Evaluation Of The Performance Of GPU Global Memory Coalescing

Dae-Hwan Kim
Department of Computer and Information,
Suwon Science College, 288 Seja-ro, Jeongnam-myun,
Hwaseong-si, Gyeonggi-do, Rep. of Korea
kimdh@ssc.ac.kr

***Abstract*—Nowadays, GPU is widely used for graphics and general-purpose parallel computations. In the GPU software development, memory coalescing is one of the most important optimization techniques, which reduces the number of memory transactions. In this paper, the performance of the global memory coalescing is evaluated in the recent GPU Titan X Pascal processor. The experimental result shows that the coalesced access improves the performance by 2.3 times compared to the uncoalesced one for the benchmark programs while the performance degradation is relatively small by the unaligned access, which ranges from 1.2% to 31.3%.**

Keywords—GPU; performance; global memory; coalescing; alignment

I. INTRODUCTION

GPU (Graphics Processing Unit) is a specialized processor designed to handle graphics operations mainly for the rendering of 2D and 3D graphics. GPU contains a powerful SIMD (Single Instruction Multiple Data) engine which is normally superior to the CPU vector processor, and thus, many researches are made to utilize GPUs for parallel processing applications [2, 4, 8, 9] in addition to graphics computation. In 2002, Harris et al. [1] coined the term GPGPU (General-Purpose computations on GPUs) for using GPUs for general-purpose computation. Nowadays, GPGPU is widely used in many parallel applications such as deep learning, image processing, and video encoding. In AlphaGo for Go game, it is reported that the number of used processors are 1,920 CPUs and 280 GPUs [10].

Major GPU vendors such as NVIDIA and AMD provide GPU as not only the rendering graphics engine but also the multicore computing platform. They supports programming languages such as CUDA (Compute Unified Device Architecture) [5, 6], and OpenCL (Open Computing Language) [3]. In 2006, NVIDIA introduced CUDA, a general purpose computing platform, which is the software layer to the GPU. Now, it is supported by all the GPUs of NVIDIA. The CUDA platform is cooperated with the widely used programming languages such as C, and C++. This enables programmers to use GPU resources easily when developing GPU software. NVIDIA defines CUDA compute capabilities to describe the features supported by the GPU hardware. The first CUDA GPUs had compute capability 1.0 while the compute capability is 6.1 for the recent Titan X Pascal GPU [7].

In GPU program, most data resides in the global memory. Therefore, it is important to maintain a large amount of coherence in memory accesses. This reduces the number of memory transactions, which leads to the performance improving.

In this paper, the performance of global memory coalescing is evaluated on the recent NVIDIA Titan X Pascal processor. The additional discussion is given to the unaligned memory accesses.

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

II. BACKGROUND

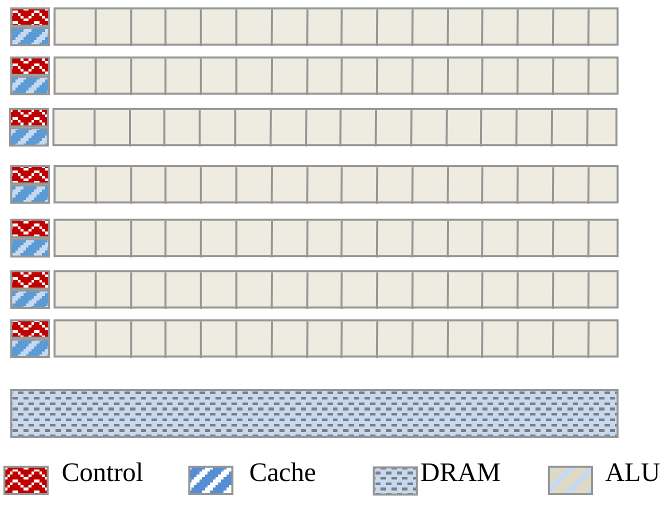

Fig. 1. *GPU architecture*

The GPU processor normally has hundreds or thousands of cores, operating on a common memory like DRAM. Fig. 1 shows the typical GPU architecture. The computing power is excellent with these cores, but, the memory latency is high. Therefore, a lot of arithmetic computation is needed to hide memory latency and achieve good performance.

In Cuda, threads are independent and can be executed in parallel. A warp is the group of 32 threads that are executed simultaneously, which is the smallest executable unit of parallelism. A block consists of warps, and a grid is composed of blocks which are independent.

There are several memory spaces in CUDA. They are register, shared, local, texture, constant, and global memory. Local memory and shared memory are visible to the thread and all the threads in the block, respectively. All the threads in a grid can access the global memory. Two additional spaces, constant and texture, are read-only and accessible by all the threads. The global memory is largest among the memory spaces. In access latency, the register file is fastest, and shared memory is next. Three slowest memory spaces are global, local, and texture. The device memory features are shown in Table I.

TABLE I. DEVICE MEMROY FEATURES [6]

Memory	Location on/off chip	Cached	Access	Scope
Register	On	n/a	R/W	1 thread
Local	Off	*	R/W	1 thread
Shared	On	n/a	R/W	All threads in block
Global	Off	*	R/W	All threads, host
Constant	Off	Yes	R	All threads, host
Texture	Off	Yes	R	All threads, host
* Cached only on devices of compute capability 2.x.				

III. GLOBAL MEMORY COALESCING

Global memory can be accessed by the 32-, 64-, or 128-byte memory transaction. This transaction needs to be aligned to its size. Only the aligned 32-, 64-, or 128-byte segment can be accessed by memory transaction. Thus, if the memory access is not aligned, more segments are transferred than are needed.

Global memory transactions of a warp are coalesced when the requirements are satisfied for the access addresses and the alignments. When a warp of 32 threads executes a global memory instruction, it coalesces the memory accesses into the memory transactions depending on the size of the data accessed by each thread and the distribution of the memory addresses.

Global memory coalescing is one of the most important optimization techniques in CUDA program because the number of memory transactions impacts on the performance. Therefore, it is important to maximize coalescing by performing optimal global memory data layout and the efficient memory addressing.

Global memory accesses are cached in L2 cache in CUDA compute capability 5.0 and above. On the other hand, the L1 caching is controlled by the -dlcm option. They can be cached in both L1 and L2 (-Xptxas -dlcm=ca) or in L2 only (-Xptxas -dlcm=cg). If only L2 cache is used, a memory access is serviced with a 32-byte memory transaction whereas the transaction is 128 bytes for both L1 and L2, which are a cache line size.

Fig. 2 shows the simplest case of coalescing. The i-th thread in a warp accesses i-th word in a cache line where the start address of memory accesses is 128. In this case, a single 128-byte transaction can service all the memory accesses of the warp.

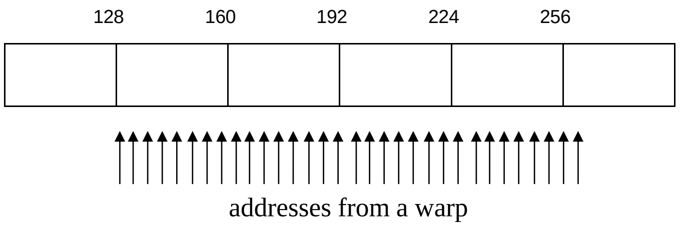

Fig. 2. *Coalesced access [6]*

Consider the case that sequential threads in a warp access sequential locations in a memory, but the first address is not aligned with a cache line. Then, two 128-byte L1 cache lines are requested. Fig. 3 shows the unaligned access pattern.

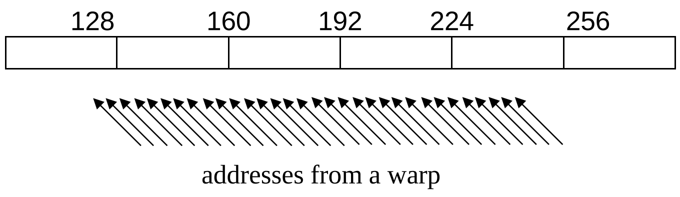

Fig. 3. *Unaligned access [6]*

Fig. 4 shows other examples of global memory accesses. In this example, suppose that compute capabilities is above 2.0, and an each thread accesses a 4-byte word. Consider first the aligned access pattern. If cached, a single 128-byte transaction services all the threads accessing from the address 128. Otherwise, four 32-byte L2 memory transactions are required. Next, consider the misaligned pattern. In this case, if cached, two 128-byte transactions are required. When L1 cache is not used, five 32-byte transactions are necessary.

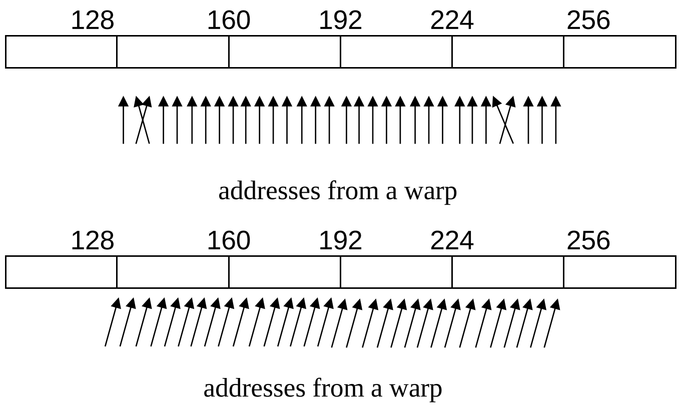

Fig. 4. *Other aligned and unaligned acccess examples [6]*

IV. EVALUATION

Experiments are performed on GTX Titan X Pascal processor [7] released in 2016. Table II shows the specification of the processor. It has 3,584 cores, and base and boost clocks are 1,417MHz, and 1,531MHz, respectively. Memory clock is 10Gpps, and standard config is 12GB. Interface width and band width are 384 bits, and 480 GB/sec, respectively.

TABLE II. TITAN X PASCAL SPECIFICATION

Titan X Pascal Specification		
Engine Spec	Cores	3.584
	Base Clock (MHz)	1,417
	Boost Clock (MHz)	1,531
Memory Spec	Clock	10 Gbps
	Standard config	12GB
	Interface width	384 bits
	Band Width (GB/sec)	480

Table III shows the CUDA kernel programs used in the experiment, which contain the coalescing and uncoalescing global memory access code. Two array elements are accessed that are 4-byte float and 1-byte char types, respectively. Suppose that the dimension is one for each of both the block of threads and the grid of blocks, and the number of threads is 1,024 for each block. Then, a global thread index is computed as blockIdx.x * 1,024 + threadIdx.x. In the coalescing code, the memory address of each thread is given by the global thread index, blockIdx.x * 1024+ threadIdx.x. For the threads in a warp, the block index (blockIdex.x) is the same, and the local thread index (threadIdx.x) is sequential. Now, memory accesses of a warp are coalesced because the addresses of a warp are sequential. On the other hand, in the uncoalescing example, the memory address of each thread is given by threadIdx.x * 1,024 + blockIdx.x. Now, the address difference of adjacent threads of a warp is 1,024, and thus, the accesses can not be coalesced.

TABLE III. EXPERIMENTAL COALESING AND UNCOALESING CODE

Assume that the number of threads is 1,024 in each code	
Type	Code

Coalesced float array access	**float Pixel[1024*1024];**
	__global__ void coal_float_kernel(float *Pixel)
	{
	Pixel[blockIdx.x*1024+ threadIdx.x] ++;
	}

Coalesced char array access	**char Pixel[1024*1024];**
	__global__ void coal_char_kernel(char *Pixel)
	{
	Pixel[blockIdx.x*1024+ threadIdx.x] ++;
	}

Uncoalesced float array access	**float Pixel[1024*1024];**
	__global__ void uncoal_float_kernel (float*Pixel)
	{
	Pixel[threadIdx.x *1024 +blockIdx.x]++ ;
	}

Uncoalesced char array access	**char Pixel[1024*1024];**
	__global__ void uncoal_char_kernel (char *Pixel)
	{
	Pixel[threadIdx.x *1024 + blockIdx.x]++ ;
	}

Fig. 5 shows the performance of the uncoalescing denoted by UNCOAL compared to the coalescing global memory access denoted by COAL. For the float type memory access, the execution speed of the UNCOAL code is just 47.5% of the coalescing accesses. For the char type, the performance degrades as the 38.5% of the coalesced one.

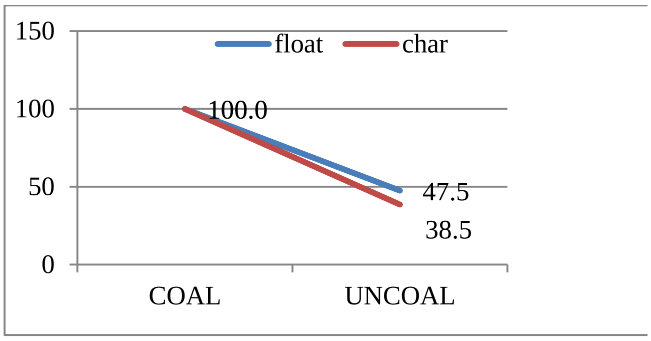

Fig. 5. Performance of uncoalescing memory access

Table IV shows the experimented CUDA kernel programs which have unaligned global memory access code. There exist 16 unaligned offsets from 1 to 16 represented by unAlign1 to unAlign16, respectively. There also exist 16 programs for the character type Pixel array, which are quite similar to the float type programs.

TABLE IV. EXPERIMENTAL UNALIGN ACCESS CODE

Assume that the number of threads is 1,024 in each code
Unaligned array access

float Pixel[1024*1024];
__global__ void unAlign1 (float *Pixel)
{
Pixel[blockIdx.x*1024+ threadIdx.x+1]++ ;
}

__global__ void unAlign2 (float *Pixel)
{
Pixel[blockIdx.x*1024+ threadIdx.x+2]++ ;
}

__global__ void unAlign3 (float *Pixel)
{
Pixel[blockIdx.x*1024+ threadIdx.x+3]++ ;
}

.

__global__ void unAlign16 (float *Pixel)
{
Pixel[blockIdx.x*1024+ threadIdx.x+16]++ ;
}

Fig. 6 shows the performance of the unaligned access pattern denoted by Un1 to Un16, which represent unaligned offsets from 1 to 16 compared to the aligned coalescing access denoted by COAL, respectively. For the float type memory access, the performance of Un1, to Un16 are 88.8%, 88.5%, 86.4%, 84.4%, 95.1%, 88.0%, 95.8%, 79.0%, 87.2%, 81.8%, 94.5%, 86.7%, 96.4%, 94.8%, 86.4%, and 75.1%, respectively. The values range from 79.0% to 96.4%. For the char type, they are 98.8%, 88.8%, 74.9%, 90.8%, 76.7%, 79.0%, 68.7%, 86.3%, 85.4%, 88.8%, 97.5%, 86.3%, 71.2%, 79.4%, 89.8%, and 86.8%, respectively. The range is from 68.7% to 98.8%. The performance degrades from 1.2% to 31.3%. The performance loss by the unaligned access is relatively small compared to the uncoalesced access.

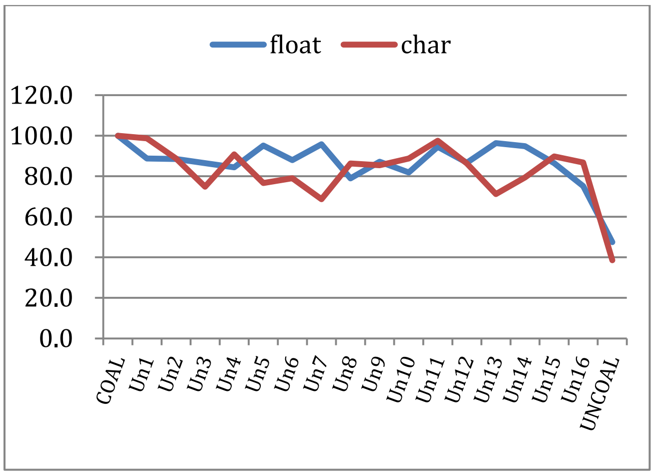

Fig. 6. Performance of unaligned memory access

V. CONCLUSIONS

In this paper, the performance of the global memory coalescing is evaluated on the GPU Titan X Pascal processor. The Uncoalescing access results in a significant performance loss by the average of 57.0%. For the unaligned accesses, the performance degradation is relatively small compared to the uncoalescing. The evaluation on other GPU processors remains as a future work.

REFERENCES

[1] M. Harris. GPGPU.org. http://www.gpgpu.org, 2002.

[2] E. S. Larsen, D. McAllister, “Fast matrix multiplies using graphics hardware,” in Proceedings of Supercomputing 2001, Denver, CO, 2001.

[3] A. Munshi (Ed.), The OpenCL Specification, Khronos OpenCL Working Group, version: 1.0, Document Revision:48, 2009.

[4] S. Mittal and J. S. Vetter. “A survey of cpu-gpu heterogeneous computing techniques. ACM Comput. Surv., 47(4), pp.1–35, July 2015.

[5] NVIDIA, CUDA C Prgoramming Guide 8.0, 2017.

[6] NVIDIA, CUDA C Best Practices Guide 8.0, 2017.

[7] NVIDIA, https://www.nvidia.com/en-us/geforce/ products/10series/titan-x-pascal/.

[8] 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 Proc. Eur. Assoc. Comput. Graph., pp. 21–51, 2005.

[9] 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

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. "Mastering the game of go with deep neural networks and tree search," Nature, 529(7587), pp. 484–489, 2016. .