Parallel Computing 71 (2018) 23–41

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Benchmarking the GPU memory at the warp level☆

Minquan Fang[a,*], Jianbin Fang[b,*], Weimin Zhang[c], Haifang Zhou[b], Jianxing Liao[a], Yuangang Wang[a]

[a] Data Center Technology Laboratory, 2012 Laboratories, Huawei Technologies Co., Ltd, Hangzhou, China
[b] College of Computer, National University of Defense Technology, Changsha, China
[c] Academy of Ocean Science and Engineering, National University of Defense Technology, Changsha, China

ARTICLE INFO

Article history:
Received 14 July 2016
Revised 23 August 2017
Accepted 2 November 2017
Available online 3 November 2017

Keywords:
Graphic process unit (GPU)
Micro-benchmarks
Warp-level latency

ABSTRACT

Graphic process units (GPUs) are widely used in scientific computing, because of their high performance and energy efficiency. Nonetheless, GPUs are featured with a hierarchical memory system, on which code optimization requires an in-depth understanding for programmers. For this, we often measure the capability (latency or bandwidth) of the memory system with micro-benchmarks. Prior works focus on the latency of a single thread to disclose the unrevealed information. This per-thread measurement cannot reflect the actual process of a program execution, because the smallest executable unit of parallelism on a GPU comprises 32 threads (a *warp* of threads). This motivates us to benchmark the GPU memory system at the warp-level.

In this paper, we benchmark the GPU memory system to quantify the capability of *parallel accessing* and *broadcasting*. Such warp-level measurements are performed on shared memory, constant memory, global memory and texture memory. Further, we discuss how to replace local memory with registers, how to avoid bank conflicts of share memory, and how to maximize global memory bandwidth with alternative data types. By analyzing the experimental results, we summarize the optimization guidelines for different types of memories, and build an optimization framework on GPU memories. Taking a case study of `maximum noise fraction rotation` in dimension reduction of hyperspectral images, we demonstrate that our framework is applicable and effective.

Our work discloses the characteristics of GPU memories at the warp-level, and leads to optimization guidelines. The warp-level benchmarking results can facilitate the process of designing parallel algorithms, modeling and optimizing GPU programs. To the best of our knowledge, this is the first benchmarking effort at the warp-level for the GPU memory system.

1. Introduction

GPUs were initially used for 3D graphics rendering. With the increase of programmability, GPUs have gained more and more attention in scientific computing [1–5]. A large amount of applications written in CUDA, OpenCL and OpenACC are running efficiently on the modern GPUs. Nevertheless, it is difficult to exploit such many-cores in order to fully mine the

☆ This paper was supported by the National Natural Science Foundation of China (Grant No. 61602501, 61272146 and 41375113).
* Corresponding author.
E-mail addresses: fmq@hpc6.com (M. Fang), j.fang@nudt.edu.cn (J. Fang).

https://doi.org/10.1016/j.parco.2017.11.003
0167-8191/

hardware potentials. Among others, taming the hierarchical memory system on GPUs is a significant obstacle [6–8]. The information offered by the GPU text books [9,10] and/or the NVIDIA's official documents [11] is very limited. In terms of GPU memory usage, we have no quantified performance gain/loss of using a specific memory type, let alone the reasons behind them. In this case, it is significant to disclose the desired information by designing micro-benchmarks and then performing actual runnings.

Prior works on benchmarking CPU/GPU memory systems often use the *pointer-chasing* technique to measure the accessing latency of a single thread [6,12–16]. Different from CPUs, threads on GPUs do not run separately in real-world applications. Instead, a group of threads (a.k.a. *warp*) are taken as the smallest executing unit on GPUs. Specifically, multiple threads within a warp run simultaneously in a *Single Instruction Multiple Threads* (SIMT) fashion. When two threads within a warp follow different code paths, they are serialized. In terms of memory accesses, the neighboring threads within a warp need access neighboring cells in the global memory space, i.e., *coalesced memory access*. Therefore, we argue that the performance behaviors from a warp are more interesting than that of a single thread. To this end, we benchmark the GPU memory system at the warp level to investigate its characteristics and gain new insights.

In this paper, we design two sets of warp-level micro-benchmarks[1] to measure the capability of *broadcasting* and *parallel accessing*. The basic idea is that a warp of threads access a batch of data elements with various patterns. By comparing the differences of the execution time (i.e., the warp-level latency) on a certain memory type, we infer whether it supports broadcasting and/or parallel accessing. Moreover, we test two memory accessing constraints (aligned accessing and contiguously accessing) to check whether they are a must of efficiently using a memory type. The experiments are running on shared memory, constant memory, global memory and texture memory of an NVIDIA Tesla K20c GPU.

Furthermore, we discuss how to replace local memory with registers, avoid bank conflicts of using shared memory, and maximize global memory bandwidth with alternative data types. By analyzing all the benchmarking results, we summarize a suite of optimization guidelines for each type of memory, and build an optimization framework on GPU memories. With a case study on `maximum noise fraction rotation` in hyperspectral images dimension reduction, we show that the optimized code can obtain a superior performance than the CUABLS version with the speedup of $1.5\times \sim 3\times$, and can obtain a speedup of up to $93\times$ comparing with the serial version. This demonstrates that our framework is applicable and effective. To the best of our knowledge, this is the first benchmarking effort at the warp level for the GPU memory system. Although the warp-based approach is for NVIDIA GPUs and we use the NVIDIA terms (e.g., warp and shared memory) in the context, we argue that it is equally applicable for other GPUs.

To summarize, we make the following contributions.

(1) We propose a warp-based approach, and design two sets of micro-benchmarks to measure the capability of *broadcasting* and *parallel accessing*.
(2) We benchmark the characteristics of shared memory, constant memory, global memory and texture memory with our approach.
(3) We quantify the performance benefits of replacing local memory with registers, avoiding bank conflicts of using shared memory, and maximizing global memory bandwidth with different data types.
(4) We summarize the optimization guidelines for different memory types towards an optimization framework on GPU memories.
(5) We demonstrate how to optimize a case study in hyperspectral image dimension reduction with the help of our framework.

2. Related work

In this section, we introduce the related work on benchmarking CPUs, Intel Xeon Phis and GPUs, with a focus on their memory systems. Then we describe GPU memory optimizations, and the concept of warps and SIMT.

Benchmarking CPUs and Phis: Various studies have been performed to investigate the microarchitectures of CPUs and Intel Xeon Phis with micro-benchmarks. Smith et al. develop a high-level program to evaluate the cache and the TLB of CPUs [24]. Peng et al. assess the performance of the traditional multi-core processors. They focus on the execution time and the throughput, and then analyze its memory hierarchy and scalability [25]. In [26] , the authors reveal many fundamental details of the Intel Nehalem processor microarchitecture by benchmarking the latency and the bandwidth between different locations of the memory subsystem. For the many integrated core (MIC) architecture, Fang et al. benchmark its five components, including core, cache, memory, ring and PCIe [27]. When benchmarking memory systems of the aforementioned microarchitectures, most researchers adpot the *pointer-chasing* approach, which is equally used in our work.

Benchmarking GPU memory systems is critical to mine the many-core's potentials. Volkov et al. firstly use the *pointer-chasing* approach to benchmark the 8800GTX GPU, and disclose the details of the texture cache and the TLB hierarchy [12]. Wong et al. demystify the GT200 GPU microarchitecture (TLB and caches) and evaluate the performance impact of branch divergence through benchmarking [6]. Baghsorkhi et al. test the C2050 GPU and disclosed the L1/L2 data cache [13]. Meltzer

[1] https://github.com/yynscfmq/warp-level-benchmark.

et al. investigate the L1/L2 data cache of the C2070 GPU, and observe that the L1 data cache does not use the least recently used (LRU) replacement policy [14]. Mei et al. measure the latencies of the memory systems on GTX560Ti and GTX780 [15]. Further, they propose a fine-grained pointer-chasing micro-benchmark to explore cache parameters, and quantify the throughput and the accessing latency of global memory and shared memory [16]. To summarize, these works focus on benchmarking the GPU memories at the per-thread level. But a GPU often has more than thousands of threads running concurrently, which leads to a context entirely different from that using a single thread. Thus, the per-thread approach is limited in revealing useful information for real-world applications. Instead, our approach focuses on performance behaviours at the warp level on GPU memory systems.

GPU memory optimizations: Both data placement and memory access patterns play an important role on GPU memory optimizations. In terms of data placement, Ma et al. obtain the optimal data placement for shared memory [17]. Chen et al. present PORPLE–a portable data placement engine that enables a new way of solving the data placement problem [8]. In terms of memory access patterns, Yang et al. propose a GPGPU compiler for memory optimization and parallelism management [18]. Zhang et al. design a pipelined online data reorganization engine to reduce memory access irregularity [19]. Wu et al. reorganize data elements to minimize non-coalesced memory accesses on GPU by complex code analysis and algorithm design [20]. Jang et al. investigate memory access patterns for various GPU memory systems, and present a vectorization algorithm for the AMD GPUs and a memory selection algorithm for the NVIDIA GPUs [7]. These works focus on shared memory, constant memory, global memory and texture memory. But we notice no discussion about local memory and/or registers. In this paper, we investigate how to replace local memory with registers, which is a complementary of these existing research.

Warps and SIMT are two key concepts to achieve high performance on GPUs. In [28] , the authors propose a GPU cache model based on the reuse distance theory with the consideration of warps and the parallel execution model (SIMT). Cui et al. discuss the effective control flow divergence optimization on the SIMT execution style [29]. Hong et al. propose a simple analytical model that estimates the execution time of massively parallel programs [30]. Its key component is to estimate the number of parallel memory requests. In [31] , the authors improve the GPU efficiency with a combination of spatiotemporal SIMT and scalarization. As can be seen in these works, warps and SIMT play important roles on GPU. In this paper, we focus on the warp-level latency of GPU memory systems.

3. Background: GPU memory and its latency

In this section, we first describe a typical GPU memory system and its organization, and then measure the thread-level accessing latency.

3.1. Overview of a GPU memory system

There are several types of programmable memories on GPU: register, local memory, shared memory, constant memory, global memory, and texture memory. How to use these memories efficiently is key to optimize GPU codes. Fig. 1 is an overview of the NVIDIA Kepler GPU memory hierarchy. We see that registers and local memory are private to threads. Registers are located in on-chip, whereas local memory is allocated in the device memory. The compiler (NVCC) maps the private variables to either registers or local memory (if there is a spill). Shared memory and L1 cache use the same memory space (sized of 64 KB) with a configurable partitions. Different from the L1 cache, shared memory is programmable and is visiable by all threads within a thread block. Constant cache is also located on-chip, in which the cacheable data is read-only. Global memory and texture memory are located in the GPU's device memory. On the host side, there are two types of memories: pageable memory and pinned memory. The pageable buffer can be swapped out of memory, while the pinned buffer is locked in memory during kernel execution.

3.2. Thread-level latencies of GPU memory systems

In this section, we quantitatively benchmark the latency of GPU memories with the *pointer-chasing* technique. We generate the input data array by allocating an integer array and filling the content of the array index by (*idx*) with (*idx+step*). The basic idea of building a *pointer-chasing* benchmark is that *the next accessing index is equal to the value read in the current iteration, i.e.,* $p = array[p]$. The micro-benchmark is designed as a dependent statement sequence on the array to avoid compiler optimizations.

The latency measurement for registers is different in that no array can be allocated in registers with the memory access pattern of pointer-chasing (p=$array[p]$). Thus we should redesign the latency measurement scheme for registers: each access

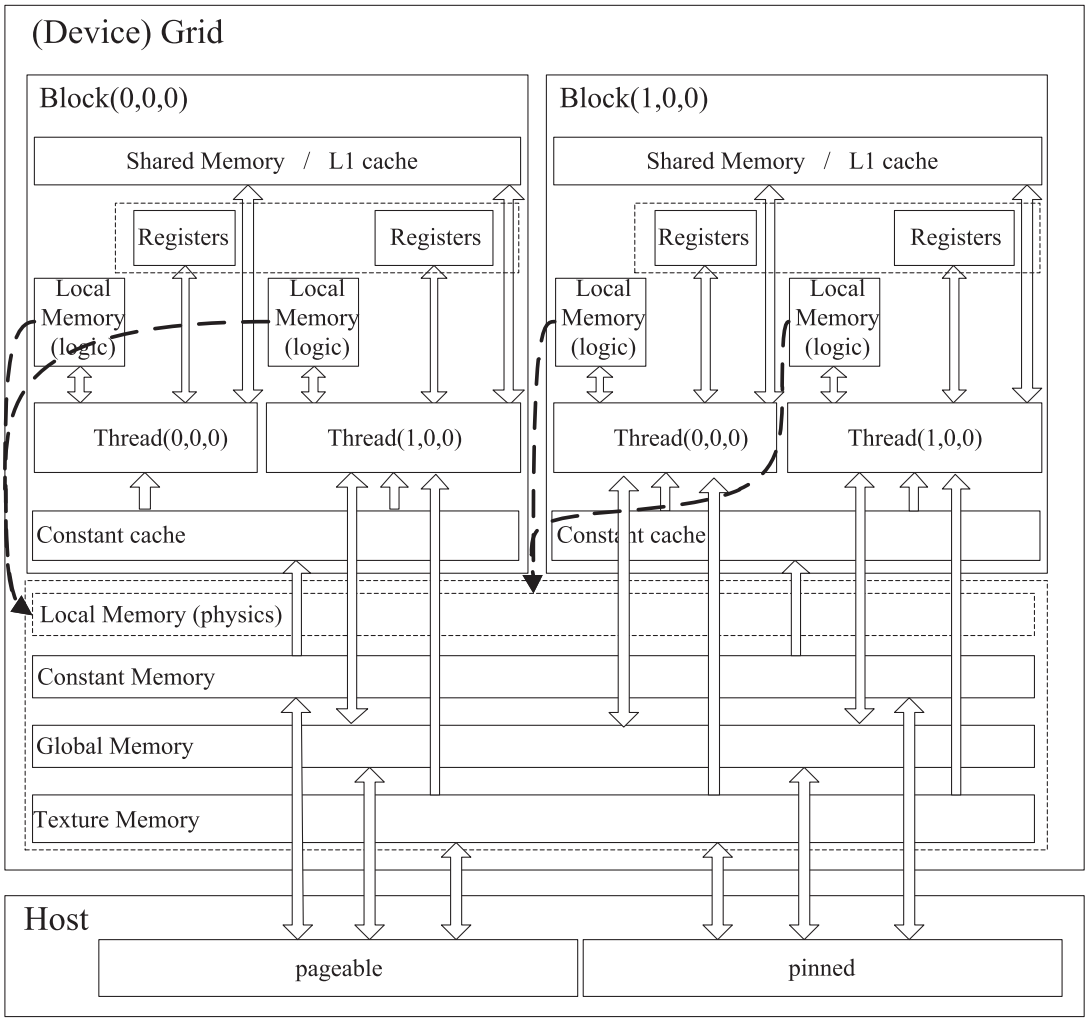

Fig. 1. The overview for GPU memory hierarchy.

depends on the lastest data. At the end of measurement, we write the variable *r* into global memory, to avoid the code being optimized out by the NVCC compiler.

Register Latency

```
1  void register_latency(out, time){
2    int r, x = 2, y = 5, z = 7, q = 1, p = 3;
3    t0 = clock();
4    for i=0 to NUM-1
5      r = p; p = q; q = x; x = y; y = z; z = r;
6    t1 = clock();
7    out[0] = r;
8    time[0] = (t1 - t0)/NUM;
9  }
```

We set the array size to be 2048, and measure the latency of all the memories with steps changing from 1 to 1024. The average latency numbers are shown in Table 1. The latency of accessing registers is so small that we need not take it into account. Shared memory is located on-chip, and its accessing latency is around 48 cycles. The data in constant memory or texture memory is cachable, and the latency is around 110 $\sim$ 115 cycles. Local memory and global memory are located in

Table 1
Thread-level latencies on GPU memories.

memory	register	constant	shared	local	global	texture
Latency/clocks	0.03	110.13	47.69	203.75	218.86	115.6

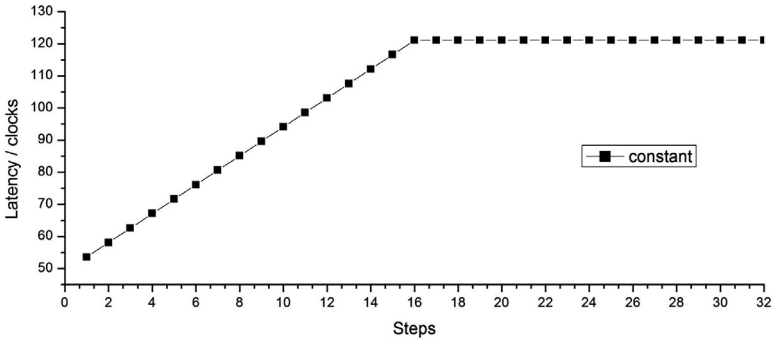

Fig. 2. The latency of constant memory (The x axis denotes the size of accessing steps, and the y axis shows the thread-level latency of constant memory).

the device memory space, and the latency ranges from 203 to 218 cycles. Further, accessing local memory is slightly faster than accessing global memory. We also notice that, the latency of constant memory increases over the accessing steps, which is shown in Fig. 2. But it remains unchanged when steps is larger than 16. By comparison, the latency of accessing texture memory remains unchanged over the step size. Thus, when configuring the constant memory, the accessed data should be stored consecutively.

4. Our warp-Level benchmarking approach

In this section, we propose our warp-level latency benchmarking, and investigate the data accessing capability of multiple threads. And then, we introduce our experiments for warp-level benchmarking: broadcasting and parallel accessing, consecutively and aligned accessing constraints.

4.1. The warp-level latency on GPU

As a coprocessor, the GPU execution model is different from that of CPU. On CPU, scalar operations run in a single instruction single data (SISD) manner, and vector operations run in a single instruction multiple data (SIMD) manner. By contrast, the kernels run in a SIMT fashion on GPUs, that all the threads within a warp run the same instruction. Despite that the threads are capable of executing different code paths, they will be serialized. Thus, warp (32 threads) is regarded as the basic execution unit on GPU.

All the 32 threads within a warp execute the same instruction when there is no divergence, but they may access different data elements. When there is a branch divergence, the threads from one branch will run simultaneously, while the threads from other branches have to wait. In this work, we investigate the data accessing capability of a warp of threads: *broadcasting* and *parallel accessing. Broadcasting* occurs when multiple threads access the same data element, i.e., multiple threads request a single data element (MTSD). We refer the case of multiple threads accessing multiple distinct data elements (MTMD) as *parallel accessing*. We argue that broadcasting and parallel accessing are two important access patterns of GPU memories. In the next section, we will design micro-benchmarks for them.

4.2. Designing micro-benchmarks

Fig. 3 shows how threads access data elements with various patterns, where the squares represent data elements, and the arrows represent threads in a warp. In pattern ①, each thread accesses distinct data elements concurrently, and there is no broadcasting (the broadcasting degree is 1). In pattern ②, every two threads access the same data element, and the broadcasting degree is 2. In pattern ③, all threads access the same data, and the broadcasting degree is the number of threads within a warp. Note that we can derive many more access patterns when varying the broadcasting degree.

Based on this idea, we design micro-benchmarks for broadcasting and parallel accessing. The pseudo kernel code for the micro-benchmark is shown in the following, where *step* denotes the broadcasting degree (in {1, 2, 4, 8, 16, 32}). We initialize

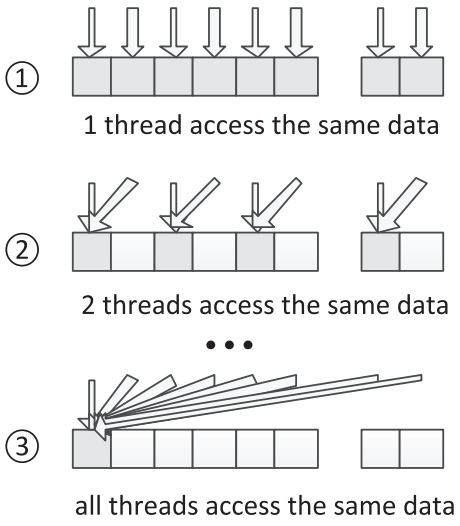

Fig. 3. Memory access patterns.

the input data as $in1[i] = i$ and $in2[i] = i$. The output result of *out* is introduced to avoid compiler optimizations. *NUM* is the number of iterations of memory accessing.

The Warp-level Benchmark

```
1  //Input: int *in1, int *in2, int step, int NUM
2  //Output: int *out, double *time
3  void warp_bench(time){
4    int p, q = (threadIdx.x/step * step);
5    int t0 = clock();
6    for i = 0 to NUM
7      p = in1[q]; q = in2[p];
8    int t1 = clock();
9    out[blockDim.x * blockIdx.x + threadIdx.x] = p + q;
10   time[blockDim.x * blockIdx.x + threadIdx.x] = (t1 - t0)/NUM/2;
11 }
```

From the experimental results, we aim to obtain derivations with the following rules: (1) If the latency of ①,②and ③ decreases one by one, we assume that the memory does not support parallel accessing, but it supports broadcasting. (2) If the latency of ①,②and ③ increases one by one, we assume that the memory supports parallel accessing, but it does not support broadcasting. (3) If the latencies are equivalent (or approximate), the memory may either support both broadcasting and parallel accessing or support none. We should compare the warp-level latency with the thread-level latency. If the warp-level latency is equal to the thread-level latency, the memory supports both; otherwise, we assume that the memory supports neither.

4.3. Accessing constraints

Once we find that the memory supports parallel accessing, we will further explore the performance impact of different memory accessing constraints. In particular, we focus on two constraints: (1) Whether the non-consecutive memory access performs differently from the consecutive one? (2) Whether the misaligned access performs differently from the aligned access? To this end, we refine the micro-benchmark to measure the performance impact of non-consecutive and/or misaligned memory accesses.

Our refined micro-benchmark is shown in Fig. 4, which corresponds to three refined memory access patterns: ① The data elements are consecutive, and the threads access them in an aligned manner. ② The accessed data elements are consecutive, but the threads access them in a misaligned manner. ③ The accessed data elements are non-consecutive, and the threads access the aligned memory. For the micro-benchmark, the key step is to structure the input data, and to enumerate the

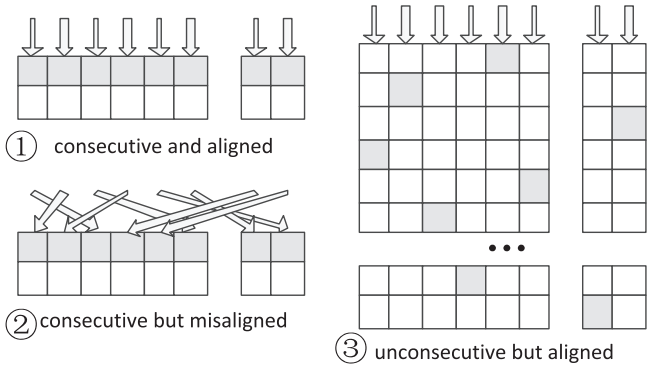

Fig. 4. Consecutively and aligned accessing constraints.

allocated memory space by pointer-chasing. For case ①, we assign the data content with the corresponding index when structuring the input data. When structuring the input data for case ②, we use the following method.

Data initialization for case ②

```
void data_initialization_2(){
  for i = 0 to 32
    p[i] = i;
  for i = 0 to n
    for j = 0 to 32
      int jj = rand()%(32 - j);
      a[i + j] = p[jj];
      for k = jj to (32 - j)
        p[k] = p[k + 1];
    for j = 0 to 32
      p[j] = a[i + j];
      a[i + j] = a[i + j] + i;
    i = i + 32
}
```

For case ③, we allocate a 2D matrix, in which each 32 elements in a row are accessed by 32 threads in a warp. The accesses per iteration are not consecutive, and each thread accesses a column of the 2D matrix. The pseudo code is shown as follows.

Data initialization for case ③

```
void data_initialization_3(){
  const int nn = n/32;
  int q[nn], b[n], p[32];
  for i = 0 to nn
    q = i;
  for i = 0 to n
    for j = 0 to nn
      int jj = rand()%(nn - j);
      b[i + j] = q[jj];
      for k = jj to (nn - j)
        q[k] = q[k + 1];
    for j = 0 to nn
      q[j] = b[i + j];
    i = i + nn
  for i = 0 to 32
    p[i] = i;
  for i = 0 to 32
    for j = 0 to nn
      a[j * 32 + i] = b[i *  nn + j] * 32 + p[i];
}
```

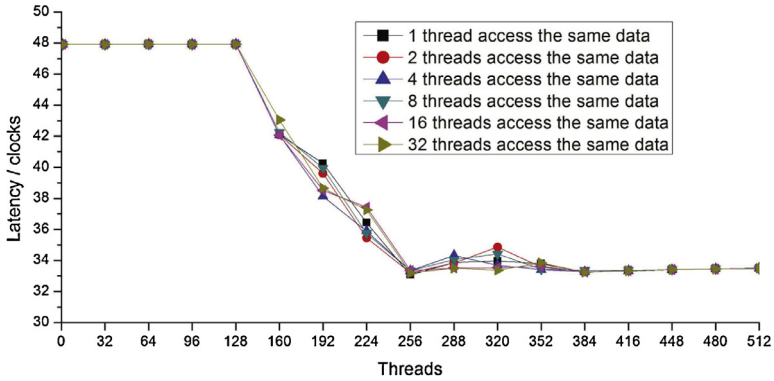

Fig. 5. Broadcasting and parallel accessing on shared memory (The x axis enumerates the threads number in a block, and there is only one block in a grid when the kernel is running; The y axis shows the warp-level latency).

By analyzing the benchmarking results with the consecutive and aligned accessing constraints, we aim to derive more memory accessing properties with the following rules: (1) We can confirm whether threads accessing aligned memory will affect the performance, by comparing the latencies of ① and ②. If ① is faster than ②, we argue that aligned accessing is necessary for this memory. (2) We can confirm whether the non-consecutive accesses will influence performance by comparing the latencies of ① and ③. While ① is faster than ③, consecutively accessing is a must.

5. The warp-level benchmarking results

With the micro-benchmarks, we measure the warp-level latency of shared memory, constant memory, global memory and texture memory on the NVIDIA Tesla K20c GPU, aiming to disclose more information. For each type of memory, we benchmark the accessing capability (*broadcasting* and *parallel accessing*) by measuring the warp-level latency, and measure the performance impact of accessing constraints.

5.1. Benchmarking shared memory

Accessing Capability. We run the micro-benchmarks on shared memory when using 1, 2, 4, 8, 16 or 32 threads to access the same data element. Fig. 5 shows that it takes almost the same time for different threads to access the same data element. In addition, the warp-level latencies are equal to the thread-level latencies (around 48 cycles). According to the rules in Section 4.2, we know that shared memory supports both parallel accessing and broadcasting. As we know, the shared memory space are divided into 32 banks and the data elements located in different banks can be accessed concurrently.

When the number of threads per block increases from 32 to 128, the latency of accessing shared memory remains constant. This is due to the fact that the Kepler architecture features four warp schedulers and eight instruction dispatchers, allowing four warps (i.e., 128 threads) to be issued and executed concurrently. When using over 128 threads, the data loading from one warp overlaps that of another. This is why the latency of accessing shared memory decreases from 128 threads to 256 threads. When over 256 threads are used, the latency of accessing shared memory remains constant again. At the time, the overlapping pipeline of loading data from shared memory is full and scheduling more warps cannot brings an increase in performance.

Accessing Constraints. The warp-level latency results with the refined micro-benchmarks are shown in Fig. 6. We see that the latency follows the same trend with Fig. 5. Also, we observe that all the three patterns take almost the same time. According to the rules in Section 4.3, we infer that neither consecutively accessing nor aligned accessing is a must when using shared memory. That is, a warp of threads can efficiently access the data elements located shared memory without any constraint (bank conflicts excluded).

5.2. Benchmarking constant memory

With 1, 2, 4, 8, 16 and 32 threads in a warp accessing the same data element, we measure the warp-level latency (Fig. 7(a)). Then, we measure the latency numbers with the thread block size of 32, 256, 512 or 1024 (Fig. 7(b)). In Fig. 7(a), the warp-level latency varies significantly. The more threads accessing the same data element, the smaller the latency is. Therefore, constant memory supports broadcasting. But constant memory does not support parallel accessing. That is, constant memory can only be accessed serially when requesting different data elements. On the one hand, constant memory is used to store a small amount of read-only data, which is not sensitive to bandwidth. So parallel accessing is not a must for constant memory. On the other hand, the constant memory is accessed by all the threads in all SMs simultaneously, and thus broadcasting is necessary.

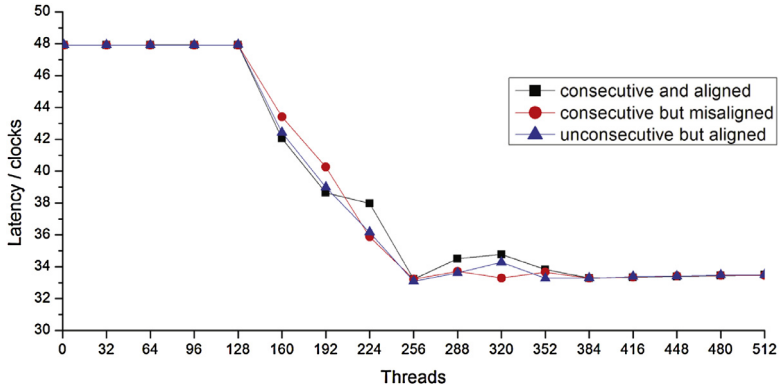

Fig. 6. Consecutively and aligned accessing on shared memory.

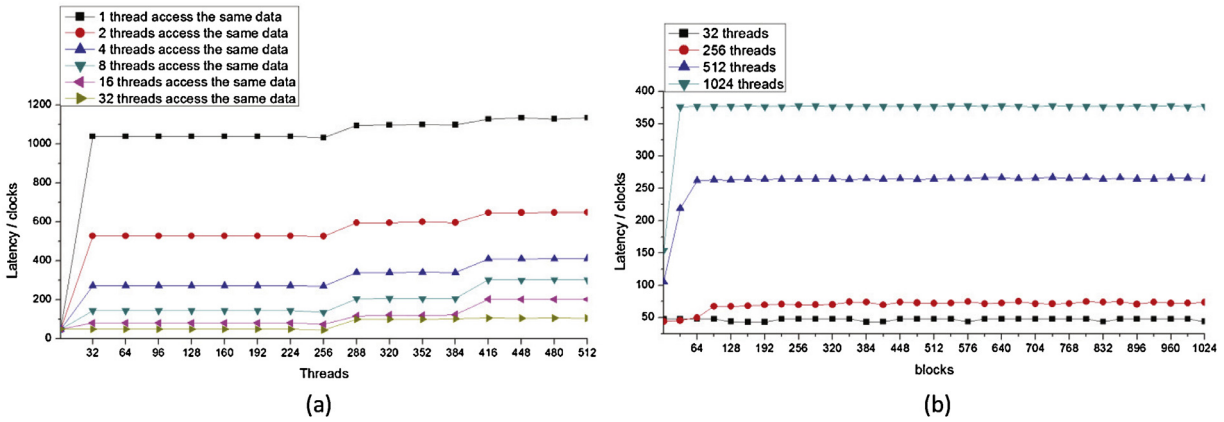

Fig. 7. Broadcasting and parallel accessing on constant memory (In figure (b), the x axis enumerates the blocks number; The y axis shows the warp-level latency).

When the number of threads changes, the memory access latency remains invariant. There are some threshold values, such as 256 and 384. When the threads number is less than the threshold value, the latency is invariant. And when the number of threads increases, the latency increases first and becomes constant thereafter. These results show the decrease of the broadcasting performance on constant memory when increasing the number of warps.

From Fig. 7(b), we have the following observations: (1) There is a threshold value. When the number of blocks is less than the threshold, the latency increases with blocks. And the latency keeps invariant when the number of blocks is larger than the threshold. (2) The threshold value is related to the number of threads within a block. (3) The only factor that affects the invariant latency is the thread number in blocks. The more threads in blocks, the larger the latency is. Thus, threads configuration in a block should be optimized, and blocks configuration is not a must.

5.3. Benchmarking global memory

Accessing Capability. We run the broadcasting and parallel accessing experiments on global memory. Fig. 8 shows that the latency of accessing global memory with multiple threads is almost the same as that with a single thread (218 cycles). And the variance among different number of threads is around 10 cycles, which can be ignored when compared with the thread-level latency. Therefore, we conclude that global memory supports both broadcasting and parallel accessing. The accessing latency becomes smaller when more threads access the same data element, i.e., the broadcasting latency is the smallest among all. This is because only one memory accessing request is generated for broadcasting, whereas multiple are generated for parallel accessing.

Accessing Constraints. We further run the consecutively and aligned accessing experiments on global memory, as shown in Fig. 9. We see that the latencies of ① and ② are equal to each other. According to the rule in Section 4.3, the aligned accessing constraint on global memory is not a must. Meanwhile, the latency of ③ is larger than ① and thus consecutively accessing on global memory is necessary to enhance performance. Fig. 9(a) shows the latency of non-consecutive accesses is 2 × as large as that of consecutive accesses. And the warp-level latency of non-consecutive memory increases over threads

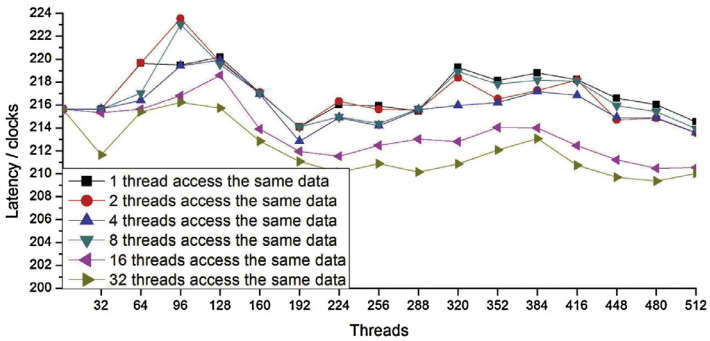

Fig. 8. Broadcasting and parallel accessing on global memory.

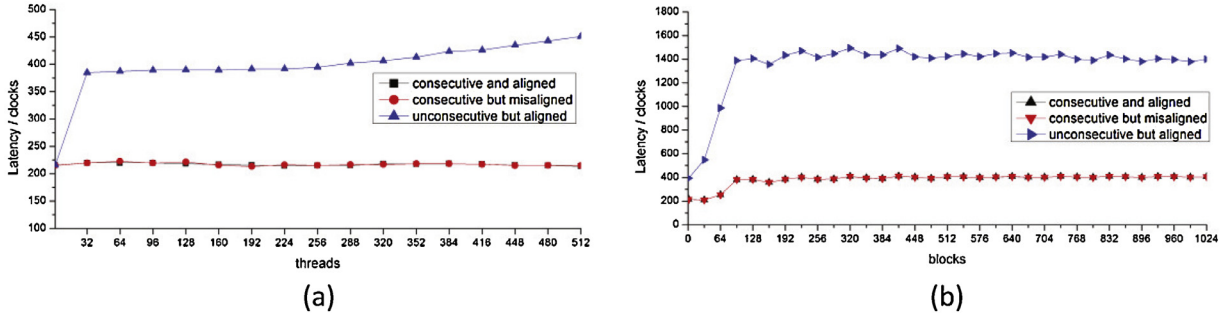

Fig. 9. Consecutively and aligned accessing on global memory (In figure (b), the x axis enumerates the blocks number with 256 threads in each block; The y axis shows the warp-level latency).

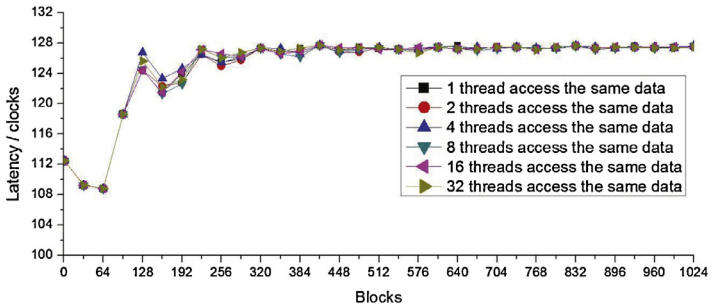

Fig. 10. Broadcasting and parallel accessing on texture memory.

per warp. Fig. 9(b) shows the warp-level latency increases and then stabilizes. The latency of non-consecutive accesses is 4 times of consecutive access when using different blocks. Therefore, non-consecutive accesses severely degrade performance, which should be avoided. If data elements are stored consecutively, they are packed and loaded in fewer requests; otherwise, many more transactions are generated to load the required data.

5.4. Benchmarking texture memory

Accessing Capability. We run the broadcasting and parallel accessing micro-benchmarks on texture memory, with 1, 2, 4, 8, 16 and 32 threads in a warp accessing the same data element. We observe that all the latency numbers are constant at around 110 cycles. According to the rule in Section 4.2, we obtain that texture memory supports both broadcasting and parallel accessing. By changing the number of blocks (each having 256 threads), we show the warp-level latency in Fig. 10. The results show all the warp-level latency numbers of different threads accessing the same data are equal. And the warp-level latency is almost equal to the thread-level latency of texture memory. When less than 96 blocks are used, the latencies are around 110 cycles. When more than 96 blocks are used, the warp-level latency increases and stabilizes at 127 cycles. The threshold of 96 blocks is small and the gap between 110 cycles and 127 cycles is not large, so configuring the number of blocks number is not critical.

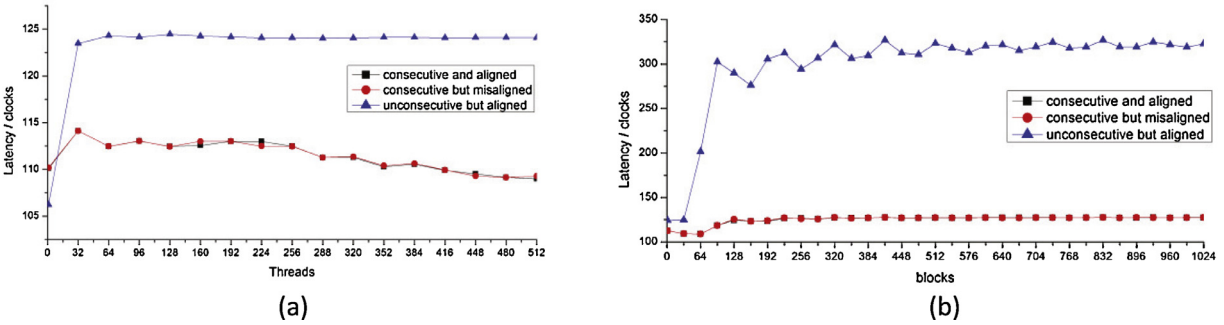

Fig. 11. Consecutively and aligned accessing on texture memory.

Table 2
Warp-level benchmarking results of GPU memories.

memory	shared	constant	global	texture
broadcast	support	support	support	support
parallel	support	not support	support	support
aligned	no impact	N/A	no impact	no impact
consecutively	no impact	N/A	large impact	small impact

Accessing Constraints. We run the consecutively and aligned accessing experiments on texture memory and measure the warp-level latency in Fig. 11. Fig. 11(a) shows the warp-level latency numbers of different threads in a block. We see that ① and ② are equal, so aligned accessing is not a must. The warp-level latency of ③ is larger than ①, and the gap is around 12 cycles, which is neglectable when comparing with the latency of texture memory. Consecutively accessing may bring a performance improvement when using texture memory, but the benefit is not expected to be large.

In Fig. 11(b), we measure the warp-level latency numbers of using different blocks (256 threads for each). When less than 32 blocks are used, the warp-level latency of ③ is constant and small. The latencies of ③ increase remarkably with the number of blocks between 32 and 96. The latencies keep constant at around 320 cycles when over 96 blocks are used. Because using more than 96 blocks leads to much large latency for the non-consecutive access, we should avoid the non-consecutive access on texture memory.

5.5. Summary

We benchmark the warp-level latencies of shared memory, constant memory, global memory and texture memory in terms of the broadcasting and parallel accessing capability. Also, we measure the performance impact of consecutively and aligned accesses. Table 2 summarizes the characteristics of each memory. We see that broadcasting is supported by all memory types, while parallel accessing is supported by all except constant memory. The aligned accessing constraint is not a must. Guaranteeing the consecutive constraint is unnecessary when using shared memory or texture memory, while it is necessary when using global memory.

6. Performance tradeoffs on GPU memories

In addition to benchmarking the GPU memory system at the warp level, we investigate the performance tradeoffs on GPU memories. In this section, we discuss how to replace local memory with registers for a private array, how to avoid bank conflicts of shared memory, and how to maximize the global memory bandwidth with different data types.

6.1. Replacing local memory with registers

As we all know, the thread-private array can be allocated in either registers or local memory. But it is not clear how the private array is allocated. To this end, we investigate when and how to replace local memory with registers. Registers and local memory are functionally equivalent. Local memory is taken as an extension to registers. Using local memory is necessary, when there is no sufficient register or the memory access pattern is complex. We have benchmarked the latencies of registers and local memory in Table 1. We see that the latency of registers is neglectable, whereas the latency of local memory equals to that of global memory. This large latency disparity between registers and local memory motivates us to replace local memory with registers.

There are two types of temporal thread variables: *scalar variables* and *arrays*. Scalar variables are typically defined in registers, while the arrays can be allocated either in registers or in local memory. NVIDIA has not disclosed the policies of

Table 3
The factors that influence the temporal array allocation.

Memory accessing pattern	sm_10	sm_35
vector addition	29	73
matrix multiplication	2*2	6*6
vector sorting	0	0

array allocation (in registers or in local memory). For that, we design three memory access patterns with different complexity, control the array size, and set up the CUDA computing capability in the compiling time. The three memory access patterns are (1) vector addition, in which there is only one loop in the code; (2) matrix multiplication, which is more complex than vector addition and has three nested loops; (3) vector sorting, in which the array elements need to be shuffled.

We record the array size when using different memory access patterns and different CUDA computing capabilities. The largest size that an array can be allocated in registers is shown in Table 3. We conclude that (1) the simpler the memory access pattern is, the larger the array can be allocated in registers; (2) The latest CUDA computing capabilities contribute to a larger array allocated in registers; (3) For the complex memory access patterns (e.g., vector sorting), the temporary array can only be allocated in local memory.

6.2. Bank conflicts on shared memory

Shared memory is partitioned into equally-sized memory modules, called banks, which can be accessed simultaneously. Each bank is of 4 bytes in width [11] , and there are 32 banks in the device whose capability is newer than SM 2.0. The memory accesses on different banks can run simultaneously. However, if two addresses of memory requests fall into the same memory bank, there will be a bank conflict and the memory access will be serialized.

We quantify the performance impact of bank conflicts with the pseudo code shown as follows. The bank conflict degree is set to be 1, 2, 4, 8, 16 and 32. Variable q is the index of the accessed array in shared memory. The input data structures of *in1* and *in2* are initialized as $in1[i] = i$ and $in2[i] = i$. The array of *smem1* and *smem2* are allocated in shared memory. We calculate the results of *p+q* to avoid compiler optimizations.

Latency for bank conflicts

```
#define conflictnum 1 //2, 4, 8, 16, 32
#define REP 128
//Input : int *in1, int *in2, int its
//Output : int *out, double *time
void latency_bankconflicts(time){
  int p; double time temp = 0;
  int q = (threadIdx.x%conflictnum) * 32 + (threadIdx.x/conflictnum);
  __shared__ int smem1[SMEMSIZE];
  __shared__ int smem2[SMEMSIZE];
  for tid = threadIdx.x to SMEMSIZE by blockDim.x
    smem1[tid] = in1[tid];
    smem2[tid] = in2[tid];
  for i = 0 to its
    __syncthreads();
    double time_start = clock();
      for j = 0 to REP
        p = smem1[q];
        q = smem2[p];
    double time_end = clock();
    time_temp = time temp + (time_end - time_start);
  time[blockDim.x * blockIdx.x + threadIdx.x] = time_temp/REP/its;
  out[blockDim.x * blockIdx.x + threadIdx.x] = p + q;
}
```

We run the experiments on the Tesla K20c GPU, and show the results in Fig. 12. We see that the memory access latency increases over the degree of bank conflicts. Therefore, bank conflicts should be carefully avoided when we optimizing GPU applications. Data padding is one technique to remove the bank conflicts. In general, accessing a column of a 2D array in shared memory may lead to 31-way bank conflicts. The method of avoiding bank conflicts is to add an padding element at the end of each row. As seen in Fig. 13, all the elements in the same column are located in different banks, so they can be accessed simultaneously. We substitute the [32][32] 2D array in shared memory with a [32][33] array. The data padding

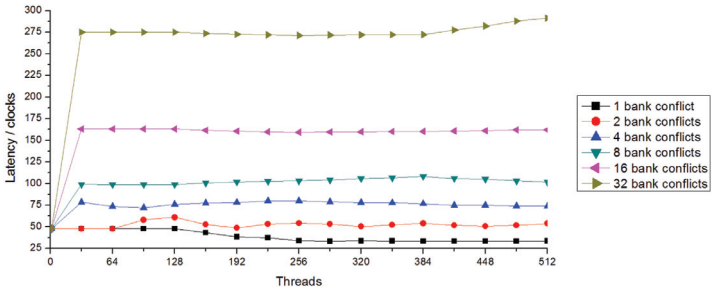

Fig. 12. Warp-level latency influenced by bank conflicts.

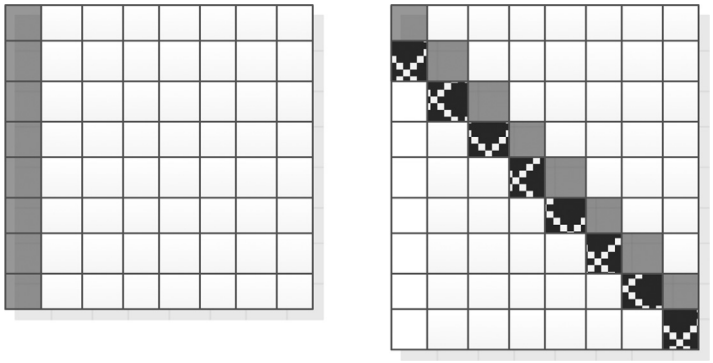

Fig. 13. Data padding method to remove bank conflict.

Table 4
Bandwidth of different types on global memory (GB/s).

data type	blocks	threads	Measured	D2D Memcpy
float	(64,64)	(32,32)	151.492	159.744
double	(1024,1024)	(16,16)	157.146	158.989
int	(64,64)	(32,32)	151.378	159.757
char	1024	1024	63.02	159.337
char4	(64,64)	(32,32)	148.599	159.286

scheme is used in both matrix transposition and scanning calculation of brent-kung method, and thus bank conflicts can be successfully avoided [9].

6.3. Bandwidth of data types on global memory

We investigate the factors that impact the achieved bandwidth on global memory. Specifically, we measure the global memory bandwidth with different data types (*float, double, integer, char* and *char4*) and various thread configurations. We record the best thread configuration and its global memory bandwidth in Table 4. From the table, we confirm that the bandwidth obtained by D2D cudaMemcpy() will reach its maximum at 160GB/s. Using the data types of 4 bytes (e.g., *float, integer* and *char4*) can reach a bandwidth of around 150GB/s, and using the *double* data type can obtain a bandwidth of 157GB/s. Nevertheless, using the *char* data type can only obtain a bandwidth of 63GB/s. Therefore, four *char* data elements should be coalesced into a *char4* element for improved memory bandwidth.

7. Towards an optimization framework on GPU memories

With the warp-level benchmarking results and analysis, we derive optimizations guidelines on the GPU memory system.

7.1. On local memory

(L_1) For the simple memory access patterns, we should allocate a sufficient small array to guarantee that it is located in registers.

(L_2) For the complex memory access patterns, we should simplify codes to exploit registers. For example, we merge a three-level loop into an one-level loop so that a larger temporal vector can be allocated in registers.

(L_3) For the memory access patterns such as vector sorting, we should use the simple and equivalent operations. By doing so, the temporary vector can be allocated in registers. For example, we replace a median filtering with a mean filtering to ensure that the temporal array is allocated in registers.

(L_4) By using the latest GPU computing capability, we can allocate the temporary array in registers instead of local memory without code changes.

7.2. On shared memory

(S_1) Bank conflicts must be avoided by the ways of e.g., data padding.

(S_2) Shared memory supports both broadcasting and parallel accessing.

(S_3) Neither consecutively accessing nor aligned accessing is a must.

(S_4) The latency decreases when the number of threads increase, and thus we should use a sufficiently large thread block.

(S_5) Replacing global memory with shared memory, because the latency of shared memory is smaller than that of global memory.

(S_6) Using shared memory bares an overhead (i.e., buffer allocation and data movement) and reusing data in it is a must for improved performance.

7.3. On constant memory

(C_1) We should reorganize data structures on constant memory, so that the data can be accessed consecutively.

(C_2) Constant memory supports the accessing capability of broadcasting.

(C_3) Constant memory does not support parallel accessing, and satisfies parallel memory requests in a serial manner.

(C_4) Thread configuration should be optimized for constant memory.

(C_5) The number of thread blocks has a very slight performance impact when using constant memory.

7.4. On global memory

(G_1) Global memory supports both broadcasting and parallel accessing.

(G_2) The data types of 4 or 8 bytes can obtain the near upper-bounded bandwidth of global memory, while the data types cannot. So the *char* data should be coalesced into the *char4* type for improved bandwidth.

(G_3) Global memory accesses should be consecutive, but aligned accessing is not necessary for global memory.

(G_4) When memory accessing is non-consecutive, the latency changes with the number of threads, but not with the number of blocks. So we should configure the thread dimensionality.

7.5. On texture memory

(T_1) Texture memory supports broadcasting and parallel accessing.

(T_2) Aligned accesses are not a must on texture memory.

(T_3) Consecutively accessing may contribute to the texture memory accessing. The latency is almost invariant when threads and blocks changing within consecutively accessing.

(T_4) When the memory accessing is non-consecutive, we should use the texture memory, instead of global memory. The latency of texture memory is much smaller, comparing Fig. 9 with Fig. 11.

7.6. Towards a memory optimization framework

By synthesizing the warp-level benchmarking results of GPU memories, the optimization considerations (i.e., the allocation of private temporal arrays on registers or local memory, the bank conflict of shared memory, the bandwidth of data types on global memory), and the optimizations deduced from them, we propose a memory optimization framework for GPUs, as shown in Fig. 14. This framework inputs CUDA codes, analyzes the data accesses of kernels, collects the usage of each type of memories, and identifies the data accessing types (e.g., temporal data, reusable data, and read-only data). Then the optimization methods deduced in our paper are used to optimize these memory accesses. Finally, the framework outputs the optimized codes.

8. Case study: maximum noise fraction rotation

In this section, we apply our memory optimization framework to the maximum noise fraction (`MNF`) rotation for hyperspectral image dimensionality reduction. We describe the `MNF` algorithm, analyze its hotspots, propose some detailed optimizations for each hotspot with our optimization framework, and show the performance improvement by comparing to the state-of-the-art.

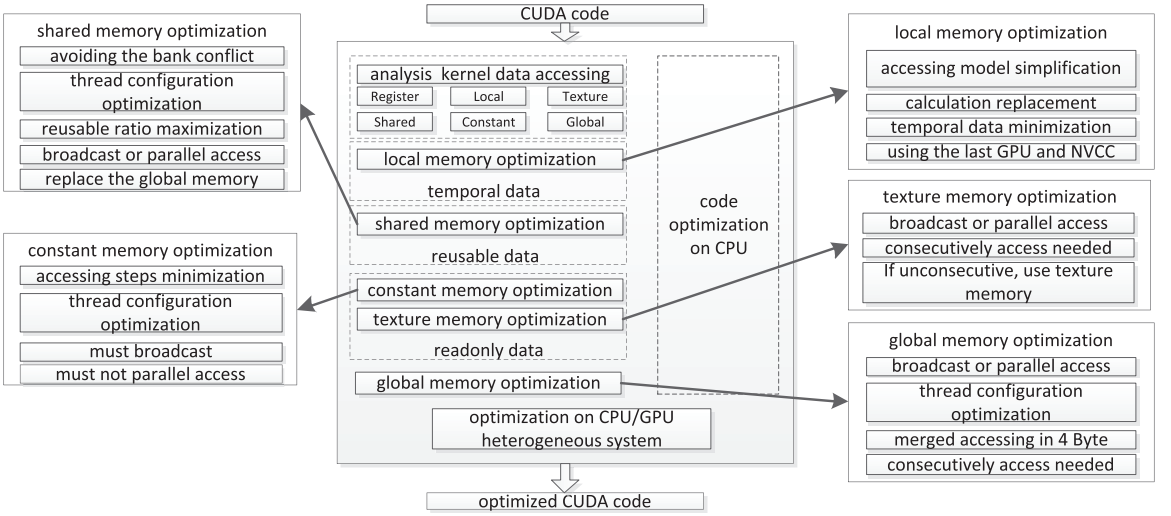

Fig. 14. Memory optimization framework on GPU.

8.1. MNF and its hotspots

MNF is a typical linear algorithm for hyperspectral image dimensionality reduction, which can separate the noise from the original data and extract the main features. The MNF algorithm includes two stages of principal component analysis (PCA) and then considers the impact of noises based on PCA.

The hyperspectral image data is represented by a $B \times S$ matrix $\boldsymbol{X}$, where B is the number of bands and S is the number of pixels in each band. S is computed by $S \leftarrow W \times H$, where W and H denote the width and the height of the input image, respectively. To reduce the dimensionality of a hyperspectral image, the following steps are required:

S1. Noise estimate (matrix $\boldsymbol{F}$) by filtering the hyperspectral image $\boldsymbol{X}$.

S2. Calculating the covariance matrix $\boldsymbol{C}_N$ for the noise matrix.

$$C_N^{ij} = \frac{1}{S-1}\left[\sum_{k=1}^{S}(F_k^i - \bar{F}^i)(F_k^j - \bar{F}^j)\right] \tag{1}$$

S3. Decompose the covariance matrix $\boldsymbol{C}_N$ into an eigenvalue vector $\boldsymbol{D}_N$ and an eigenvector matrix $\boldsymbol{U}$:

$$D_N = U^T C_N U \tag{2}$$

And then calculate the transformation matrix $\boldsymbol{P}$:

$$P = U D_N^{-\frac{1}{2}} \tag{3}$$

S4. Calculate the covariance matrix $\boldsymbol{C}_D$ for the hyperspectral image matrix $\boldsymbol{X}$.

$$C_D^{ij} = \frac{1}{S-1}\left[\sum_{k=1}^{S}(X_k^i - \bar{X}^i)(X_k^j - \bar{X}^j)\right] \tag{4}$$

S5. Transform for the matrix $\boldsymbol{C}_D$.

$$C_{D-adj} = P^T C_D P \tag{5}$$

S6. Decompose the covariance matrix $\boldsymbol{C}_{D-adj}$.

$$D_{D-adj} = V^T C_{D-adj} V \tag{6}$$

in which $\boldsymbol{D}_{D-adj}$ is the eigenvalue vector, and $\boldsymbol{V}$ is the eigenvector matrix.

S7. Calculate the transformation matrix $\boldsymbol{T}$:

$$T = PV \tag{7}$$

S8. Calculate the MNF transformation matrix $\boldsymbol{Z}$.

$$Z = T^T X \tag{8}$$

We implement the MNF algorithm and run it with a hyperspectral image with 224 bands and 614×1087 elements in each band. By measuring the running time for each step, we show that the time consumption of the hotspots are noise filtering (S1), covariance matrix calculation (S2 and S4), and MNF transformation (S8). Then, we will focus on tuning the hotspots of the MNF algorithm with our memory optimization framework on GPU.

Table 5
The optimization impact on noise filtering (ms).

Image NO.	V0	V1	V2	V3
1	123.38	25.96 (4.8)	22.00 (5.6)	20.88 (5.9)
2	254.08	57.89 (4.4)	55.55 (4.6)	44.97 (5.6)
3	1011.15	221.36 (4.6)	206.18 (4.9)	168.59 (6.0)

Table 6
Optimization effects of covariance matrix calculation (ms).

Image NO.	V0	V1	V2	V3	V4
1	8262.08	774.98 (10.7)	750.53 (11.0)	432.08 (19.1)	269.42 (30.7)
2	24163.80	1692.44 (14.3)	1649.07 (14.7)	938.11 (25.8)	626.61 (38.6)
3	163899.69	6377.08 (25.7)	6181.60 (26.5)	3537.05 (46.3)	2366.40 (69.3)

8.2. Performance optimization

Based on our GPU memory optimization framework, we design and implement several detailed optimizations for each hotspot. Then we measure the optimization effects with three hyperspectral images on the Tesla K20c GPU. All the three AVIRIS (airborne visible infrared imaging spectrometer) hyperspectral images have 224 bands. Image NO.1 includes 614×1087 elements in each band, image NO.2 includes 753×1924 elements in each band, and image NO.3 includes 781×6955 elements in each band.

8.2.1. Optimizing noise filtering

We map the filtering task of each element in the hyperspcetral image onto a thread of the GPU. Based on this, we optimize the noise filtering with the L_3, S_5, and S_6 optimizations proposed in Section 7.

(1) Replacing local memory with registers (L_3). In the serial code of noise filtering, the median filtering is used. We need the sorting operation when performing the median filtering. As we have shown in Section 5.1, the temporal array cannot be allocated in the register space, but only in the local memory space, with the memory access pattern of sorting. By replacing the median filtering with the mean filtering, we can guarantee that the temporal array is allocated in registers.
(2) Improving data reuse with shared memory (S_5). A row data can be used by 3 rows of the noise filtering calculation. Thus, these data elements are reusable and can be staged in shared memory.
(3) Maximizing data reuse of shared memory (S_6). For a single element in the data matrix, the element itself together with its 8 neighbors are used. In this way, we can improve the reuse ratio of shared memory.

We measure the kernel execution time before and after each optimization using the three hyperspectral images. The performance results are shown in Table 5, in which V0 denotes the one without optimization, and V1 ~ V3 incrementally adopt the aforementioned three optimizations. We see that, all these three optimizations can improve the kernel performance. With our optimizations, we can obtain a speedup of up to $6\times$ comparing with the parallel version on Tesla K20c GPU.

8.2.2. Optimizing covariance matrix calculation

For the covariance matrix calculation, we design the parallel scheme of mapping the covariance elements calculation onto threads rather than blocks, and apply the optimizations of S_5, S_5+S_6, and S_1 mentioned in our GPU memory framework. We propose four optimizations to improve the performance of covariance matrix calculation.

(1) Accelerating reduction of temporal results with shared memory (S_5). Calculating a single covariance is similar to a vector-vector production. Thus, shared memory is used to stage the temporal results.
(2) Allocating the temporal variable onto registers.
(3) Replacing global memory with shared memory for data reuse (S_5), and maximizing the reuse ratio of shared memory (S_6). The covariance matrix calculation is a kind of matrix multiplication. So the input hyperspectral image can be staged in shared memory for data sharing. To maximize the reuse ratio, we propose a scheme, where the threads within a block load the data elements of a small grid into shared memory.
(4) Avoiding bank conflicts of shared memory (S_1). When accessing the data elements in shared memory (optimization 3), bank conflicts might occur. By padding data or using transposition, we can avoid the bank conflicts.

The execution time of the kernel is measured before and after optimizations. The experimental results are shown in Table 6, in which V0 denotes the one without optimization, and V1 ~ V4 apply the four optimizations one by one. Based on the parallel version on GPU, We can achieve a speedup ranging from $30.7\times \sim 69.3\times$ with these optimizations.

When comparing to our previous work on covariance matrix calculation of the PCA [21] and FastICA [22] algorithm, we obtain a further performance improvement using the proposed optimization. In [22] , the covariance matrix calculation

Table 7
Optimization effects of the MNF transformation (ms).

Image NO.	V0	V1	V2
1	137.38	88.84 (1.5)	67.99 (2.0)
2	298.27	200.20 (1.5)	158.67 (1.9)
3	599.02	413.20 (1.4)	273.15 (2.2)

Table 8
CUDA-MNF vs. CUBLAS-MNF (ms).

Image NO.	serial	CUDA-MNF	CUBLAS-MNF
1	62393.56	958.75	2923.20
2	141221.63	1761.99	3969.39
3	538294.40	5796.17	8888.73

consumes 4070.48 ms for the hyperspectral image NO.3, while the same step only takes 2366.40 ms with our optimization framework.

8.2.3. Optimizing MNF transformation

MNF transformation is a kind of matrix multiplication. But CUBLAS has no support of the type *uchar* for this operation, and transforming *uchar* into *float* takes extra time and memory space. So we customize this step on GPU, instead of using the CUBLAS. We optimize the kernel of MNF transformation with C_2, S_5 and S_6 derived in Section 7.

(1) Optimizing the transformation matrix with constant memory (C_2). The size of transformation matrix is $m \times B$, where m is in dozens and B is in hundreds. So the matrix size is sufficiently small to be stored in the constant memory space. Moreover, the matrix is accessed in a broadcasting fashion, which is supported by constant memory.
(2) Optimizing the hyperspectral image matrix with shared memory (S_5 and S_6). The hyperspectral image matrix is stored on global memory, and there are reuses when the hyperspectral image is used in MNF transformation. Therefore, it can be optimized by shared memory.
The kernel execution time is measured before and after optimizations. The results are shown in Table 7, in which V0 does not use any optimization, whereas V1 ∼ V2 adopts the two optimizations one by one. We observe that, these optimizations can speed up the MNF transformation significantly.

8.3. Comparing with the CUBLAS implementation

Based on the aforementioned work, we optimize MNF by simultaneously using the CPU and the GPU. This code version optimized with our framework is denoted as CUDA-MNF. Further, we implement a code version of the MNF algorithm based on CUBLAS, in which *cublasSasum()* and *cublasSsyrk()* are used. When compared with *cublasSegemm()* used for the MNF algorithm in[23] , the *cublasSsyrk()* can achieve a better performance. Noise filtering cannot be implemented with CUBLAS in a straightforward way. Thus, we implement the kernel for filtering with CUDA, and adopt the optimizations in Section 8.1 for filtering on GPU. This code version is labeled as CUBLAS-MNF.

We run these experiments and collect the running time of the serial version, the CUDA-MNF version and the CUBLAS-MNF version. As shown in Table 8, CUDA-MNF can achieve better performance than CUBLAS-MNF, with a speedup of up to 92.9× compared with the serial version. The results demonstrate that our framework is applicable and effective.

8.4. Discussion

By analyzing and optimizing MNF, we obtain a significant speedup for this real-world application. We argue that there are two key factors influencing the speedups: (1) the ratios of the time consumption for hotspots, and (2) the speedups from parallelization as well as optimizations in each hotspot.

The MNF algorithm has a total of 8 steps. Among them, 4 steps take the majority of the overall execution time and thus are parallelized and optimized. The hotspot of noise filtering (S1 in Section 8.1) takes 12.3%, and obtains a speedup of 62.3× by parallelization, and a further speedup of 6.0× times with optimizations when compared with the native version. The hotspot of covariance matrix calculation (S2 and S4) takes almost 82.2%, and obtains a speedup of 3.1× by parallelization, and a further speedup of 30.7× with optimizations. For Image NO.1, the parallel version gets a speedup of 3.6×, and the optimized version obtains a speedup of 65× compared with the serial version. Therefore, both parallelization and optimizations contribute to the performance improvement.

For the scientists from different domains, they should first identify the hotspots of their applications. Then for each hotspot, they design suitable parallel schemes for GPUs. Finally, our GPU memory optimization framework can facilitate the optimization process.

9. Conclusion

In this paper, we propose a warp-level benchmarking approach for GPU memory systems and design a suit of micro-benchmarks. With the micro-benchmarks, we measure the warp-level latency of shared memory, constant memory, global memory and texture memory. The experimental results show that all the memories support broadcasting, and it is only constant memory that does not support parallel accessing. We observe that the aligned accessing constraint is not a must for all memory types. Meanwhile, the consecutive constraint should be guaranteed when using global memory. Moreover, we disclose that there is a performance impact of thread configurations on latency.

By exploring the private temporal array allocation on registers or local memory, we show that keeping simpler memory access patterns, allocating smaller arrays and using the latest CUDA computing capability can increase the possibility of allocating the temporal array on registers. Then we show the warp-level latency is in a positive correlation with the bank conflict numbers, which explains why bank conflicts should be avoided. Also, the *char* data type should be packed to be *char4* type to obtain a larger bandwidth.

By analyzing the optimizations, we propose a memory optimization framework on GPU. We show the applicability and effectiveness of our framework with a case study (`MNF`). With the framework, we can obtain a significant performance improvement when compared with the native parallel code. The proposed benchmarking approach and our experimental results can help with the algorithm design, performance modeling and optimization on GPUs.

For future work, we will extend our work in two folds. On the one hand, we will implement an automated or semi-automated tuning framework to replace manual optimizations. On the other hand, we will extend our framework to other many-core processors (e.g., MIC).

References

[1] S.W. Keckler, W.J. Dally, B. Khailany, et al., GPUs and the future of parallel computing., IEEE Micro (5) (2011) 7–17.
[2] Y. Li, H. Chi, L. Xia, et al., Accelerating the scoring module of mass spectrometry-based peptide identification using GPUs., BMC Bioinf. 15 (1) (2014) 1–11.
[3] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, et al., Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming., ACM, 2008, pp. 73–82.
[4] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: Proceedings of 2nd workshop on general purpose processing on graphics processing units., ACM, 2009, pp. 79–84.
[5] K. Zhao, X. Chu, G-BLASTN: accelerating nucleotide alignment by graphics processors[j]., Bioinformatics 30 (10) (2014) 1384–1391.
[6] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi, et al., Demystifying GPU microarchitecture through microbenchmarking, in: Performance Analysis of Systems & Software (ISPASS), IEEE, 2010, pp. 235–246. 2010 IEEE International Symposium on.
[7] B. Jang, D. Schaa, P. Mistry, et al., Exploiting memory access pat-terns to improve memory performance in data-parallel architectures., in: Parallel and Distributed Systems IEEE Transactions on, 22, 2011, pp. 105–118.
[8] G. Chen, B. Wu, D. Li, et al., Porple: an extensible optimizer for portable data placement on GPU, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture., IEEE Computer Society, 2014, pp. 88–100.
[9] W. Nicholas, The CUDA Handbook: A Comprehensive Guide to GPU Programming., China Machine Press, Beijing, 2014.
[10] B. David, W. Wen-mei, Programing Massively Parallel Processors:A Hands-on Approach, Second Edition., Tsinghua University Press, Beijing, 2013.
[11] 2016, CUDA C programming guide (v8.0). NVIDIA Corporation. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[12] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: High Performance Computing, Networking Storage and Analysis, 2008, IEEE, 2008, pp. 1–11. SC 2008. International Conference for.
[13] S.S. Baghsorkhi, I. Gelado, M. Delahaye, et al., Efficient perfor-mance evaluation of memory hierarchy for highly multithreaded graphics processors, in: ACM SIGPLAN Notices., 47, ACM, 2012, pp. 23–34.
[14] R. Meltzer, C. Zeng, C. Cecka, Micro-benchmarking the c2070, in: GPU Technology Conference, 2013.
[15] X. Mei, K. Zhao, C. Liu, et al., Benchmarking the memory hierarchy of modern GPUs, in: Network and Parallel Computing, Springer Berlin Heidelberg, 2014, pp. 144–156.
[16] X. Mei, X. Chu, Dissecting GPU memory hierarchy through microbenchmarking., 2015, ArXiv preprintarXiv:1509.02308.
[17] W. Ma, G. Agrawal, An integer programming framework for optimizing shared memory use on GPUs, in: High Performance Computing (HiPC), IEEE, 2010, pp. 1–10. 2010 International Conference on.
[18] Y. Yang, P. Xiang, J. Kong, et al., A GPGPU compiler for memory optimization and parallelism management, in: ACM Sigplan Notices., 45, ACM, 2010, pp. 86–97.
[19] E.Z. Zhang, Y. Jiang, Z. Guo, et al., On-the-fly elimination of dy-namic irregularities for GPU computing, in: ACM SIGARCH Computer Architecture News, 39, ACM, 2011, pp. 369–380.
[20] B. Wu, Z. Zhao, E.Z. Zhang, et al., Complexity analysis and algo-rithm design for reorganizing data to minimize non-coalesced memory accesses on GPU, in: ACM SIGPLAN Notices., 48, ACM, 2013, pp. 57–68.
[21] M.Q. Fang, H.F. Zhou, X. Shen, Multilevel parallel algorithm of PCA dimensionality reduction for hyperspectral image on GPU., Dongbei Daxue Xuebao/J. Northeastern Univ. (2010) 238–243. S1.
[22] M.Q. Fang, H.F. Zhou, X. Shen, A parallel algorithm of fastICA dimensionality reduction for hyperspectral image on GPU, Guofang Keji Daxue Xuebao/J. Natl.Univ. Defense Technol. 37 (4) (2015) 65–70.
[23] Y. Wu, L. Gao, H. Zhang, et al., Real-time implementation of opti-mized maximum noise fraction transform for feature extraction of hyperspectral images., J. Appl. Remote Sens. 8 (2014) 1–16.
[24] R.H. Saavedra, A.J. Smith, Measuring cache and TLB performance and their effect on benchmark runtimes., IEEE Trans. Comput. 44 (10) (1995) 1223–1235.
[25] L. Peng, J.K. Peir, T.K. Prakash, et al., Memory hierarchy performance measurement of commercial dual-core desktop processors., J. Syst. Archit. 54 (8) (2008) 816–828.
[26] D. Molka, D. Hackenberg, R. Schone, et al., Memory performance and cache coherency effects on an intel nehalem multiprocessor system, in: Parallel Architectures and Compilation Techniques, IEEE, 2009, pp. 261–270. 2009. PACT'09. 18th International Conference on.
[27] J. Fang, H. Sips, L. Zhang, et al., Test-driving intel xeon phi, in: Proceedings of the 5th ACM/SPEC international conference on Performance engineering., ACM, 2014, pp. 137–148.
[28] C. Nugteren, B.G.J. van den, H. Corporaal, et al., A detailed GPU cache model based on reuse distance theory, in: High Performance Computer Architecture (HPCA), IEEE, 2014, pp. 37–48. 2014 IEEE 20th International Symposium on.

[29] Y. Liang, M.T. Satria, K. Rupnow, et al., An accurate GPU performance model for effective control flow divergence optimization., IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 35 (7) (2016) 1165–1178.
[30] S. Hong, H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: ACM SIGARCH Computer Architecture News., 37, ACM, 2009, pp. 152–163.
[31] J. Lucas, M. Andersch, M. Alvarez-Mesa, et al., Spatiotemporal SIMT and scalarization for improving GPU efficiency, ACM Trans. Archit. Code Optim. (TACO) 12 (3) (2015). 32.