Benchmarking the Memory Hierarchy of Modern GPUs

Xinxin Mei, Kaiyong Zhao, Chengjian Liu, Xiaowen Chu

Benchmarking the Memory Hierarchy of Modern GPUs

Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu

Department of Computer Science, Hong Kong Baptist University
{xxmei,kyzhao,cscjliu,chxw}@comp.hkbu.edu.hk

Abstract. Memory access efficiency is a key factor for fully exploiting the computational power of Graphics Processing Units (GPUs). However, many details of the GPU memory hierarchy are not released by the vendors. We propose a novel fine-grained benchmarking approach and apply it on two popular GPUs, namely Fermi and Kepler, to expose the previously unknown characteristics of their memory hierarchies. Specifically, we investigate the structures of different cache systems, such as data cache, texture cache, and the translation lookaside buffer (TLB). We also investigate the impact of bank conflict on shared memory access latency. Our benchmarking results offer a better understanding on the mysterious GPU memory hierarchy, which can help in the software optimization and the modelling of GPU architectures. Our source code and experimental results are publicly available.

1 Introduction

GPUs have become popular parallel computing accelerators; but their further performance enhancement is limited by the sophisticated memory system [1–6]. In order to reduce the default memory access consumption, developers usually utilize some specially designed memory spaces empirically [3–5]. It is necessary to have a clear and comprehensive documentation on the memory hierarchy. Despite the need, many details of memory access mechanism are not released by the manufacturers. To learn the undisclosed characteristics through third-party benchmarks becomes compelling.

Some researchers benchmarked the memory system of earlier GPU architectures [7–10]. They studied the memory latencies and revealed cache/translation lookaside buffer (TLB) structures. According to reports from vendor, recent generations of GPUs, such as Fermi, Kepler and Maxwell show significant improvement on memory access efficiency [11–16]. The memory hierarchies are different from those of earlier generations. To the best of our knowledge, a state-of-art work remains vacant.

In this paper, we explore the memory hierarchies of modern GPUs: Fermi and Kepler. We design a series of benchmarks to investigate their structures. Our experimental results confirm the superiority of recent architectures in memory access efficiency. Our contributions are summarized as follows:

Table 1. Comparison of NVIDIA Tesla, Fermi and Kepler GPUs

Generation	Tesla	Fermi	Kepler
Device	GeForce GTX 280	GeForce GTX 560 Ti	GeForce GTX 780
Compute Capacity	1.3	2.1	3.5
Shared Memory	size: 16 KB bank No: 16 bank width: 4 byte	size: 16/48 KB bank No: 32 bank width: 4 byte	size: 16/32/48 KB bank No: 32 bank width: 4/8 byte
Global Memory	non-cached size: 1024 MB	cached in L1&L2 L1 cache size: 64 KB subtract shared memory size; L2 cache size: 512 KB size: 1024 MB	cached in L2 or read-only data cache L1 cache size: 64 KB subtract shared memory size; L2 cache size: 1536 KB size: 3071 MB
Texture Memory	per-TPC texture units	per-SM texture units	per-SM texture units

1. We develop a novel fine-grained P-chase benchmark to expose GPU cache and TLB features.

2. Based on the benchmark, we find a number of unconventional designs not disclosed by previous literatures, such as the special replacement policy of Fermi L1 data cache, 2D spacial locality optimized set-associative mapping of texture L1 cache and the unequal L2 TLB sets.

3. Our source code and experimental results are publicly available.[1]

The remainder of this paper is organized as follows. Section 2 introduces some background knowledge of GPU memory hierarchy. Section 3 presents our fine-grained benchmark design. Section 4 discusses the microarchitecture of various memory spaces of Fermi and Kepler. We conclude our work in Section 5.

2 Background: Modern GPU Memory Hierarchy

In the popular GPU programming model, CUDA (compute unified device architecture), there are six memory spaces, namely, register, shared memory, constant memory, texture memory, local memory and global memory. Their functions are described in [14–18]. In this paper, we limit our scope to the discussion of three common but still mysterious ones: shared memory, global memory and texture memory. Specifically, we aim at disclosing the impact of bank conflict on shared memory access latency, and the cache mechanism and latency of global/texture memory.

In Table 1, we compare the memory characteristics of the Tesla GPU discussed in [8, 9] and our two targeting GPU platforms. The *compute capacity* is used by NVIDIA to distinguish the generations. The Fermi devices are of compute capacity 2.x, and the Kepler devices are of 3.x. The two GPU cards we use, MSI N560GTX-Ti Hawk (repack of GeForce GTX 560 Ti) and GIGABYTE GeForce GTX 780 (repack of GeForce GTX 780), are of compute capacity 2.1 and 3.5 respectively. As we can find in Table 1, the most distinctive difference is the global memory. On Tesla devices, the global memory access is non-cached while on Fermi devices, it is cached in both L1 and L2 data cache. Kepler has L1 data cache; but it is designed for local memory accesses rather than global memory accesses. Besides L2 data cache, Kepler global memory accesses can be cached in read-only data cache for compute capacity 3.5 or above. It is also

[1] `http://www.comp.hkbu.edu.hk/~chxw/gpu_benchmark.html`

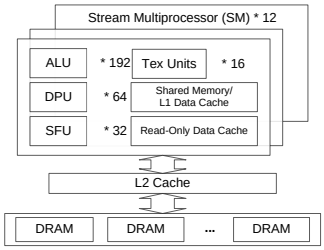

Fig. 1. Block Diagram of GeForce GTX 780

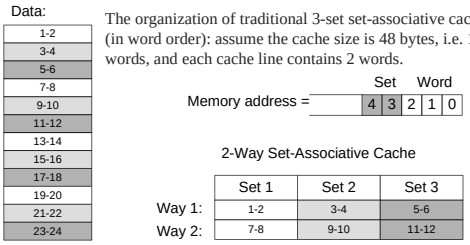

Fig. 2. Traditional Set-Associative Cache

notable that modern GPUs have larger shared memory spaces and more shared memory banks. Tesla shared memory size is fixed as 16 KB. On Fermi and Kepler devices, shared memory and L1 data cache share a total of 64 KB memory space. The texture memory is cached in all generations. Tesla texture units are shared by three streaming multiprocessors (SMs), namely a thread processing cluster (TPC). However, Fermi and Kepler texture units are per-SM.

As shown in Fig. 1, the shared memory, L1 data cache and the read-only data cache are on-chip, i.e., they are within SMs. L2 cache and DRAMs are off-chip. The L2 cache is accessed by all the SMs, and a GPU board has several DRAM chips.

For ease of reference, we also review some fundamentals of cache systems. The cache backs up a piece of main memory on-chip to offer very instant memory accesses. Due to the performance-cost tradeoff, the cache sizes are limited. Fig. 2 shows the structure of a traditional set-associative cache. Data is loaded from main memory to cache at the granularity of a cache line. Memory addressing decides the location in the cache of a particular copy of main memory. For set-associative cache, each line in the main memory is mapped into a fixed cache set and can appear at any cache ways of the corresponding set. For example, in Fig. 2, word 1-2 can be in way 1 or way 2 of the first cache set. If the required data is stored in cache, there is a cache hit, otherwise a cache miss. When the cache is full and a cache miss occurs, some existing contents in the cache is replaced by the required data. One popular replacement policy is least-recently used (LRU), which replaces the least recently accessed cache line. Modern architectures usually have multi-level and multi-functional caches. In this paper, we discuss the data cache and TLB (cache for virtual-to-physical memory translation page tables). Previous cache studies all assume a cache model of equal cache sets, typical set-associative addressing and LRU replacement policy [7–10, 19, 20]. Based on our experimental results, such model is sometimes incorrect for GPU cache systems.

3 Methodology

3.1 Shared Memory Bank Conflict: Stride Memory Access

GPU shared memory is divided into banks. Successive words are allocated to successive banks. If some threads belonging to the same warp access memory spaces in the same bank, bank conflict occurs.

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

Listing 1. Shared Memory Stride Access

To study the impact of bank conflict on shared memory access latency, we utilize the stride memory access introduced in [15]. We launch a warp of threads on GPU. Listing 1 is the kernel code of our shared memory benchmark. We multiply the thread id with an integer, called *stride*, to get a shared memory address. We do 64 times of such memory accesses and record the total time consumption. This consumption is actually the summation of 63 times of shared memory access and 64 times of *clock*() overhead. We then calculate the average memory latency of each memory access. If a bank conflict occurs, average memory latency is much longer.

3.2 Cache Structure: Fine-grained Benchmark

The P-chase benchmark is the most classical method to explore cache memory hierarchy [7–10, 19, 20]. Its core idea is to traverse an array A by executing $j = A[j]$ with some stride. The array elements are initialized with the indices of the next memory access. We measure the time consumption of a great number of such memory accesses and calculate the average consumption of each access. Listing 2 and Listing 3 give the P-chase kernel and the array initialization respectively. The memory access pattern can be inferred from the average memory access latency. The smallest memory latency indicates cache hit and bigger latencies indicate cache misses.

For simplicity, we define the notations for cache and P-chase parameters in Table 2. Note that we access GPU memory k times but only N/s array elements are accessed ($k >> N/s$). Memory access pattern is decided by the combination of N and s [19].

Saavedra *et al.* varied both N and s in one experiment to study CPU memory hierarchy [19, 20]. Volkov *et al.* applied the same method on a G80 GPU [7]. Wong *et al.* developed the footprint experiment: fixing s and varying N, to study the multi-level caches one by one of a Tesla GPU [8, 9]. Recently, Meltzer *et al.* used both Saavedra's and Wong's footprint experiment to investigate Fermi L1 and L2 data cache structure [10]. They utilized Saavedra's method to get an overall idea and then analyzed each cache structure with footprint experiment. Experimental results based on the two methods coincided with each other perfectly in [10]. However, we got different results of cache line size of texture L1 cache when we applied the two methods. What happened?

The problem is caused by the usage of total or average time consumption. It indicates the existence of cache miss, but little information on the miss percentage or the causes of cache miss. In order to get all the information, we need

to know the full memory access process. Motivated by the above, we design a fine-grained benchmark utilizing GPU shared memory to display the latency of every memory access.

```
start_time = clock();
for(it=0;it<iterations;it++){
    j=A[j];
    }
end_time=clock();
    //average memory latency
tvalue=(end_time-start_time)/
    iteration;
```

Listing 2. P-chase Kernel

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

Listing 3. Array Initialization

```
__global__ void KernelFunction(...){
    //declare shared memory space
__shared__ unsigned int s_tvalue[];
__shared__ unsigned int s_index[];

for(it=0;it<iterations;it++) {
    start_time=clock();
    j=my_array[j];
        //store the element index
    s_index[it]=j;
    end_time=clock();
        //store the access latency
    s_tvalue[it]=end_time-start_time;
    }
}
```

Listing 4. Fine-grained P-chase Kernel

Listing 4 gives the kernel code of our fine-grained benchmark. We launch one thread on GPU devices each time. By repeatedly executing $j = my\_array[j]$, the thread visits the array elements whose indices are multiples of s. For ease of analysis, we also record the visited array indices. We time each procedure of reading the array element and storing the index into the shared memory. Because the CUDA compiler automatically omit meaningless data readings, we need to write the shared memory with the updated index, namely the index of the next element rather than of the current one [15]. In addition, for operations of calling *clock*() and writing shared memory are synchronous, to get convincible memory latency, we need to imbed writing shared memory in the timing. Although this brings extra measurement error, the error is relatively small compared with the memory latency and does not affect the disclosure of memory structures.

Specifically, we apply our benchmark with strategies below to get the cache characteristics. N and s are calculated on every word (i.e., the length of an unsigned integer) basis.

(1) Determine C: $s = 1$. We initialize N with a small value and increase it gradually until cache misses appear. C equals the maximum N where all memory accesses fit in the cache.

(2) Determine b: $s = 1$. We begin with $N = C + 1$ and increase N gradually again. When $N < C + b + 1$, only memory accesses to the first cache set are missed. If $N = C + b + 1$, memory accesses to the second cache set are also missed. Based on the increase of missed cache lines, we can find b.

(3) Determine a: $s = b$. We start with $N = C$ and increase N at the granularity of b. The cache miss patterns are decided by N, as shown in Fig. 3. Every increment of N causes cache misses of a new cache set. When $N > C + (a-1)b$, all cache sets are missed. We can get a from cache miss patterns accordingly. The cache associativity, i.e., number of cache ways, equals $C/(ab)$.

(4) Determine cache replacement policy. In our fine-grained benchmark, we set $k > N/s$ so that we traverse the array multiple times. Because the array

Table 2. 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	No. of cache sets	k	iterations

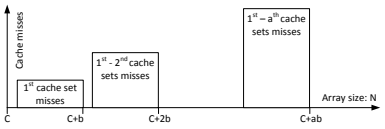

Fig. 3. Cache Miss Patterns of Various N

elements are accessed in order, if the cache replacement policy is LRU, then the memory access process should be periodic. For example, given a cache shown in Fig. 2, $N = 13$ and $s = 1$, the memory access pattern is repeated every 13 data loadings: whenever we visit the i^{th} array element, it is fixed as a cache miss/hit. If the memory access process is aperiodic, then the replacement policy cannot be LRU. Under this circumstance, we set $N = C + b, s = b$, and follow the full memory access process with a considerable k. All cache misses belong to the first cache set. Because we also have information of accessed array indices, we can find which cache line is replaced of every cache miss. Based on this method, we get the particular Fermi L1 data cache replacement policy.

In addition, we design a special array initialization with non-uniform strides. We are motivated to exhibit as many memory latencies as possible within one experiment, similar with [19]. We apply this initialization on studying various global memory latencies. We manually fill the array elements with the indices rather than execute Listing 3.

To conclude, we propose a fine-grained benchmark that utilizes GPU shared memory to store all memory access latencies. This benchmark enables exhaustive study of GPU cache structures. We explore the global memory and texture memory hierarchy with our fine-grained benchmark. We also design a sophisticated array initialization to exhibit various memory latencies within one experiment.

Experimental platform: the CPU is Intel Core$^{\text{TM}}$ i7-3820 @3.60 GHz with PCI-e 3.0. Our operating system is a 64-bit CentOS release 6.4. CUDA runtime/driver version is 5.5. We use CUDA compiler driver NVCC, with options *-arch=sm_21* and *-arch=sm_35* to compile all our files on Fermi and Kepler devices respectively.

4 Experimental Results

4.1 Shared Memory

GPU shared memory is on-chip and non-cached. In many CUDA applications, researchers utilize shared memory to speed up memory accesses [3–5]. However, based on our experimental results, the shared memory access can be slower than global memory access if there are considerable bank conflicts. In this section, we investigate the impact of bank conflict on shared memory access latency.

Fig. 4 illustrates a 2-way bank conflict caused by stride memory access on Fermi architecture. The bank width is 4-byte. E.g., word 0 and word 32 are mapped into the same bank. If the stride is 2, thread 0 and thread 16 will visit word 0 and word 32 respectively, causing a bank conflict. The way of bank conflict equals the greatest common divisor of stride and 32. There is no bank conflict for odd strides.

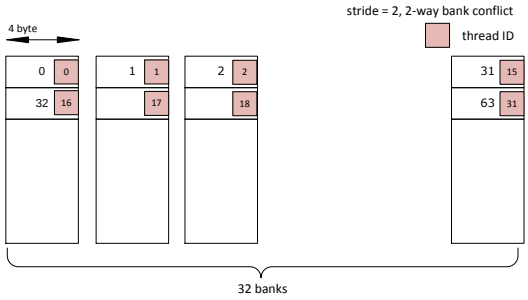

Fig. 4. Fermi Shared Memory Banks

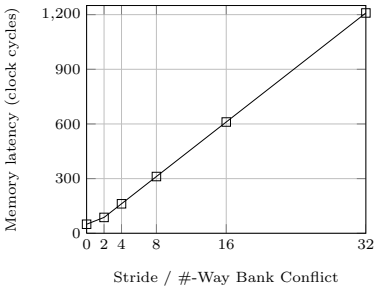

Fig. 5. Latency of Fermi Bank Conflict

In Fig. 5, we plot the average shared memory latency of Fermi. If stride is 0, i.e., the data is broadcasted [15], memory latency is about 50 cycles. Memory latency increases to 88 cycles for 2-way bank conflict, and 1210 cycles for 32-way bank conflict. The increment indicates that memory loads of different spaces in the same bank are executed sequentially. GPU kernel efficiency could be seriously degraded when there are considerable bank conflicts.

Kepler shared memory outperforms Fermi in terms of avoiding bank conflicts [18]. Kepler improves shared memory access efficiency by introducing the 8-byte wide bank. The bank width can be configured by calling *cudaDeviceSetShared-MemConfig()* [15]. Fig. 6 gives a comparison of memory mapping between the two modes: 4-byte and 8-byte. We use 32-bit data so that each bank row contains two words. In 8-byte mode, 64 successive integers are mapped into 32 successive banks. In 4-byte mode, 32 successive integers are mapped into 32 successive banks. Different from Fermi, bank conflict is only caused by two or more threads accessing different bank rows.

Fig. 7 shows the Kepler shared memory latency with even strides of both 4-byte and 8-byte modes. When stride is 2, there is no bank conflict for either 4-byte or 8-byte mode, whereas there is 2-way bank conflict on Fermi. When stride is 4, there is 2-way bank conflict, half as Fermi. When stride is 6, there is 2-way bank conflict for 4-byte mode but no bank conflict for 8-byte mode. We illustrate this situation in Fig. 6. For 4-byte mode, half of the shared memory banks are visited. Thread i and thread $i + 16$ are accessing separate rows of the same bank ($i = 0, ..., 15$). For 8-byte mode, 32 threads visit 32 banks without conflict. Similarly, 8-byte mode is superior to 4-byte mode for other even strides whose number is not a power of two.

In summary, Kepler can provide higher shared memory access efficiency by the following two ways. First, compared with Fermi, 4-byte mode Kepler shared memory can halve the chance of bank conflict. Second, 8-byte mode further reduces bank conflict.

4.2 Global Memory

The CUDA term, global memory, includes physical memory spaces of DRAM, L1 and L2 cache. Previous studies show that there are two levels of TLB: L1 TLB

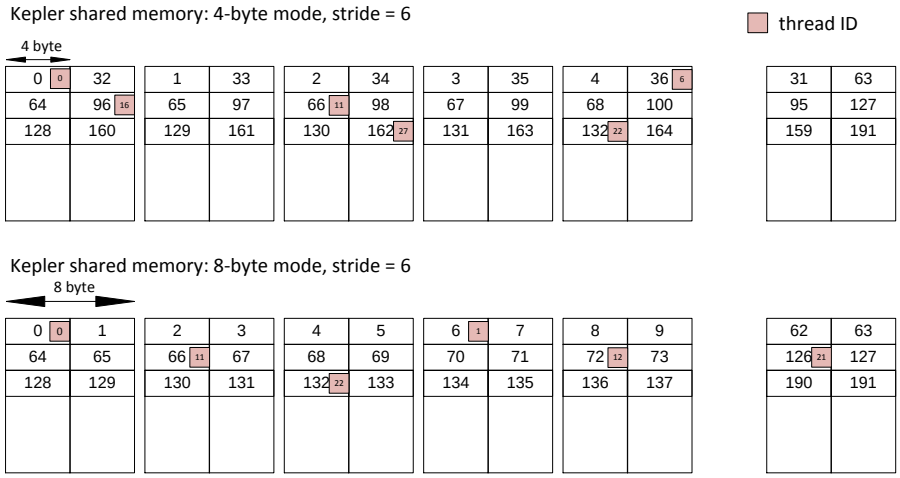

Fig. 6. Kepler Shared Memory Access: 4-Byte Bank v.s. 8-Byte Bank (Stride=6)

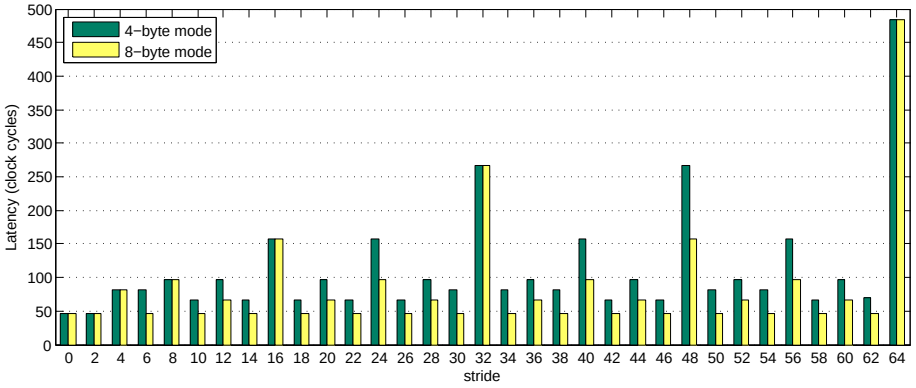

Fig. 7. Latency of Kepler Shared Memory: 4-Byte Mode v.s. 8-Byte Mode

and L2 TLB to support GPU virtual memory addressing [7–9]. In this section, we first exhibit memory latencies of various access patterns. We visit global memory spaces with non-uniform strides to collect as many access patterns as possible within one experiment. We then focus on the architectures of micro units, such as L1 data cache and TLB.

Table 3. Global Memory Access Latencies of Various Patterns

Pattern	1	2	3	4	5	6
Kepler	230	236	289	371	734	1000
Fermi: L1 enabled	116	404	488	655	1259	–
Fermi: L1 disabled	371	398	482	639	1245	–

Pattern	1	2	3	4	5	6
Data cache	hit	hit	hit	miss	miss	miss
L1 TLB	hit	miss	miss	hit	miss	miss
L2 TLB	–	hit	miss	–	miss	miss

Global Memory Latency We collect global memory latencies of six access patterns in Table 3. Fermi global memory accesses are cached in both L1 and L2 data cache. The L1 data cache can be manually disabled by applying compiler

Table 4. Common GPU Cache Characteristics

Parameters	Fermi L1 data cache	Fermi/Kepler L1 TLB	Fermi/Kepler L2 TLB	Fermi/Kepler texture L1 cache
N	16 KB	32 MB	130 MB	12 KB
b	128 byte	2 MB	2 MB	32 byte
a	32	1	7	4
LRU	no	yes	yes	yes

option *-Xptxas -dlcm=cg*. We measure the memory latencies with Fermi L1 data cache both enabled and disabled, as listed in the last two rows of left side table.

Note that Kepler gets a unique memory access pattern (pattern 6 in Table 3) of page table context switching. We find that when a kernel is launched on Kepler, only memory page entries of 512 MB are activated. If the thread visits a page entry that is inactivated, the hardware needs a rather long time to switch among the page tables. This is so-called page table "miss" in [10].

View from Table 3, on Fermi devices, if the data is cached in L1, the L1 TLB miss penalty is 288 cycles. If data is cached in L2, the L1 TLB miss penalty is 27 cycles. Because the latter penalty is much smaller, we infer that physical memory places of L1 TLB and L2 data cache are close. Similarly, physical memory places of L1 TLB and L2 TLB are also close, which means that L1/L2 TLB and L2 data cache are off-chip shared by all SMs.

We can also find that unless the L1 data cache is hit, caching in L1 does not really save time. For four out of five patterns, enabling L1 data cache is about 6 or 15 clock cycles slower than disabling it.

Another interesting finding is that unless Fermi L1 data cache is hit, Kepler is about 1.5-2 times faster than Fermi although it does not utilize L1 data cache. Kepler has much smaller L2 data cache memory latency, L2 data cache miss penalty and L1/L2 TLB miss penalty. It confirms the superiority of Kepler in terms of memory access efficiency.

Fermi L1 Data Cache We list the characteristics of Fermi L1 data cache and some other common caches in Table 4. Fermi cache can be either 16 KB or 48 KB, and we only report the 16 KB case in this paper due to limited space. According to [10], cache associativity is 6 if it is configured as 48 KB .

One distinctive feature of Fermi L1 cache is that its replacement policy is not LRU, because the memory access process is aperiodic. We apply our fine-grained benchmark on arrays varying from 16 KB to 24 KB to study the replacement policy. Fig. 8 gives the L1 cache structure based on our experimental results. L1 cache has 128 cache lines mapped into way 1-4. Of all 32 sets, one cache way has triple the chance to be replaced than other three ways. It is updated every two cache misses. In our experiment, way 2 is replaced most frequently. The replacement probabilities of the four cache ways are $\frac{1}{6}, \frac{1}{2}, \frac{1}{6}, \frac{1}{6}$ respectively.

Fig. 9 shows the effect of the non-LRU replacement policy. The y-axis label, cache miss rate, is obtained from dividing the missed cache lines by the total cache lines. For the traditional cache, the maximum cache miss rate should be 100% [9, 19] yet the non-LRU Fermi cache has a maximum miss rate of 50% based on our experimental result.

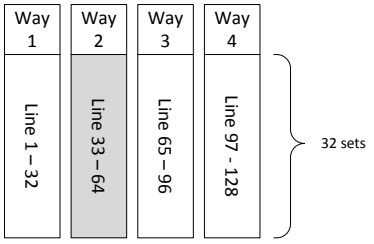

Fig. 8. Fermi L1 Cache Mapping

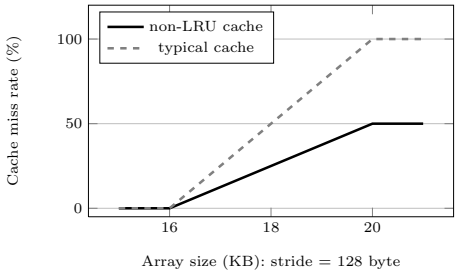

Fig. 9. Miss Rate of Non-LRU Cache

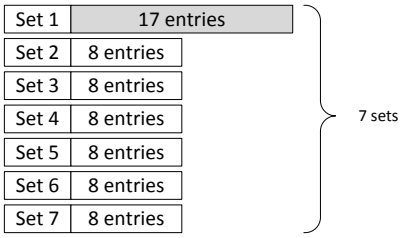

Fig. 10. Kepler/Fermi L2 TLB Structure

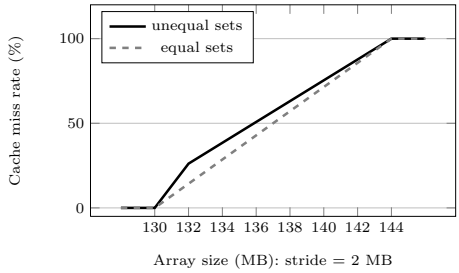

Fig. 11. Miss Rate of Unequal-Set Cache

Fermi/Kepler TLBs Based on our experimental results, Fermi and Kepler have the same TLB structure: L1 TLB is 16-way fully-associative and L2 TLB is set-associative with 65 ways. The L2 TLB has unequal cache sets as shown in Fig. 10.

We plot the L2 TLB miss rate in Fig. 11. For the traditional cache, the miss rate increases linearly while the measured miss rate increases piecewise linearly: $N = 132$ MB causes 17 missed entries at once and varying N from 134 MB to 144 MB with $s = 2$ MB causes 8 more missed entries each time. Thus the big set has 17 entries, while the other six sets have 8 entries.

4.3 Texture Memory

Texture memory is read-only and cached. Fermi/Kepler texture memory also has two levels of cache. Here we discuss texture L1 cache only.

Table 5. Texture Memory Access Latency

Device	Texture cache		Global cache	
	L1 hit	L1 miss, L2 hit	L1 hit	L1 miss, L2 hit
Fermi	240	470	116	404
Kepler	110	220	–	230

Texture L1 Cache We bind an unsigned integer array to linear texture, and fetch it with *tex1Dfetch*(). We measure the texture memory latency of both Fermi and Kepler as listed in Table 5. The Fermi texture L1 cache hit/miss consumption is about 240/470 clock cycles and Kepler texture L1 cache hit/miss consumption is about 110/220 clock cycles. The latter one is about two times faster.

In Table 5, we also find that Fermi texture L1 cache access is much slower than global L1 data cache access. In contrast, Kepler texture memory management is of low cost.

In addition, our experimental results suggest a special set-associative addressing as shown in Fig. 12. The 12 KB cache can store up to 384 cache lines. Each line contains 8 integers/words. 32 successive words/128 successive bytes are mapped into successive cache sets. The 7-8th bits of memory address define the cache set, while in traditional cache design, the 5-6th bits define the cache set. Each cache set contains 96 cache lines. The replacement policy is LRU. This mapping is optimized for 2D spatial locality [14]. Threads of the same warp should visit close memory addresses to achieve best performance, otherwise there would be more cache misses.

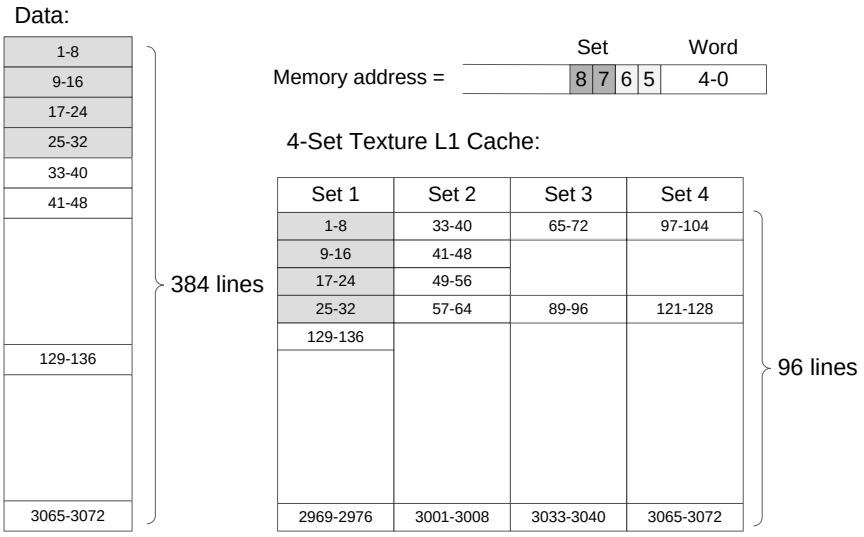

Fig. 12. Fermi & Kepler Texture L1 Cache Optimized Set-Associative Mapping

5 Conclusions

In this paper, we have explored many unexposed features of memory system of Fermi and Kepler GPUs. Our fine-grained benchmark on global memory and texture memory revealed some untraditional designs used to be ignored. We also explained the advantage of Kepler's shared memory over Fermi. We consider our work inspiring for both GPU application optimization and performance modeling. However, our work still has two limitations. First, we restrict ourselves to single thread or single warp memory access. The memory latency could be much different due to the multi-warp scheduling. Second, due to our preliminary experimental results on L2 cache investigation, the L2 cache design is even more complicated. Our fine-grained benchmark is incapable of L2 cache study due to the limited shared memory size. We leave these two aspects for our future study.

Acknowledgement

This research work is partially supported by Hong Kong GRF grant HKBU 210412 and FRG grant FRG2/13-14/052.

References

1. Li, Q., Zhong, C., Zhao, K., Mei, X., Chu, X.: Implementation and analysis of AES encryption on GPU. In: High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on. (2012) 843–848
2. Chu, X., Zhao, K.: Practical random linear network coding on GPUs. In: GPU Solutions to Multi-scale Problems in Science and Engineering. Springer (2013) 115–130
3. Li, Y., Zhao, K., Chu, X., Liu, J.: Speeding up K-Means algorithm by GPUs. Journal of Computer and System Sciences **79** (2013) 216–229
4. Micikevicius, P.: 3D finite difference computation on GPUs using CUDA. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM (2009) 79–84
5. Zhao, K., Chu, X.: G-BLASTN: accelerating nucleotide alignment by graphics processors. Bioinformatics (2014)
6. Mei, X., Yung, L.S., Zhao, K., Chu, X.: A measurement study of GPU DVFS on energy conservation. In: Proceedings of the Workshop on Power-Aware Computing and Systems. Number 10, ACM (2013)
7. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Number 31, IEEE Press (2008)
8. Papadopoulou, M., Sadooghi-Alvandi, M., Wong, H.: Micro-benchmarking the GT200 GPU. Computer Group, ECE, University of Toronto, Tech. Rep (2009)
9. Wong, H., Papadopoulou, M.M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, IEEE (2010) 235–246
10. Meltzer, R., Zeng, C., Cecka, C.: Micro-benchmarking the C2070. In: GPU Technology Conference. (2013)
11. NVIDIA Corporation: Fermi Whitepaper. (2009)
12. NVIDIA Corporation: Kepler GK110 Whitepaper. (2012)
13. NVIDIA Corporation: Tuning CUDA Applications for Kepler. (2013)
14. NVIDIA Corporation: CUDA C Best Practices Guide - v6.0. (2014)
15. NVIDIA Corporation: CUDA C Programming Guide - v6.0. (2014)
16. NVIDIA Corporation: Tuning CUDA Applications for Maxwell. (2014)
17. Micikevicius, P.: Local Memory and Register Spilling. NVIDIA Corporation. (2011)
18. Micikevicius, P.: GPU performance analysis and optimization. In: GPU Technology Conference. (2012)
19. Saavedra, R.H.: CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking. PhD thesis, EECS Department, University of California, Berkeley (1992)
20. Saavedra, R.H., Smith, A.J.: Measuring cache and TLB performance and their effect on benchmark runtimes. Computers, IEEE Transactions on **44** (1995) 1223–1235