GPGPU 를 위한 공유 메모리 최적화

쟌 느앗 프엉, 이명호*, 홍석원
명지대학교 컴퓨터공학과, 경기도 용인시 처인구 남동 산 38-2
e-mail: myunghol@mju.ac.kr

요 약

최근 GPU 의 뛰어난 부동 소수점 연산 능력을 활용하여 그래픽 이외에 다양한 응용 프로그램들의 병렬화 및 성능최적화가 활발하게 이루어지고 있다. 이러한 GPU 의 성능을 극대화하기 위해서는 메모리 계층구조 및 shared memory 를 비롯한 on-chip 메모리의 사용을 최적화하는 것이 필수적이다. 본 논문에서는 이러한 shared memory 의 사용을 최적화하기 위한 기법들을 제안하고, 이를 패턴 매칭 응용 프로그램에 적용하여 효용성을 검증한다.

Optimizing Shared Memory Accesses for GPGPU Computations

Nhat-Phuong Tran, Myungho Lee*, Sugwon Hong
Dept. of Compute Science and Engineering, Myongji University
38-2 San Namdong, Cheo-In Gu, Yong In, Kyung Ki Do, Korea

Abstract

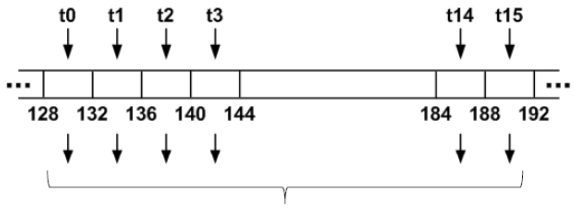
Recently, a lot of general-purpose application programs in addition to graphic applications have been parallelized for boosting their performance using Graphic Processing Unit (GPU)'s excellent floating-point performance. In order to maximize the application performance on GPUs, optimizing the memory hierarchy and the on-chip caches such as the shared memory is essential. In this paper, we propose techniques to optimize the shared memory, and verify its effectiveness using a pattern matching application program.

1. Introduction

Recently, the Graphic Processing Unit (GPU) is becoming increasingly popular for various applications. The latest GPU architectures have incorporated Shader, Vertex, Pixel units into multiple uniform programmable processing units or cores which were separate processing units in the earlier GPU architectures. With the new architecture, the number of fine-grain cores has dramatically increased and huge floating-point performance improvements are made possible [4], [5]. Furthermore, on-chip caches such as the shared memory and the texture/constant caches are built on chip to effectively reduce the memory access latencies, whereas previous GPU's memory hierarchy was designed mainly to maximize the memory bandwidths.

In order to optimize the application's performance on the GPU, it is crucial to optimize the on-chip memories such as the shared memory, texture/constant cache, etc. The shared memory, for example, is totally under programmer's control whereas the other texture/constant caches are managed with the hardware control. Therefore, user-level techniques to optimize the use of the shared memory have great influences on the overall applications' performance. In this paper, we propose techniques to optimize the shared memory by coalescing the global memory accesses and by avoiding bank conflicts using an efficient scheme to store the data returned from the off-chip global memory loads. By applying these techniques to a pattern matching application, Aho-Corasick algorithm [1], we've observed significant performance improvements.

In the following sections, we present our techniques (Section 2) and performance evaluation results (Section 3), with conclusions (Section 4).



(Figure 1) Coalesced accesses: 16 threads cooperate to read 64 bytes together

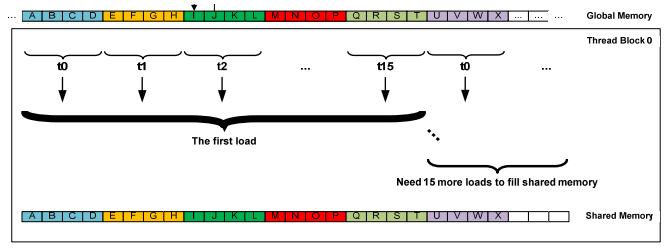
2. Optimizing Shared Memory Accesses

When data is loaded from the off-chip global memory to the on-chip shared memory for computations on the GPU, we need to consider techniques to optimize the memory performance. One of the most important performance considerations is to coalesce the global memory accesses. For example, let's assume that the input text data is buffered in a sequential fashion in the global memory and each character contains one byte. The data is divided amongst the multiple threads for the parallel computation. If each thread slides on its own data and naively loads each character from the global memory to the shared memory, each thread reads a long sequence of data sequentially. Thus the total latency to

* Corresponding author: Myungho Lee (myunghol@mju.ac.kr)

load data needed for each thread will be high. In order to solve this problem, we let threads of a block cooperate to read as a half warp and each thread read four bytes (one word of integer) at one time. Figure 1 above show the case where 16 threads cooperate to load 64 bytes data. These 16 requests are combined into 1 global memory access request as they fall within 128 bytes memory address boundaries. Thus it can save a lot of time to access the shared memory. It also saves a lot of the memory bandwidths.

When the data needed to be loaded into the shared memory is longer than the size of the data loaded by threads of a block at one time, threads of a block need to load data multiple times cooperatively. For example, if we assume the size of the data to be loaded into the shared memory as 1024 -bytes and there are 16 threads per block (see Figure 2). Because each thread read one 4-byte word at one time, 16 threads of a block need 1024 / (4*16) = 16 loads from the global memory to the shared memory to load up the 1024-bytes data.

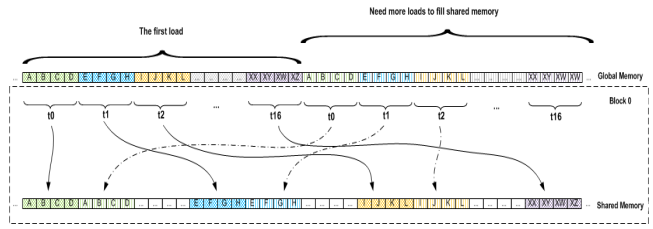


(Figure 2) Loading data from global memory to the shared memory (Data size to be loaded into the shared memory=1024-bytes, number of threads per thread block = 16, each thread reads 4 bytes at one time)

Synchronization of data will be done to make sure that all threads transferred data from the global memory to the shared memory successfully before threads process other works. After transferring data to the shared memory, each thread will apply AC algorithm on its own chunk as mentioned in global memory approach.

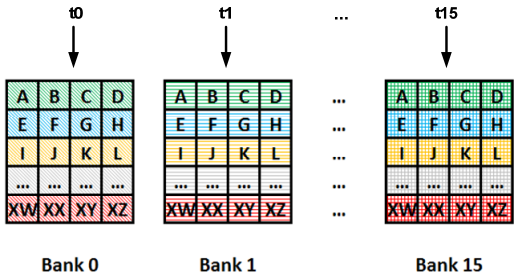
After reading data to the shared memory using coalescing explained above, each thread of block needs to apply its computations with respect to its portion of the data. In order to service a number of simultaneous accesses at the same time and achieve high memory bandwidth, the shared memory is divided into multiple banks. If there are multiple simultaneous accesses to the same bank, it results in a bank conflict. Conflicting accesses to be same bank are serialized which affects the program performance.

When the data for each thread is long, it spreads out through many banks. If each thread reads data in the sequential fashion, threads access data banks randomly. These accesses make a lot of bank conflicts and makes the program performance drop noticeable.



(Figure 3) Arranges the data loaded from global memory to shared memory to avoid bank conflicts

In order to avoiding the bank conflicts, stores of the loaded data from the global memory to the shared memory need to be carefully arranged. We let multiple threads cooperate to read as half warp and each thread loads four bytes at one time as explained above. In order to avoid the shared memory bank conflicts, bytes loaded by threads are stored alternately into the shared memory at addresses which are multiple of 16 and mapped to the same shared memory bank (see Figure 3). Using the above storing scheme, data stored in each bank of the shared memory has the data layout as shown in Figure 4. Therefore, each thread accesses its own data from single shared memory bank and the bank conflicts can be avoided.



(Figure 4) Data distributed to 16 banks of the shared memory to avoid bank conflicts

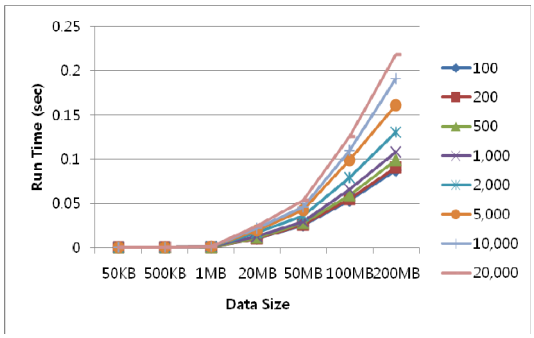
3. Experimental Results

We applied the techniques explained in Section 2 to a pattern matching algorithm, Aho-Corasick (AC). AC algorithm [1] is a multiple patterns matching algorithm which can simultaneously match a number of patterns for a given finite set of strings (or dictionary). The AC algorithm is commonly used in various pattern matching applications such as network intrusion detection, genome/protein matching for bio-sequence analysis, image processing, among many others. In network intrusion detection, for example, intensive pattern matching operations are performed for a deep packet inspection using the AC algorithm. In order to speed up the pattern matching and meet the real-time performance requirement imposed on these applications, achieving high performance for AC algorithm is crucial.

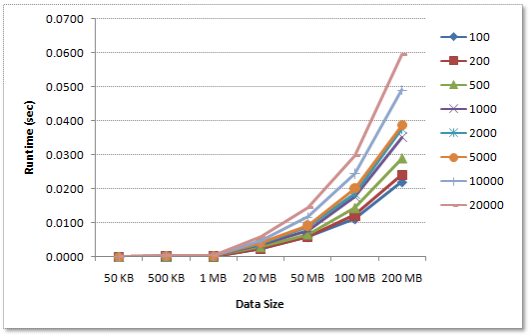
We implemented three experiments. In the first one, we used the off-chip global memory only (GM experiment). In the second one, we used the shared memory with the memory access coalescing only when accessing the global memory for loading the data to the shared memory (SM-1 experiment). In the third one, we further used an optimization technique to avoid the shared memory bank conflicts (SM-2 experiment). Our experiments were conducted on a system including Intel multi-core processor (2.2 Ghz Intel Core 2 Duo) with 2GB of main memory, Nvidia Geforce GTX 285 GPU with 240 thread processors (cores) organized in 8 streaming multiprocessors, operating at 1.48 Ghz with 1 GB device memory. The OS is Centos 5.5. We used the input data sizes in the range of 50KB – 200MB and the number of patterns in the range of 100 – 20,000.

Figure 5, 6, and 7 show the run times of 3 experiments for a range of input data sizes and for a range of patterns. First of all, the SM approaches (SM-1, SM-2) perform faster than the GM approach by 5~10 times. Therefore the benefit of the shared memory is significant. In general, the run times

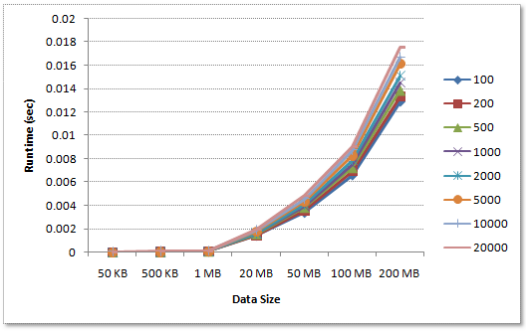
increase as the data size increase and as the number of patterns increase. As the input data size increases, however, the run time increase for the SM-2 slows down with the increase in the number of patterns.



(Figure 5) Runtimes of the GM using different input data sizes and different numbers of patterns

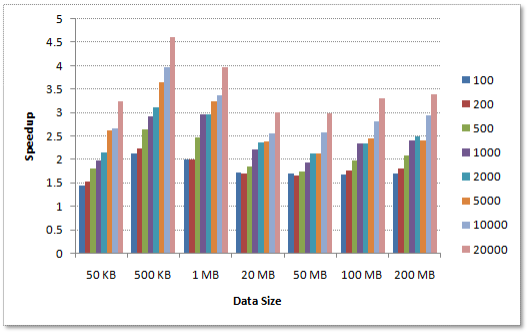


(Figure 6) Runtimes of the SM-1 using different input data sizes and different numbers of patterns



(Figure 7) Runtimes of the SM-2 using different input data sizes and different numbers of patterns

Figure 8 shows the speedup of the SM-2 approach over the SM-1. Thus it shows the benefit of avoiding the shared memory bank conflicts. The run time of the SM-2 performs from 1.5 ~ 4.6 times faster than the SM-1. This shows that the bank conflict avoidance has significant performance impacts.



(Figure 8) Speedups of the optimized SM approach (SM-2) compared with the SM-1 using different input data size and different number of patterns

4. Conclusion

In this paper, we proposed new techniques to optimize the shared memory performance on the GPU. The proposed approach coalesces the data accesses to the global memory and arranges the stores of the loaded data from the global memory to the shared memory so that the shared memory bank conflicts can be avoided. Applying the new techniques to the AC pattern matching algorithm achieved significant performance improvements.

5. Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (Grant No: 2012-0002264).

References

[1] A.V. Aho and M.J. Corasick, “Efficient string matching: An aid to bibliographic search”, Communications of the ACM, vol. 20, Session 10, Oct. 1977, pp. 761–772.

[2] Marc Norton, “Optimizing Pattern Matching for Intrusion Detection”, http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf, July 2004

[3] NVIDIA, “CUDA Best Practices Guide: NVIDIA CUDA C Programming Best Practices Guide – CUDA Toolkit 4.0”, May, 2011.

[4] NVIDIA, "Nvidia gtx280", http://kr.nvidia.com/object/geforce_family_kr.html

[5] Giorgos Vasiliadis, Spiros Antonatos, “Gnort: High Performance Network Intrusion Detection Using Graphics Processors”, RAID, 2008, pp. 116-134.