Experiences of the GPU Thread Configuration and Shared Memory

Dae-Hwan Kim

Abstract—Nowadays, GPU processors are widely used for general-purpose parallel computation applications. In the GPU programming, thread and block configuration is one of the most important decisions to be made, which increases parallelism and hides instruction latency. However, in many cases, it is often difficult to have sufficient parallelism to hide all the latencies, where the high latencies are often caused by the global memory accesses. In order to reduce the number of those accesses, the shared memory is instead used which is much faster than the global memory being located on a chip. The performance of the proposed thread configuration is evaluated on the GPU 960 processor. The experimental result shows that the best configuration improves the performance by 7.3 times compared to the worst configuration in the experiment. The experiences are also discussed for the shared memory performance when compared to that of the global memory.

Index Terms—GPU; Performance; Thread; Shared Memory.

I. INTRODUCTION

Nowadays, GPU (Graphics Processing Unit) processors are widely used in computation intensive and general-purpose parallel processing applications [1]-[4] due to their powerful SIMD (Single Instruction Multiple Data) engines. The GPU-powered applications cover a broad range of areas including machine learning, data science, finance, climate, manufacturing, media and entertainment, and safety [5].

NVIDIA Inc., the dominant GPU vendor, provides the parallel programming computing platform for general computing on GPUs, which is named CUDA (compute unified device architecture) [6]-[7]. The company develops APIs (Application Programming Interfaces), a lot of sample code, and many useful libraries such as CUFFT for fast fourier transform. CUDA is proprietary to NVIDIA Inc. and works only on NVIDIA’s GPUs. On the other hand, OpenCL [8] proposed by the Khronos Group is an open programming standard for general-purpose computations on heterogeneous systems targeting multi-core CPUs, DSPs (Digital Signal Processors), and GPUs.

To achieve high performance in the GPU programming, it is important to have sufficient parallelism to fully utilize the GPU cores. This also can hide instruction latencies where the high latencies are often caused by the global memory accesses. However, it is often difficult to have enough parallelism to hide all the global memory latencies. Thus, it is also necessary to relocate the data in the global memory into the faster memory. Shared memory is such a candidate which is located on chip, and thus it is much faster than the global memory.

In this paper, two techniques are evaluated, and their effects are measured on the GPU performance. The first one is the thread and block configuration, and the second is the data relocation from the global memory to the shared memory. The details and experiences are discussed in the evaluation section.

The rest of this paper is organized as follows. Section II shows the overview of the GPU architecture, and Section III presents the proposed technique. Detailed evaluation is presented in Section IV, and conclusions are given in Section V.

II. GPU ARCHITECTURE

The overall CPU and GPU architectures are significantly different as shown in Fig. 1 [6]. In CPU, a large portion of the chip is devoted to the cache and the control logic, and only a small portion is taken by arithmetic logic unit (ALU). On the other hand, GPU uses more transistors for the ALUs, which is suitable for a lot of parallel computation.

CUDA is a heterogeneous model where both CPU and GPU are used. Each of CPU and GPU has its own memory. Code run on CPU launches kernels which are executed on GPU. Kernels are executed by many GPU threads in parallel. Multiple threads are organized into thread blocks, and all thread blocks together form a grid. Fig. 2 shows the example of the grid, block, and thread configuration. There are 9 blocks in a grid, and each block has 4 threads.

GPU consists of multiple streaming multiprocessors (SMs) where there exists 1 to 30 SMs per chip. Each SM contains 8, 32, or 192 processing elements (PEs). SMs and PEs run blocks and threads, respectively. Fig. 3 shows the streaming processors and the processing elements in the GPU architecture. Blocks are serially distributed to all the SM by the hardware, and note that not all the blocks may be resident at the same time. A thread block consists of 32 threads called warps. Warps are the scheduling units in SM,
such that a warp is executed physically in parallel SIMD on a processor. SM schedules and executes warps that are ready to run, and warp scheduling overhead is zero in SM.

Fig. 2. Example of grid, block, and thread

Inside GPU, ALUs (processing elements) are split into blocks as shown in the Fig. 4 [6]. On a streaming processor, each thread runs the same instruction while the different streaming processor can run different code. CPU and GPU are referred to as host and device, respectively. There are several memory types in GPU also as shown in the Fig. 4. Global memory is shared by every processors. This memory is off-chip and large, but very slow. It takes hundreds of clock cycles to access the single memory address. The device memory contains the texture memory, which is of the same speed as the global memory, but there is a special texture cache inside the chip. This is the same for the constant cache for constant memory on the device memory. Local memory is private per thread, which contains auto variables and spilled variables. The last important memory type is the shared memory, each of which exists per multiprocessor. This memory is shared by the threads of the same block, and used for the inter-thread communication while threads in different blocks cannot cooperate. It takes only one cycle to access because it resides physically on GPU. Therefore, it can serve as a software managed cache for the global memory.

III. OPTIMIZATION TECHNIQUES

A. THREAD AND BLOCK CONFIGURATION

The performance of a CUDA kernel often depends on the number of threads per block, and the optimal configuration differs depending on the GPU hardware. A warp is not scheduled until all threads in the warp have finished the previous instruction, and some instructions can have high latencies such as global memory access instructions. Therefore, it is necessary to have enough warps to keep the GPU busy during the waiting time. The number of threads per block is generally set to be a multiple of warp size, and 128 to 256 threads per block are normally known to be a good initial configuration.

Fig. 5. Overall loop structure of the program

Fig. 5. shows the overall computation structure of the benchmark program. The 1536x1536x1536 voxel images are reconstructed by the 480 view projections. This computation is very expensive, and it is important to have enough parallelism to utilize the GPU efficiently. The maximum number of threads per block is 1,024 even in the latest CUDA compute capability 7.0, and the maximum x-dimension, y-dimension, and z-dimension of a grid of thread blocks can be $2^{31}$, 65,535, and 65,535, respectively. Because 1,536 threads per block is not possible, 768(1,536/2) is chosen as the number of threads, and the final configuration is decided to be 768 threads and 1,536 blocks as shown in Table I. In this configuration, each block processes each y, and one thread processes two x values and 1,536 z values. For example, thread number 0 in block number 0 processes 0 value for y, 0 and 768 x values, and 1,536 z values ranging from 0 to 1,535.

| Category | Number |
|----------|--------|
| blocks   | 1,536  |
| threads  | 768    |

TABLE I: OPTIMAL CONFIGURATION FOR CODE IN FIG. 5

DOI: http://dx.doi.org/10.24018/ejers.2018.3.7.788
B. Shared Memory Optimization

GPU provides shared memory, which is much faster than global memory. The block of threads runs on an SM, each of which has a small amount of shared memory, which can be accessed by the threads in a block. The maximum amount of shared memory per block is 48KB while the maximum amount of shared memory per multiprocessor ranges from 16KB to 112Kb. For example, if each SM has 32KB of shared memory, and there are 8 thread blocks that can simultaneously run on an SM, then the maximum amount of shared memory available to each block is 4KB.

To improve the kernel performance, it is often necessary to minimize the number of the global memory accesses because of their long waiting time. One simple way is to use shared memory as a scratchpad or a cache for the frequently used global memory data. However, careful consideration is necessary because the shared memory size is limited such that it is not big enough to accommodate all the data in the global memory.

The shared memory consists of multiple banks where the number of memory banks is 16 or 32 depending on the CUDA compute capability. This enables concurrent accesses across the different memory banks, but bank conflicts inevitably occur when multiple threads simultaneously access the same bank. This results in serialized access, and accordingly performance degradation.

Consider the code in the Fig. 8 where Proj, and Voxel are global variables. For each view, voxel values are computed using the Proj data which is different for each view. The range of z values are reduced for description. The total number of Voxel data accesses is 0.5 Tera which is 480 (view) x 1536 (y) x 1536(x) x 256(z) x 2 (read, write). Thus, it is not easy have enough parallelism to hide all the latencies of this global memory accesses. Shared memory version is suggested to reduce the latencies by these accesses.

Voxel data in global memory is replaced by its shared memory version denoted by S_Voxel, which is declared with the CUDA C keyword __shared__ to make this variable resident in the shared memory. At each step, data is copied from the global memory to the shared memory. Then S_Voxel in the shared memory is computed within four nested loops. In the last part of the code, the final value is rewritten to Voxel global memory. Various shared memory versions are tried with its thread configuration adequate for the shared memory size. However, no significant improvement is achieved with any shared memory version.

IV. Evaluation

Experiments are performed on GTX 960 processor [10] and GTX Titan X Pascal processor [11]. Table II shows the specification of the GTX 960 processor. It has 1,024 cores, and base and boost clocks are 1,127MHz, and 1,178MHz, respectively. Memory clock is 7Gbps, and standard config is 2GB. Interface width and band width are 128 bits, and 112 GB/sec, respectively.
The number of cores is 3,584, and base and boost clocks are 1,417MHz, and 1,531MHz, respectively. Memory clock is 10Gbps, and standard configuration contains 12GB. Interface and band widths are 384 bits, and 480 GB/sec, respectively.

Table IV shows the performance of each configuration on the GTX 960 processor where the baseline is the kernel with 192 blocks and 2 threads. In the experiment, the range of y values is reduced to 368 values ranging from 0 to 367, x from 0 to 1,535, and z from 0 to 255 to meet the GTX 960 memory constraint. The best performance is obtained when 368 blocks and 768 threads per block are deployed for the kernel, which increases parallelism to utilize GPU cores more efficiently. To compare block and thread preference, the total 384 threads are partitioned in seven configurations such as 192 blocks and 2 threads per block. In the benchmark program, the preference of block or thread is not evident in the partition, and there is no significant performance improvement in any of the seven configurations.

Table V compares the performance on the GTX960 and Titan X processors. For Titan X, no reductions are made, and the numbers of view, x, y, and z values are 480, 1,536, 1,536, and 1,536, respectively. On the GTX 960 processor, the reductions are made for the benchmark program to fit into the GTX960 memory. The iteration counts are reduced such that the numbers of view, y, and z values are 48, 384, 256, respectively. The number of x values is not reduced. Overall, the iteration count is reduced by 240 in total. But due to the advanced computational power of the Titan X processor, the execution time is not increased by the amount of 240, and the increase ratio is just by 25.7(180/7).

V. CONCLUSION

In this paper, the performance of the block and thread configuration is evaluated on the GTX 960 and GTX Titan X processors. The efficient configuration results in a significant performance improvement, and it is important to have enough parallelism. The shared memory efficiency is not clear for some cases, and may require careful use of the shared memory considering bank conflicts. The evaluation on other GPU processors remains as a future work.

REFERENCES

[1] E. S. Larsen, D. McAllister, “Fast matrix multipies using graphics hardware,” in Proceedings of Supercomputing 2001, Denver, CO, 2001.
[2] S. Mittal, and J. S. Vetter. “A survey of cpu-gpu heterogeneous computing techniques,” ACM Computing Survey, 47(4), pp.1–35, July 2015.
[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” in Proceedings of European Association for Computer Graphics, pp. 21–51, 2005.
[4] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, “A Survey of General-Purpose Computation on Graphics Hardware,” in Computer Graphics Forum, Volume 26, number 1, pp. 80-113, 2007
[5] NVIDIA, https://www.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf
[6] NVIDIA, CUDA C Programming Guide 8.0, 2017.
[7] NVIDIA, CUDA C Best Practices Guide 8.0, 2017.
[8] A. Munshi, The OpenCL Specification, Khenos OpenCL Working Group, version: 1.0, Document Revision:48, 2009.
[9] D. Kirk, and W. W. Hwu, University of Illinois, “Programming Massively Parallel Processes,” Urbana-Champaign, 2010.
[10] NVIDIA, https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960/specifications
[11] NVIDIA, https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/