Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

CWLP: coordinated warp scheduling and locality-protected cache allocation on GPUs*

Yang ZHANG[†‡], Zuo-cheng XING, Cang LIU, Chuan TANG

National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology, Changsha 410073, China

[†]E-mail: zhangyang@nudt.edu.cn

Received Jan. 19, 2017; Revision accepted Aug. 29, 2017; Crosschecked Feb. 14, 2018

Abstract: As we approach the exascale era in supercomputing, designing a balanced computer system with a powerful computing ability and low power requirements has becoming increasingly important. The graphics processing unit (GPU) is an accelerator used widely in most of recent supercomputers. It adopts a large number of threads to hide a long latency with a high energy efficiency. In contrast to their powerful computing ability, GPUs have only a few megabytes of fast on-chip memory storage per streaming multiprocessor (SM). The GPU cache is inefficient due to a mismatch between the throughput-oriented execution model and cache hierarchy design. At the same time, current GPUs fail to handle burst-mode long-access latency due to GPU's poor warp scheduling method. Thus, benefits of GPU's high computing ability are reduced dramatically by the poor cache management and warp scheduling methods, which limit the system performance and energy efficiency. In this paper, we put forward a coordinated warp scheduling and locality-protected (CWLP) cache allocation scheme to make full use of data locality and hide latency. We first present a locality-protected cache allocation method based on the instruction program counter (LPC) to promote cache performance. Specifically, we use a PC-based locality detector to collect the reuse information of each cache line and employ a prioritised cache allocation unit (PCAU) which coordinates the data reuse information with the time-stamp information to evict the lines with the least reuse possibility. Moreover, the locality information is used by the warp scheduler to create an intelligent warp reordering scheme to capture locality and hide latency. Simulation results show that CWLP provides a speedup up to 19.8% and an average improvement of 8.8% over the baseline methods.

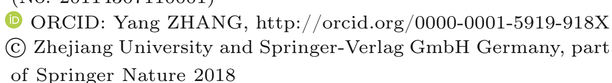
Key words: Locality; Graphics processing unit (GPU); Cache allocation; Warp scheduling
https://doi.org/10.1631/FITEE.1700059 **CLC number:** TP368.1

1 Introduction

Due to their high efficiency and easily-programming characteristics, graphics processing units (GPUs) have been a popular platform in industry and academia for accelerating various applications, ranging from the simplest matrix addition and transformations to popular large-scale deep learning algorithms. As a type of many-core processor, GPUs have thousands of processors. Different from the traditional multiprocessors, such as central processing units (CPUs), GPUs have few control units and many compute units. Due to their massive threads, they can hide most of the long latencies and gain a high throughput. However, a large number of threads introduce an enormous number of accesses to the on-chip memory. Since there are only a few megabytes of on-chip memory, many accesses

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61170083) and the Specialized Research Fund for the Doctoral Program of Higher Education, China (No. 20114307110001)



ORCID: Yang ZHANG, http://orcid.org/0000-0001-5919-918X

© Zhejiang University and Springer-Verlag GmbH Germany, part of Springer Nature 2018

easily cause cache contention and thrashing problems (NVIDIA, 2015). At the same time, for such a large number of accesses, traditional warp scheduling methods have difficulties in hiding long latencies.

Protecting data locality in the cache is an effective way to relieve the cache problem. There are a few ways to preserve data locality. One way is to tune the GPU programs, which is direct and effective but painstaking for programmers to optimise a highly tuned program. Therefore, this method has only a limited effect in practice. As a result, there is an urgent need to design an intelligent cache management strategy that is invisible to programmers, to alleviate these problems and promote higher performance.

Latency hiding is another way to improve performance. A good warp scheduling method can hide a long latency. A simple scheduler regardless of the locality information is not suitable for different kinds of programs. Thus, it is better to have a scheduler that uses the locality information from the cache for scheduling. Moreover, a locality information based warp scheduler should be simple enough to avoid a high overhead.

GPUs are built around an array of streaming multiprocessors (SMs) (Fig. 2). A multi-threaded program is partitioned into blocks of threads which execute independently from each other; thus, a GPU with more cores will automatically execute the program faster than one with fewer cores. Because most early applications are computation-intensive, and massive multi-threaded execution patterns can hide long access time to memory, the early generations of NVIDIA GPUs do not consider cache as one of the components for simplicity (NVIDIA, 2015). However, as the GPU architecture has been developed and the fields for its application have been extended, a generic on-chip memory, which is invisible for programmers, is imperative. This can preserve most of the locality. Therefore, later generations of GPUs introduce fast on-chip cache to improve the GPU performance (NVIDIA, 2009). From Fermi architectures, NVIDIA GPUs feature on-chip L1 cache and the programmers are able to turn on/off the L1 cache at the application or instruction level (Xie et al., 2013). However, in most cache-sensitive applications, L1 cache has important effects on single-thread performance, and will affect the GPU performance.

From the Kepler architecture onward, L1 cache has been used for only local access, and the global loads are cached in L2 cache. The poor efficiency of L1 cache is one of the main reasons for this change. Therefore, there is an urgent need to find better solutions in cache management policy for the next generation GPUs to relieve this problem. Since this study does not involve new features in the Kepler architecture and improvements in Kepler, and the following Maxwell architecture is related mainly to extending functional units and memory capacity (Harris, 2014), we consider the Fermi architecture for our research and the strategies applied to it can be popularised to other new architectures.

Recent studies focus mainly on reducing the access number to on-chip memory by bypassing and throttling (Rogers et al., 2013; Xie et al., 2013; Chen et al., 2014; Lee et al., 2014; Sethia et al., 2015; Xie et al., 2015, 2017). Bypassing is a method applied first to CPUs. It is used when the memory system detects that the memory is full of data locality and a new data insertion will destroy the data locality. The new data skips this memory level and accesses the lower memory. Throttling has been another research hotspot in recent years. It can relieve cache contention and thrashing by fundamentally decreasing the number of blocks on the cores. There are other studies on data locality (Rhu et al., 2013) and reuse distance in GPU (Nugteren et al., 2014). Rhu et al. (2013) retained the advantages of coarse-grained access for spatially and temporally local programs while permitting selective fine-grained access to memory. Nugteren et al. (2014) showed that the reuse distance theory can be used to model the GPU caches in detail by extending the theory. In conclusion, the latest research shows that cache optimisation plays a critical role in improving GPU performance.

However, it is still quite difficult to manage cache efficiently and to optimise warp scheduling. For cache-sensitive applications, an enormous number of memory accesses easily render the cache capacity insufficient, and this will cause a long latency. Therefore, we need to coordinate a cache management strategy with a thread scheduling method to preserve data locality and simultaneously hide the memory latency. In this paper, we propose a new strategy that uses a single locality collection unit to manage cache allocation and to optimise warp scheduling. The main contributions of this work are

as follows:

1. We analyze the data locality in L1 cache for two typical programs. We show that early eviction impairs the performance, and focus on a way to preserve locality to solve the problem.

2. We compare the program counter (PC) and the address based reuse detection methods, and illustrate why we choose the PC-based method.

3. We propose a novel locality detector called the 'locality-protected cache allocation method based on the instruction program counter (LPC)', and a prioritised cache allocation unit which evicts the cache line with lower reuse possibilities, by using the collected reuse information and time-stamp information.

4. A novel locality-based warp scheduler is proposed to use the reuse information from the locality detector to instruct the warp reordering scheme to preserve locality and hide latency at the same time.

5. We evaluate the coordinated warp scheduling and locality-protected (CWLP) cache allocation scheme on a simulated Fermi architecture and achieve an IPC improvement of up to 19.8% over the baseline scheme with a low overhead. We also inspect L1-cache performance for LPC, and analyze the relationship between L1_D-cache performance and instructions-per-cycle (IPC) performance. The results can be achieved for a variety of applications with different characteristics.

2 Motivation

Data locality has become increasingly important in designing high throughput and energy efficient GPUs (Nugteren et al., 2014). This reasoning relies on two aspects. First, early GPU applications occur mainly in image processing, which has special access patterns. Thus, GPUs are almost customised for these applications and have special types of on-chip memory such as texture and constant caches (Dally et al., 2003; Drew, 2008). However, modern generic applications have few streaming memory access patterns. Second, recent universal and irregular applications do not have enough parallelism, compared with image processing programs. As a result, preserving data locality becomes especially important for these applications, because it can reduce the number of redundant accesses to the global memory, resulting in lower data access time and fewer blocks.

Continuous reuse number (CRN) is the number of continuous memory accesses at a time to the same address. It is different from the reuse distance, which equals the number of the only addresses accessed between the current and the most recent previous accesses to the same address. Fig. 1 gives a clear distribution of CRNs for k-means (KMN) clustering and breadth first search (BFS) applications. CRN can be used as locality information. For a cache line, a larger CRN indicates that more localities exist in it. Fig. 1 shows that the number of memory accesses in L1_D-cache decreases as CRN increases. For BFS, the downward trend is more remarkable than that in KMN. Fig. 1 implies that, for some programs (such as BFS), most memory accesses have no or little locality.

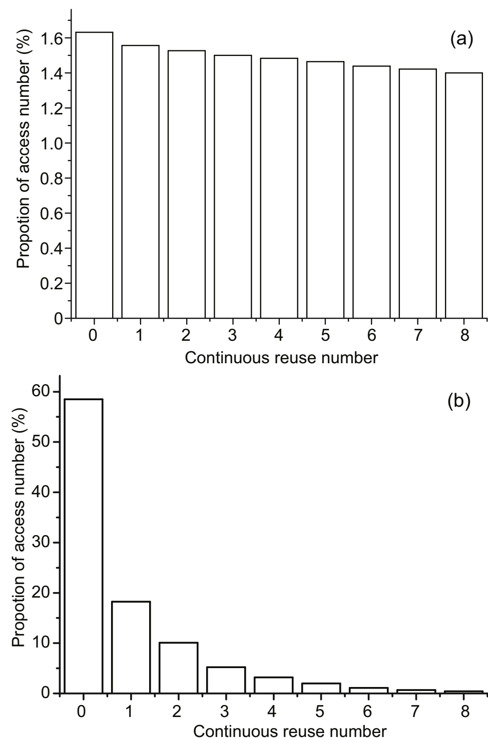


Fig. 1 Distributions of the reuse number for memory accesses in k-means (KMN) clustering (a) and breadth first search (BFS) (b)

Memory hiding is another important method for improving the GPU performance. In GPUs, multithreading is used to hide the long latency and to achieve a high throughput. A good warp scheduler can preserve locality and hide long latency at the same time. It gives issue priority to warps based on the characteristics of the executed instruction to achieve reordering. As reordering may destroy

locality, the reordering decision is based on locality information from the cache. Therefore, if there is a good locality in the cache, it will not perform reordering to preserve locality. Otherwise, it does perform reordering to hide latency.

3 Baseline architecture

As Fig. 2 shows, modern GPUs consist of multiple SMs or computation units (CUs), interconnects, memory partitions, and global memory. An SM executes instructions in the manner of single instruction multiple data (SIMD). At each cycle, an instruction is fetched and decoded in the front-end, and then it is dispatched to a warp (typically with 32 threads) pool which contains multiple warps to run the instruction. After the dispatch, the scoreboard will check for data hazards. If an instruction passes the scoreboard, it is qualified to be issued to the execution units. The instruction execution process is performed in multiple SIMD lanes, which constitute the hardware execution units. The lanes have many computing resources to process the data in parallel. Due to the sharing of the same fetch and the decoding stage for multiple instructions in a warp, considerable hardware expense is saved in GPU. Each SM is connected on chip to the L2 cache through interconnects. The L2 cache is shared by SMs and the data in L2 cache is transferred to DRAM through multiple data channels. The GPUs use mainly a multi-threaded execution model to achieve high throughput.

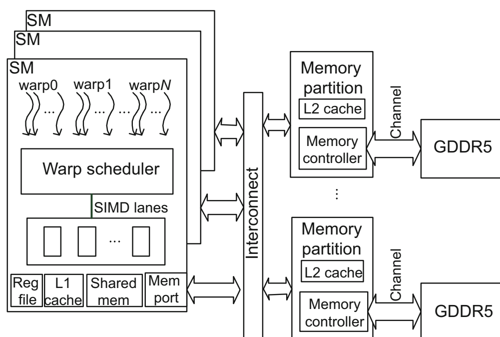


Fig. 2 Baseline architecture of our graphics processing unit (GPU)

SM: streaming multiprocessor; SIMD: single instruction multiple data; Reg: register; Mem: memory

The compute unified device architecture (CUDA) is the computing platform for the GPU architecture. In CUDA, a GPU kernel is offloaded onto the GPU for execution. The kernel executes instructions through massive threads, and the programmer can assign the number of threads. The numerous threads are organised hierarchically. A kernel has several thread blocks, and each block consists of many threads. The threads in one block run in an SIMT model. They can synchronise among themselves through barriers.

Each warp in an SM takes advantages of the coalesced access to global memory to enhance memory access efficiency. More clearly, multiple accesses in a warp to a single block can be coalesced into one to reduce the number of long access times to global memory. Furthermore, the address will be checked to see which chunk in the block will be accessed by the address. In this way, smaller chunks can be accessed instead of the entire cache block, reducing memory bandwidth.

4 Comparison of address and program counter based reuse information

To use data locality, GPUs need to detect locality first. There are two ways to collect reuse information. One is based on the instruction PC and the other is based on the address information. They each have different advantages and weaknesses, and are suitable for different kinds of processors.

In the traditional CPU, each instruction issues only one memory request. After the address is issued to the cache, the hardware circuit will compare the corresponding address segment with the cache tag to see whether the accessed data hits on the cache. If the request hits, the data in the block will be read or written. Otherwise, the request is sent to the lower level. The address-based reuse information is direct and simple. Fig. 3a shows CPU's memory access pattern.

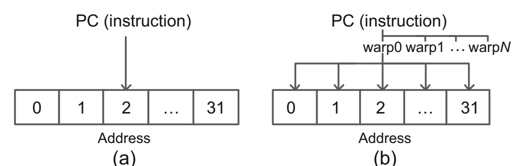


Fig. 3 Address access patterns for central processing unit (CPU) (a) and graphics processing unit (GPU) (b)

In the GPU architecture, the memory access procedure is almost the same as that in CPU, except

that each warp with an instruction can issue up to 32 memory addresses (Fig. 3b). The 32 accesses are not separated completely and have the same instruction PC. Although some of the 32 accesses can be coalesced by the coalescer, the access number is still large and will change with the program's features. If we adopt an address-based method to collect reuse information, the memory access process will be more complicated and will not be suited for GPU's single instruction multiple thread (SIMT) running pattern. The PC information for an application is more stable than the address information. The number of PCs will not change with the increase in the size of the input dataset (Zheng, 2014). Therefore, we use the PC-based method to collect reuse information to accommodate the characteristics of the GPU architecture. The locality value in a cache block increases when two accesses to the block have the same PC, and decreases when they do not. The detailed computation process for the locality is shown in Section 5.

As the coalescer merges the many memory accesses in a warp into fewer or even one, some intra-warp reuse information is hidden. The access number after the coalescer does not show the real memory access number of threads. We will consider the locality problem after the coalescence operation, and deem the coalescing problem beyond our consideration.

5 Coordinated warp scheduling with locality-protected cache allocation

In this section, we will introduce CWLP which uses locality information from the cache to instruct cache allocation and warp scheduling simultaneously. The locality-protected cache allocation and the warp scheduling constitute CWLP.

5.1 Instruction program counter: locality-protected method based on program counter information

In this section, we discuss implementation of the LPC allocation scheme. First, we propose a novel PC-based reuse information collector to obtain the locality information. Then, we demonstrate an arbitrator unit called 'PCAU', which consists of a predictor and an eviction unit to evict the cache block with the low reuse possibility. The architecture of LPC is based on the locality-protected method based on instruction PC (LPP) presented by Zhang et al. (2017); however, their implementation and hardware overhead are different.

5.1.1 Locality detection and prediction with program counter information

As shown in Fig. 1, for some programs, a proportion of the memory accesses has little or no reuse. Our target is to acquire the reuse information in these programs dynamically and evict these cache lines with a low reuse possibility.

To protect the lost locality in the GPU cache, we need to collect the reuse information in the cache to distinguish the cache lines with a low locality from those with a high locality. Our locality detector has two parts: an extensive cache line architecture and the locality prediction logic. As shown in Fig. 4, we add the PC information in each block of the L1_D-cache. Each line consists of tag bits, data bits, PC, and a reuse bit. The tag bits are used to check the hit or miss of the access. The data bits store the data. PC, shown as W_PC, is for the instruction which has accessed the cache line recently. The reuse bit records the reuse information of this block, indicating whether this block has a locality or not.

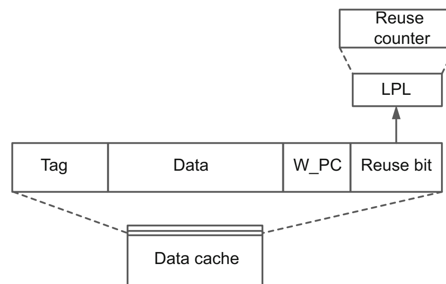


Fig. 4 Implementation of the locality detector on the graphics processing unit (GPU)
LPP: locality-protected method based on the instruction program counter

We develop the locality prediction logic (LPL) to monitor the locality of each line and predict the reuse possibility for each block by making an AND operation between the last and the current reuse values of the same block. LPL includes 128 entries (covering all of the cache blocks). Each entry comprises a one-bit locality counter to record one block's reuse information in the last access. LPL also has a common AND logic to compute reuse values and to predict reuse possibility. The output from LPL

is used by our proposed prioritised cache allocation unit for a more efficient cache management.

When a memory request is sent to the memory, the PC for instruction, which sends the current address, will be compared with that in the line. If the two are the same, the block's reuse bit will be set as 1. Otherwise, it will be reset. After that, the reuse bit of the accessed block will perform an AND operation with its corresponding bit in the LPL entry. If the result is 1, it means that the last and the current accesses both have a locality, and we predict that the next access will also have a locality. Otherwise, we predict that the next access will have no locality. Then, the reuse bit in the block will replace the reuse bit of this entry and the AND result will be used by PCAU as the locality information (Fig. 5). Since a warp has the same PC, we can use the PC-based reuse information to indicate warp information. As the locality between different memory instructions is far less than that within one instruction, the locality detection overhead between different instructions is considerable. Therefore, we consider mainly the locality within one instruction.

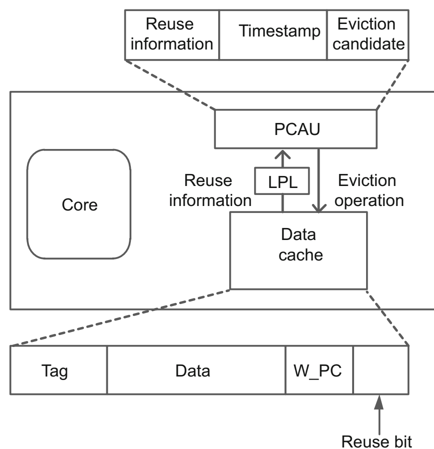


Fig. 5 Implementation of the locality-protected method based on the instruction program counter (LPC) on the graphics processing unit (GPU)

PCAU: prioritised cache allocation unit; LPL: locality prediction logic

5.1.2 Prioritised cache allocation unit with reuse and time-stamp information

After the collection of reuse information for each cache block, the reuse information is transformed to PCAU to help decide which cache line will be evicted on the cycle. In the traditional bit-pseudo-LRU (LRU: least recently used) eviction policy, each cache line has time-stamp information on the last access time for the block, and the eviction unit will choose the LRU cache block to evict. In this way, the temporal locality of the cache lines is preserved.

Our implementation is based on the bit-pseudo-LRU replacement policy and has made some improvements according to the GPU's memory access pattern. It uses the reuse information from the cache lines to help determine whether the cache block will be a candidate for eviction. It is clear that the reuse information and the time-stamp information are used together to predict whether the cache block will be the candidate for eviction. If a cache block is LRU and is without a locality, it will be evicted. Therefore, the time-stamp and locality information can coordinate with each other to decide the cache allocation scheme.

The reuse information from LPL shows the value of the locality counter, which predicts the reuse possibility of the currently accessed block. The time-stamp information indicates whether the block is LRU. The time-stamp information indicates if the cache block is the LRU one (if yes, it will be set as 0; otherwise, 1), and will create an OR operation with the reuse information (from LPL) which indicates if the block has locality or not in the latest two accesses (if yes, it will be set as 1; otherwise, 0). If the result of the OR operation is 1, the eviction prediction bit is set as 1, indicating that this block should not be evicted; otherwise, it is 0, meaning that this block should be evicted. Using reuse information with time-stamp information can predict the reuse possibility of the block more accurately, since the time-stamp information can represent only temporal locality, and reuse information from LPL, which uses the real access traces to instruct cache replacement, is more convincing.

5.1.3 Implementation of the prioritised cache allocation unit

LPC is implemented in Algorithm 1, where we first declare and initialise two variables 'r_use' and 'classify'. 'r_use' records the reuse information of each cache line and the 'classify' variable can distinguish L1 cache from all of the on-chip memories. Then, each cache line will be checked to see if the access hits on the cache. If it hits, the current time-stamp will be assigned to the cache line. After that,

whether the memory access comes from L1 cache will be checked. If it is from L1_D-cache and the PC information of the access matches the last PC in cache, r_use will be set as 1. Otherwise, r_use will be reset. r_use will then perform an AND operation with the corresponding reuse value in LPL. Finally, the AND result will carry out an OR operation with the time-stamp information to decide the eviction.

Algorithm 1 Program counter based locality-protected cache allocation

```
1: classify ← false
2: for Idx ← 1 to set_index×assoc do
3:    m_lines[Idx].r_use ← 0;
4: end for // One-best to initialise
5: if access L1D cache then
6:    classify = true;
7: end if
8: for Idx ← 1 to set_index×assoc do
9:    if STATUS=HIT then
10:      m_lines[Idx].m_time_stamp = current_time;
11:      if classify = true then
12:         if PC = m_lines[Idx].m_pc then
13:            m_lines[Idx].r_use ← 1; // Collect the
                  // reuse information
14:         else
15:            m_lines[Idx].r_use ← 0;
16:            m_lines[Idx].m_pc ← PC;
17:         end if
18:         if (m_lpl[Idx].r_use&m_lines[Idx].r_use)
             ==1 then
19:            m_lpl[Idx].r_use=1; // Predict reuse
                  // information of the next access to
                  // the block
20:         else
21:            m_lpl[Idx].r_use=0;
22:         end if
23:      end if
24:   end if
25:   if !(m_lines[Idx].m_time_stamp<valid_time)||
        (m_lpl[Idx].r_use) then
26:      valid_time=m_lines[Idx].m_time_stamp;
27:      evicted_line=Idx; // Determine which cache
            // block to evict
28:   end if
29: end for
```

5.2 Locality information based reordering

In this section, we will introduce the locality information based warp scheduling method which uses locality information from PCAU to instruct the warp scheduling. The warp scheduling method and the locality-protected cache allocation discussed in Section 5.1 constitute CWLP. The whole structure diagram of CWLP is shown in Fig. 6.

5.2.1 Locality information collection

Because the reuse information in PCAU can predict cache locality, it is used to guide the cache allocation to preserve data locality as shown in Section 5.1. Furthermore, the reuse information can be used to instruct warp scheduling in the warp issue stage of the pipeline.

The locality information score (LIS) update logic is made up of a 10-bit storage capacity and several adders. The 10-bit storage stores 10 instances of reuse information from the latest 10 accesses. The adders are used to calculate the reuse values in the storage to evaluate the locality status in the cache. The reuse information in PCAU will update LIS every cycle. The output from LIS will help the warp priority logic create reorder operations in the warp queue.

The maximum value for the 10 bits in LIS is 10. Therefore, if the addition of the 10 bits is larger than five, it means that the number of cache lines accessed with locality is larger than that accessed without locality. When the addition is larger than five, we think that there is locality in the L1 data cache. Otherwise, we think there is no locality in the L1 data cache. If the addition of the 10 bits is larger than five, LIS shows that there is locality and the warp scheduler will not reorder the warp queue. Otherwise, the warp scheduler will reorder the warp queue to preserve locality. The reuse information in PCAU will update LIS in every cycle. The priority logic creates the reordering operations in the warp queue in every cycle. To detect the locality more accurately, we create a fine-grained locality information clean-up operation in LIS update logic. For every 10 L1 cache accesses, LIS update logic buffer will be cleaned up and the buffer will count the reuse values from the next cycle. Using this fine-grained locality clean-up method, the locality information can be passed to the priority logic in a timely manner.

5.2.2 Warp reordering process and effects

We propose a novel warp scheduling scheme to preserve inter-warp locality and hide memory

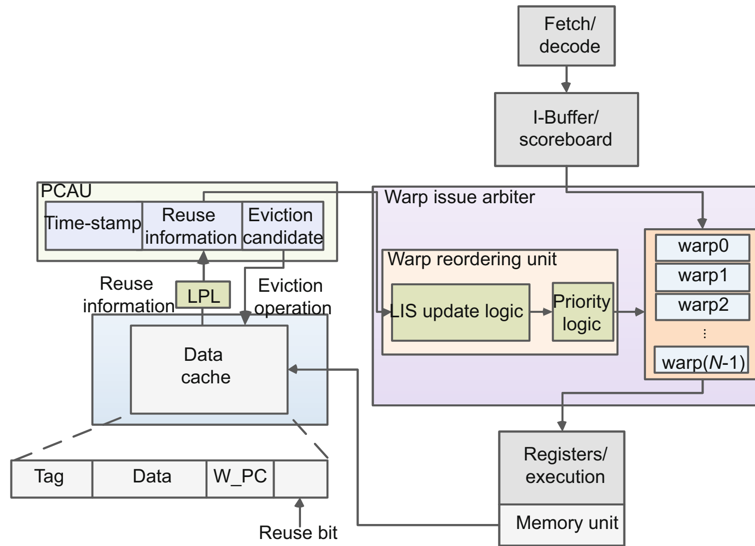


Fig. 6 Coordinated warp scheduling with locality-protected cache allocation

PCAU: prioritised cache allocation unit; LPL: locality prediction logic; LIS: locality information score; I-Buffer: instruction buffer

latency. We use the locality information from LIS to decide whether to reorder the warps. Our method is based on the idea that the locality information based scheduling can hide latency and preserve locality by either rebalancing the warp queue or not, which has significant effects on the GPU performance.

CWLP has the ability to preserve locality and hide memory latency. LRR scheduling can preserve locality well but can not hide long latency. Reordering can hide long latency but may destroy locality. Therefore, we use locality information from memory to issue a reorder decision to preserve locality and hide long latency at the same time. If there is locality in memory, it will not reorder the warps and the locality is preserved. Otherwise, it will reorder the warps and memory latency is hidden.

In CWLP, warps are divided into short and long latency warps as shown in Fig. 7a, where short latency warps are on-chip and long latency warps have accesses to the global memory. If LIS shows that the cache has no locality, the scheduler will perform reordering. Long- and short-latency warps are put into the warp pool in an interleaved fashion (Fig. 7b). Short latency warps will proceed to overlap the long latency warps to reduce idles in CU. If LIS shows that the cache has locality, the warp queue will not reorder to keep locality. In this way, locality is maintained and latency is hidden. The choice of insertion place is based on balance and simplicity. We have done experiments on different reordering schemes and found a method which makes short latency warps separated by long latency warps one by one. Fig. 7c shows that this can hide latency well. The balanced reordering method avoids the occurrence of a burst of long operations that stall CU. The saved stall cycles and the reordering effects are shown in Fig. 8.

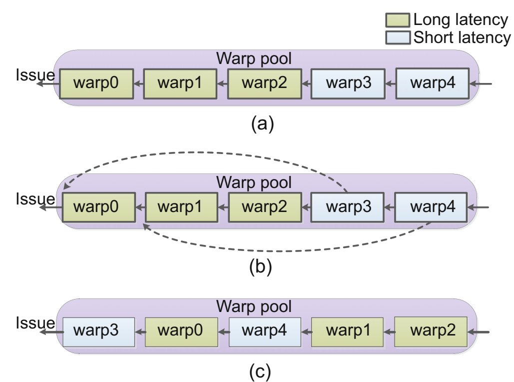


Fig. 7 Graphical representation of the warp scheduling process: (a) original warp queue; (b) locality-protected reordering; (c) after reordering (References to color refer to the online version of this figure)

Fig. 8 shows the reordering process and the effects of the scheduling method in CWLP, an optimised warp scheduling method which uses locality preservation and latency hiding, and its effects are on the baseline LRR scheduling. In Figs. 8a and 8b, the upper graphs show that the CU stall

benefits from reordering, and the graphs below show that the locality changes before and after reordering. In the upper graphs, arrows indicate the time of memory access, and the saved cycles with reordering are shown in the graphs. In the bottom of Figs. 8a and 8b, dark boxes illustrate long latency warps and light ones show short latency warps. The ticks and crosses illustrate whether there is locality between successive warps or not.

We use a typical and a common memory access pattern to show the process and effects of the reordering method. In a typical case where a burst of long memory warps access the memory (Fig. 8a), the warps are classified into long (warp0, warp1, and warp2) and short (warp3 and warp4) latency warps. The long and short latency warps are interleaved one by one (Fig. 7). Because the consecutive long latency warps are separated and the short ones can use the computation units as soon as possible, CWLP saves stall cycles with acceptable data locality misses compared with LRR scheduling. In a common case, where the long memory access pattern is not successive (i.e., if there is no consecutive long operation, the accesses are not burst) (Fig. 8b), CWLP issues classification and reordering to the candidate warps and saves some cycles with misses of some locality.

In general, this method performs well in a typical case and has no performance degradation in common cases. Compared with LRR scheduling, CWLP's reordering method achieves a better latency hiding with few locality misses. CWLP scheduling shows potential for performance improvements in the processor. In Section 6, we will analyze the hardware cost of our method.

6 Implementation of coordinated warp scheduling and locality-protected cache allocation scheme

CWLP is implemented in Algorithm 2, where the collection process for locality information is the same as that in LPC. The LPL entries are used to record the reuse information of each line. If the number of cache lines with locality is larger than that without and the whole access number is 10, the warp queue will perform reordering. Otherwise, the warp queue will remain unchanged. The reuse information needs to be cleaned up every 10 accesses for more accurate locality detection.

The reordering process is shown in Fig. 9. It uses time-stamp and latency information in warp to select a warp to issue. As the HDL code shows (Fig. 9), the condition includes two parts, an equation to judge whether the warp is the oldest among the remaining warps, and the XOR logic to decide whether the latency of the warp is different from that of the last issued warp. If one warp satisfies the two conditions simultaneously, the warp will be issued. Otherwise, the XOR logic has a priority to be checked separately to allow the issuing of the warp. If there is still no

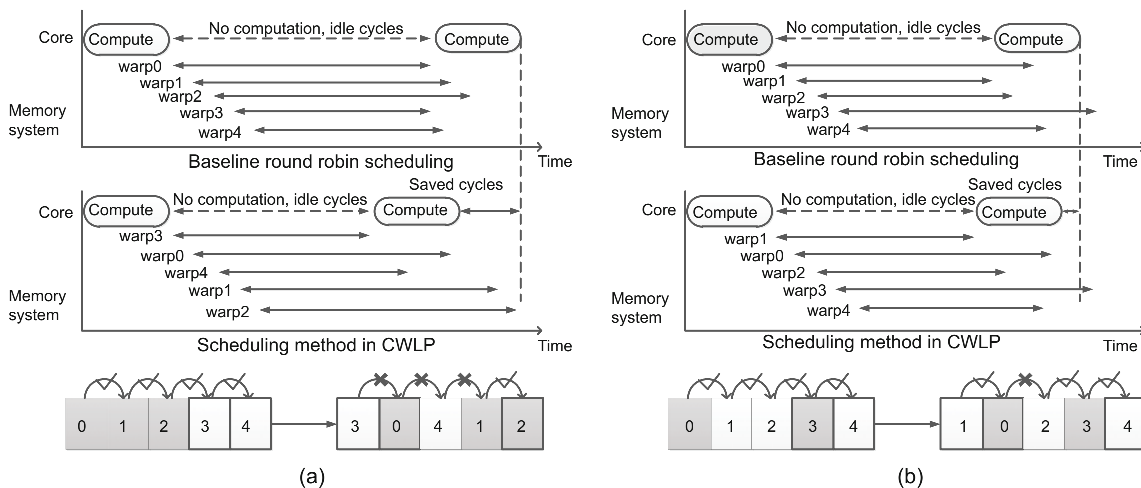


Fig. 8 Processes and effects of the scheduling method in the coordinated warp scheduling and locality-protected (CWLP) cache allocation scheme when reordering is enabled in a typical case where the scheduling uses memory hiding with acceptable data locality misses (a) and a common case where the scheduling uses memory hiding with acceptable data locality misses (b)

Algorithm 2 Coordinated warp scheduling and locality-protected algorithm

```
1:  Algorithm 1  // The same as in Algorithm 1
2:  for Idx ← 1 to set_index×assoc do
3:    if m_lpl[Idx].r_use=1 then
4:      dist1++; // The accumulation of locality
          // information
5:    else
6:      dist0++;
7:    end if
8:    if dist0<dist1 then
9:      r_use_info=1;
10:   else
11:     r_use_info=0;
12:   end if
13:   if dist0+dist1>10 then
14:     dist0=0;
15:     dist1=0;
16:   end if
17:   if (r_use_info==1) && (dist0+dist1==10)
      then
18:     m_warps.reordering(); // Take reordering
          // operation based on locality information
19:   end if
20: end for
```

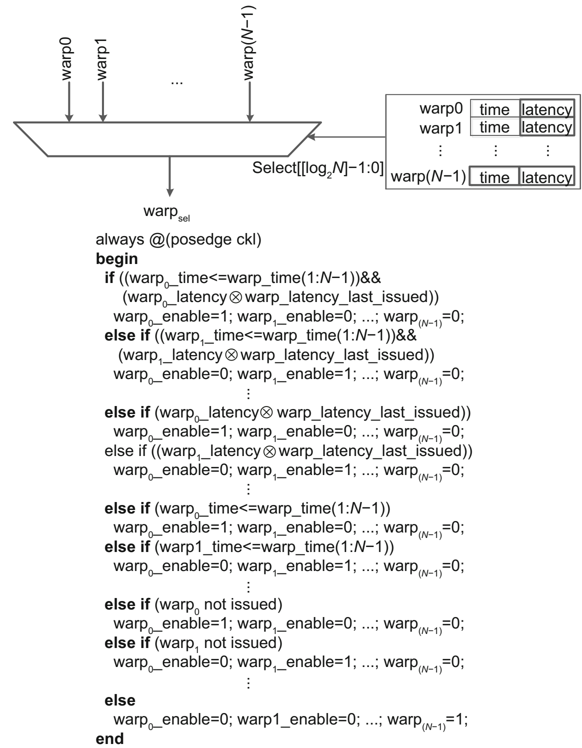


Fig. 9 Hardware implementation of the priority logic

warp with permission to issue, the time information in warps will be compared to issue the oldest warp. If all of the remaining warps have the same time information, the priority logic will issue the warp according to the warp number sequentially.

7 Hardware cost and complexity

CWLP requires the introduction of several hardware elements, and the hardware overhead is comparatively small. On one hand, the L1_D-cache must be enlarged to add information within each line about the PC that requested it (the W_PC field) and the reuse bit. On the other hand, new structures are required, such as LPL, PCAU, LIS, and priority logic, which also introduce extra overhead. Since L1_I-cache has a memory space of 2 KB (Table 1), the W_PC field needs 11 bits for each line to cover the whole instruction space and 1 bit for each line to indicate the locality in the line. The L1_D-cache has 128 lines and the overhead is 192 B, which is only about 1% of the L1_D-cache capacity. LPL comprises 128 one-bit entries (16 B) and the AND logic. The hardware cost is low. As for PCAU, since it records the information for only the current access, its memory overhead is only three bits. The AND logic in PCAU is also inexpensive to implement. The LIS update logic consists of 10 bits to record the reuse values of the recent lines and nine adders (five 1-bit, two 2-bit, one 3-bit, and one 4-bit adder to add to the 10 bits) to determine the cache locality. The hardware overhead can be omitted. The warp priority logic has 1 bit for each warp to indicate whether it has a long or short latency, which is updated by information from the scoreboard. It also has some logic to reorder the warp queue. As shown in Fig. 9, the multiple selector uses an enable signal to select a warp to issue. The enable signal is decided by the logical operation result of the time and latency information among the warps. The decision logic is simple as the HDL code shows. The whole hardware overhead for the priority logic is dependent mainly on the warp number. Since the warp number in an SM is no more than 1536/32=48 (Table 1) and the number will decrease as the warps are issued from the warp pool, the overhead for the priority logic is small. Thus, we can conclude that the total hardware cost in CWLP is small and its complexity is low.

8 Experimental evaluation

8.1 Methodology

We use the GPGPU-sim (version 3.2.2) simulator (Bakhoda et al., 2009) to evaluate our locality-protected design across various GPU programs. GPGPU-sim is a cycle-accurate performance simulator which models a general-purpose GPU architecture supporting NVIDIA CUDA (NVIDIA, 2015) and its parallel thread execution and instruction set architecture (PTX ISA). We take the architecture of GTX480 as our simulated architecture and the optimisation method can be popularised to other new architectures. We use the default simulator parameters, and the relevant configurations are shown in Table 1.

Table 1 GPGPU-sim configurations

Item	Value
Numbers of streaming multiprocessors	32
Warp size	32
SIMD pipeline width	16
Number of threads/core	1536
Number of registers/core	32 768
Shared memory/core	48 KB
Constant cache size/core	8 KB
Texture cache size/core	12 KB, 128 B line, 24-way assoc.
Number of memory channels	6
L1 instruction cache	2 KB
L1 data cache	16 KB, 128 B line, 4-way assoc. LRU
L2 unified cache	64 K/mem. channel, 128 B line, 8-way assoc. LRU
Compute core clock	700 MHz
Interconnect clock	700 MHz
Memory clock	924 MHz
DRAM request queue capacity	16
Memory controller	Out of order
GDDR5 memory timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6
Memory channel bandwidth	8 bytes/cycle

SIMD: single instruction multiple data; DRAM: dynamic random access memory; LRU: least recently used

8.2 Evaluation results and analysis

To perform our evaluation on the CWLP mechanism, we use GPU-enabled workloads from Rodinia (Che et al., 2009), Bakhoda et al. (2009), CUDA SDK, and Mars (Fang et al., 2011). The applications have different parallelisms and memory-access patterns. We classify the benchmarks into two categories, cache-sensitive and cache-insensitive applications. We use the default GPU configurations to evaluate the benchmarks in Table 2.

Table 2 Benchmarks of the proposed locality-protected design

Application	Abbreviation	Sensitivity
k-means clustering	KMN	Sensitive
Histogram	His	Sensitive
Speckle reducing anisotropic diffusion V2	SRAD-V2	Sensitive
Convolution separable	Convolution	Sensitive
Breadth first search	BFS	Sensitive
Inverted index	IIX	Sensitive
Balance tree	B+tree	Sensitive
N queen	NQU	Insensitive
Vector add	VADD	Insensitive
String match	SM	Insensitive
Word count	WC	Insensitive
Mummer GPU	Mum	Insensitive
Hot spot	Hotspot	Insensitive
Store GPU	STO	Insensitive

We use loose round robin (LRR), two-level scheduling, cache conscious wavefront scheduling (CCWS), and greedy then oldest (GTO) for comparison with LPC and CWLP. LRR is the scheduling scheme that the base architecture adopts. Two-level scheduling is an optimised scheme for LRR. Through prioritising warps with short latency, the number of long memory stalls is reduced. CCWS is a novel scheduling method which can achieve high performance by reducing cache contention. These scheduling schemes are all practical and can promote GPU performance effectively. GTO is the scheduling method that the default simulator adopts. We use them for performance comparison with our scheme.

8.2.1 Performance results for the L1 cache

LPC performs optimisations on the L1_D-cache and we will evaluate LPC's effects on it. The L1 cache is used to store the most frequently used data in GPU. Its performance has a great impact on GPU's performance. The cache miss rate is one of the commonly used metrics to evaluate cache performance. A higher cache miss rate indicates that more accesses will be sent to the lower memory hierarchy, leading to performance degradation. As Fig. 10 shows, KMN, BFS, and SM have clear decreases in

the cache miss rate. This indicates that LPC has relieved some of the cache conflict problems through locality-protected methods. The other applications have almost no change in terms of the miss rate, meaning that they are not sensitive to cache locality. In Section 8.2.2, we will analyze the relationships between L1_D-cache performance and IPC results in detail over different applications.

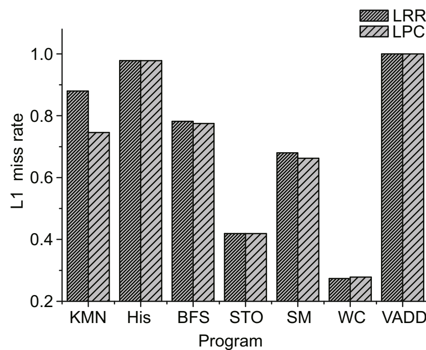


Fig. 10 Miss rate for the L1 cache with locality-protected method based on the instruction program counter (LPC) and loose round robin (LRR) for different programs

8.2.2 Instructions per cycle (IPC) performance of GPU

Fig. 11 shows the normalised instructions per cycle (IPC) for the different optimisation methods across different applications. With CWLP, programs such as KMN and Convolution obtain 19.8% and 16.8% improvement, respectively, over the baseline LRR method. They can obtain 20.1% and 23.3% benefit, respectively, over the newest CCWS scheme. They both have many reused data accesses (Fig. 1) and can achieve high performance. Some programs that have less locality, such as BFS (Fig. 1), IIX, and SRAD, can obtain an improvement of 5.2%, 11%, and 5.2%, respectively, over the baseline, and a slight decrease compared with CCWS. This is because the locality-protected method based on locality has fewer benefits for these applications, and warp reordering begins to take effect. For the other applications, they have a slight increase or decrease over the baseline and the CCWS scheme. They are insensitive to the preservation of data locality and latency hiding because there are little reuse information and few long latencies within them. In particular, we find steep decreases in some applications with CCWS. However, CWLP will always have a good performance. In general, CWLP is more robust compared with the other schemes and can achieve good performance for a variety of applications.

IPCs for KMN, BFS, and SM are consistent with their L1_D-cache miss rate, because the lower L1_D-cache miss rate has better IPC performance, indicating that their performance improvements result mainly from locality preservation in the L1_D-cache. His and STO have a clear advantage in IPC performance; however, their L1 miss rate remains unchanged. This is because their performance advantages are derived mainly from latency hiding. The other applications have few improvements in IPCs and their L1 miss rates stay the same. This is because their performances are insensitive to L1_D-cache and latency hiding, and the locality protection and warp reordering method has limited effects on them.

Overall, our CWLP can achieve a good performance for a variety of cache-sensitive applications with different characteristics. It can achieve an average improvement of 8.8% over the baseline LRR scheduling method, and 4.8% over the novel CCWS scheduling scheme. We can also see that the relatively irregular programs have gained a better performance from the CWLP mechanism, because warp reordering has its greatest effects in these programs. Compared with some feedback-based schedulers such as CCWS and DAWS, CWLP uses PC information as the feedback to reduce reuse information collection and address-comparison expenses. Meanwhile, the new thread scheduler is more efficient and the hardware overhead is small. It is a promising method for future GPU architectures.

9 Related work

Previous work has looked at various ways to solve issues that limit GPU performance, including thread scheduling schemes, using data locality in the GPU cache, and memory optimisation methods coordinated with thread scheduling. In this section, we will discuss research related to these three aspects and illustrate our method's advantage.

9.1 Thread scheduling

The two-level scheduler on GPUs was first proposed to reduce energy (Gebhart et al., 2011). It

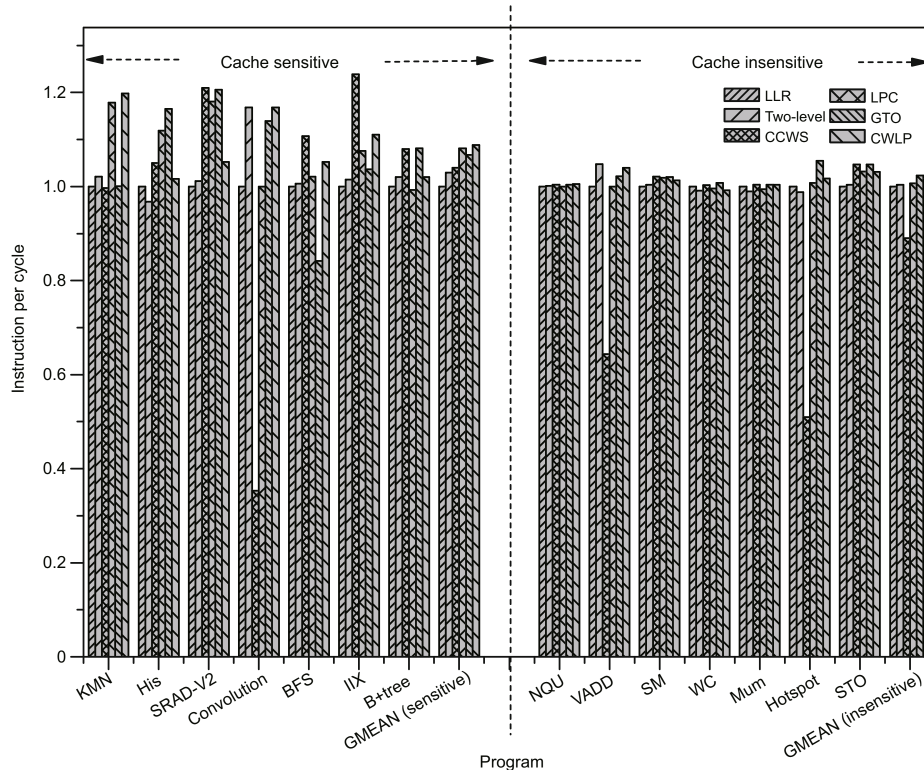


Fig. 11 The CWLP cache allocation scheme performance compared with the LRR, two-level, LPC, CCWS, and GTO schemes for different programs

CWLP: coordinated warp scheduling and locality-protected; LPC: locality-protected method based on the instruction program counter; CCWS: cache conscious wavefront scheduling; GTO: greedy then oldest

maintains a small set of warps to hide short-latency warps, and uses a large set of pending threads to hide long memory latencies. It can reduce energy considerably because most of the warps are not active. Narasiman et al. (2011) expanded the previous two-level scheduler to partition all warps into warp groups. Warps within the active group are qualified for scheduling while they are executing short-latency instructions. Once all the warps in the active group are stalled on long-latency loads, the warps within the pending group will start to execute to avoid stalls. Lee et al. (2014) explored alternative thread blocks or CTA scheduling in GPU. In particular, they proposed two novel scheduling methods, the lazy and block CTA scheduling methods, to exploit the interaction between the thread block scheduler and the warp scheduler to improve performance. These methods are effective and can achieve good performance over the baseline; however, their objective is optimising the scheduling scheme, while our work combines a cache management scheme with thread scheduling on GPUs.

9.2 Data reuse in GPUs

Nugteren et al. (2014) showed that reuse distance theory can be used to model GPU caches in detail by extending the theory with: (1) scheduling of the GPU's threads, warps, threadblocks, cores, and sets of active threads; (2) in-flight memory requests and conditional and non-uniform latencies; (3) cache associativity; (4) miss-status holding-registers (MSHRs); (5) warp divergence. It is a more accurate model for evaluating GPU cache performance using reuse distance theory. However, this is not meant to promote cache performance. Gupta et al. (2013) investigated the locality of reference at different cache levels in the memory hierarchy. They looked into the locality behaviour at the warp, thread-block, and SM levels. Since multiplication was used as only a case study, the insights in Gupta et al. (2013) are limited to some programs similar to multiplication, and do not make optimisations to the memory hierarchy based on the insight revealed from locality analysis. In contrast, since

CWLP is not proposed for certain applications where intra- and inter-warp locality can be preserved, it can be used for a wider range of applications. Meanwhile, it performs optimisations on the cache management strategy based on reuse information to achieve better performance.

9.3 Cache optimisation by scheduling threads

Rogers et al. (2012) put forward CCWS to optimise memory access efficiency on GPUs by using a victim cache tag array to record inter-warp cache contention. The scheduler uses this feedback information to throttle the active warp count to reduce cache contention. Memory request prioritization buffer (MRPB) (Jia et al., 2014) attacks the intra-warp contention that cannot be addressed by warp schedulers. It uses request reordering and cache bypassing to improve performance. Other strategies such as criticality-aware warp acceleration (CAWA) (Lee et al., 2015) target the removal of significant execution time disparity on GPU to improve resource utilization for GPU workloads. It can obtain a good performance improvement; however, the introduction of two predictors makes the scheduling process complicated. Chen et al. (2013) introduced a novel algorithm to efficiently use multi-thread parallelism and overlap short-latency compute instructions with long-latency memory accesses. Data locality, however, cannot be preserved directly in the method. Jog et al. (2013) carried out a locality-aware warp scheduling method to reduce contention and increase reuse in the L1 cache; however, it is based on the shift in priority in warps, and its performance improvements come from the smaller number of CTAs. In contrast, CWLP can maintain intra-warp locality by evicting cache lines with less locality and preserve inter-warp locality through warp scheduling. Its simpler predictor, compared with CAWA's, is highly efficient with a lower hardware overhead.

10 Conclusions and future work

We have proposed a locality-protected method called 'CWLP' to improve GPU performance. The key premise of CWLP is to use GPU features to design a novel cache management scheme to preserve cache blocks with a high locality in the cache and a thread scheduling method to hide the latency, and to improve the overall throughput. CWLP achieves this by: (1) collecting the reuse information from each memory block based on the instruction PC and designing a locality predictor to predict the possibility of eviction for each line; (2) proposing a coordinated cache line evictor which coordinates reuse information with an LRU replacement scheme to evict cache blocks without reuse possibility and to preserve space for cache lines with a high locality; (3) using the locality information to instruct the warp scheduling process to hide latency and preserve locality.

Our experimental evaluations on the GPGPU-sim platform demonstrated that CWLP can effectively improve GPU performance. It can achieve an average improvement of 8.8% over the baseline LRR scheduling with a low overhead. We concluded that our proposed CWLP method is an effective way to enhance GPU performance and it is practical for future GPU architectures. In the future, we plan to extend the CWLP scheme to L2 cache to further improve GPU performance.

References

Bakhoda A, Yuan G, Fung W, et al., 2009. Analyzing CUDA workloads using a detailed GPU simulator. ISPASS IEEE Int Symp on Performance Analysis of Systems and Software, p.163-174. https://doi.org/10.1109/ISPASS.2009.4919648

Che S, Boyer M, Meng J, et al., 2009. Rodinia: a benchmark suite for heterogeneous computing. IISWC IEEE Int Symp on Workload Characterization, p.44-54. https://doi.org/10.1109/IISWC.2009.5306797

Chen J, Tao X, Yang Z, et al., 2013. Guided region-based GPU scheduling: utilizing multi-thread parallelism to hide memory latency. IEEE 27$^{\text{th}}$ Int Symp on Parallel & Distributed Processing, p.441-451. https://doi.org/10.1109/IPDPS.2013.95

Chen X, Chang L, Rodrigues C, et al., 2014. Adaptive cache management for energy-efficient GPU computing. Proc 47$^{\text{th}}$ Annual IEEE/ACM Int Symp on Microarchitecture, p.343-355. https://doi.org/10.1109/MICRO.2014.11

Dally W, Labonte F, Das A, et al., 2003. Merrimac: supercomputing with streams. Proc ACM/IEEE Conf on Supercomputing, Article 35. https://doi.org/10.1145/1048935.1050187

Drew Y, 2008. A closer look at GPUs. *Commun ACM*, 51(10):50-57. https://doi.org/10.1145/1400181.1400197

Fang W, He B, Luo Q, et al., 2011. Mars: accelerating mapreduce with graphics processors. *IEEE Trans Parall Distr Syst*, 22(4):608-620. https://doi.org/10.1109/TPDS.2010.158

Gebhart M, Johnson D, Tarjan D, et al., 2011. Energy-efficient mechanisms for managing thread context in throughput processors. Proc 38th Annual Int Symp Computer Architecture, p.235-246. https://doi.org/10.1145/2000064.2000093

Gupta S, Xiang P, Zhou H, 2013. Analyzing locality of memory references in GPU architectures. Proc ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, Article 12.
https://doi.org/10.1145/2492408.2492423

Harris M, 2014. Maxwell: the Most Advanced CUDA GPU Ever Made. https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made

Jia W, Shaw K, Martonosi M, 2014. MRPB: memory request prioritization for massively parallel processors. IEEE 20th Int Symp on High Performance Computer Architecture, p.272-283.
https://doi.org/10.1109/HPCA.2014.6835938

Jog A, Kayiran O, Nachiappan C, et al., 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. *ACM SIGARCH Comput Arch News*, 41(1):395-406.
https://doi.org/10.1145/2490301.2451158

Lee M, Song S, Moon J, et al., 2014. Improving GPGPU resource utilization through alternative thread block scheduling. IEEE 20th Int Symp on High Performance Computer Architecture, p.260-271.
https://doi.org/10.1109/HPCA.2014.6835937

Lee S, Arunkumar A, Wu C, 2015. CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. Proc 42nd Annual Int Symp on Computer Architecture, p.515-527.
https://doi.org/10.1145/2872887.2750418

Narasiman V, Shebanow M, Lee CJ, et al., 2011. Improving GPU performance via large warps and two-level warp scheduling. Proc 44th Annual IEEE/ACM Int Symp on Microarchitecture, p.308-317.
https://doi.org/10.1145/2155620.2155656

Nugteren C, van den Braak G, Corporaal H, et al., 2014. A detailed GPU cache model based on reuse distance theory. IEEE 20th Int Symp on High Performance Computer Architecture, p.37-48.
https://doi.org/10.1109/HPCA.2014.6835955

NVIDIA, 2009. NVIDIA's next generation CUDA compute architecture: FERMI. v1.1. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

NVIDIA, 2015. NVIDIA CUDA C Programming Guide v7.5. http://developer.nvidia.com/nvidia-gpu-computing-documentation

Rhu M, Sullivan M, Leng J, et al., 2013. A locality-aware memory hierarchy for energy-efficient GPU architectures. Proc 46th Annual IEEE/ACM Int Symp on Microarchitecture, p.86-98.
https://doi.org/10.1145/2540708.2540717

Rogers T, O'Connor M, Aamodt T, 2012. Cache-conscious wavefront scheduling. Proc 45th Annual IEEE/ACM Int Symp on Microarchitecture, p.72-83.
https://doi.org/10.1109/MICRO.2012.16

Rogers T, O'Connor M, Aamodt T, 2013. Divergence-aware warp scheduling. Proc 46th Annual IEEE/ACM Int Symp on Microarchitecture, p.99-110.
https://doi.org/10.1145/2540708.2540718

Sethia A, Jamshidi D, Mahlke S, 2015. Mascar: speeding up GPU warps by reducing memory pitstops. IEEE 21st Int Symp on High Performance Computer Architecture, p.174-185.
https://doi.org/10.1109/HPCA.2015.7056031

Xie X, Liang Y, Sun G, et al., 2013. An efficient compiler framework for cache bypassing on GPUs. IEEE/ACM Int Conf on Computer-Aided Design, p.516-523.
https://doi.org/10.1109/ICCAD.2013.6691165

Xie X, Liang Y, Wang Y, et al., 2015. Coordinated static and dynamic cache bypassing for GPUs. IEEE 21st Int Symp on High Performance Computer Architecture, p.76-88. https://doi.org/10.1109/HPCA.2015.7056023

Xie X, Liang Y, Li X, et al., 2017. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. IEEE/ACM Int Symp on Microarchitecture, p.395-406.
https://doi.org/10.1109/TC.2017.2776272

Zhang Y, Xing Z, Zhou L, et al., 2017. Locality protected dynamic cache allocation scheme on GPUs. IEEE Trustcom/BigDataSE/ISPA, p.1524-1530.
https://doi.org/10.1109/TrustCom.2016.0237

Zheng Z, 2014. Research on Key Technologies for Cache Power and Performance Optimization on Many-Core Heterogeneous Architecture. PhD Thesis, National University of Defense Technology, Changsha, China (in Chinese).