Dissecting GPU Memory Hierarchy through Microbenchmarking

Xinxin Mei, Xiaowen Chu, *Senior Member, IEEE*

Abstract—Memory access efficiency is a key factor in fully utilizing the computational power of graphics processing units (GPUs). However, many details of the GPU memory hierarchy are not released by GPU vendors. In this paper, we propose a novel fine-grained microbenchmarking approach and apply it to three generations of NVIDIA GPUs, namely Fermi, Kepler and Maxwell, to expose the previously unknown characteristics of their memory hierarchies. Specifically, we investigate the structures of different GPU cache systems, such as the data cache, the texture cache and the translation look-aside buffer (TLB). We also investigate the throughput and access latency of GPU global memory and shared memory. Our microbenchmark results offer a better understanding of the mysterious GPU memory hierarchy, which will facilitate the software optimization and modelling of GPU architectures. To the best of our knowledge, this is the first study to reveal the cache properties of Kepler and Maxwell GPUs, and the superiority of Maxwell in shared memory performance under bank conflict.

Index Terms—GPU, CUDA, memory hierarchy, cache structure, throughput

1 Introduction

The past decade has witnessed a boom in the development of general-purpose graphics processing units (GPGPUs). These GPUs are embedded with hundreds to thousands of arithmetic processing units on one die and have tremendous computing power. They are one of the most successful types of many-core parallel hardware and are deployed in a great variety of scientific and commercial applications. The prospect of more thorough and broader applications is very promising [1], [2], [3], [4], [5], [6]. However, their realistic performance is often limited by the huge performance gap between the processors and the GPU memory system. For example, NVIDIA's GTX980 has a raw computational power of 4,612 GFlop/s, but its theoretical memory bandwidth is only 224 GB/s [7]. The realistic memory throughput is even lower. The memory bottleneck remains a significant challenge for these parallel computing chips [2], [3]. The GPU memory hierarchy is rather complex, and includes the GPU-unique shared, texture and constant memory. According to the literature, appropriate leverage of GPU memory hierarchies can provide significant performance improvements [4], [5], [6], [8]. For example, on GTX780, the memory-bound G-BLASTN achieves an overall 14.8x speedup compared with the sequential NCBI-BLAST by coordinating the use of GPU texture and shared memory [4]. On GTX980, the performance of a naive compute-bound matrix multiplication kernel without memory optimization is only 148 GFlop/s, that of a kernel with clever application of shared memory is 598 GFlop/s, and that of a kernel with extremely efficient optimization of memory is as high as 1,225 GFlop/s [9], [10]. Hence, it is vital to expose, exploit and optimize GPU memory hierarchies.

NVIDIA has launched three generations of GPUs since 2009, codenamed as Fermi, Kepler and Maxwell, with compute capabilities of 2.x, 3.x and 5.x, respectively. Compared with its former 1.x hardware, NVIDIA has devoted much effort to improving GPU memory efficiency, yet the memory bottleneck is still a primary limitation [7], [11], [12], [13], [14], [15], [16]. Because NVIDIA provides very limited information on its GPU memory systems, many of their details remain unknown to the public. Existing work on the disclosure of GPU memory hierarchy is generally conducted using third-party benchmarks [17], [18], [19], [20], [21], [22]. Most of them are based on devices with a compute capability of 1.x [17], [18], [19], [20]. Recent explorations of Fermi architecture focus on a part of the memory system [21], [22]. To the best of our knowledge, there are no state-of-the-art works on the recent Kepler and Maxwell architectures. Furthermore, the above benchmark studies on GPU cache structure are based on a method that was developed for early CPU platforms [23], [24] with a simple memory hierarchy. As memory designs have become more sophisticated, this method has become out of date and inappropriate for current generations of GPU hardware [25].

In this paper, we investigate the GPU memory hierarchy of three recent generations of NVIDIA GPUs: Fermi, Kepler and Maxwell. We investigate them using a series of microbenchmarks targeting their cache mechanism, memory throughput, and memory latency. In particular, we propose a fine-grained pointer chasing (P-chase) microbenchmark, which reveals that many of the characteristics of a GPU cache differ from those of a CPU. All our experimental results are based on many rounds of experiments and are reproducible. Our work illuminates the currently mysterious architecture of GPU memory. In addition, by comparing the properties of three generations of GPU memory hierarchy, we can clearly perceive the evolution of GPU memory de-

- *Xinxin Mei and Xiaowen Chu are with the Department of Computer Science, Hong Kong Baptist University. Xiaowen Chu is also with HKBU Institute of Research and Continuing Education.*
  E-mail: {xxmei, chxw}@comp.hkbu.edu.hk
- *Our source code and experimental data are publicly available at: http://www.comp.hkbu.edu.hk/~chxw/gpu_benchmark.html.*

signs. The Kepler device is designed to maximize compute performance by aggressively integrating many emerging technologies, whereas the latest Maxwell device is more conservative and aims at energy efficiency rather than pure compute performance.

We highlight the contributions of our work as follows.

1) We propose a novel fine-grained P-chase microbenchmark to explore the unknown GPU cache parameters. Our results indicate that GPUs have many features that differ from those of traditional CPUs. We discover the unequal sets of L2 translation look-aside buffer (TLB), the 2D spatial locality optimized set-associative mapping of the texture L1 cache, and the non-traditional replacement policy of the L1 data cache.
2) We quantitatively benchmark the throughput and access latency of a GPU's global and shared memory. We study the various factors that influence the memory throughput, and the effect of the shared memory bank conflict on the memory access latency. For the first time, we verify that Maxwell is highly optimized to avoid long latency under shared memory bank conflict.
3) Our work provides comprehensive and up-to-date information on the GPU memory hierarchy. Our microbenchmarks cover the architecture, throughput and latency of recent generations of GPUs. To the best of our knowledge, this paper is the first to study the new features of the Kepler and Maxwell GPUs.

The remainder of this paper is organized as follows. Section 2 summarizes the related work and Section 3 gives an overview of GPU memory hierarchy. Section 4 introduces the fine-grained P-chase microbenchmark and how we apply it to dissect the GPU cache micro-architectures. Section 5 presents our study on the effective global memory throughput and memory access latencies under different access patterns. Section 6 investigates the shared memory in terms of latency, throughput, and the impact of bank conflict. We conclude our findings in Section 7.

2 Related Work

Many studies have investigated GPU memory system, some of which have confirmed that the performances of many GPU computing applications are limited by the memory bottleneck [2], [3], [6], [26]. Using characterization approaches, several studies located the causes of low memory throughput to relative memory spaces [6], [26], [27]. Some designed a number of data mapping/memory management algorithms with the aim to improve memory access efficiency [27], [28], [29], [30], [31]. Recently, Li et al. proposed a locality monitoring mechanism to better utilize the L1 data cache for higher performance [32]. All of these studies have contributed to the field and inspired our work.

We summarize the related GPU microbenchmark work in Table 1. Most of the work on memory structure and access latency is based on the P-chase microbenchmark, which was first introduced in [23], [24] (referred to as Saavedra1992 hereafter). Saavedra1992 was originally designed for CPU hardware and was quite successful for various CPU platforms [33]. Duchateau et al. developed P-ray, a multi-threaded version of P-chase to explore multi-core CPU cache architectures [34]. They exploited false sharing to quickly determine the cache coherence protocol block size. As the multi-threading feature on GPU is different from that of CPU, we cannot apply their method on GPU directly. Volkov and Demmel used P-chase for a very early GPU device, Nvidia 8800GTX, with a relatively simple memory hierarchy [17]. Papadopoulou et al. applied Saavedra1992 to explore the global memory TLB structure [18]. They also proposed a novel footprint experiment (referred to as Wong2010 hereafter) to investigate the other GPU caches [19]. Baghsorkhi et al. applied Wong2010 to benchmark a Fermi GPU and disclosed its L1/L2 data cache structure [21]. Different from previous studies, we proposed a novel fine-grained P-chase and disclosed some unique characteristics that both Saavedra1992 and Wong2010 neglected [25]. Our target hardware includes the caches of three recent generations of GPUs.

Meltzer et al. used both Saavedra1992 and Wong2010 to study the L1/L2 data cache of Fermi architecture [22]. They found that the L1 data cache does not use the least recently used (LRU) replacement policy, which is one of the basic assumptions of the traditional P-chase [19]. They also found that the L2 cache associativity is not an integer. Our experimental results coincide with theirs. Moreover, our fine-grained P-chase microbenchmarks allow us to obtain the L1 cache replacement policy.

Zhang and Owens quantitatively benchmarked the global/shared memory throughput from the bandwidth perspective [20]. Our work also includes a throughput study, but we are more interested to study the major factors that affect the effective memory throughput. Moreover, we include the study of memory access latencies which are also important factors for performance optimization.

TABLE 1
Summary of GPU Memory Microbenchmark Studies

Reference	Year	Device	Scope
[17]	2008	8800GTX	Architectures and latencies
[18], [19]	2009	GTX280	Architectures and latencies
[20]	2011	GTX285	Throughput
[21]	2012	Tesla^{TM}C2050	Architectures and latencies
[22]	2013	Tesla^{TM}C2070	Architectures and latencies
[25]	2014	GTX560Ti GTX780	Architectures and latencies

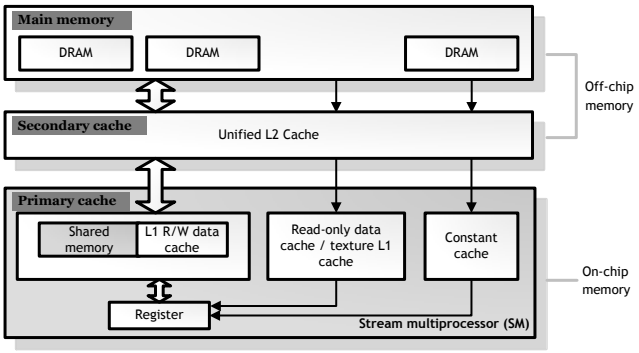


Fig. 1. Memory hierarchy of the GeForce GTX780 (Kepler).

TABLE 2
Features of Common GPU Memory

Memory	Type	Cached	Scope
Global	R/W	Yes (CA 2.0 or above)	All threads
Shared	R/W	N/A	Thread block
Texture	R	Yes	All threads

3 Overview of GPU Memory Hierarchy

Following the terminologies of CUDA, there are six types of GPU memory space: register, constant memory, shared memory, texture memory, local memory, and global memory. Their properties are elaborated in [15], [16]. In this study, we limit our scope to the three common types: global, shared, and texture memory. Specifically, we focus on the mechanism of different memory caches, the throughput and latency of global/shared memory, and the effect of bank conflicts on shared memory access latency.

Table 2 lists some salient characteristics of the target memory spaces. Unlike the early devices studied in [19], in recent GPUs the global memory access has become cached. The cached global/texture memory uses a two-level caching system. The L1 cache is located in each stream multiprocessor (SM), while the L2 cache is off-chip and shared among all SMs. It is unified for instruction, data and page table access. Furthermore, page table is used by GPU to map virtual addresses to physical addresses, and is usually stored in the global memory. The TLB is the cache of the page table. Once a thread cannot find the page entry in the TLB, it would access the global memory to search the page table, which causes significant access latency. Although the global memory and texture memory have similar dataflows, the former is read-and-write (R/W) and the latter is read-only. Both of them are public to all threads in the kernel function. The GPU-specific R/W shared memory is also located in the SMs. On the Fermi and Kepler devices it shares memory space with the L1 data cache, whereas on the Maxwell devices it has a dedicated space. In CUDA, the shared memory is declared and accessed inside a cooperative thread array (CTA, a.k.a. thread block), which is a programmer-assigned set of threads executed concurrently. Fig. 1 shows the block diagram of the memory hierarchy of a Kepler device, GeForce GTX780. The arrows indicate the dataflow. The architecture of the L1 cache in the Maxwell device is slightly different from that shown in Fig. 1 due to the separate shared memory space.

In Table 3, we compare the memory characteristics of the old Tesla GPU discussed in [18], [19] and our three target GPU platforms. The compute capability is used by NVIDIA to distinguish the generations. Table 3 shows that the most distinctive difference lies in the global memory. On the Tesla device, the global memory access is not cached, whereas on the Fermi device it is cached in both the L1 and the L2 data cache. The Kepler device has an L1 data cache, but it is designed for local rather than global memory access. In addition to the L2 data cache, global memory data that is read-only for the entire lifetime of a kernel can be cached in the read-only data cache with a compute capability of 3.5 or above. On the Maxwell device, the L1 data cache, texture on-chip cache and read-only data cache are combined in one physical space. Note that the L1 data cache of the Fermi and the read-only data cache of the Maxwell can be turned on or off. It is also notable that modern GPUs have larger shared memory spaces and more shared memory banks. On the Tesla device, the shared memory size of each SM is fixed at 16 KB. On the Fermi and Kepler devices, the shared memory and L1 data cache share 64 KB of memory space. On the Maxwell device, the shared memory is independent and has 96 KB. The maximum volume of shared memory that can be assigned to each CTA has been increased from 16 KB on the Tesla device to 48 KB on the later devices. The texture memory is cached on every generation of GPUs. The Tesla texture units are shared by three SMs (i.e., thread processing cluster). However, texture units on later devices are per-SM. The texture L2 cache shares space with the L2 data cache. The size of the texture L1 cache depends on the generation of the GPU hardware.

4 Cache Structures

The greatest difference between recent GPUs and the old Tesla GPUs lies in their cache systems. In this section, we first present a novel fine-grained P-chase method, and then explore two kinds of cache: the data cache and the TLB. We focus on the architectures of the Fermi/Maxwell L1 data cache, Fermi/Kepler/Maxwell texture memory L1 cache, read-only data cache, L2 cache and TLBs.

4.1 Why Not Typical P-chase?

By exploiting the principle of locality, cache memory is used to back up a piece of main memory for faster data access and plays a major role in modern computer architectures. Most existing GPU microbenchmark studies on cache architecture assume a classical set-associative cache model with the least recently used (LRU) replacement policy, the same as that of a conventional CPU cache [23], [24]. The cache size (C) is much smaller than main memory size. Data is loaded from main memory to cache with the basic unit of a cache line. The number of words in a cache line is referred to as the line size (b). For the classical *LRU set-associative cache*, the cache memory is divided into T cache sets, each of which consists of a cache lines. Fig. 2 shows an example of a 12-word set-associative cache and its memory mapping. There are three essential assumptions for this kind of cache model:

Assumption 1. All cache sets have the same size, and the cache parameters satisfy $T*a*b = C$. If any three of the four parameters are known, the remaining one can be found.

Assumption 2. In the memory address, the bits that identify the cache set are immediately followed by the bits that identify the offset (the intra-cache line location of data).

Assumption 3. The cache replacement policy is LRU.

Assumption 1 implies that all cache sets have the same number of cache lines. Assumption 2 indicates that the data mapping from the main memory to the cache follows a predictable, regular pattern. For instance, in Fig. 2, two out of every six consecutive words are mapped to one cache set, and they may appear in either of the two cache lines in the set. Assumption 3 implies that if we perform sequential

TABLE 3
Comparison of the Memory Properties of the Tesla, Fermi, Kepler and Maxwell Devices

Device	Tesla GTX280	Fermi GTX560Ti	Kepler GTX780	Maxwell GTX980
Compute capability	1.3	2.1	3.5	5.2
SMs * cores per SM	30 * 8	8 * 48	12 * 192	16 * 128
Global memory				
Cache mechanism	N/A	L1 and L2	L2, or read-only	L2, or unified L1
Cache size	N/A	L1: 16/48 KB L2: 512 KB	Read-only: 12 KB L2: 1.5 MB	Unified L1: 24 KB L2: 2 MB
Total size	1024 MB	1024 MB	3072 MB	4096 MB
Shared memory				
Size per SM	16 KB	48/16 KB	48/32/16 KB	96 KB
Maximum size per CTA	16 KB	48 KB		
Bank No.	16	32		
Bank width	4 B		8 B	4 B
Texture memory				
Texture units	per-TPC	per-SM		
L1 cache size	6-8 KB	12 KB	12 KB	24 KB

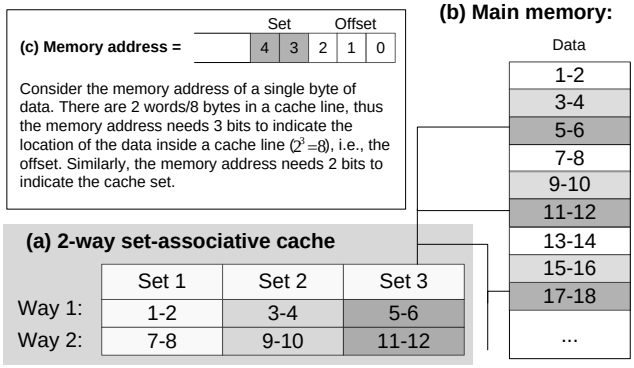


Fig. 2. An example of a 12-word 2-way set-associative cache. Assume each word has 4 bytes, each cache line can store 2 words ($b = 2$), and the data array is sequentially accessed. The cache lines are grouped into 3 separate cache sets ($T = 3$), each of which has 2 cache lines (i.e., Way 1 and Way 2), and we say its *cache associativity* is 2 ($a = 2$).

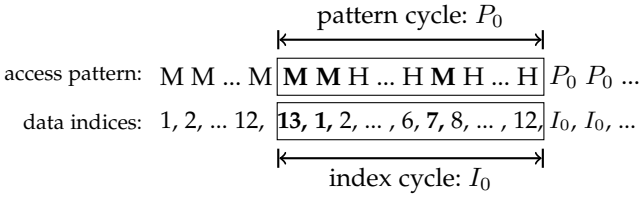


Fig. 3. Periodic memory access pattern of a classical LRU set-associative cache. M means cache miss and H means cache hit.

loading of a piece of data, the memory access is periodic. Taking the cache model in Fig. 2 as an example, we initialize an array with 13 words and read it one word by one word. Fig. 3 shows the full memory access process and its access pattern (a cache miss or a cache hit generated by visiting one array element). As the array size is one word larger than the cache size, the cache miss occurs. With the exception of the first 12 data accesses, which are cold cache misses, those data accesses to the 1st, 7th and 13th array elements are cache misses while the rest are cache hits. The 13-25th memory accesses form a pattern P_0, which recurs until the end of the data loading process. The period of this memory

TABLE 4
Notations for Cache and P-chase Parameters

Notation	Description	Notation	Description
C	cache size	N	array size
b	cache line size	s	stride size
a	cache associativity	k	iterations
T	number of cache sets	r	cache miss rate

access pattern is 13, which equals the array length.

```
1  for(i=0;i<array_size;i++){
2      A[i]=(i+stride)%array_size;
3  }
```

Listing 1. P-chase: array initialization

```
1  start_time = clock();
2  for(it=0;it<iterations;it++){
3      j=A[j];
4  }
5  end_time=clock();
6  //calculate average memory latency
7  tvalue=(end_time-start_time)/iterations;
```

Listing 2. P-chase: kernel function

The P-chase microbenchmark is a successful method for obtaining cache parameters [17], [18], [19], [21], [22], [23], [24], [33]. The core idea of P-chase is to traverse an array whose elements are initialized as the indices for the next memory access. The distance between two consecutively accessed array elements is called *stride* and is usually fixed in an experiment. The memory access latency is highly dependent on the *stride* due to the cache effect. By measuring **the average memory access latency** of a great number of memory accesses, the cache parameters can be deduced from the array size and the stride size. Listing 1 and Listing 2 give the array initialization and the kernel function of P-chase. In Listing 2, *j=A[j]* is repeatedly executed over $iterations$ of times, so that the array A is sequentially traversed with a fixed *stride*. Before the timing, we load the array elements for a number of times to eliminate the cold instruction cache misses. The average memory access latency, $tvalue$, is calculated by dividing the total clock

cycles by *iterations*. We denote the array size, stride size, and $iterations$ by N, s and k, respectively. We summarize the notations in Table 4.

Based on Assumptions 1-3, the output of P-chase, i.e., the average memory access latency, t_{avg}, satisfies

$$t_{avg} = t_0 * (1 - r) + (t_0 + t_m) * r = t_0 + t_m * r$$

where r denotes the cache miss rate, t_0 denotes the cache access latency and t_m denotes the cache miss penalty. Because t_0 and t_m are hardware-dependent constants, the typical P-chase method actually relies on the cache miss rate, r.

It has been believed that the cache parameters can be deduced from the $tvalue$-s graph (Saavedra1992) or the $tvalue$-N graph (Wong2010). As mentioned, under Assumptions 1-3, the memory access, or the cache miss patterns are periodic. Moreover, both Saavedra1992 and Wong2010 suggest that not only the cache miss patterns are predictable, but also the possible values of r are predictable.

In particular, Saavedra1992 suggests to run the experiments for multiple times, each with a different stride. Both array size N and stride size s are usually set to be power-of-two. If N is much larger than the cache size C, and s is smaller than cache line size b, there is a cache miss when loading the data mapped to the beginning address of a cache line, i.e., the cache miss rate is s/b. If $s \geq b$ but not exceeding N/a, every data loading is a cache miss. When s continues growing, the loaded data can fit into the cache so that there is no cache miss. To summarize, the cache miss rate satisfies Eq. (1) for all (N, s) pairs.

$$r \in \{0, s/b, 1\}, N >> C \quad (1)$$

Wong2010 suggests visiting arrays of various sizes with a fixed stride, which is chosen carefully and should be around cache line size. If we choose $s = b$, then every time we increase array size by b, there are much more cache misses. The cache miss rate satisfies Eq. (2) for all (N, s) pairs.

$$r \in \{0, \frac{1}{T}, ..., \frac{k}{T}, ..., 1\}, N \in [C, C + T * b], s = b \quad (2)$$

Fig. 4 and Fig. 5 show the experimental results when we apply Saavedra1992 and Wong2010 on the texture L1 cache on GTX780. Surprisingly, we obtain different results from the two methods. In Fig. 4, the N=12KB line suggests that $C = 12$ KB. The N=48KB line at $log_2(s) = 5$ suggests $b = 32$ bytes, and at $log_2(s) = 11$ suggests $a = N/s = 24$ so that $T = C/(ab) = 16$. In Fig. 5, there are 4 plateaus between the minimum and maximum memory latency, which indicates there are 4 cache ways in a cache set. The cache line size equals the width of every plateau. Overall, it suggests that $C = 12$ KB, $b = 128$ bytes, $T = 4$, and $a = C/(bT) = 24$. Here we face a contradiction: Fig. 4 and Fig. 5 are based on the same hardware, yet they lead to different cache parameters. This motivates us to seek the underlying causes.

Both Saavedra1992 and Wong2010 methods are based on Assumptions 1-3 so that the cache miss rates satisfy Eqs. (1) and (2). However, our experimental results reveal that Assumptions 1-3 seldom hold for different types of GPU cache, consequently Eqs. (1) and (2) are ineffective. Thus, the typical P-chase results become inappropriate to expose the GPU cache structure. For example, if Assumptions 1 and 2 hold but Assumption 3 does not, and the cache replacement

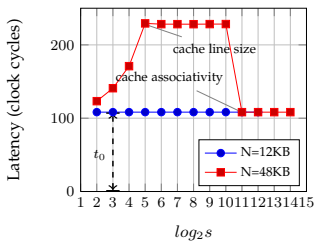


Fig. 4. $tvalue$-s of the Kepler texture L1 data cache.

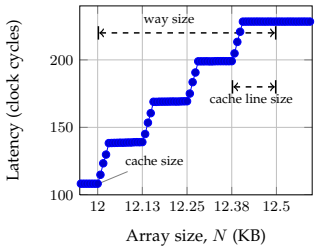


Fig. 5. $tvalue$-N of the Kepler texture L1 data cache (8-byte stride).

policy is random, then the measured t_{avg} can vary even for a given (N, s) pair. The value of r also varies and may not belong to those listed in (1) or (2). Hence, t_{avg} alone fails to serve as an indicator of GPU cache architecture.

Motivated by the above observation, we designed a microbenchmark that utilizes GPU shared memory to display the latency of every single memory access. We refer to it as fine-grained P-chase microbenchmark because it provides the most detailed information on the data access process.

4.2 Our Methodology: Fine-grained P-Chase

```
1  __global__ void KernelFunction(...){
2    //declare shared memory space
3    __shared__ unsigned int s_tvalue[];
4    __shared__ unsigned int s_index[];
5    preheat the data;
6    for(it=0;it<iterations;it++) {
7      start_time=clock();
8      j=my_array[j];
9      //store the array index
10     s_index[it]=j;
11     end_time=clock();
12     //store the access latency
13     s_tvalue[it]=end_time-start_time;
14   }
15 }
```

Listing 3. Fine-grained P-chase kernel (single thread, single CTA)

The core idea of our fine-grained P-chase is to record and analyze every single data access latency in a kernel with a single thread and single CTA. Such method is difficult to be used for CPU cache because of the challenge of recording every data access latency without interfering the normal data access. However, we can exploit GPU shared memory to store a sequence of data access latencies, based on which we can deduce the cache structure and parameters. The shared memory access is prompt and does not affect the data cache. Listing 3 gives the kernel code of our single-thread fine-grained P-chase. Notice that before the measurement, we need to visit the data in an initial iteration, aiming to load the data into L2 cache. Doing so can avoid the cold instruction cache miss and the interference from possible hardware pre-fetching. The core statement in line 8, *j = my_array[j]*, is the same as in the conventional P-chase. The difference lies in the location of the timing function. We put the timing statements inside a long loop, as shown in lines 7 and 11. The *clock*() function provided by CUDA is implemented by reading a special register, the value of which is incremented every clock cycle. We measure the overhead of *clock*() as the difference between two consecutive *clock*() calls in a single kernel thread. Based on our experimental

results, the overhead of *clock*() is 14, 16, and 6 cycles on Fermi, Kepler, and Maxwell platforms, respectively.

Although the idea of fine-grained P-chase is simple, we need to address the following major challenge: due to instruction-level parallelism (ILP), function *clock*() may overlap with its previous instruction and even return before the previous instruction finishes. E.g., if we put the second *clock*() (line 11) immediately after statement *j = my_array*[*j*] (line 8), it may lead to incorrect memory latency measurements because the second *clock*() could return before line 8 finishes. We overcome this problem by introducing a new statement, *s_index*[*it*] = *j* (line 10), that has data dependency on line 8, to ensure that the memory access completes when line 11 is issued. We use a separate program to measure the overhead of the code segment of lines 10-11, which is 20, 32, 16 cycles on Fermi, Kepler, and Maxwell, respectively. We can then deduce the latency of line 8 alone.

Our fine-grained P-chase microbenchmark outputs two arrays, *s_tvalue*[] and *s_index*[], the lengths of which are equal to the value of $iterations$. The former contains the data access latencies and the latter contains the accessed data indices. With these two arrays, we can reproduce the entire memory loading process and obtain all of the data access latencies rather than the average.

We work out a procedure to find the cache parameters using our fine-grained P-chase microbenchmark with different (N, s) configurations. Fig. 6 shows the flowchart of our two-stage procedure. We could use brute-force N testing to get the cache size. Then in the first stage, we overflow the cache with one element, getting the cache line size. We can also find whether the cache replacement policy is LRU or not in this stage. In the second stage, we gradually overflow the cache with the granularity of a cache line, until all the data accesses become cache miss. We can deduce the cache associativity and the memory addressing from the second stage. We further elaborate our method as follows. Notice that the basic unit of (N, s) is the length of an array element.

1) Determine cache size C. We set s to 1. We then initialize N with a small value and increase it gradually until the first cache miss appears. C equals the maximum N where all memory accesses are cache hits.
2) Determine cache line size b. We set s to 1. We begin with $N = C + 1$ and increase N gradually again. When $N < C + b + 1$, the numbers of cache misses are close. When N is increased to $C + b + 1$, there is a sudden increase on the number of cache misses, despite that we only increase N by 1. Accordingly we can find b. Based on the memory access patterns, we can also have a general idea on the cache replacement policy.
3) Determine number of cache sets T. We set s to b. We then start with $N = C$ and increase N at the granularity of b. Every increment causes cache misses of a new cache set. When $N > C + (T - 1)b$, all cache sets are missed. We can then deduce T from cache miss patterns accordingly.
4) Determine cache replacement policy. As mentioned before, if the cache replacement policy is LRU, then the memory access process should be periodic and all the cache ways in the cache set are missed. If memory access process is aperiodic, then the replacement policy cannot be LRU. Under this circumstance, we set $N = C + b, s = b$ with a considerable large k ($k >> N/s$) so that we can traverse the array multiple times. All cache misses are from one cache set. Every cache miss is caused by its former cache replacement because we overflow the cache by only one cache line. We have the accessed data indices thus we can reproduce the full memory access process and find how the cache lines are updated.

Applying the above method, we sketch the structures of texture L1 cache, read-only data cache, L1/L2 TLBs, and on-chip L1 data cache in the following sections. We also present some preliminary results of the off-chip L2 data cache.

4.3 Texture L1 Cache and Read-only Data Cache

We apply our P-chase microbenchmark on the texture L1 cache with the two-stage methodology. We bind an unsigned integer array to the linear texture, and fetch it with *tex1Dfetch*(). In the first stage, we find out the cache size C, which is 12 KB; and then set s to 1 element (i.e., 4 bytes) and overflow the cache gradually to get the cache line size, which is 32 bytes. In the second stage, we increase N from 12 KB to 12.5 KB with $s = 32$ bytes. Our results suggest a 12 KB set-associative cache with a special memory address format, as shown in Fig. 7, on Fermi and Kepler devices, and a 24 KB cache with similar organization on Maxwell device.

On the Fermi and Kepler GPUs, the 12 KB texture L1 cache is divided to 4 cache sets and can store up to 384 cache lines. Each cache set contains 96 cache lines and each cache line contains 8 words (i.e., 32 bytes). Each consecutive 32 words (i.e., 128 bytes) is mapped onto 4 successive cache sets. In particular, the 7-8th bits of the memory address determine the corresponding cache set, whereas the 5-6th bits do so in the traditional set-associative cache design. This mapping is optimized for 2D spatial locality in graphic processing [16], [35]. To take advantage of this mapping, in generalized applications, threads within a warp need to visit adjacent memory addresses, otherwise there would be more cache misses. The Maxwell texture L1 cache has a similar structure except it contains 768 cache lines.

Devices with a compute capability of 3.5 or above have an on-chip per-SM read-only data cache, which is an improvement on the texture memory cache [12]. The read-only data cache is loaded by calling $\_\_ldg(const\ \_\_restricted\_\_ * address)$. On our GTX780, we find a 12 KB read-only data cache, the same as the texture L1 cache. We overflow the read-only data cache with a single 4-byte element and find that the cache line size is 32 bytes and the replacement policy is LRU. We then examine it with $s = 32$ bytes and N varying from 12 KB to 60 KB. When the array is larger than 12.5 KB, each data access results in a cache miss. We infer that the read-only cache structure is the same as the texture L1 cache: 4 cache sets, with a 32-byte cache line and 96 lines in each set. Similarly, 128 successive bytes are mapped onto the same set, but the data mapping is not bits-defined. On the GTX980, the structure of the read-only data cache is also the same as that of the texture L1 cache except for the rather random data mapping.

4.4 Translation Look-Aside Buffer

Previous studies show that the GTX280 has two levels of TLB to support GPU virtual memory addressing on the

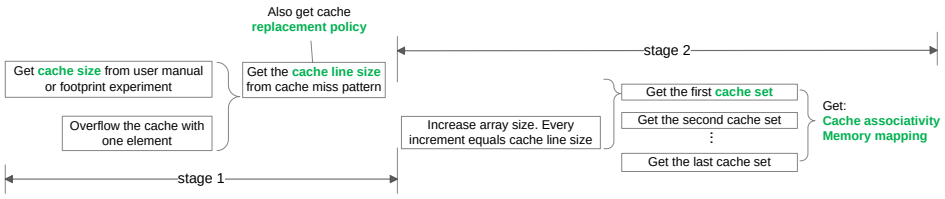


Fig. 6. Flowchart of applying fine-grained P-chase.

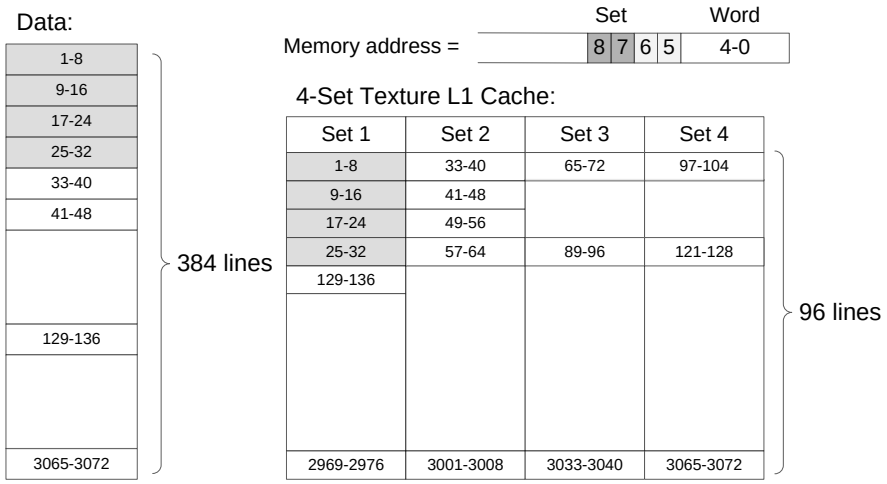


Fig. 7. The texture L1 cache structure of the Fermi and Kepler device and the memory address.

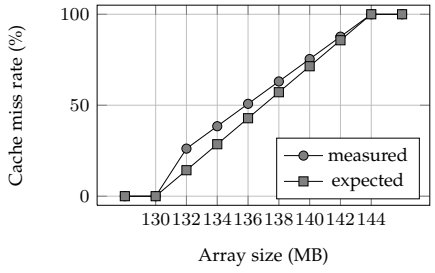


Fig. 8. Miss rate of L2 TLB (2 MB stride).

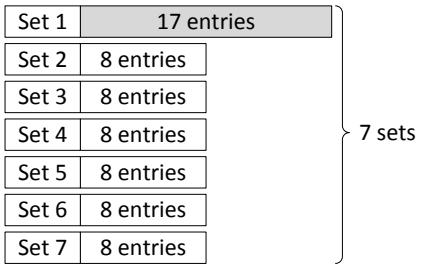


Fig. 9. L2 TLB structure.

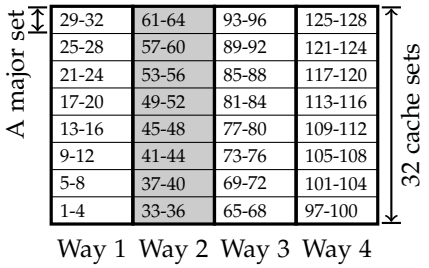


Fig. 10. L1 data cache structure (16 KB).

GTX280 [18], [19], where the L1 TLB is 16-way fully associative and the L2 TLB is 8-way set-associative. We apply our fine-grained P-chase method to investigate the TLB of three recent GPU architectures and find them have the same 16-way fully associative L1 TLB, and the page size is 2 MB. We plot the cache miss rate of the L2 TLB in Fig. 8 based on our microbenchmark results. The traditional LRU cache with equal sets triggers the same number of cache misses each time, thus the expected cache miss rate increases linearly. In contrast, our measured miss rate increases piecewise linearly. When N equals 132 MB, we observe 17 missed entries; varying N from 134 MB to 144 MB with $s = 2$ MB causes 8 more missed entries each time. Considering that cache misses are triggered set by set, the only explanation for the piecewise linear increase is that the first cache set has more cache ways than others. In addition, we deduce that the replacement policy is LRU, as the number of cache ways is equal to the number of missed cache entries. This gives us the conjectured L2 TLB structure as shown in Fig. 9: 1 large set with 17 entries and 6 small sets with 8 entries each.

4.5 L1 Data Cache

On the Fermi and Kepler devices, the L1 data cache and shared memory are physically implemented together. On the Maxwell devices, the L1 data cache is unified with the texture cache.

The Fermi L1 data cache can be either 16 KB or 48 KB. We only report the 16 KB case here for brevity. We vary the array size from 15 KB to 24 KB with $s = 4$ bytes or $s = 128$ bytes, and observe the memory access patterns. Fig. 10 gives the Fermi 16 KB L1 cache structure based on our experimental results. The 16 KB L1 cache has 128 cache lines mapped onto four cache ways. For each cache way, 32 cache sets are divided into 8 major sets. Each major set contains 16 cache lines. The data mapping is also unconventional. The 12-13th bits in the memory address define the cache way, the 9-11th bits define the major set, and the 0-6th bits define the memory offset inside the cache line.

One distinctive feature of the Fermi L1 cache is that its replacement policy is not LRU, as pointed out by Meltzer et al. in [22]. In our experimental results, the memory access

TABLE 5
Parameters of Common GPU Caches

Parameters	Default Fermi L1 data cache	Fermi/ Kepler/ Maxwell L1 TLB	Fermi/ Kepler/ Maxwell L2 TLB	Fermi/ Kepler texture L1 cache/ Kepler read-only data cache	Maxwell L1 data/ texture L1 cache/ read-only data cache
C	16 KB	32 MB	130 MB	12 KB	24 KB
b	128 byte	2 MB	2 MB	32 byte	32 byte
T	32	1	7	4	4
LRU	no	no	yes	yes	yes

	Way1	Way2	Way3	Way4		
1^{st} cache set:	3	35	68	100	129	
read 129:	3	129	68	100		miss
read 3:	3	129	68	100		hit
read 35:	3	129	35	100		miss
read 68:	3	68	35	100		miss
read 100:	3	68	35	100		hit
read 129:	129	68	35	100		miss
read 3:	129	3	35	100		miss
read 35:	129	3	35	100		hit
read 68:	129	3	35	68		miss
read 100:	129	100	35	68		miss
read 129:	129	100	35	68		hit
read 3:	129	100	3	68		miss
read 35:	129	35	3	68		miss

Fig. 11. Aperiodic memory access of the Fermi L1 data cache. In the figure, the numbers are the data line indices. In the second row, "read 129" stands for loading the 129th data line, and "miss" is the memory access status given by the output memory latency array. The highlighted data blocks represent the replaced cache ways according to the output index array when cache misses occur.

process does not reveal periodicity. We demonstrate part of the memory access process with $N = 16.125$ KB (i.e., 129 data lines), $s = 128$ bytes in Fig. 11. Because we overflow the cache with only one line, all cache misses are from a single cache set. In our experiment, cache misses occur when accessing data line 3, 35, 68, 100 and 129, which therefore belong to the first cache set. When we read the 129th data line, it sometimes leads to a cache miss and sometimes a cache hit. This cannot happen in the conventional LRU cache model. We find that among the four cache ways, cache way 2 is three times more likely to be replaced than the other three cache ways. It is updated once every two cache misses. The replacement probabilities of the four cache ways are $\frac{1}{6}$, $\frac{1}{2}$, $\frac{1}{6}$ and $\frac{1}{6}$, respectively.

For sequential data loading in our experiment, this non-LRU cache reduces the number of cache misses compared with the conventional cache; for example, in Fig. 11, the listed memory accesses should all be cache misses if the LRU replacement policy were used.

4.6 L2 Data Cache

The GTX560Ti, GTX780 and GTX980 report the maximum L2 cache size as 512 KB, 1536 KB and 2048 KB, respectively. Our fine-grained P-chase microbenchmark method is restricted by the shared memory size. At least 64 KB of shared memory is required for a single CTA to store one round of the smallest Fermi L2 cache accesses, much more than our hardware device can offer. However, our fine-grained P-chase can still find the following interesting results.

1) The replacement policy of the L2 cache is not LRU, either, because our experimental results show that the memory access processes are aperiodic again.
2) The L2 cache line size is 32 bytes by observing the memory access pattern of overflowing the cache and visiting array element one by one. The data mapping is sophisticated and not conventional bits-defined, either, since the cache miss pattern is very irregular.
3) We detect a hardware-level pre-fetching mechanism from the DRAM to the L2 data cache on all three platforms. For example, when we visit an array with uniform stride P-chase, we only observe a long latency for the first data item; the latencies of the following data items all match the L2 cache latency. The pre-fetching size is about 2/3 of the L2 cache size and the pre-fetching is sequential. This is deduced from that if we load an array smaller than 2/3 of the L2 data cache size, there is no cold cache miss patterns.

To summarize, in this section, we study the various GPU caches of three generations of GPUs. We propose a novel fine-grained P-Chase microbenchmark that provides the most detailed measurements. We list the derived parameters of various GPU caches in Table 5. According to our experimental results, the GPU caches are quite different from those of a CPU: they have unequal cache sets and a special replacement policy or data mapping. None of the GPU caches use the traditional bits-defined memory addressing stated in Assumption 2. To the best of our knowledge, most of these characteristics have been ignored in previous microbenchmark GPU studies.

5 Global Memory

In CUDA terms, global memory access involves accessing the DRAM, L1 and L2 data caches, TLBs and page tables. It is the most frequently accessed memory space in GPU programming. In this section, we use a series of microbenchmarks to quantitatively study the global memory throughputs and data access latencies on recent GPU platforms.

5.1 Global Memory Throughput

Although GPUs are designed with high memory bandwidth, their peak performance can rarely be achieved in reality. The theoretical bandwidth is calculated as f_{mem} * bus width * DDR_factor, where f_{mem} is the memory frequency,

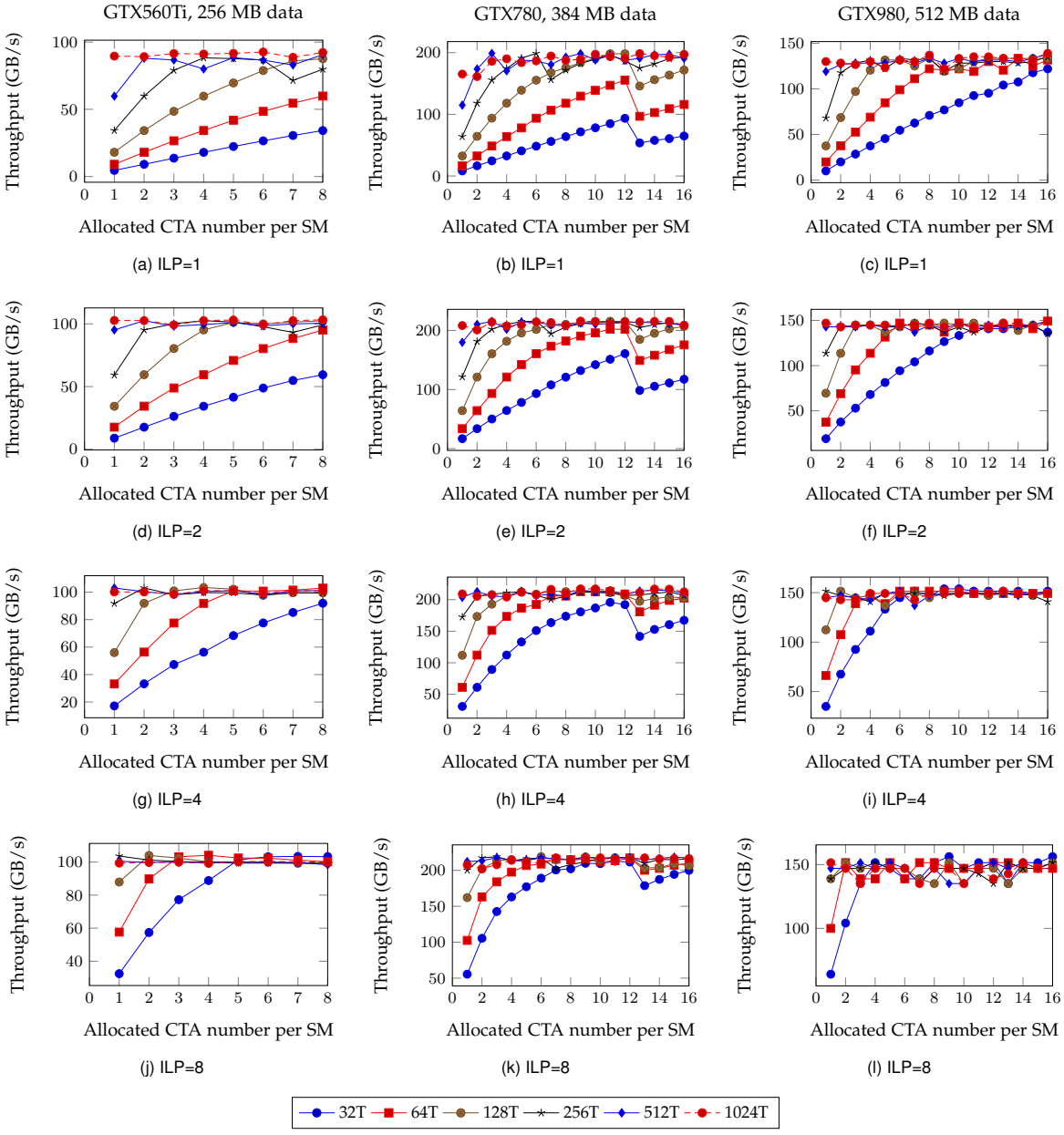


Fig. 12. Achieved throughput of global memory copy against the number of CTAs, CTA size and ILP.

TABLE 6
Theoretical and Achieved Bandwidth of Global Memory

Device	GTX560Ti	GTX780	GTX980
f_{mem} (MHz)	1050	1502	1753
Bus width (bits)	256	384	256
Theoretical bandwidth (GB/s)	134.40	288.38	224.38
Maximum throughput (GB/s)	109.38	215.92	156.25
Efficiency (%)	81.38	74.87	69.64

and the DDR_factor is 4 on all three target platforms. Table 6 lists the theoretical peak bandwidth and our measured maximum throughput of the three devices.

The global memory throughput is affected by many factors. According to Little's law, it requires as many memory requests on the fly as possible to fully utilize the bandwidth. We perform a plain memory copy on our three devices with large, fixed amounts of data. We measure the total elapsed time on the CPU. The throughput is calculated as $2 * datasize/time$. For each group of experiments, we vary the CTA number, the CTA size (number of threads in each CTA) and the ILP [36]. The ILP is defined as the number of 4-byte words that each thread copies at one time. Note that we allocate a number of CTAs to an SM, but these CTAs do not always execute in parallel because the number of activate threads in each SM is limited. Each thread executes the data copying for hundreds of times to ensure there are sufficient memory requests. We plot the achieved throughput in Fig. 12, where T stands for the number of threads per CTA. In general, the throughput converges to its maximum when the ILP/CTA size and the number of CTAs are large. We find that the throughput is limited by the number of active warps: when the size and the number of CTAs are both small, throughput increases almost linearly. The ILP also

influences the throughput. Fig. 12 shows that for all three devices, the throughput of a larger ILP saturates faster. The GTX560Ti relies on ILP the most, because its SM can launch the fewest warps/CTAs, and a larger ILP helps to handle more memory requests. The GTX780 has the highest throughput as it benefits from the highest bus width, but its convergence speed is the slowest, i.e., it requires the most memory requests to hide the pipeline latency. Considering that such a large amount of parallel memory requests is hardly ever reached in real applications, the higher bus width is somewhat wasteful. This could be part of the reason that NVIDIA reduced the bus width back to 256 bits in Maxwell devices.

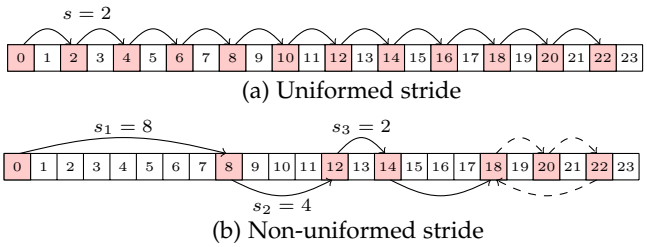
5.2 Global Memory Latency

In this section, we report the global memory latencies of various data access patterns. The global memory access latency is the whole time accessing a data located in DRAM/L2 or L1 cache, including the latency of page table look-ups. We apply our fine-grained P-chase with a novel self-defined data initialization so that we can collect as many memory latencies as possible in one experiment. We manually set the values of the array elements to create non-uniform stride accesses, rather than executing Listing 1. We are motivated by the convenience of Saavedra1992 method that a single $tvalue$-s graph can show memory latencies of different memory access patterns. Fig. 13 illustrates the difference of the data access process between the conventional P-chase and our non-uniform stride fine-grained P-chase.

We measure the global memory latencies with the L1 data cache of the GTX980 and GTX560Ti turned both on and off through the command options. By default, the Maxwell L1 cache is turned off and the Fermi L1 cache is turned on.

Fig. 14 shows the global memory latency cycles of six access patterns (noted as P1-P6). In our fine-grained P-chase initialization, we first set a very large $s_1 = 32$ MB to construct the TLB/page table miss and cache miss (P5&P6). We then set $s_2 = 1$ MB to construct the L1 TLB hit but cache miss (P4). After a total of 65 data accesses, 65 data lines are loaded into the cache. We then visit the cached data lines with s_1 again for several times, to construct cache hit but TLB miss (P2&P3). At last, we set $s_3 = 1$ element and repeatedly load the data in a cache line so that every memory access is a cache hit (P1). The latency values in Fig. 14 are based on the average of ten times of experiments. The data cache represents the L1 cache with the GTX980 and GTX560Ti L1 data cache turned on, otherwise it represents the L2 cache. We list some of our findings as follows.

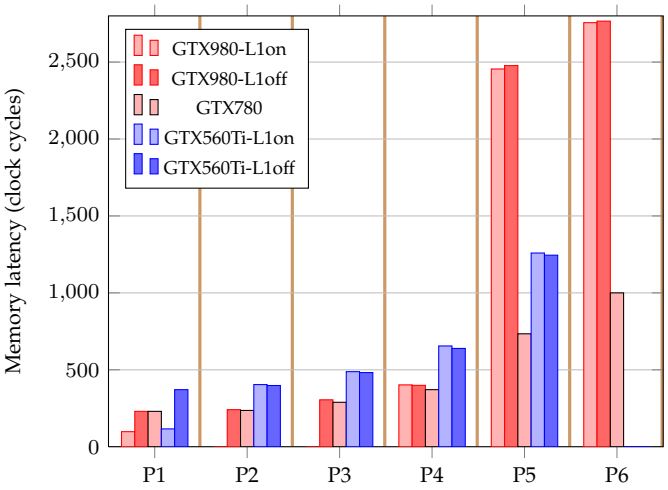
1) The Maxwell and Kepler devices have a unique memory access pattern (P6) for page table context switching. When a kernel is launched, only memory page entries of 512 MB are activated. If the thread visits an inactivate page entry, the hardware needs a rather long time to switch between page tables. This phenomena is also reported in [22] as page table "miss".
2) The Maxwell L1 data cache addressing does not go through the TLBs or page tables. On the GTX980, there is no TLB miss pattern (i.e., P2 and P3) when the L1 data cache is hit. Once the L1 cache is missed, the access latency increases from tens of cycles to hundreds or even thousands of cycles.



(a) Uniformed stride

(b) Non-uniformed stride

Fig. 13. Comparison between normal P-chase array access and our non-uniform stride array access. The numbers inside the square blocks are the array indices. The arrows indicate the values of the array elements, for example, the 0th data block pointing to the 2nd block means that we initialize the 0th array element with 2. In Fig. (a), the array is initialized with a single stride $s = 2$ that it forms a single memory access pattern: loading every one of two array elements. The measured memory latency is also of this single pattern. In Fig. (b), the array is initialized with various stride, s_1, s_2 and s_3, likewise, we can get the memory latencies of various patterns.



Pattern	P1	P2	P3	P4	P5	P6
Data cache	hit	hit	hit	miss	miss	miss
L1 TLB	hit	miss	miss	hit	miss	miss
L2 TLB	–	hit	miss	–	miss	miss

Latency	P1	P2	P3	P4	P5	P6
GTX980-L1on	82	–	–	385	2439	2740
GTX980-L1off	214	225	289	383	2461	2750
GTX780	198	204	257	339	702	968
GTX560Ti-L1on	96	384	468	635	1239	–
GTX560Ti-L1off	351	378	462	619	1225	–

Fig. 14. Global memory access latency spectrum.

3) The TLBs are off-chip. Fig. 14 shows that on the GTX560Ti, if the data are cached in L1, the L1 TLB miss penalty is 288 cycles. If data are cached in L2, the L1 TLB miss penalty is 27 cycles. Because the latter penalty is much smaller, we infer that the physical memory locations of the L1 TLB and L2 data cache are close. The physical memory locations of the L1 TLB and L2 TLB are also close, which means that the L1/L2 TLB and L2 data cache are shared off-chip by all SMs.
4) The GTX780 generally has the shortest global memory latencies, almost half that of the Fermi, with an access pattern of P2-P5. By default, the GTX980 has similar latencies to those of the GTX780 for P1-P4. However, for P5 (caused by the cold cache misses), the access latency

TABLE 7
Theoretical and Achieved Throughput of Shared Memory

Device	GTX560Ti	GTX780	GTX980
W_{bank} (byte/cycle)	2	8	4
f_{core} (GHz)	0.950	1.006	1.279
W_{SM} (GB/s)	60.80	257.54	163.84
W'_{SM} (GB/s)	34.90	83.81	137.41
Efficiency (%)	57.4	32.5	83.9

is about 3.5 times longer than on the Kepler and twice as long as on the Fermi. The page table context switching of the GTX980 is also much more expensive than that of the GTX780.

To summarize, the Maxwell device has long global memory access latencies for cold cache misses and page table context switching. Except for these rare access patterns, its access latency cycles are close to those of the Kepler device. In our experiment, because the GTX980 has higher f_{mem} than the GTX780, it actually offers the shortest global memory access time (P2-P4).

6 Shared Memory

The shared memory is designed with high bandwidth and very short memory latency, and each SM has a dedicated shared memory space. In CUDA programming, different CTAs assigned to the same SM have to share the same physical memory space. On the Fermi and Kepler platforms, the shared memory is physically integrated with the L1 cache. On the Maxwell platform, it occupies a separate memory space. Storing data in shared memory is a recognized optimization strategy for GPU-accelerated applications [4], [5], [9]. Programmers move the data into and out of shared memory from global memory before and after arithmetic execution, to avoid the frequent occurrence of long global memory access latencies.

In this section, we micro-benchmark the throughput and latency of shared memory. In particular, we discuss the effects of the bank conflict on shared memory access latency. We report a dramatic improvement in performance for the Maxwell device.

6.1 Shared Memory Throughput

On all three GPU platforms, the shared memory is organized as 32 memory banks [15]. The bank width of the Fermi and Maxwell devices is 4 bytes, while that of the Kepler device is 8 bytes. Each bank has a bandwidth of W_{bank}, as shown in Table 7. The theoretical peak throughput of each SM (W_{SM}) is calculated as $f_{core} * W_{bank} * 32$. Our microbenchmark results indicate that although the bandwidth of shared memory is considerable, the real achieved throughput could be much lower. This is most obvious on our Fermi and Kepler devices.

The microbenchmark is designed as follows. We copy a number of integers from one shared memory region to another with various grid configurations and ILP levels. Each thread copies ILP of 4-byte data and consumes 8*ILP bytes of shared memory. For each SM, we measure the total elapsed clock cycles with the *__syncthreads*() and *clock*() for all its active warps. The overhead of a pair of *__syncthreads*() and *clock*() is measured as 78, 37, and 36 cycles for Fermi, Kepler, and Maxwell platforms, respectively. The achieved throughput per SM is calculated as 2 * f_{core} * sizeof(int) * (number of active threads per SM) * ILP / (total latency of each SM). We run the microbenchmark with CTA size = {32, 64, 128, 256, 512, 1024}, CTAs per SM ={1, 2, 3, 4, 5, 6}, and ILP={1, 2, 4, 6, 8}, subject to the constraint of shared memory size per SM. Usually a large value of ILP results in less active warps per SM. The peak throughput W'_{SM} denotes the respective maximum throughput of the above combinations. Two key factors that affect the throughput are the number of active warps per SM and the ILP level.

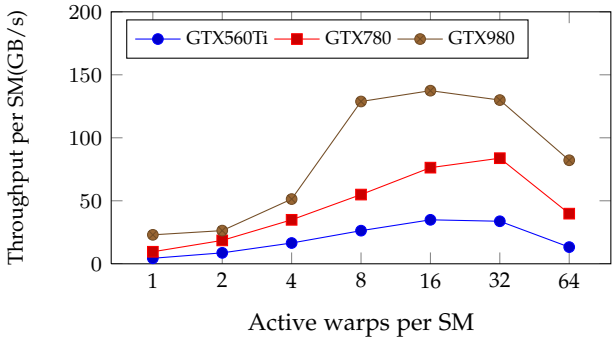


Fig. 15. Achieved shared memory peak throughput per SM.

We plot the achieved shared memory peak throughput per SM against the number of active warps in Fig. 15. In general the peak shared memory throughput grows with the increase of active warps, until it reaches some threshold. The peak shared memory throughput of the GTX560Ti occurs when the CTA size = 512, CTAs per SM = 1 and ILP = 4, i.e., 16 active warps per SM. The peak throughput is 34.90 GB/s, which is about 58.7% of the theoretical bandwidth. The GTX780 reaches its peak throughput when the CTA size = 1024, CTAs per SM = 1 and ILP = 6, i.e., 32 active warps per SM. The peak throughput is 83.81 GB/s, which is only 32.5% of the theoretical bandwidth. The GTX980 reaches its peak throughput when the CTA size = 256, CTAs per SM = 2 and ILP = 8, i.e., 16 active warps per SM. The peak throughput is 137.41 GB/s, about 83.9% of the theoretical bandwidth. The Maxwell device shows the best use of its shared memory bandwidth, and the Kepler device shows the worst.

Fig. 16 shows the achieved shared memory throughputs for different combinations of ILP and number of active warps per SM. Notice that on GTX560Ti and GTX780, when there are 32 active warps, the maximum ILP is 6 due to limited shared memory size. On the GTX560Ti, the achieved throughput grows with the increase of ILP until it reaches 4. On the GTX780, for low SM occupancy (i.e., 1 to 4 active warps), ILP = 4 gives the highest throughput; while for higher SM occupancy (i.e., 8 to 32 active warps), ILP = 6 or 8 give the highest throughput. GTX980 exhibits similar behavior as GTX780: high ILP is required to achieve high throughput for high SM occupancy.

According to Little's Law, we roughly have: number of active warps * ILP = latency cycles * throughput. Applying the latency values in Section 6.2, the GTX780 requires about 94 active warps if ILP = 1, but the Kepler device allows

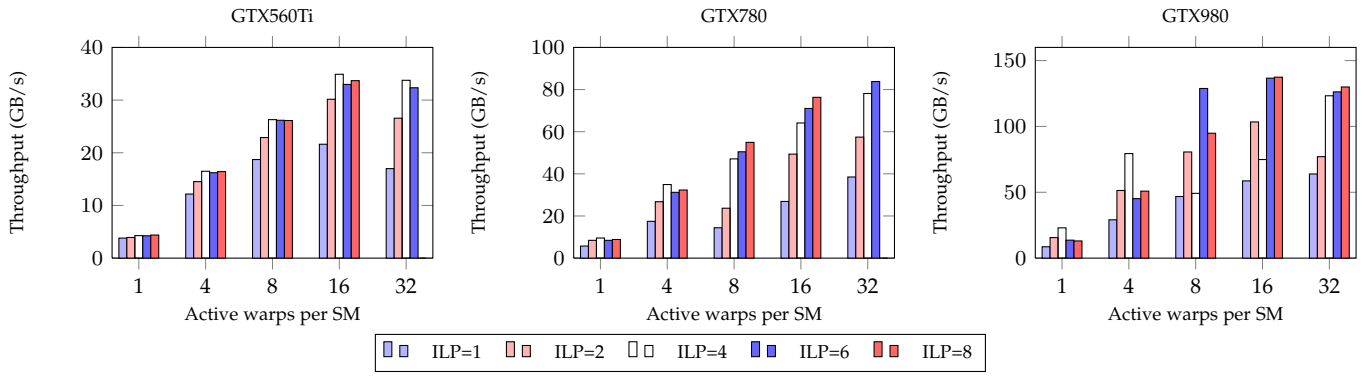


Fig. 16. Shared memory throughput per SM vs. ILP.

64 warps at most to be executed concurrently [15]. The gap between the number of required active warps and the number of allowed concurrent warps is particularly obvious on the GTX780. We consider this to be the main reason the achieved throughput of the GTX780 is poor compared with its designed value. For the Maxwell device, due to the significantly reduced access latency, we observe a higher shared memory throughput.

6.2 Shared Memory Latency

```
1  for ( i=0;i <= iterations; i++ ) {
2      data=threadIdx.x*stride;
3      if(i==1) sum = 0; //omit cold miss
4      start_time = clock();
5      repeat64( data=sdata[data];);
6          //64 times of stride access
7      end_time = clock();
8      sum += (end_time - start_time);
9  }
```

Listing 4. Kernel function of shared memory stride access

We first use the P-chase kernel in Listing 4 with single thread and single CTA to measure the shared memory latencies without bank conflict. The shared memory latencies on Fermi, Kepler and Maxwell devices are 50, 47 and 28 cycles, respectively. However, the shared memory access latency will grow when bank conflicts occur. In this section, we focus on the effect of bank conflicts on shared memory access latency.

The shared memory space is divided into 32 banks. Successive words are allocated to successive banks. If two threads in the same warp access memory spaces in the same bank, a 2-way bank conflict occurs. Listing 4 is also used to measure the shared memory access latency with bank conflicts. Different from the previous case, we launch a warp of threads with a single CTA to access stride memory. We multiply the thread id with an integer, *stride*, to get a shared memory address. We perform the memory access 64 times and record the total time consumption. We then calculate the average memory latency for each memory access.

Fig. 17 illustrates a 2-way bank conflict caused by stride memory access on the Fermi architecture. For example, word 0 and word 32 are mapped onto the same bank. If the stride is 2, threads 0 and 16 will visit words 0 and 32, respectively, which causes a 2-way bank conflict. The number of potential bank conflicts equals the greatest common divisor of the stride number and 32. There is no bank conflict

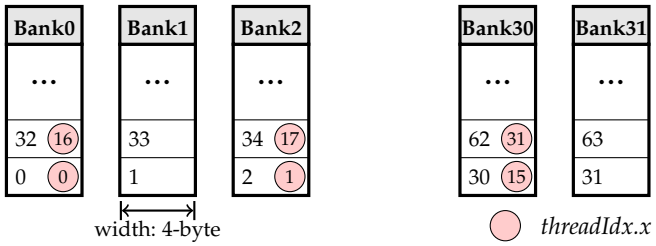


Fig. 17. 2-way shared memory bank conflict (stride=2).

TABLE 8
Shared Memory Access Latency with Bank Conflicts

Bank conflict	2-way	4-way	8-way	16-way	32-way
GTX980	30	34	42	58	90
GTX780	82	96	158	257	484
GTX560Ti	87	162	311	611	1209

for odd strides. Fermi and Maxwell devices have the same number of potential bank conflicts because they have the same architecture.

Kepler outperforms Fermi in terms of avoiding shared memory bank conflicts by doubling the bank width [37]. The bank width of Kepler device is 8 bytes, yet it offers two configurable modes to programmers: 4-byte mode and 8-byte mode. In the 8-byte mode, 64 successive integers are mapped onto 32 successive banks, whereas in the 4-byte mode, 32 successive integers are mapped onto 32 successive banks. Fig. 18 illustrates the data mapping of the two modes. A bank conflict only occurs when two or more threads access different bank rows. Fig. 19 shows the Kepler shared memory latencies with even strides for the 4-byte and 8-byte modes. When the stride is 2, there is no bank conflict in either mode, whereas there is a 2-way bank conflict on Fermi. When the stride is 4, both modes show a 2-way bank conflict. When the stride is 6 (Fig. 18), there is a 2-way bank conflict for the 4-byte mode but no bank conflict for the 8-byte mode. For the 4-byte mode, half of the shared memory banks are visited. Thread i and thread $i+16$ access separate rows in the same bank ($i = 0, ..., 15$). For the 8-byte mode, 32 threads visit 32 different banks with no conflict. Similarly, the 8-byte mode is superior to the 4-byte mode for other even strides if their number is not to the power of two.

We list our measured shared memory access latencies according to the number of potential bank conflicts in Table

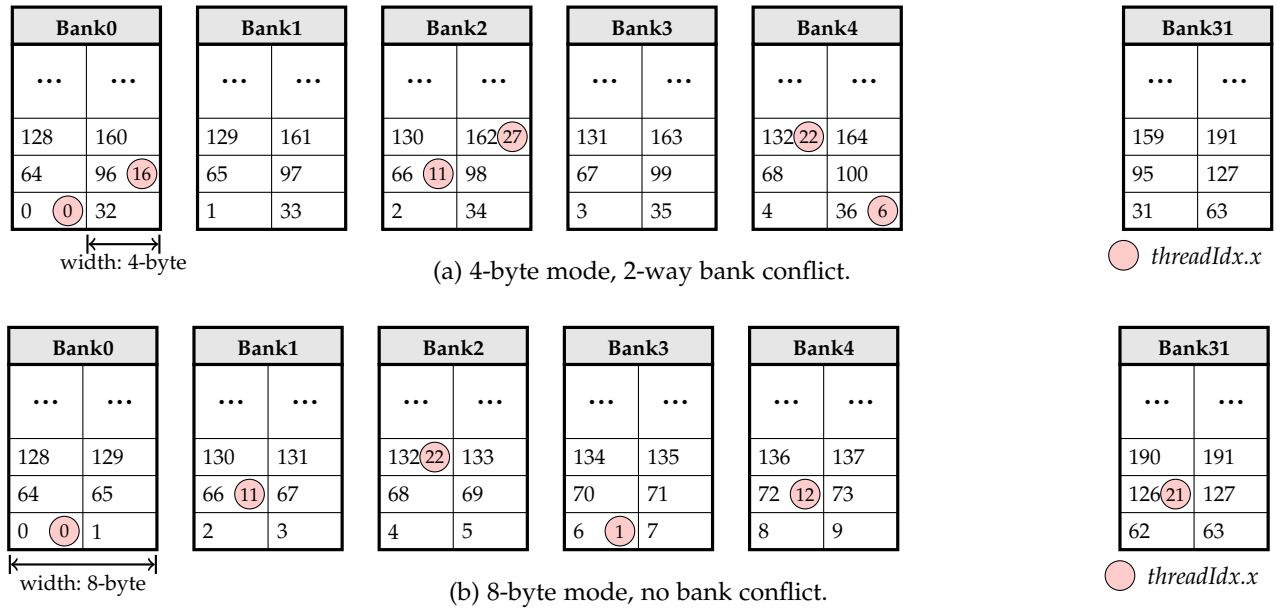


Fig. 18. Kepler shared memory bank conflict (stride = 6).

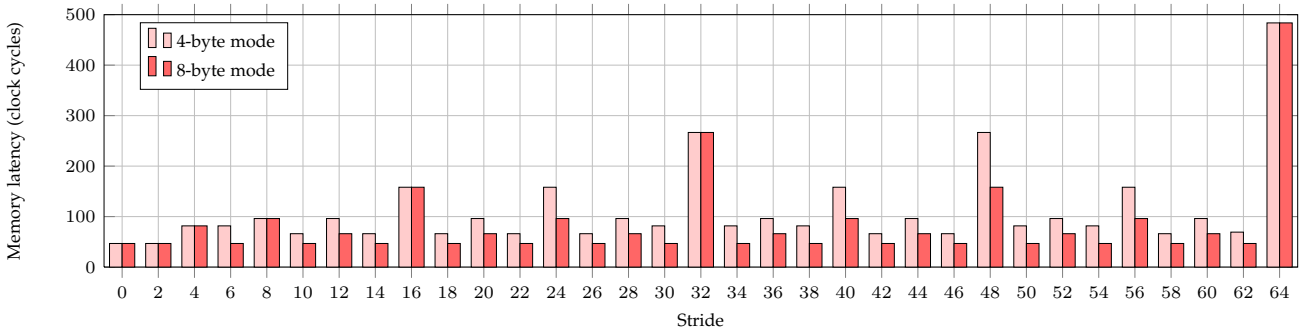


Fig. 19. Latency of Kepler Shared Memory with bank conflict: 4-byte mode v.s. 8-byte mode.

8. The memory access latency increases almost linearly with the number of potential bank conflicts. This confirms that the data access instructions are sequentially executed in case of a bank conflict. For the Fermi and Kepler devices, where there is a 32-way bank conflict, it takes much longer to access shared memory than regular global memory (TLB hit, cache miss). Surprisingly, the effect of a bank conflict on shared memory access latency on the Maxwell device is mild. Even the longest shared memory access latency is still at the same level as L1 data cache latency.

In summary, although the shared memory has very short access latency, it can be rather long if there are many ways of bank conflicts. This is most obvious on the Fermi hardware. The Kepler device tries to solve it by doubling the bank width of shared memory. Compared with the Fermi, the Kepler's 4-byte mode shared memory halves the chance of bank conflict, and the 8-byte mode reduces it further. However, we also find that the Kepler's shared memory is inefficient in terms of throughput. The Maxwell device has the best shared memory performance. With the same architecture as the Fermi device, the Maxwell hardware shows a 2x size, 2x memory access speedup and achieves the highest throughput. Most importantly, the Maxwell device's shared memory has been optimized to avoid the long latency caused by bank conflicts. As many GPU-accelerated applications rely on shared memory performance, this improvement certainly leads to faster and more efficient GPU computations.

7 Conclusions

In this study, we microbenchmarked the cache characteristics, memory throughput, and memory latencies of three recent generations of NVIDIA GPUs: Fermi, Kepler and Maxwell. We perceive an evolution of the NVIDIA GPU memory hierarchy. The memory capacity is significantly enhanced in both Kepler and Maxwell as compared with Fermi. The Kepler device is performance-oriented and incorporates several aggressive elements in its design, such as increasing the bus width of DRAM and doubling the bank width of shared memory. These designs have some side-effects. The theoretical bandwidths of both global memory and shared memory are difficult to saturate, and hardware resources are imbalanced with a low utilization rate. The Maxwell device has a more efficient and conservative design. It has a reduced bus width and bank width, and the on-chip cache architectures are adjusted, including doubling the shared memory size and the read-only data cache size. Furthermore, it sharply decreases the shared memory latency caused under bank conflicts. With its optimized

memory hierarchy, the Maxwell device not only retains good performance but is also more economical.

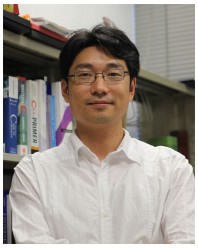
ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments. This work is partially supported by Hong Kong GRF grant HKBU 210412, HKBU FRG2/14-15/059, Shenzhen Basic Research Grant SCI-2015-SZTIC-002.

REFERENCES

[1] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.
[2] W. mei Hwu, "What is ahead for parallel computing," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2574–2581, 2014.
[3] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 0007–17, 2011.
[4] K. Zhao and X. Chu, "G-BLASTN: accelerating nucleotide alignment by graphics processors," *Bioinformatics*, vol. 30, no. 10, pp. 1384–91, 2014.
[5] Y. Li, H. Chi, L. Xia, and X. Chu, "Accelerating the scoring module of mass spectrometry-based peptide identification using GPUs," *BMC bioinformatics*, vol. 15, no. 1, pp. 1–11, 2014.
[6] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2008, pp. 73–82.
[7] *NVIDIA GeForce GTX 980 Whitepaper*, NVIDIA Corporation, 2014.
[8] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 79–84.
[9] NVIDIA, "matrixMul," *CUDA SDK 6.5*, 2014.
[10] ——, "matrixMulCUBLAS," *CUDA SDK 6.5*, 2014.
[11] *Fermi Whitepaper*, NVIDIA Corporation, 2009.
[12] *Kepler GK110 Whitepaper*, NVIDIA Corporation, 2012.
[13] *Tuning CUDA Applications for Kepler*, NVIDIA Corporation, 2013.
[14] *Tuning CUDA Applications for Maxwell*, NVIDIA Corporation, 2014.
[15] *CUDA C Programming Guide - v7.5*, NVIDIA Corporation, 2015.
[16] *CUDA C Best Practices Guide - v7.5*, NVIDIA Corporation, 2015.
[17] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, no. 31. IEEE Press, 2008.
[18] M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong, "Micro-benchmarking the GT200 GPU," *Computer Group, ECE, University of Toronto, Tech. Rep*, 2009.
[19] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. of Performance Analysis of Systems and Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.
[20] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. of High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 382–393.
[21] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu, "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 23–34.
[22] R. Meltzer, C. Zeng, and C. Cecka, "Micro-benchmarking the C2070," 2013, poster presented at GPU Technology Conference, March 18-21, San Jose, California.
[23] R. H. Saavedra, "CPU performance evaluation and execution time prediction using Narrow spectrum benchmarking," Ph.D. dissertation, EECS Department, University of California, Berkeley, Feb 1992.
[24] R. H. Saavedra and A. J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *Computers, IEEE Transactions on*, vol. 44, no. 10, pp. 1223–1235, 1995.
[25] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern GPUs," in *Proc. of Network and Parallel Computing, 2014 IFIP 11th International Conference on*, 2014, pp. 144–156.
[26] S. Lal, J. Lucas, M. Andersch, M. Alvarez-Mesa, A. Elhossini, and B. Juurlink, "GPGPU workload characteristics and performance analysis," in *Proc. of Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*. IEEE, 2014, pp. 115–124.
[27] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proc. of the 26th ACM International Conference on Supercomputing*. ACM, 2012, pp. 15–24.
[28] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. of Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 2013, pp. 516–523.
[29] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 105–118, 2011.
[30] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 13:1–13:11.
[31] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 513–522.
[32] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 67–77.
[33] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *Proc. of USENIX Annual Technical Conference*. San Diego, CA, USA, 1996, pp. 279–294.
[34] A. X. Duchateau, A. Sidelnik, M. J. Garzarán, and D. Padua, "P-ray: A software suite for multi-core architecture characterization," in *Proc. of Languages and Compilers for Parallel Computing, 21th International Workshop on*. Springer, 2008, pp. 187–201.
[35] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2, pp. 108–120, 1997.
[36] V. Volkov, "Better performance at lower occupancy," in *the 1st GPU Technology Conference*. San Jose, CA, USA, 2010.
[37] P. Micikevicius, "GPU performance analysis and optimization," in *the 3rd GPU Technology Conference*. San Jose, CA, USA, 2012.

Xinxin Mei received the B.E. degree in electronic information engineering from the University of Science and Technology of China, P.R.C., in 2010. She is currently a Ph.D. student in the Department of Computer Science, Hong Kong Baptist University. Her research interests include distributed and parallel computing and GPU-accelerated parallel partial differential equation solvers.



Xiaowen Chu received the B.E. degree in computer science from Tsinghua University, P.R. China, in 1999, and the Ph.D. degree in computer science from the Hong Kong University of Science and Technology in 2003. Currently, he is an associate professor in the Department of Computer Science, Hong Kong Baptist University. His research interests include distributed and parallel computing and wireless networks. He is serving as an Associate Editor of IEEE Access and IEEE Internet of Things Journal. He is a senior member of the IEEE.