and launches the GPU kernels to the GPGPU-Sim simulator.

Table 7. Configurations of the gem5-gpu Simulator

Number of SMs	15
Number of CPU Cores	1
GPU SM Clock Cycle	500 Ticks
CPU Core Clock Cycle	500 Ticks
L1 Data Cache Size	64KB
L1 Cache Line Size	128B
L1 Cache Associativity	4
L2 Data Cache Size	256KB/512KB/1024KB
L2 Cache Line Size	128B
L2 Cache Associativity	32
L1/L2 Cache Replacement Policy	LRU
GPU Warp Size	32
GPU Warp Scheduling Policy	Pure Round-Robin
Max Number of Active Warps	48
Max Number of Active Blocks	8

Table 7 shows some of the basic configuration values of the gem5-gpu simulator. Since the focus of this work is the analysis of the shared data LLC and its impact on GPU kernel analysis, the CPU part in the system is relatively simple with 1 CPU core, while there are 15 GPU SMs. The periods of one clock cycle for the GPU SM and GPU core are set to 500 ticks. One tick is the basic cycle at which the whole simulator cycles. There is an L1 data cache for each GPU SM and CPU core, with the

size, the cache line size and associativity as shown in the table. There are separate instruction caches, which are modified and configured as perfect caches (as this work focuses on analyzing the data LLC). All the caches use the LRU replacement policy. To enable the static timing analysis, the Pure Round-Robin warp scheduling policy is used. The other basic configurations for the GPU SMs are shown in the rest of the table, which basically follows the configuration for the Fermi architecture [17] in the GPGPU-Sim simulator. Although the simulated GPU architecture is an early version of the CUDA architecture and does not have the recent advanced architectural features, such as dynamic parallelism, the simulated architecture has the fundamental GPGPU architecture components, which are the major architectural parts in every architecture version and critical parts for applying GPGPU to real-time computing.

6.6.1.2 Benchmarks

The GPU kernels used in the evaluations are from the Rodinia [31] benchmark suite. Table 8 shows the names of the GPU kernel benchmarks and the sizes of the inputs to the kernels. The names *k1-10* are used in Section 6.6.2 to refer to these benchmarks. It should be noted that, although the Rodinia benchmark suite is not one that is specially collected and set for the real-timing computing, there is not any GPGPU real-time benchmark available that is shared and used publicly by researchers. Therefore, the Rodinia benchmark suite, though being a GPGPU benchmark suite, is used in this work, since it is already widely used in GPGPU related studies and can represent some characteristics of the real-time GPGPU computing applications.

With the access interval method, extra delays can be introduced in accessing the LLC, which can lead to performance overhead. To measure this, each benchmark is executed with out the access interval regulation first to get the baseline performance

results. Then, with the access interval enabled, each benchmark is executed again to get the performance results with possible performance overhead and the results of the actual miss rate in the shared data LLC, which the estimated LLC miss rate is compared with.

Table 8. Benchmarks

	Benchmark Name	Input Size
k1	cfd1	4096
k2	cfd2	4096
k3	gaussian	128
k4	gaussian	256
k5	lud	128
k6	lud	256
k7	nw	1024
k8	nw	2048
k9	srad	128
k10	srad	256

6.6.1.3 Assumptions

Since the focus of this work is the analysis of the shared data LLC in the GPU-CPU system and its impact on the GPU kernel WCET estimation, the following assumptions are made.

Instruction Caches. To separate the impact of the accesses for instruction contents from the memory system. Each SM and CPU core has its own ideal L1 instruction cache, the accesses to which always result in hits. Therefore, only data accesses go to the shared LLC in the memory system.

L1 Data Cache Miss Rate. The hit and miss results of the L1 data caches in the system are assumed to be known by profiling method, so that the sequences of the accesses to the LLC from each SM or core can be generated for the LLC analysis.

Memory Accesses From CPU. For the memory accesses from CPU cores, profiling is used to get the sequence of memory accesses to the LLC as well as the information of which access interval a certain memory access belongs to.

Memory Access Latencies. Modeling the latencies of accessing different levels in the memory system is not part of this work, therefore the profiling method is used to get the longest latencies of L1 hit, L2 (LLC) hit and miss.

6.6.2 Experiment Results

6.6.2.1 Shared Data LLC Miss Rate Estimation Results

Fig. 35. Miss Rate Estimation Results of a 512KB LLC

Fig. 35 shows the actual and estimated miss rate results of a 512KB LLC. The results show that, for different actual miss rates across the benchmarks, the proposed estimation method can provide a safe upper bound, among which only *k6* has relatively higher overestimation.

Fig. 36 shows the actual and estimated miss rate results of 3 different LLC sizes,

including 256KB, 512KB and 1024KB. As shown in the figure, for most of the kernels the overestimation reduces as the LLC size increases. For example, the overestimation in *k3* reduces from over 100% with 256KB LLC to less than 1% with 1024KB LLC. This is because that a larger LLC has more cache sets and hence the number of possible conflicting accesses that are mapped to the same set is reduced, which leads to a tighter estimation of reuse distance values.

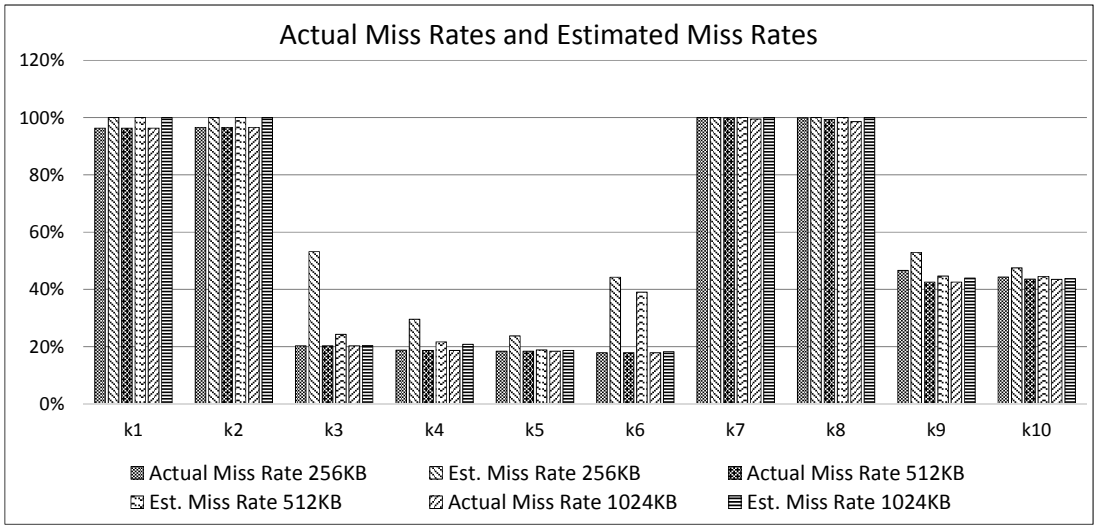


Fig. 36. Miss Rate Estimation Results of Different LLC Sizes

Fig. 37 shows the normalized performance results of the benchmarks with 3 different LLC sizes. The results are the execution cycles of the GPU kernel benchmarks with the access interval normalized to the execution cycles without the access interval regulations. The performance overhead in *k6* is higher than the others, because synchronizations are used in this kernel, together with which the access interval regulations lead to longer delays for warps to reach the synchronization barriers. As shown in the figure, the average performance overhead is less than 8%, which is not prohibitive considering the benefit of much tighter timing analysis. The average results are the geometric means of the results of the benchmarks.

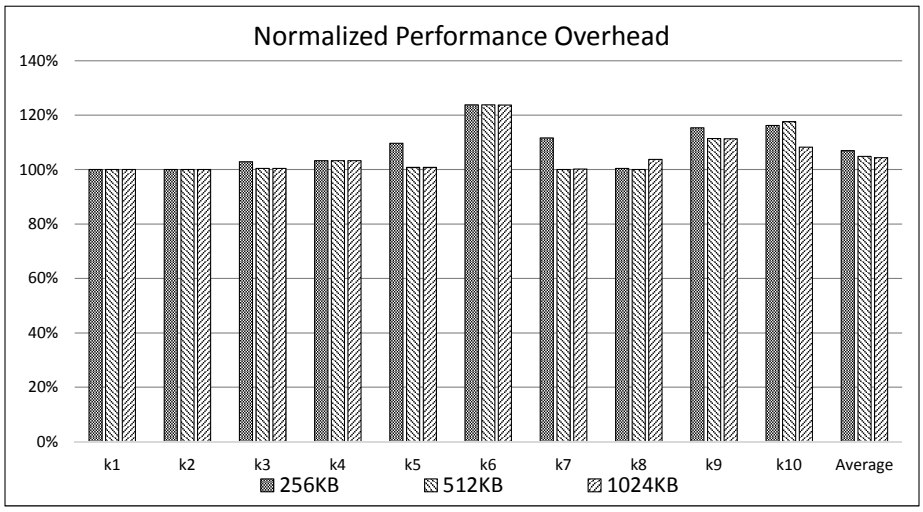


Fig. 37. Normalized Access Interval Method Performance Results of Different LLC Sizes

6.6.2.2 WCET Estimation Results of GPU Kernels

Fig. 38 shows the normalized performance results of the benchmarks with different shared data LLC caches. The numbers of execution cycles are normalized to those with a 256KB LLC for each benchmark. As shown in the figure, some benchmarks benefit from larger cache sizes, while some don't. Part of the reason is that for some benchmarks, larger LLC does not necessarily result in lower miss rate. For those that have smaller LLC miss rate with larger LLC sizes, e.g. *k7, k8 and k9*, the performance is well improved.

Fig. 39 shows the normalized performance and WCET estimation results with different shard LLC sizes. The performance and estimation results in the figure are normalized to the actual performance results with a 256KB shared data LLC for each benchmark. The results show that high overestimation in the LLC miss rate can result in high overestimated WCET result, such as *k6* with more than 140% in overestimation of LLC miss rate and more than 35% overestimation in WCET with

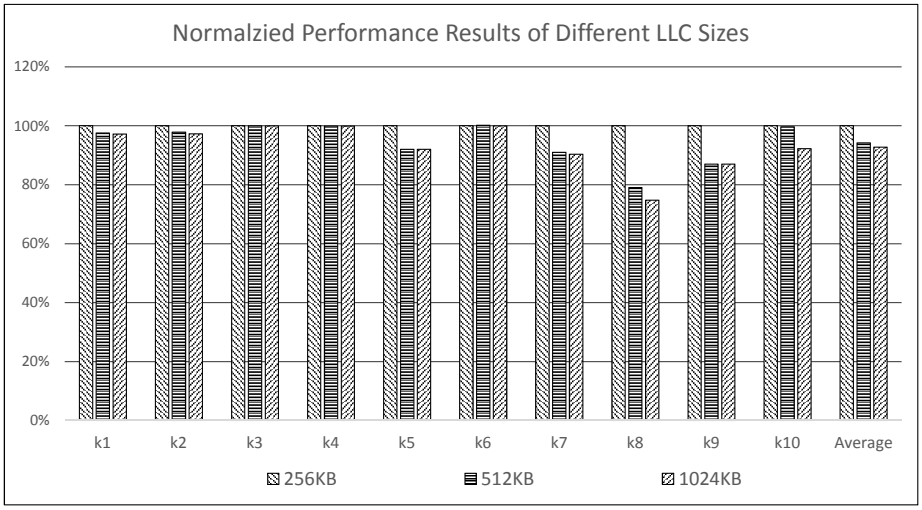


Fig. 38. Normalized Performance Results of Different LLC Sizes

a 256KB LLC.

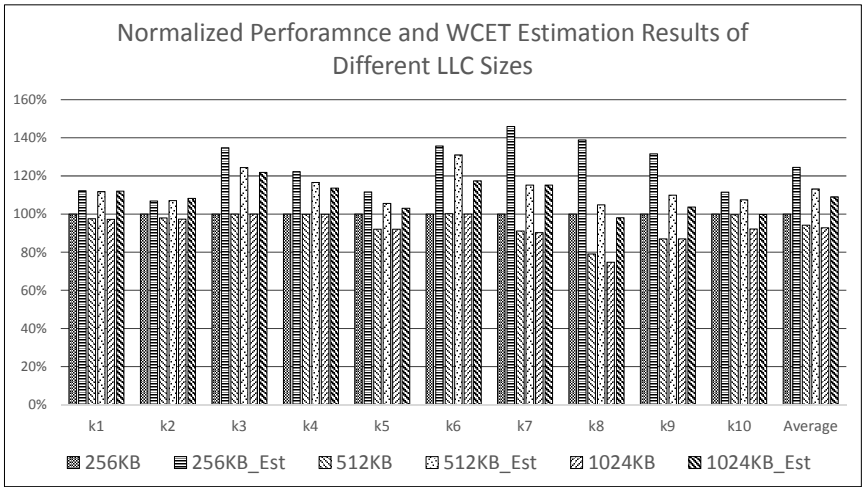


Fig. 39. Normalized WCET Estimation Results of Different LLC Sizes

It should be noted that the overestimation is also related to the ratio between the maximum and the average latencies of accessing different levels of the memory system. For example, although the overestimation of the LLC miss rate is very low for benchmark *k7* and *k8* as shown in Fig. 36, the overestimation in WCET is high (35%

to 40%). This is because the ratio between the maximum and average latencies in accessing the off-chip memory is around 2.5 for these two benchmarks while this ratio is below 1.5 for other benchmarks, and the WCET analyzer has to use the maximum latency for every access in the estimation.

CHAPTER 7

CONCLUSIONS

GPUs are no longer just used as accelerators for graphics computing. The parallel computing capability and high throughput of GPUs have put them into all kinds of general-purpose computing applications. Such potentials of GPUs make it promising to apply GPUs to real-time applications as well, where, however, good time-predictability characteristic is critical for the purposes of safety and reliability of the system. Nevertheless, the current architectural features in GPUs are designed for the improvement of average-case performance, rather than time-predictability. Therefore, the research topics in this dissertation focus the analysis and improvement of GPUs, so that they can be safely applied to real-time applications.

7.1 Profiling-Based GPU L1 Data Cache Bypassing

Cache memories are widely used to decrease the performance gap between processor/core and physical memory. The different programming and execution models in GPUs, however, generate different access patterns to the cache memory in GPUs. Specifically, memory accessing by coalescing different requests from different warps leads to the issue of memory access and traffic efficiency. The profiling results of some GPU kernel benchmarks show that, in GPGPU applications/kernels, there can be a large percentage of data that is never reused or only reused for very few times. It is also shown that sometimes not all the data in a coalesced memory transfer is useful (low utilization rate), which leads to unnecessary memory traffic. Based on such observations, a profiling-based method is proposed to identify the memory accesses with

low utilization rate and a small number of reuse times and to let such accesses bypass the GPU L1 data cache. The experiment results show that the proposed method can effectively reduce the memory traffic and improve the performance, which indicates that the proposed method can use GPU L1 data caches in a more effective way.

7.2 Warp-Based Load/Store Reordering for Time-Predictability Improvement

Cache memories are known as harmful to the time-predictability characteristic of a system. Furthermore, the dynamic behaviors, such as dynamic warp scheduling, can make the behavior of a GPU kernel even harder to analyze statically. On the other hand, GPUs, however, rely these dynamic behaviors to achieve high performance and throughput. Putting too many constraints can introduce performance overhead. Therefore, in this work, the load/store reordering framework is implemented, to regulate the order of memory requests before they reach the L1 data cache, while allowing the dynamic warp scheduling inside each SM. The experiment results show that the proposed reordering framework can give accurate miss rate estimations for GPU L1 data cache, while the performance overhead introduced by the reordering framework is very small.

The works on the first two topics show that cache memories are still desirable in GPU architecture for real-time applications. However, special efforts, from either architecture side or compiler size or both, are needed to make sure that the usage of cache memories can benefit the performance and have good time-predictability characteristic.

7.3 Static WCET Analysis Timing Model for GPUs

To apply GPUs in real-time system and applications, it is important to have a WCET performance model for the GPU architecture and kernels. Therefore, the time-predictability of GPU architecture needs to be improved to be more analyzable. To address this, the pure round-robin scheduling policy that has predictable behaviors is chosen as the warp scheduling policy, based on which a WCET timing model for GPU kernels is built. To the best of our knowledge, this timing model is the first one that estimates the WCET for GPU kernels at the instruction scheduling level, by analyzing the details of GPU architecture. The experimental results show that our WCET analyzer can effectively provide WCET estimations for both soft and hard real-time application purposes.

7.4 Static WCET Analysis on Shared Data LLC in CPU-GPU Architectures

The integrated CPU-GPU architectures can take the advantages of tightly-coupled CPUs and GPUs to further boost performance. In such architecture, the shared LLC is an important architectural component for performance improvement and a key source of time-unpredictability as well. Since different cores access the shared LLC simultaneously, the run-time behavior of the shared LLC is hard, if not impossible, to predict statically, In this work, a technique of regulating the accesses to the shared LLC by enforcing access intervals is proposed to improve the time-predictability of the LLC. The results show that the proposed technique can significantly reduce the over-estimation in the miss rates of the shared data LLC, without significantly impacting the average-case performance.

The works in the last two topics show the techniques that can enable the WCET

analysis on GPU and integrated CPU-GPU architectures and allow such architectures to be applied in real-time applications and systems. The proposed WCET timing model and shared data LLC analysis method can give tight worst case estimations, indicating that, with architectural modifications and extensions, the time-predictability of GPU and integrated CPU-GPU architectures can be improved and applied to real-time applications safely.

7.5 Future Work

Besides the key architectural components that are studied in the previous four topics, the improvement of time-predictability of some other components in the GPU and CPU-GPU architectures can also be studied. For example, the time-predictability of the Network-on-Chip (NoC), which connects the different processing units and memory components, may be analyzed and improved with some new topology or accessing and scheduling polices. Also, new warp scheduling policies may be studied to find a good balance between time-predictability and performance.

Furthermore, the new features in the recent GPU architectures bring new problems that can become the focuses of some future researches as well. For instance, the Dynamic Parallelism[80] in CUDA programming model allows a CUDA kernel to launch a child CUDA kernel, while the High Bandwidth Memory (HBM) technique is used to achieve high-performance RAM interfacing[81]. In the version of CUDA 9, the Cooperative Groups technique is introduced for the CUDA programming model to have the ability to do collective operations, such as synchronization, at sub-block or multi-block level[82]. According to these new features, additional studies may be done to analyze their impacts on the time-predictability of the system and some approaches to improve it.

REFERENCES

[1] J. D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.917757.

[2] ARM Corp. "Mali Graphics Hardware". In: (Accessed on Oct 2016). URL: http://www.arm.com/products/graphics-and-multimedia/mali-gpu.

[3] NVIDIA Corp. "Nvidia Tegra Mobile Processors". In: (Accessed on Oct 2016). URL: http://www.nvidia.com/object/tegra.html.

[4] NVIDIA Corp. "Nvidia DRIVE PX 2". In: (Accessed on Oct 2016). URL: http://www.nvidia.com/object/drive-px.html.

[5] Uri Verner, Assaf Schuster, and Mark Silberstein. "Processing data streams with hard real-time constraints on heterogeneous systems". In: (2011), p. 120. DOI: 10.1145/1995896.1995915.

[6] Glenn A Elliott and James H Anderson. "Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs". In: (2012), pp. 267–276. DOI: 10.1109/ECRTS.2012.20.

[7] A. Kurdila et al. "Vision-based control of micro-air-vehicles: progress and problems in estimation". In: *Decision and Control, 2004. CDC. 43rd IEEE Conference on.* Vol. 2. Dec. 2004, 1635–1642 Vol.2. DOI: 10.1109/CDC.2004.1430279.

[8] Jamie Shotton et al. "Real-time human pose recognition in parts from single depth images". In: (2011), pp. 1297–1304. ISSN: 0001-0782. DOI: 10.1109/CVPR.2011.5995316.

[9] Anuj Pathania et al. “Integrated CPU-GPU Power Management for 3D Mobile Games”. In: *Proceedings of the 51st Annual Design Automation Conference*. DAC ’14. San Francisco, CA, USA: ACM, 2014, 40:1–40:6. ISBN: 978-1-4503-2730-5. DOI: 10.1145/2593069.2593151. URL: http://doi.acm.org/10.1145/2593069.2593151.

[10] A. Bernhardt et al. “Real-Time Terrain Modeling Using CPU-GPU Coupled Computation”. In: *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*. Aug. 2011, pp. 64–71. DOI: 10.1109/SIBGRAPI.2011.28.

[11] Cheng KT. Wang YC. Donyanavard B. “Energy-Aware Real-Time Face Recognition System on Mobile CPU-GPU Platform”. In: *Trends and Topics in Computer Vision*. 2012, pp. 411–422. DOI: 10.1007/978-3-642-35740-4_32.

[12] Sfffdbastien Roujol et al. “Online real-time reconstruction of adaptive TSENSE with commodity CPU/GPU hardware”. In: *Magnetic Resonance in Medicine* 62.6 (2009), pp. 1658–1664. ISSN: 1522-2594. DOI: 10.1002/mrm.22112. URL: http://dx.doi.org/10.1002/mrm.22112.

[13] Y. Huangfu and W. Zhang. “Real-Time GPU Computing: Cache or No Cache?” In: *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. Apr. 2015, pp. 182–189. DOI: 10.1109/ISORC.2015.12.

[14] Y. Huangfu and W. Zhang. “Warp-Based Load/Store Reordering to Improve GPU Data Cache Time Predictability and Performance”. In: *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. May 2016, pp. 166–173. DOI: 10.1109/ISORC.2016.31.

[15] Ali Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: (2009), pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[16] Jason Power et al. “gem5-gpu: A Heterogeneous CPU-GPU Simulator”. In: *Computer Architecture Letters* 13.1 (Jan. 2014). ISSN: 1556-6056. DOI: 10.1109/LCA.2014.2299539. URL: http://gem5-gpu.cs.wisc.edu.

[17] NVIDIA Corp. “NVIDIAs Next Generation CUDA Compute Architecture Fermi”. In: (Accessed on Oct 2016). URL: $http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf$.

[18] NVIDIA Corp. “CUDA Programming Guide”. In: (Accessed on Oct 2016).

[19] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Comput Sci Eng* 12.3 (2010), pp. 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69.

[20] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. “MRPB: Memory request prioritization for massively parallel processors”. In: (2014), pp. 272–283. DOI: 10.1109/HPCA.2014.6835938.

[21] Antonio González, Carlos Aliagas, and Mateo Valero. “A data cache with multiple caching strategies tuned to different types of locality”. In: (1995), pp. 338–347. DOI: 10.1145/224538.224622.

[22] G Tyson et al. “A modified approach to data cache management”. In: (), pp. 93–103. DOI: 10.1109/MICRO.1995.476816.

[23] TL Johnson et al. “Run-time cache bypassing”. In: *Ieee T Comput* 48.12 (1999), pp. 1338–1354. ISSN: 0018-9340. DOI: 10.1109/12.817393.

[24] Haiming Liu et al. “Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency”. In: (2008), pp. 222–233. DOI: 10.1109/MICRO.2008.4771793.

[25] M Kharbutli and Yan Solihin. "Counter-Based Cache Replacement and Bypassing Algorithms". In: *Ieee T Comput* 57.4 (2008), pp. 433–447. ISSN: 0018-9340. DOI: 10.1109/TC.2007.70816.

[26] Youfeng Wu et al. "Compiler managed micro-cache bypassing for high performance EPIC processors". In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* 2002, pp. 134–145. DOI: 10.1109/MICRO.2002.1176245.

[27] Zhenlin Wang et al. "Using the compiler to improve cache replacement decisions". In: *Proceedings.International Conference on Parallel Architectures and Compilation Techniques.* 2002, pp. 199–208. DOI: 10.1109/PACT.2002.1106018.

[28] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. "Characterizing and improving the use of demand-fetched caches in GPUs". In: (2012), p. 15. DOI: 10.1145/2304576.2304582.

[29] Vineeth Mekkat et al. "Managing Shared Last-level Cache in a Heterogeneous Multicore Processor". In: *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques.* PACT '13. Edinburgh, Scotland, UK: IEEE Press, 2013, pp. 225–234. ISBN: 978-1-4799-1021-2. URL: http://dl.acm.org/citation.cfm?id=2523721.2523753.

[30] Yun Liang et al. "An Efficient Compiler Framework for Cache Bypassing on GPUs". In: *Ieee T Comput Aid D* 34.10 (2015), pp. 1677–1690. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2424962.

[31] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC).* Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[32] Gurulingesh Raravi, Björn Andersson, and Konstantinos Bletsas. "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors". In: *Real-time Syst* 49.1 (2013), pp. 29–72. ISSN: 0922-6443. DOI: 10.1007/s11241-012-9161-1.

[33] Glenn A Elliott and James H Anderson. "Globally scheduled real-time multiprocessor systems with GPUs". In: *Real-time Syst* 48.1 (2012), pp. 34–74. ISSN: 0922-6443. DOI: 10.1007/s11241-011-9140-y.

[34] G. A. Elliott, B. C. Ward, and J. H. Anderson. "GPUSync: A Framework for Real-Time GPU Management". In: *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. Dec. 2013, pp. 33–44. DOI: 10.1109/RTSS.2013.12.

[35] Xavier Vera, Björn Lisper, and Jingling Xue. "Data cache locking for higher program predictability". In: *Acm Sigmetrics Perform Eval Rev* 31.1 (2003), p. 272. ISSN: 0163-5999. DOI: 10.1145/885651.781062.

[36] Vivy Suhendra and Tulika Mitra. "Exploring locking & partitioning for predictable shared caches on multi-cores". In: (2008), p. 300. DOI: 10.1145/1391469.1391545.

[37] Huping Ding, Yun Liang, and Tulika Mitra. "WCET-centric partial instruction cache locking". In: (2012), p. 412. DOI: 10.1145/2228360.2228434.

[38] R Banakar et al. "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems". In: (), pp. 73–78. DOI: 10.1109/CODES.2002.1003604.

[39] Martin Schoeberl. "A Time Predictable Instruction Cache for a Java Processor". In: *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops: OTM Confederated International Workshops and Posters, GADA,*

JTRES, MIOS, WORM, WOSE, PhDS, and INTEROP 2004, Agia Napa, Cyprus, October 25-29, 2004. Proceedings. Ed. by Robert Meersman, Zahir Tari, and Angelo Corsaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–382. ISBN: 978-3-540-30470-8. DOI: 10.1007/978-3-540-30470-8_52. URL: http://dx.doi.org/10.1007/978-3-540-30470-8_52.

[40] D. Hardy and I. Puaut. "WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches". In: *Real-Time Systems Symposium, 2008*. Nov. 2008, pp. 456–466. DOI: 10.1109/RTSS.2008.10.

[41] Jun Yan and Wei Zhang. "WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches". In: (2008), pp. 80–89. DOI: 10.1109/RTAS.2008.6.

[42] Yun Liang et al. "Timing analysis of concurrent programs running on shared cache multi-cores". In: *Real-time Syst* 48.6 (2012), pp. 638–680. ISSN: 0922-6443. DOI: 10.1007/s11241-012-9160-2.

[43] B. K. Huynh, L. Ju, and A. Roychoudhury. "Scope-Aware Data Cache Analysis for WCET Estimation". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2011, pp. 203–212. DOI: 10.1109/RTAS.2011.27.

[44] Christian Ferdinand et al. "Cache behavior prediction by abstract interpretation". In: *Sci Comput Program* 35.2-3 (1999), pp. 163–189. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(99)00010-6.

[45] Jaewoong Sim et al. "A performance analysis framework for identifying potential benefits in GPGPU applications". In: (2012), p. 11. ISSN: 0362-1340. DOI: 10.1145/2145816.2145819.

[46] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. "Improving GPU Performance Prediction with Data Transfer Modeling". In: (2013), pp. 1097–1106. DOI: 10.1109/IPDPSW.2013.236.

[47] Yun Liang et al. "An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization". In: *Ieee T Comput Aid D* 35.7 (2016), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2501303.

[48] Yuki Abe et al. "Power and Performance Analysis of GPU-Accelerated Systems". In: *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*. Hollywood, CA: USENIX, 2012. URL: https://www.usenix.org/conference/hotpower12/workshop-program/presentation/Abe.

[49] Krzysztof Rojek, Lukasz Szustak, and Roman Wyrzykowski. *Performance Analysis for Stencil-Based 3D MPDATA Algorithm on GPU Architecture*. Vol. 8384. springer, 2014. ISBN: 9783642552236. DOI: 10.1007/978-3-642-55224-3_15.

[50] Christian Feichtinger et al. "Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU-GPU clusters". In: *Parallel Comput* 46 (2015), pp. 1–13. ISSN: 0167-8191. DOI: 10.1016/j.parco.2014.12.003.

[51] Adwait Jog et al. "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance". In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 395–406. ISSN: 0362-1340. DOI: 10.1145/2499368.2451158. URL: http://doi.acm.org/10.1145/2499368.2451158.

[52] Veynu Narasiman et al. "Improving GPU performance via large warps and two-level warp scheduling". In: (2011), p. 308. DOI: 10.1145/2155620.2155656.

[53] Yijie Huangfu and Wei Zhang. "Warp-Based Load/Store Reordering to Improve GPU Data Cache Time Predictability and Performance". In: (2016), pp. 166–173. DOI: 10.1109/ISORC.2016.31.

[54] Adam Betts and Alastair Donaldson. "Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis". In: (2013), pp. 193–202. DOI: 10.1109/ECRTS.2013.29.

[55] Kostiantyn Berezovskyi et al. "WCET Measurement-based and Extreme Value Theory Characterisation of CUDA Kernels". In: (2014), pp. 279–288. DOI: 10.1145/2659787.2659827.

[56] Wilson W. L. Fung et al. "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware". In: *ACM Trans. Archit. Code Optim.* 6.2 (July 2009), 7:1–7:37. ISSN: 1544-3566. DOI: 10.1145/1543753.1543756. URL: http://doi.acm.org/10.1145/1543753.1543756.

[57] Green500. "The Green500 List". In: (Accessed on Oct 2016). URL: https://www.top500.org/green500/lists/2016/06/.

[58] Inc. Advanced Micro Devices. "AMD Athlon APUs". In: (Accessed on Oct 2016). URL: http://www.amd.com/en-us/products/processors/desktop/athlon.

[59] ARM Ltd. "big.LITTLE Technology". In: (Accessed on April 2017). URL: https://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[60] Lui Sha et al. "Real-Time Computing on Multicore Processors". In: *Computer* 49.9 (2016), pp. 69–77. ISSN: 0018-9162. DOI: doi.ieeecomputersociety.org/10.1109/MC.2016.271.

[61] R. Mancuso et al. “Real-time cache management framework for multi-core architectures”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. Apr. 2013, pp. 45–54. DOI: 10.1109/RTAS.2013.6531078.

[62] B. C. Ward et al. “Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms”. In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 157–167. DOI: 10.1109/ECRTS.2013.26.

[63] Marco Paolieri et al. “Hardware Support for WCET Analysis of Hard Real-time Multicore Systems”. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. Austin, TX, USA: ACM, 2009, pp. 57–68. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555764. URL: http://doi.acm.org/10.1145/1555754.1555764.

[64] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. “Modeling Shared Cache and Bus in Multi-cores for Timing Analysis”. In: *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*. SCOPES ’10. St. Goar, Germany: ACM, 2010, 6:1–6:10. ISBN: 978-1-4503-0084-1. DOI: 10.1145/1811212.1811220. URL: http://doi.acm.org/10.1145/1811212.1811220.

[65] Gregorio Bernabe, Javier Cuenca, and Domingo Gimenez. “Optimization Techniques for 3D-FWT on Systems with Manycore GPUs and Multicore CPUs”. In: *Procedia Computer Science* 18 (2013), pp. 319–328. ISSN: 1877-0509. DOI: 10.1016/j.procs.2013.05.195.

[66] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. “A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures”. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 57:1–57:20. ISSN: 1544-3566. DOI:

10.1145/2400682.2400716. URL: http://doi.acm.org/10.1145/2400682.2400716.

[67] Chao Yang et al. "A Peta-scalable CPU-GPU Algorithm for Global Atmospheric Simulations". In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 1–12. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442518. URL: http://doi.acm.org/10.1145/2442516.2442518.

[68] T. P. Stefanski. "Implementation of FDTD-compatible green's function on heterogeneous cpu-GPU parallel processing system". In: *Progress In Electromagnetics Research* 135 (2013), pp. 297–316. DOI: 10.2528/PIER12111702.

[69] Jacques A. Pienaar, Srimat Chakradhar, and Anand Raghunathan. "Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 24:1–24:12. ISBN: 978-1-4673-0804-5. URL: http://dl.acm.org/citation.cfm?id=2388996.2389029.

[70] Klaus Kofler et al. "An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA: ACM, 2013, pp. 149–160. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465007. URL: http://doi.acm.org/10.1145/2464996.2465007.

[71] W. Jiang and G. Agrawal. "MATE-CG: A Map Reduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters". In: *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. May 2012, pp. 644–655. DOI: 10.1109/IPDPS.2012.65.

[72] T. Odajima et al. “GPU/CPU Work Sharing with Parallel Language XcalableMP-dev for Parallelized Accelerated Computing”. In: *2012 41st International Conference on Parallel Processing Workshops*. Sept. 2012, pp. 97–106. DOI: 10.1109/ICPPW.2012.16.

[73] Kyle L. Spafford et al. “The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures”. In: *Proceedings of the 9th Conference on Computing Frontiers*. CF ’12. Cagliari, Italy: ACM, 2012, pp. 103–112. ISBN: 978-1-4503-1215-8. DOI: 10.1145/2212908.2212924. URL: http://doi.acm.org/10.1145/2212908.2212924.

[74] M. Daga, A. M. Aji, and W. c. Feng. “On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing”. In: *2011 Symposium on Application Accelerators in High-Performance Computing*. July 2011, pp. 141–149. DOI: 10.1109/SAAHPC.2011.29.

[75] Y. Ukidave et al. “Quantifying the energy efficiency of FFT on heterogeneous platforms”. In: *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. Apr. 2013, pp. 235–244. DOI: 10.1109/ISPASS.2013.6557174.

[76] Kristof Beyls and Erik H. DfffdHollander. “Reuse distance as a metric for cache behavior”. In: *IN PROCEEDINGS OF THE IASTED CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS*. 2001, pp. 617–662.

[77] Chen Ding and Yutao Zhong. “Predicting Whole-program Locality Through Reuse Distance Analysis”. In: *SIGPLAN Not.* 38.5 (May 2003), pp. 245–257. ISSN: 0362-1340. DOI: 10.1145/780822.781159. URL: http://doi.acm.org/10.1145/780822.781159.

[78] C. Nugteren et al. "A detailed GPU cache model based on reuse distance theory". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 37–48. DOI: 10.1109/HPCA.2014.6835955.

[79] Nathan Binkert et al. "The gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[80] NVIDIA Corp. "Dynamic Parallelism in CUDA". In: (Accessed on Apr 2017). URL: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf.

[81] NVIDIA Corp. "NVIDIA Tesla P100". In: (Accessed on Apr 2017). URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[82] NVIDIA Corp. "CUDA 9 Features Revealed: Volta, Cooperative Groups and More". In: (Accessed on May 2017). URL: https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/#more-7874.

Appendix A

PUBLICATION

1. Y. Huangfu, W. Zhang. Warp-Based Load/Store Reordering to Improve GPU Data Cache Time Predictability and Performance. IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC), 2016

2. Y. Huangfu, W. Zhang. Hardware-Based Performance Enhancement Guaranteed Caches. IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC), 2015

3. Y. Huangfu, W. Zhang. Boosting GPU Performance by Profiling-Based L1 Data Cache Bypassing. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015

4. Y. Huangfu, W. Zhang. Hardware-Based and Hybrid L1 Data Cache Bypassing to Improve GPU Performance. IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015

5. Y. Huangfu, W. Zhang. Profiling-based L1 data cache bypassing to improve GPU performance and energy efficiency. ACM SIGBED Review 12 (1). 2015

6. Y. Huangfu, W. Zhang. Real-Time GPU Computing: Cache or No Cache? IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC), 2015

7. Y. Huangfu, W. Zhang. A Real-Time Instruction Cache with High Average-Case Performance. IEEE 17th International Symposium on Real-Time Distributed Computing (ISORC), 2014

8. Y. Huangfu, W. Zhang. Worst-case performance guaranteed data cache. IEEE 33rd International Performance Computing and Communications Conference (IPCCC), 2014

9. Y. Huangfu, W. Zhang. Compiler-directed leakage energy reduction for instruction scratch-pad memories. 15th International Symposium on Quality Electronic Design (ISQED), 2014

10. Y. Huangfu, W. Zhang. PEG-C: Performance Enhancement Guaranteed Cache for Hard Real-Time Systems. IEEE Embedded Systems Letters 6 (2). 2014

11. Y. Huangfu, W. Zhang. Compiler-based approach to reducing leakage energy of instruction scratch-pad memories. IEEE 31st International Conference on Computer Design (ICCD), 2013