Thorough Evaluation of GPU Shared Memory Load and Store Instructions

Satoshi Okamoto, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {okamoto, yasuaki, nakano}@cs.hiroshima-u.ac.jp

Jacir L. Bordim
Department of Computer Science
University of Brasilia
70910-900, Brasilia - DF - Brazil
Email: bordim@unb.br

***Abstract*—This work focuses on measuring the number of GPU clock cycles necessary to execute load/store instructions in both bank conflict and bank conflict-free shared memory access patterns. To this end, a varying number of parameters have been considered in the experiments, including the number of warps (w), the number of memory bank conflicts (k) as well as the number of load/store instructions (l) per warp. From the analysis of the experimental results, it was possible to obtain an estimate (E) on the number of the clock cycles necessary to execute l load/store instructions. The estimate is given by $E = w \cdot l \cdot k \cdot c_1 + c_2$, where c_1 and c_2 are constants assuming values 1.047 and 337.7, respectively. From the above results, we believe that obtained estimated can be used as an approximation on the number of clock cycles necessary to execute load and store instructions.**

***Keywords*-bank conflict; shared memory; GPU; clock cycle measurement.**

I. Introduction

Owing to its programmability and highly parallel processing features, GPUs (Graphics Processing Units) have captured the attention of many application developers. When the application software uses a large number of parallel threads, GPUs are usually more efficient than multicore processors since they have hundreds of processor cores and very high memory bandwidth [1].

The CUDA (Compute Unified Device Architecture), introduced by NVIDIA, allows the developer to gain access to the virtual instruction set and memory of the parallel computational elements of an NVIDIA GPU [2]. CUDA architecture allows the access to two types of memories: (1) the *shared memory*; and (2) the *global memory*. The shared memory is an extremely fast, on-chip memory, with lower capacity (usually between 16 to 64 Kbytes). The global memory, on the other hand, is implemented as an off-chip DRAM with much higher capacity (currently around 1.5 to 6 Gbytes). Despite that, access latency is much higher than that of shared memory. Efficient usage of the shared memory and the global memory is mandatory to accelerate applications using GPUs [1]. This work focuses on shared memory access patterns and its impact on performance.

The GPU is built around an array of Streaming Multiprocessors (SMs), where a multithreaded program is executed in SIMT (Single-Instruction, Multiple-Thread) fashion [2]. A thread block is assigned to a single SM. Each SM is comprised of a shared memory that can be accessed by the threads in the block assigned to it. The threads are connected to shared memory banks (MBs) through the memory management unit. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the (i mod m)-th MB, where m is the number of the MBs [3]. The multiprocessor creates, manages, schedules, and executes threads in groups of parallel threads, called *warps*. In this work we assume that number of threads in a warp is the same as the number m of memory banks. The best performance is achieved when all threads in a warp access different memory banks. Figure 1(a) depicts a number of threads (T) accessing different memory banks (MB).

When a number of threads in a warp attempt to access different addresses of the same memory bank, a *conflict* occurs [4]. In this case, the shared memory access requests are processed sequentially. The development of efficient algorithms should avoid conflicts as much as possible. In this work we are interested in measuring the number of GPU clock cycles necessary to execute *load/store* instructions, in bank conflict and bank conflict-free situations. Let k denote the *maximum* number of threads in a given clock cycle that attempt to access different addresses of memory bank MB(i), ($0 \leq i \leq 31$). Whenever $k > 1$, we say that there is a k-*congestion* access, while a 1-*congestion* denotes a conflict-free access. The number k of congestion takes values in the interval $[1, m]$. Figure 1 shows three congestion examples, with 1, 4, and 2-congestion, respectively. The figure illustrates the cases where a number of threads attempt to access memory location on the same memory banks, resulting in 4 and 2-congestion scenarios (Figure 1(b) and (c), respectively). Recall that 1-congestion is the most favorable access pattern, as it has no memory conflicts.

II. Shared Memory Clock Cycle Measurement

The NVIDIA GeForce GTX780Ti GPU has been used to measure the shared memory clock cycles while executing *load/store* instructions. The GTX780Ti has 3GB of memory, 2880 processing cores (15 Streaming Multiprocessors, each having 192 processing cores) running at 928MHz. In this

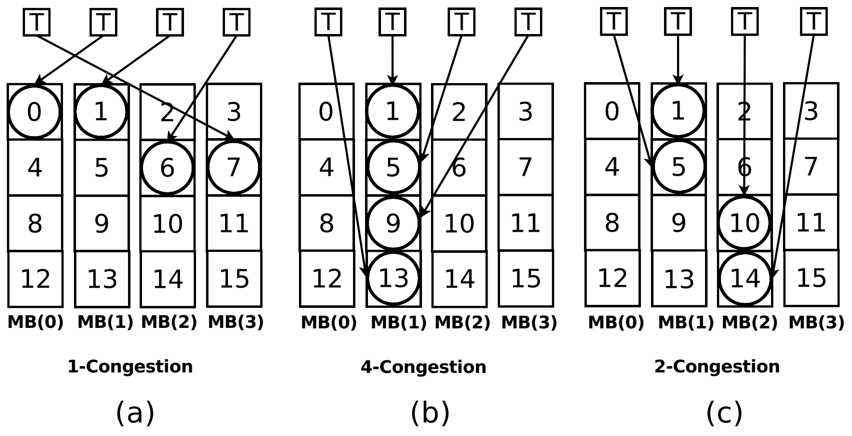

Figure 1: Shared memory congestion examples with $m = 4$.

Inline PTX Assembly Code Snippet

```
1.  asm volatile
2.  (  ...
3.    "bar.sync 0;\n\t"
4.    "mov.u32 %0, %%clock;\n\t"
5.     shared memory load/store instructions
7.    "mov.u32 %1, %%clock;\n\t"
8.    .... );
```

Figure 2: Assembly code snippet for measuring clock cycles for instructions load/store.

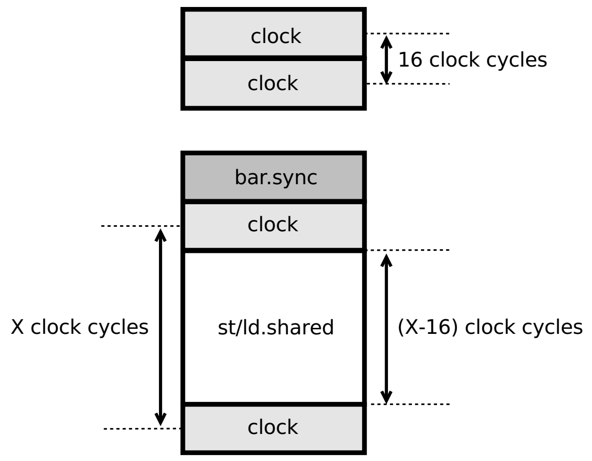

Figure 3: Obtaining the number of clock cycles to execute load/store instructions.

work we have restricted the kernel execution to a single block.

To obtain the number of clock cycles, the "inline PTX assembly language" for NVIDIA GPUs has been used [5]. Figure 2 shows a snippet of the inline PTX assembly code. Lines 4 and 7 of the snippet are used to record the number of clock cycles taken by a load/store instruction (shown in line 5). Note that a *barrier synchronization* is used to ensure that all threads are synchronized before the first clock instruction. Let X denote the number of clock cycles to execute a load/store instruction as well as the clock measuring instructions, as depicted in Figure 3. Note that two consecutive clock instructions take exact 16 clock cycles in the GTX780Ti GPU. Thus, by computing $X - 16$, it is possible to obtain the number of clock cycles to execute load/store instructions in line 5. The aforementioned inline PTX code has been inserted into the CUDA C program, thus allowing to measure the number of clock cycles to execute shared memory load/store instructions.

III. Experiments

This section presents the experimental setting used to measure the number of clock cycles to execute load/store instructions in the shared memory of the GeForce GTX780Ti GPU. The following parameters have been considered in the experiments:

- the number w of warps;
- the number k of congestion; and
- the number l of load/store instructions per warp.

The number of warps and the number of instructions take values in the interval $[1, 32]$. As there are 32 threads in a warp, k can be at most 32. The results are taken from the average of 100 executions. Note that the GPU may store data into local registers to speed up computations. As this could interfere with the measurements, *volatile* instructions were used to avoid data of being kept into the shared

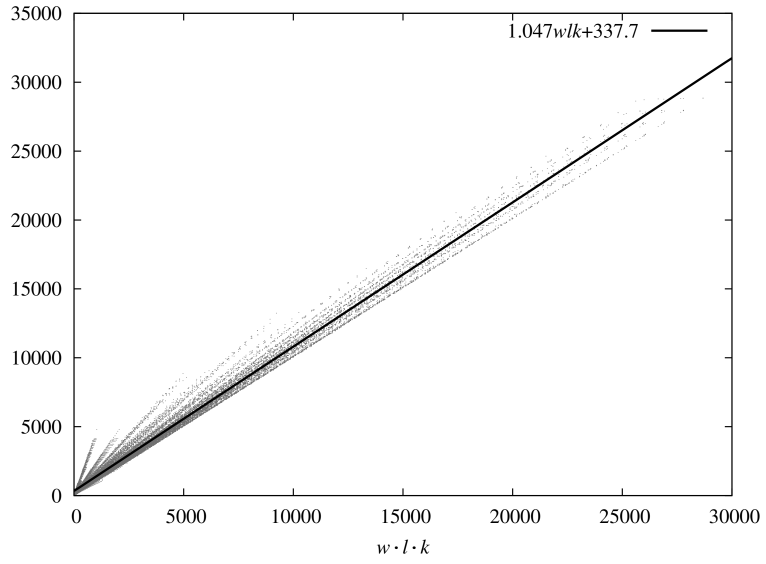

Figure 4: Number of clock cycles and best fit line for load instructions with $w \cdot l \cdot k$.

memory registers [2]. The shared memory has been set to work in 64-bit mode in the experiments. Furthermore, all load/store instructions in the experiments manipulate 64-bits floating numbers. In the GeForce GTX780Ti, each SM has 32K registers capable of storing 64-bits floating numbers. Hence, the number of instructions and the number of warps considered in this work are bounded by this limit. Due to space limitation, only the clock cycle measurement results for load instructions will be shown. We note, however, that store instructions have shown similar results.

As mentioned in [3], it is supposed that modern GPU architecture employs multistage interconnection networks to route memory access requests. However, memory access becomes serialized in the event of memory bank conflict. In such cases, it is presumable that the benefits of a fast memory access to be significantly reduced. Furthermore, the number of warps usually implies an increase on the number of instructions to be executed. Thus, it is expected that the number of clock cycles to increase significantly with the number of warps and load/store instructions. Given the above facts, we conjecture that:

1) an increase on the number of bank conflicts would produce a similar increase in the number of clock cycles necessary to complete the execution of the load/store instructions; and
2) increasing the number of warps and instructions would yield an equivalent increase in the number of clock cycles.

To verify the above, we have conduced a number of experiments with a varying number of warps, instructions and congestions. As these variables are not independent, we have combined the averaged results so that a trend line could be obtained. To this end, the linear regression method has been used. The resulting best fit line is shown in Figure 4. The abscissa shows the averaged product of $w \cdot l \cdot k$, while the ordinate shows the number of clock cycles necessary to complete the execution. As can be observed in the figure, the best fit line for the estimated number of clock cycles can be expressed by $E = w \cdot l \cdot k \cdot c_1 + c_2$, where c_1 and c_2 are constants assuming values 1.047 and 337.7, respectively. For instance, the obtained experimental results for $w = l = 32$ and $k = 8$ was 8524.46 clock cycles. Using the above estimate, the number of clock cycles is $E = 8914.72$, a difference of less than 5%. With twice as much bank conflicts, that is $k = 16$ with $w = l = 32$, the estimated the number of clock cycles raises to $E = 17491.75$. From the above results, we believe that the presented estimate can be used to obtain an approximation on the number of clock cycles necessary to execute load and store instructions.

References

[1] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[2] NVIDIA Corporation, *CUDA C Programming Guide*, 6.5 ed., August 2014.

[3] K. Nakano, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

[4] K. Nakano, S. Matsumae, and Y. Ito, "The random address shift to reduce the memory access congestion on the discrete memory machine," in *International Symposium on Computing and Networking (CANDAR)*, pp. 95–103, Dec 2013.

[5] NVIDIA Corporation, *Inline PTX Assembly in CUDA*, 6.5 ed., August 2014.