CUDA'S Mapped Memory to Get High Performance using GPU

Tofik R. Kacchi
Department of Computer Science and Engineering, Government College of Engineering, Amravati, India

Pushpanjali Chauragade
Assistant Professor, Department of Computer Science and Engineering, Government College of Engineering, Amravati, India

ABSTRACT
The API interfaces provided by CUDA help programmers to get high performance CUDA applications in GPU, but they cannot support most I/O operations in device codes. The characteristics of CUDA's mapped memory are used here to create a dynamic polling service model in the host which can satisfy most I/O functions such as read/write file and "printf".The technique to implement these I/O functions has some influence on the performance of the original applications. These functions quickly respond to the users' I/O requirements with the "printf" performance better than CUDA's. An easy and effective real-time method is given for users to debug their programs using the I/O functions. These functions improve productivity of converting legacy C/C++ codes to CUDA and broaden CUDA's functions.

Keywords: CUDA's Introduction, Architecture, input-output functions, mapping of memory.

1. INTRODUCTION
Parallel applications for GPU can be easily developed using CUDA with the various API interfaces provided by CUDA providing the powerful tools to manage the GPU and the high memory bandwidth. Besides traditional image processing, CUDA has also enabled analyses at oil reconnaissance, astronomical timing, hydrodynamics, molecular kinetics, biology, audio frequency decoding, and video frequency decoding. The number of applications using the GPU have been accelerated many fold, even a hundred times more than in a CPU[1] with key run-time libraries such as CUBLAS, CUFFT, and CUDPP.

However, the API interfaces and libraries in CUDA are self-contained, so most of the I/O functions are not supported in the device (referring to the GPU and its memory) codes, so developing and debugging application is difficult. For example, developers often use "printf" in debugging to access application information. Although CUDA supports "printf" in device codes in version 3.1[2] and higher, users cannot see the results of "printf" in real-time until the kernel function is finished, which is not satisfactory. If the kernel function hangs, the users will not get any of the "printf" information. Therefore, the I/O functions are not convenient for programmers. Especially, the system will spend extra energy dealing with legacy codes that contain I/O operations. For example, the only method to complete file read/write operations in the device is to add some memory copies between the host (referring to the CPU and the system memory) and the device instead of directly reading/writing files.

Most research on CUDA has been based on existing programming models and compilers that are legacy codes, not CUDA programs. Generally, these studies have changed the programs written in other programming languages to CUDA or to executable target codes on the GPU directly. For example, Lee et al.[3] designed a source-to-source compiler which can change OpenMP program to CUDA and used different program optimization methods in OpenMP and CUDA. HMPP[4] used compiling directives to translate C or FORTRAN programs to CUDA or OpenCL[5] program.PyCUDA[6] allows programmers to directly use CUDA's parallel compute APIs in Python[7] codes. CuPP[7] integrates CUDA programs into an existing C++ framework. All these try to make the GPU programming easier and more effective. These tools can quickly transplant legacy codes to the GPU. Therefore, CUDA needs to also support I/O functions and the other common functions in these codes so as to not increase the designers' workloads and to reduce the compiler's applicability.

This study uses the characteristics of the mapped memory to support I/O functions, such as reading/writing files and "printf" in device codes. These efficient I/O functions enable application developers to conveniently get data at run time to further enable transplanting legacy codes. At the same time, the I/O agent introduced here can be extended to memory operations, message sending and receiving, and socket operations, even for entire web server. This paper introduces CUDA and GPU, and the challenges faced when implementing efficient I/O functions in GPUs with descriptions of the implementation ideas and techniques. Examples are given using CUDA SDK 4.0

2. ARCHITECTURE OF CUDA
Figure 2.1 shows an overview of the CUDA memory model. The memory hierarchy of a CUDA device has several parts including the global memory, constant memory, shared memory, texture memory, and local memory. Each thread has a private local memory and each thread block has shared memory visible to all threads of the block with the same lifetime as the block. All threads can access the same global memory[2]. The constant memory and texture memory are read-only memory spaces accessible by all threads.

Each block contain following:

- Set of local registers per thread.
- Parallel data cache or shared memory that is shared by all the threads.
- Read-only constant cache that is shared by all the threads and speeds up reads from constant memory space.

- Read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space.

Local memory is in scope of each thread. It is allocated by compiler from global memory but logically treated as independent unit.Shared memory :

- Accessible by any threads within a block where it was created.
- Lifetime of a block.

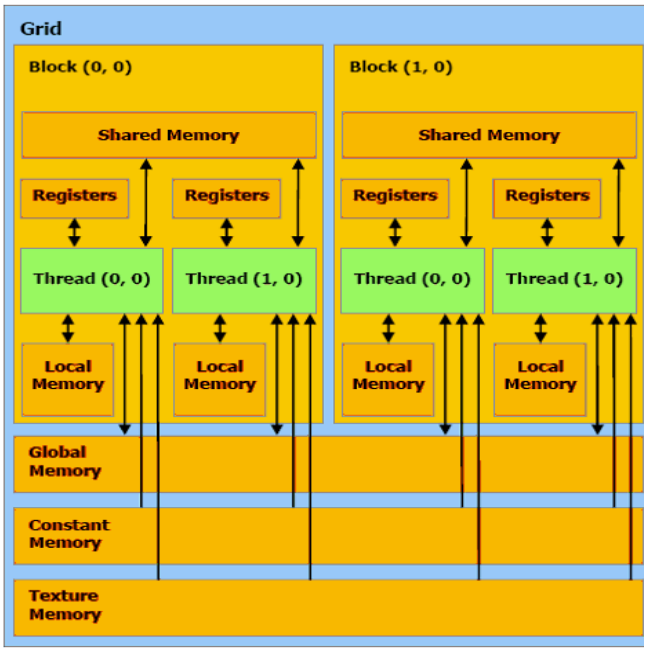

Figure 2.1 CUDA Memory Model

The mapped memory is used here for efficient I/O on the GPU. However, the system must ensure the order and coherence of operations to the same mapped memory spaces because of the features of mapped memory, which is a big challenge.

3. INPUT-OUTPUT FUNCTIONS

The API interfaces and libraries in CUDA are self-contained, so most of the I/O functions are not supported in the device (referring to the GPU and its memory) codes, so developing and debugging application is difficult.This section shows the basic framework of I/O functions, which is as follows :

3.1 Overview

Figure 3.1 shows the basic framework of implementation of the Input-Output functions, which includes a preprocessor module, a support library of I/O functions, and a host agent module. The preprocessor inputs are GPU codes containing I/O functions which require processing and code generation work to provide the arguments for the device functions. The I/O function device library provides the required APIs. These APIs record the I/O function data and parameters, send agent requests, complete interactions with the host, and finally get the return values. The host calls pthread create to create a process to do the agent job. The process denoted by Agent tid scans the data structures stored in mapped memory to record information for the agent requests. When the host detects a request, the APIs in the host are called to deal with the agent's request.

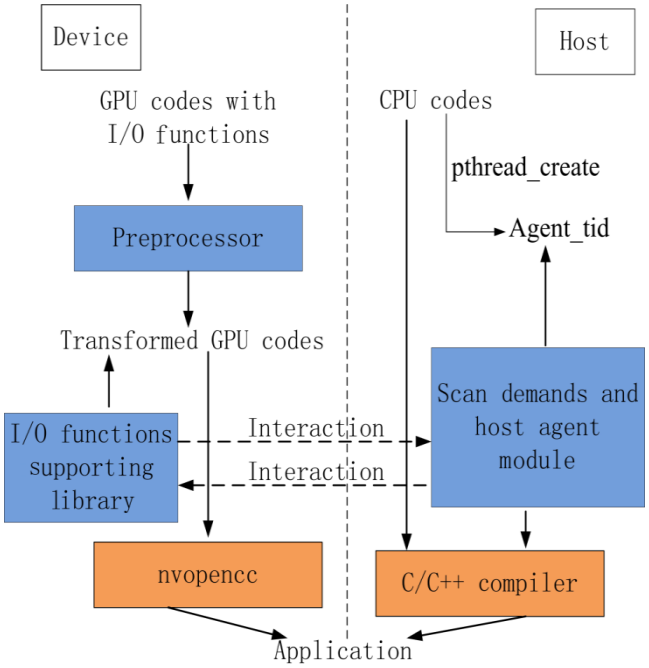

Figure 3.1 Basic Frame-work for I/O Functions

4. MAPPING OF MEMORY

There are several built in variables that are available to kernel call:

- blockIdx - block index within grid.
- threadIdx - thread index within block.
- blockDim - number of threads in a block.

Equation Used :

idx =blockIdx.x * blockDim.x + threadIdx.x;

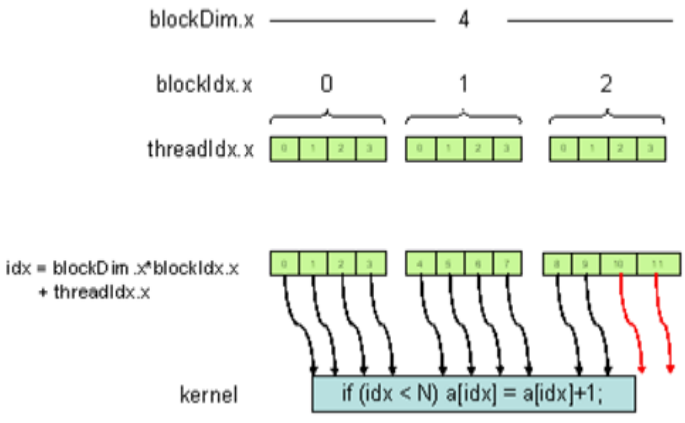

Figure 4.1 Diagram of block breakdown and thread assignment for our array

5. CONCLUSION

Thus, we have inferred that CUDA is nothing but a few extensionto C programming with application programming interfaces which is supporting heterogeneous data. As GPUs contain much larger number of dedicated ALUs than CPUs and GPUs is providing extensive support of Stream Processing paradigm i.e SIMD processing, so we can employ GPU for complex operations at the same time we can parallel employ CPU for performing execution of other operations. CUDA is better than C programming or C++ one, because it takes the advantages of both procedure oriented and object oriented programming by using the features of both C and C++. Hence, CUDA is needed for high performance parallel applications.In short, it is summarized as follows :

- Essentially, a few extension to C + API supporting heterogeneous data
- Parallel CPU + GPU execution
- Uses features of both C & C++
- Needed for high performance parallel applications

6. REFERENCES

[1] IEEE Paper on "CUDA's Mapped Memory To Support I/O functions on GPU" , presented at TSINGHUA SCIENCE AND TECHNOLOGY, by Wei Wu, FengbinQi,WangQuan He and Shanshan Wang, Volume 18,Number 6,December 2013.

[2] NVIDIA Corporation, CUDA Toolkit 3.1 Downloads, https://developer.nvidia.com/cuda-toolkit-31-downloads, 2010.

[3] S. Lee, S. Min, and R. Eigenmann, OpenMP to GPGPU: A compiler framework for automatic translation and optimization, presented at the 14th ACM SIGPLANSymposium on Principles and Practice of Parallel Programming, Raleigh, NC, USA, 2009.

[4] R. Dolbeau, S. Bihan, and F. Bodin, HMPP?: A hybrid multi-core parallelprogramming environment, presented at the 1st Workshop on General Purpose Processing on Graphics Processing Units, Boston, USA, 2007.

[5] Khronos Group, The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl, 2011.

[6] J. Breitbart, Cupp-A framework for easy CUDA integration, in Proc. the 2009 IEEEInternational Symposium on Parallel&Distributed Processing, Washington, DC, USA, 2009, pp. 1-8.

[7] S. Zhang, Y. Zhu, K. Zhao, and Y. Zhang, GPU High Performance Computing withCUDA, (in Chinese). Beijing, China: China WaterPower Press, 2009.

[8] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Burlington, MA, USA: Morgan Kaufmann Publishers, 2010.

[9] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. Boston, MA, USA: Addison-Wesley,2010.

[10] G. Diamos, A. Kerr, and S. Yalamanchili, Ocelot: A dynamic optimizationframework for bulk-synchronousapplications in heterogeneous systems, presented atthe19th International Conference on Parallel Architectures andCompilation Techniques, Vienna, Austria, 2010.

7. AUTHOR BIOGRAPHY

Tofik R. Kacchi has received his B.Tech degree in Computer Science and Engineering from Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded, India in 2014. At present, he is persuing Master of Technology in department of Computer Science and Engineering at Government College of Engineering, Amravati, India. His research interest includes High Performance Computing, Parallel Computing, Artificial Neural Network.

Pushpanjali M. Chouragade has received her Diploma in Computer Science and Engineering from Government Polytechnic, Amravati, India, in 2007, the B.Tech. degree in Computer Science and Engineering from Government College of Engineering, Amravati, India in 2010 and her M.Tech. in Computer Science and Engineering from Government College of Engineering, Amravati, India, in 2013. She was a Lecturer with Department of Computer Science & Engineering, in Government College of Engineering Amravati, in 2010-11. Her research interest includes Data Mining, Web Mining, Image Processing. At present, she is an Assistant professor with department of Computer Science and Engineering at Government College of Engineering, Amravati, India, since 2011.