GPU APPLICATION IN CUDA MEMORY

Khoirudin and Jiang Shun-Liang

Department of Computer Applied Technology, Nanchang University, China

ABSTRACT

Nowadays modern computer GPU (Graphic Processing Unit) became widely used to improve the performance of a computer, which is basically for the GPU graphics calculations, are now used not only for the purposes of calculating the graphics but also for other application. In addition, Graphics Processing Unit (GPU) has high computation and low price. This device can be treat as an array of SIMD processor using CUDA software. This paper talks about GPU application, CUDA memory and efficient CUDA memory using Reduction kernel. High-performance GPU application requires reuse of data inside the streaming multiprocessor (SM). The reason is that onboard global memory is simply not fast enough to meet the needs of all the streaming multiprocessor on the GPU. In addition, CUDA exposes the memory space within the SM and provides configurable caches to give the developer the greatest opportunity of data reuse.

KEYWORD

GPU, CUDA, NVIDIA, CUDA Memory

1. INTRODUCTION

GPU is the special processor that is in charge to relieve the main processor in a computer graphic of a computation. GPUs were originally created for a high-performance workstation and certainly the price is also very expensive [1,2]. In early 1990, 3D games with rendering processor were appearing, since then 3D accelerator Hardware was made [2,9]. API OpenGL was basically from a graphic application of professional workstation and adopted to make a graphic 3D game programming, the same as the emergence of the DirectX and Direct3D. In addition, GPU was becoming more affordable and more powerful than ever and the development of GPU was faster than the CPU development, this is because the encouragement of the Business gaming on PC [2].

GPU continued evolving; GPU GeForce from NVIDIA appeared. In 2001, This GPU was the pioneer of the Shader programmable GPU. Shader was the data processing units in GPU. Generally, each GPU has more than one shader. During this period, the GPU can be programmed by aiming to increase the GPU works to process the data graphics in parallel [1,2]. On a modern GPU, shader number or often called the Stream Processor (for stream input and output/Stream), has reached the hundreds or even thousands. GPU calculation abilities can reach Terra FLOPS, the hundreds of times faster than the CPU. These capabilities were ready to be used to perform other calculations besides graphics calculations [5,6]. GPU Computing is on the growth in popularity and that makes the future very familiar [7,11]. The comparison of the GPU and the CPU is show in Table 1.

DOI:10.5121/acij.2015.6201

Table 1. Comparison between CPU and GPU

CPU	GPU
Parallelism through time multiplexing	Parallelism through space multiplexing
Emphasis on low memory latency	Emphasis on high memory throughput
Allows wide rangeof control flows + control flow optimization	Very control flow restricted
Optimized for low latency access to caches data set	Optimized for data parallel, throughput computation
Very high clock speed	Mid-tempo clock speed
Peak computation capability low	Higher peak computation capability
Off-chip bandwidth lower	Higher off-chip bandwidth
Handle sequential code well	Requires massively parallel computing
CPU are great for task parallelism	GPU are great for data parallelism

2. GPU

GPU refer to Graphics Processing Unit and is a single chip processor used for 3D application. GPU functionality has traditionally beenvery limited. In fact, since long time ago the GPU just used to accelerate certain parts of the graphics pipeline [4,8]. The GPU is limited to independent vertices and fragments on the processing capacity. However, this processing can be improved in parallel using the multiple cores which is away now to available to the GPU. This is more effective when the programmer wants to process a lot of vertices or fragments in the similar way [9,14].

It creates lighting effect and transforms objects every time when a 3D scene is redrawn. These are mathematically intensive tasks, and put quite a strain on the CPU. Free this burden from the CPU give some spaces for cycles that can be used for another task[1,6]. GPU produce not only high computational power but also with low costs. More transistors can be devoted to data computation rather than data caching and flow control as in the case of CPU. With multiple cores that control by very high memory bandwidth, nowadays GPU serve with incredible resources for both non-graphics processing and graphics processing at the same time[1,6].

3. CUDA

At November 2006, NVDIA introduce CUDA, a general purpose computing platform and programming model that leverages the parallel compute engine in NVDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU[3,5].Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processing Unit or GPU has evolved into a highly parallel, multithread, many core processors with tremendous computational horsepower and very high memory bandwidth [3,4], as illustration on figure 1.

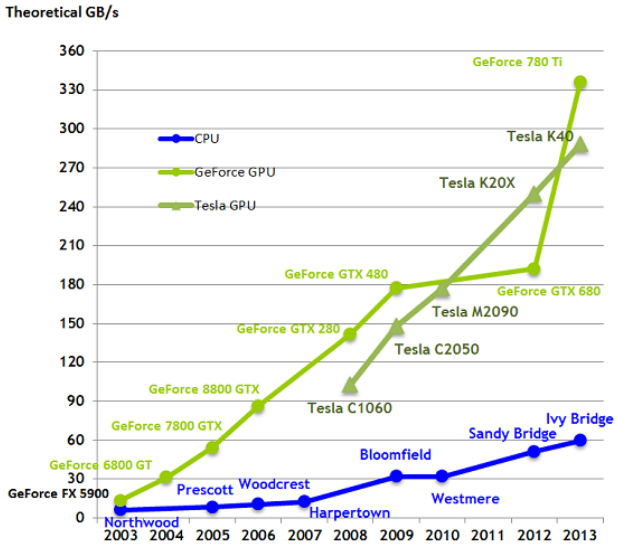


Figure 1. Memory Bandwidth of GPU and CPU

3.1 Memory Hierarchy

The CUDA threads can access trough the data from multiple memory spaces during their execution process. Every thread has its own private local memory. Every thread block has shared memory visible to every thread of the block and with the similar lifetime as the block. Every thread has the access to the same global memory [3]. There are another two additional read-only memory spaces that can be access by every of the threads; the texture memory and constant memory spaces. Every memory spaces include the global, constant and texture memory spaces are optimized for non-similar memory function [3, 5]. Texture memory in other ways offers the other different addressing modes, the same with data filtering, for some specific data formats. The memory spaces that consist of the global memory, constant, and texture memory space are persistent across kernel launching by using the same application [3, 5].

3.2 Architecture

GPU is a massively parallel architecture; many problems can be efficiently solve using GPU computing. GPU have large among of arithmetic capability. They increase the amount of programmability in the pipeline[7,15,16].

Fig. 2. Show architecture of a typical CUDA-capable GPU. CUDA can be likely the array of streaming processors that is have capability of high level of threading. In Fig. 2, two SMs form a building block; moreover, the number of the Streaming Multiprocessing in a building block may be varying from one another generation of CUDA GPUs to another generation. In complexion, each Streaming Multiprocessing has a number of streaming processors (SPs) that share instruction cache and control logic. Every GPU currently already comes with up to 4 gigabytes of graphics double data rate (GDDR) DRAM, referring as the global memory. These RAMs of the GPU are non-similar with the CPU that they are functionally as frame buffer memories for rendering graphics. For graphics applications, they keep video images, and texture information for three-dimensional (3D) rendering, but for computing their function as very high bandwidth, off-chip memory with something that more latency than typical system memory. For massively parallel applications, the higher bandwidth made for the longer latency [7,15].

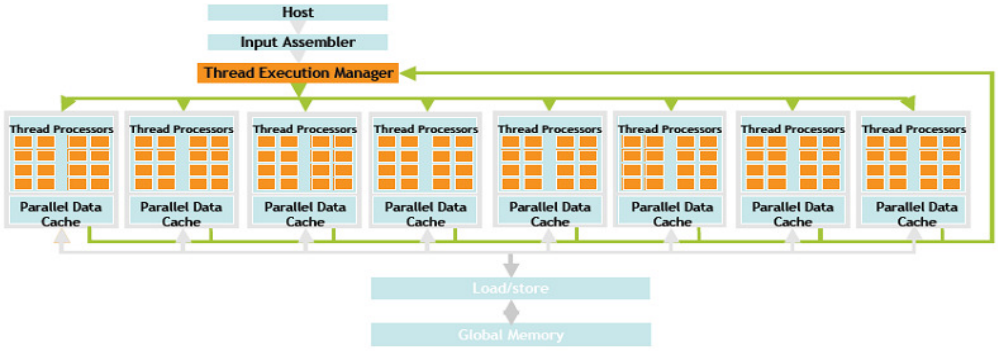


Figure 2. GPU architecture

4. CUDA Memory

CUDA supports several types of memory that can be used by programmers to achieve high CGMA (Compute to Global Memory Access) ratios and thus high execution speeds in their kernels [4, 5]. Figure 3 shows this CUDA device memory. At the part of the figure 3, we can see the constant memory and global memory. These many types of memory can be read (R) and written (W) by the host, that using API functions. Register and shared memory are GPU on-chip memories. This Variable that reside in these memory types that access able at very high speed in high parallel manner [3].

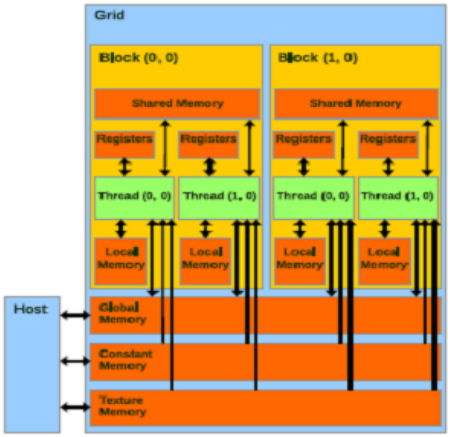


Figure 3. CUDA memory model

In CUDA memory structure there are some memory in which each memory has duties and functions of each, Here are the types and characteristics of CUDA memory [3,4,5].

Table 2. Characteristics and Type of CUDA Memory

Memory types	position	Caches	Accessibility	Area
Register	On chip	No	W/R	One thread
Local	On chip	Yes	W/R	One thread
Shared	On chip	N/A	W/R	All threads in block
Global	Off chip	Yes	W/R	All host+ threads
Constant	Off-chip	Yes	R	All host+ threads
Texture	Off-chip	Yes	W/R	All host+ threads

4.1 Register Memory

The fastest memory on the GPU is the Register memory. Because they are only memory on the GPU with enough bandwidth and a low latency to deliver peak performance [4,5].

Each GF100 SM support 32 K 32 bit registers. The biggest number of registers that using by a CUDA kernel is 63, occur to the limited number of bits available for indexing in the register memory store. The number of ready able registers varies on a Fermi SM:

a. If the SM running 1,536 thread, only registers can be used
b. The number of available register degrades gracefully from 63 to 21 as the workload (and hence resource requirement) increases by number of thread.

4.2 Local Memory

The accesses of the Local memory occur for only some automatic variable. An automatic variable is launch in the device code with none of any of the **__shared__, __device__, or __constant__** qualifier. Generally, an automatic variable reside in a register except for the following [4, 5]:

a. Array that the compiler cannot be determine to be index with constant quantities.
b. Large structure or array that would consume too much register space
c. Every variable of the compiler decides to spill to local memory when a kernel using much more registers memory than are available on the SMs.

4.3 Shared Memory

Shared memory also as know as to smem, can be not only 16 KB but also 48 KB per SMs arranged in 32 banks that are 32 bits wide. This is different with early NVDIA documentation; shared memory is not fast enough like the register memory. Shared memory can be distribute in three different ways [4, 5];

a. In static number within the kernel or globally within the file that shows in the declaration in the sample code below, "A static Shared Memory declaration":

```
__shared__ int_data[256;]
```

b. Dynamically within the kernel by the driver API function calling.
c. Dynamically via the execution configuration.

Only shared memory of the single block can be allocate by using the execution configuration. Used more than one dynamically allocated shared memory variable on the kernel need the manually generating offsets for every variable.

4.4 Constant Memory

For the compute 1.x device, using constant memory is an excellent way to store and broadcast read-only data to all the thread on the GPU. The constant cache is limited up to 64 KB. In addition, can be broadcast 32-bits per two clocks per warp per multiprocessor and suppose to be use when every thread in a warp read the same address. Moreover, the accesses will come serialize on compute 1.x device [4,5]. Higher devices and compute 2.0 allow developers to access global memory using the efficiency of constant memory when the compiler can use and recognize the LDU instruction. Specifically, the data must:

a. Reside in the global memory
b. Be read-only in the kernel (programmer can enforce using the **const** keyword).
c. Not depend on the thread ID.

4.5 Texture Memory

Texture are bound to global memory and can provide both cache and some limited 9-bits processing capabilities. How the global memory, which the texture binds to is allocateto dictate some of the capabilities the texture can provide. Due to this reason, it is important to distinguish between three kinds of memory types that can be bound to the texture memory (look at Table 3). For CUDA programmer, the most salient points about using texture memory [4,5]:

a. Texture memory is generally used in visualization
b. The cache improved for 2D spatial locality.
c. It consist only 8 KB of cache per SMs.

Table 3 The way Memory Was Create Defines the Texture memory Capability

Type	How build	Capability	Texture Update
Linear	cudaMalloc()	Acts as a linear cache	If the incoherence is safe, Free to write to the global memory from threads.
CUDA arrays	cudaMallocArray(), cudaMalloc3D()	- Cache improved for spatial locality - Wrapping, Interpolation, and clamping	It is not allows Writing to array from kernel.
2D pitch Linear	cudaMallocPitch()	- Cache improved for spatial locality - Wrapping, Interpolation, and clamping	If the incoherence is safe, Free to write to the global memory from threads.

4.6 Global memory

Understanding how to efficiently used global memory is an essential to becoming an adept CUDA programmer. Focusing on data reuse within the SM and cache avoids memory bandwidth limitations. There are three most important rules of high-performance GPU programming [4,5].

a. Take the data on the GPU and then keep it.
b. Give GPU enough work to do.
c. Concerning the data reuse within the GPU to avoid memory-limited bandwidth.

For some part, it is impossible to keep off global memory, where in some case it is important to understand how to use the global memory efficiently. Especially the Fermi architecture made some important changing that CUDA programmers should think about and use the global memory efficiently.

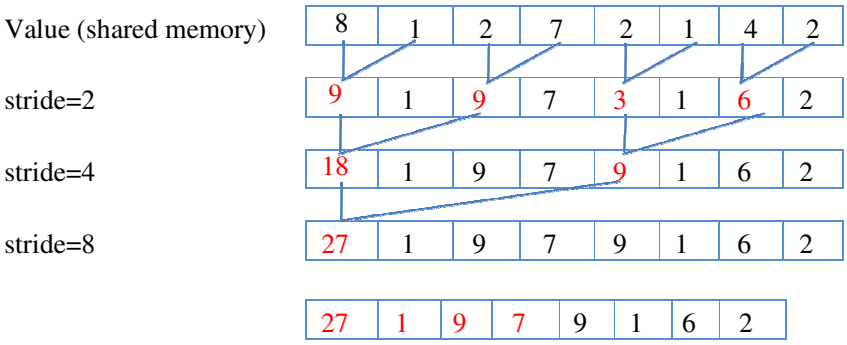
5. Efficient CUDA memory Using Reduction Kernel

The importance of efficiently using memory in CUDA cannot be overstated. There are three orders of magnitude un similar speed between the fastest on-chip register memory and mapped host memory that should be traverse the PCIe bus, CUDA developers must have understanding about the most efficient way to memory used[13,17].

Reduction operation perform common task such as finding the minimum, maximum, or sum of a vector. The thrust API provides a simple interface that hides all the complexity of reduction, making both flexible and easy to use. Thrush uses a reduction algorithm designed by Mark Harris [8].

a. Reduction algorithm 1

Single block parallel reduction



Interleaved addressing with divergent branching

```
for (int stride=1;stride<blockDim.x;stride*=2) {
__syncthreads();
if (tid%(2*stride)==0)sm[tid]=sm[tid+stride]; }
if (tid==0) d[blockIdx.x]=sm[0];
 }
```

b. Reduction algorithm 2

Replace the divergent branch with a non-divergent one.

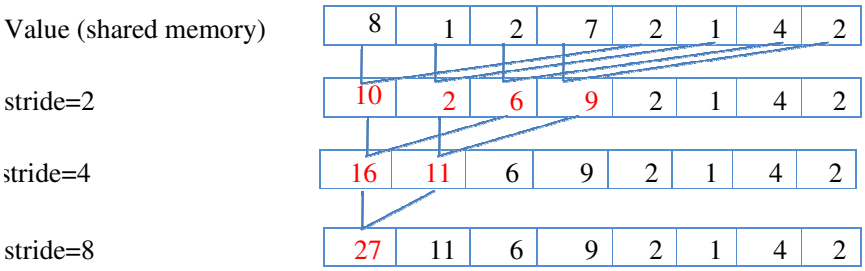
```
if (tid%(2*stride)==0)sm[tid]+=sm[tid+stride];
```

↓

```
Int index=2*stride*tid;
If (index<blockDim.x)
        Sm[index]+=sm[index+stride];
```

c. Reduction algorithm 3

Single block parallel reduction



Replace the stride loop with a reversed one.

```
Int index=2*stride*tid;
If (index<blockDim.x)
        Sm[index]=sm[index+stride];
```

↓

```
Int index=2*stride*tid;
If (index<blockDim.x)
        Sm[index]+=sm[index+stride];
```

Table 4. Comparison reduction algorithm

Algorithm	Speedup	Time	Bandwidth
Algorithm 1	1	7.800 ms	4.18 GB/sec
Algorithm 2	1.96	3.975 ms	8.41 GB/sec
Algorithm 3	2.94	2.650 ms	12.43 GB/sec

6. EXAMPLE OF EXPERIMENT

In the example, research by Huang Jing-Jing entitle "Research and application of simulation for particle system based on GPU" [12]. In this research, he use shared memory to efficiencies the system and compare different strategies to choose the optimal strategy.

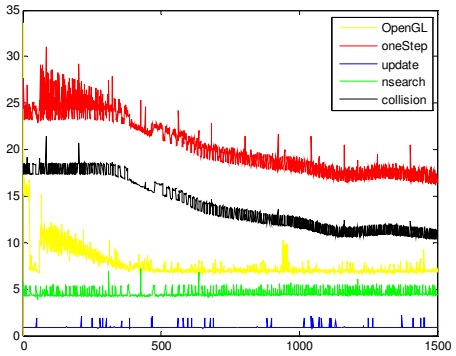


Figure 4a. Simulation Computer A

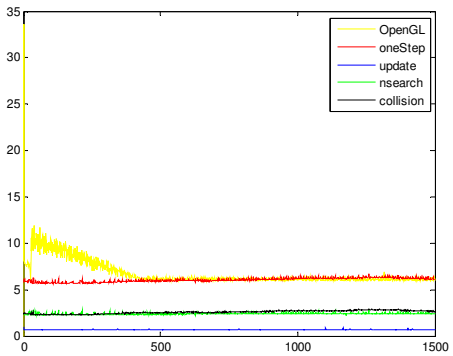


Figure 4b. Simulation Computer B

On simulation computer A and B, respectively, the optimized operation system 1500 steps, the kernel function is time-consuming. Observation two pictures in the curve, the curve of the initial value is very high, and then for a higher value in steady decline, until the basic stability. When program began to run, you need to allocate space and some initialization, so the curve of the initial value is higher.

In this system using the Shared memory of the kernel function is hashsort() and findCellParticle(). Which hashsort() sorting process using CUDA cstd in the library of cudppSort() function, although have no function on the source code, but through CUDA profiler, still can analyze the function of the basic situation, first of all, this function is run on the GPU, and the function is executed, each block using the Shared memory with 4096 bit. In addition, when the function findCellParticle() is executed, each block using the Shared memory size of 1028 bit.

Table 5. Before and after using Shared memory

	OpenGL	*oneStep*	*update*	*Nsearch*	*collision*
A：shared memory	8.1610	22.4092	0.8873	4.4877	13.9956
A：No shared memory	8.1730	22.5342	0.8957	4.4553	13.9863
B：shared memory	6.1472	6.7001	0.6477	2.3902	2.5957
B：No shared memory	6.2896	6.8708	0.7013	2.4561	2.5835

On the computer A and B respectively improved Nsearch kernel function execution speed 0.0324 m/s and 0.1305 m/s. After the test, it got the proven that using the Shared memory of Nsearch execution speed did not improve at all.

7. CONCLUSION

CUDA makes various hardware spaces available to the programmer. It is important that the CUDA programmer use the available memory space with the three orders of magnitude un similar in bandwidth between the kinds of CUDA memory types. Malfunction of usage of memory can result in low performance. CUDA defines shared memory, register, and constant memory that can be access at maximum speed and in parallel manner than global memory. Using memory effectively seems like require redesign of the algorithms, a popular technical strategy to maximize the locality of data access and enable effective use of shared memory.

Moreover, it is the most essential think for CUDA programmers to be aware of the limited sizes of this type of CUDA memory, their capacities are implementing dependently. Once their capacities are exceed, they become limited factors for the number of thread that can be simultaneously executing in each SM. The application must exhibit locality in data access in order to make efficient use of high-speed memory in the system operation.

REFERENCES

[1]. vanden Boer, Dirk. "General Purpose Computing on GPU's." (2005).
[2]. BomaAnantasyaadhi, “Implementation General Purpose GPU to Process Singular Value Decompotition on Simple-O” Indonesia University.2010.
[3]. “CUDA C programming guide version 6.5”, NVDIA Corporation, August 2014
[4]. Jason Sanders, Edward Kandrot, “Cuda by Example” Nvdia
[5]. Rob Farber, “Cuda Application Design And Development”, NVDIA
[6]. Harris, Mark. "Gpgpu: General-purpose computation on gpus." SIGGRAPH 2005 GPGPU COURSE (2005).
[7]. Ghorpade, Jayshree, et al. "Gpgpu processing in cuda architecture." arXiv preprint arXiv:1202.4347 (2012).
[8]. Harris, Mark. "Optimizing parallel reduction in CUDA." NVIDIA Developer Technology 2.4 (2007).
[9]. CalleLedjfors , “High Level GPU Programming”, Department of Computer Science Lund University. 2008.
[10]. Ueng, Sain-Zee, et al. "CUDA-lite: Reducing GPU programming complexity."Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg, 2008. 1-15.
[11]. Thomas, Winnie, and Rohin D. Daruwala. "Performance comparison of CPU and GPU on a discrete heterogeneous architecture." Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014 International Conference on. IEEE, 2014.
[12]. Huang Jiangjiang, “Reserch and application of simulation for particle system based on GPU”.Nanchang University. 2010

[13]. Yeh, Tsung Tai, et al. "Efficient parallel algorithm for nonlinear dimensionality reduction on gpu." Granular Computing (GrC), 2010 IEEE International Conference on. IEEE, 2010.
[14]. Garland, Michael, et al. "Parallel Computing in CUDA." IEEE micro 28.4 (2008): 13-27.
[15]. Kirk, David. "NVIDIA CUDA software and GPU parallel computing architecture."ISMM. Vol. 7. 2007.
[16]. Zhao, Xiang Jun, MeiZhen Yu, and Yong Beom Cho. "GPU_CPU based parallel architecture for reduction in power consumption." Global High Tech Congress on Electronics (GHTCE), 2012 IEEE. IEEE, 2012.
[17]. Roger, David, Ulf Assarsson, and Nicolas Holzschuch. "Efficient stream reduction on the GPU." Workshop on General Purpose Processing on Graphics Processing Units. 2007.

Authors

Khoirudin

(Master Degree of Computer applied and technology)
Nanchang University, China
Area interest is in Software Engineering, Database and Artificial Intelligent

Jiang Shun-Liang

Professor, Nanchang University, China
Area of interest is in Computer Modeling and Simulation, Numerical Computation, Algorithm Design and Analysis, Artificial Intelligence