ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

CUDA: High Parallel Computing Performance

Prasad C. Shinde[1], Tejaswi Kadam[2], Rushikesh V. Mahadik[3], Professor Kumkum Bala[4]

Student, Dept. of Computer Engineering, BVCOEL, Pune University, India[1]

Student, Dept. of Computer Engineering, BVCOEL, Pune University, India[2]

Student, Dept. of Computer Engineering, BVCOEL, Pune University, India[3]

Professor, Dept. of Computer Engineering, BVCOEL, Pune University, India[4]

ABSTRACT: Graphics process units (GPUs) originally designed for computer video cards have emerged because the most powerful chip in a superior digital computer. Not like multicore CPU architectures, that presently ship with two or four cores, GPU architectures are "multicore" with many multiple cores capable of running thousands of threads in parallel. NVIDIA's CUDA could be a co-evolved hardware-software architecture that allows superior computing developers to harness the tremendous power of computation and memory bandwidth of GPU in a very familiar programming surroundings – the C programming language. We describe the CUDA programming model and inspire its use.

KEYWORDS: CUDA, Parallel programming, GPGPU, high performance, NVIDIA.

I. INTRODUCTION

Formally introduced in 2006, CUDA is steady winning in scientific and engineering fields. Many years gone, pioneering programmers discovered that GPUs might be harnessed for tasks aside from graphics. However, their improvised programming model was clumsy, and also the programmable component shaders on the chips weren't the engines for general purpose computing. NVIDIA's has seized upon this chance to form a stronger programming model and to boost the shaders. In fact, for the superior computing.

GPU computing, or the utilization of graphics processors for general-purpose computing, began in earnest many years ago. Work so far as enclosed a lot of promising analysis in the medical specialty domain. However, this analysis at first concerned programming the GPU via a graphics language, that restricted its flexibility and was arcane for non-graphics consultants. NVIDIA's CUDA platform changed that, providing a massively multithreaded general purpose architecture with up to 128 processor cores and thousands of many billions of floating point operations each second. CUDA runs on all current NVIDIA GPUs including the HPC-oriented Tesla product. The ubiquitous nature of those GPUs make them a compelling platform for fast high performance computing (HPC) applications.

II. RELATED WORK

Many researchers have done their work on the implementation of parallel programming for the efficient and effective parallel processing on a large data and processes. They propose an OpenGL based implementation that used the graphics pipeline. It incorporates a load balancing scheme that helps to scale back the computational cost. NVIDIA's similarly projected a parallel programming approach by using a CUDA programming can achieve a highly parallel computation for best performance of an applications.

Scherl et al. show a CUDA based method with a comparison to a Cell-based method. They claim that their methodology to reduces the number of instructions and the usage of registers. In contrast, our acceleration techniques focus on reducing the number and amount of off-chip memory accesses and hiding the memory latency with computation. Such memory optimisation is very important to enhance the performance of the FDK algorithm, which can be classified into a memory-intensive problem.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

III.CUDA OVERVIEW

Graphics Processing Units (GPUs) are used as for non-graphics computation for several years this can be mentioned as General-Purpose Computation on GPUs (GPGPU). The GPU is specialised for compute-intensive, extremely parallel computation (exactly what graphics rendering is about) so, extra transistors may be dedicated to process rather than data caching and flow management. The GPU is good at a data-parallel processing. The same computation executed on several data components in parallel with high arithmetic intensity. Several calculations per access same computation suggests that lower demand for sophisticated flow management High arithmetic intensity and plenty of data components mean that access latency may be hidden with calculations rather than huge data caches.

Fig.1. Basic CPU and GPU architecture

Fig. 1. Shows that the basic architectural difference of both CPU and GPU. Architecturally, the CPU consists of simply few cores with a lot of cache memory which will handle some computer code threads at a time. In distinction, a GPU consists of many cores will handle thousands of threads at the same time. The power of a GPU with 100+ cores to method thousands of threads will accelerate some computer code by 100x over a CPU alone. What's more, the GPU achieves this acceleration whereas being additional power and cost-effective than a CPU.

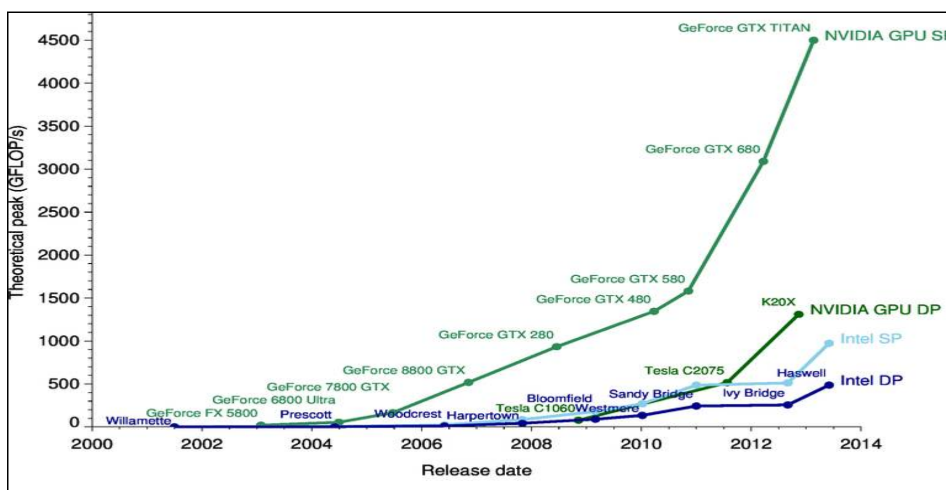

Fig.2. CPU vs GPU Performance

Fig. 2. Shows the Peak performance between CPU and GPU the graph shows us how a NVIDIA's GPU has more peak performance more than a CPU and it always get increased more than the CPU.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

IV. CUDA ARCHITECTURE

Each thread processor in a GPU can manage 96 concurrent threads, and these processors have their own FPUs, registers and shared local memory.

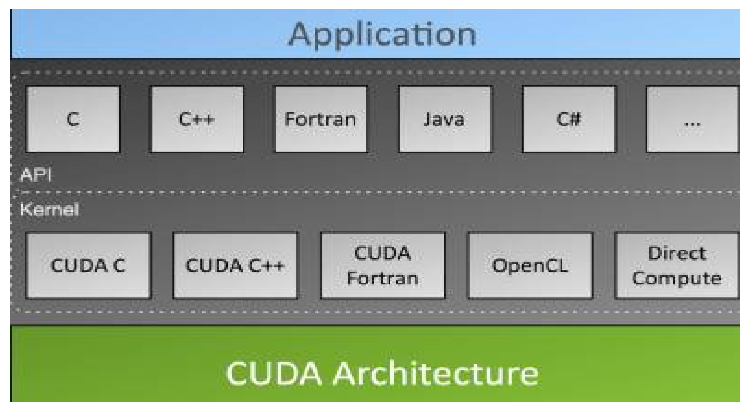

Fig.3. Language Supported by CUDA Architecture

Fig 3. Show which languages can be supported by the CUDA architecture for GPU computing and makes it possible to run a standard C on NVIDIA's GPU. To make this possible, NVIDIA has defined a general computing instruction set (PTX) and small set of C language extensions that allow developers to take advantage of the massively parallel processing capabilities in NVIDIA's GPU. The Portland Group is providing support for FORTRAN on the CUDA architecture, and others provide support for JAVA, PYTHON, .NET and other languages.

CUDA includes a C/C++ software development tools to perform a library, and a hardware abstraction mechanism that hides the GPU hardware from developers. Though CUDA needs programmers to write down a special code for parallel processing, it doesn't need them to explicitly mange threads in the conventional sense, that greatly simplifies the programming model.

CUDA development tolls work alongside a conventional C/C++ compiler, therefore programmers can compile GPU code with general-purpose code for the host CPU (Central Processing Unit). For now, CUDA aims at data intensive applications that needs single-precision floating point math. Double precision floating point is on NVIDIAs road map for a new GPU in close to a future.

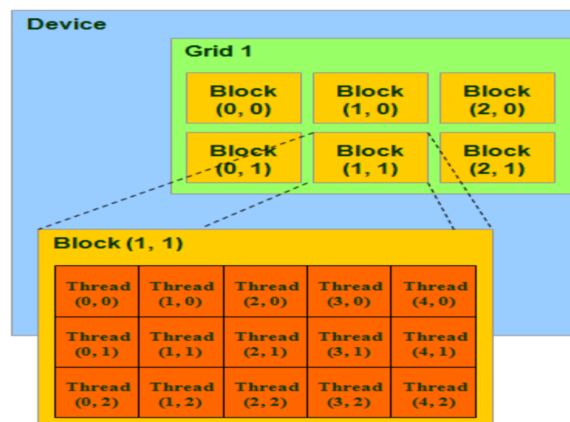

Fig.4. CUDA Architecture.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

As shown in fig.4. CUDA Architecture provides a three basic parts which help the programmer to effectively utilise the full computational capability of the graphics card on the system. CUDA Architecture splits the device into a grids, blocks and threads in a hierarchical structure.

A. The Grid

A grid is a cluster of threads all running identical kernel. These threads don't seem to be a synchronized. Each decision to CUDA from CPU is formed through one grid. Beginning a grid on CPU is a synchronous operation however, multiple grids will run directly. On Multi-GPU systems, grids cannot be shared between GPUs as a result of they use many grids for max potency.

B. The Block

Grids are composed of blocks. Every block is a logical unit containing variety of co-ordinating threads, an explicit quantity of shared memory. Even as grids don't seem to be shared between GPUs. Blocks don't seem to be shared between multiprocessors. All blocks in a grid use the same program. A in built variable "blockldx" may be wont to establish this block. Block ID's may be one dimensional or two dimensional. Typically, there are 65,535 blocks in a GPU.

C. The Thread

Blocks are composed of threads. Threads are run on the individual cores of the multiprocessors. However not like grids and blocks, they're not restricted to one core, like blocks, every thread has an ID (threadidx). Thread IDs may be one dimensional, two or three dimensional. The thread ID is relative to block it's in threads have an explicit quantity of register memory. Typically, there may be 512 threads per block.

V. CUDA PROGRAMMING MODEL

A. Thread Blocking

- A kernel is executed as a grid of thread blocks.
  - All threads share the data memory space.
- A thread block is a batch of threads that can co-operate with one another by:
  - Synchronizing their execution.
- For hazard free shared memory accesses.
  - Efficiently sharing data through a low latency shared memory.
- Two threads from two different blocks cannot co-operate.
- Threads and blocks have IDs.
  - So every thread will decide what data to work on.
  - Block ID can be one dimensional or two dimensional.
  - Thread ID can be one dimensional, two dimensional or three dimensional.

B. CUDA Device Memory Space

- Each thread can:
  - Read/Write per thread registers.
  - Read/Write per thread local memory.
  - Read/Write per block shared memory.
  - Read/Write per grid global memory.
  - Read only per grid constant memory.
  - Read only per grid texture memory.
- The host can Read/Write global, constant, and texture memories:

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

– Global memory is the main means of communicating Read/Write data between host and device and the contents are visible to all threads.
– Texture and constant Memories are constants initialized by host and contents visible to all threads.

VI. CUDA MEMORY TYPES

A. *REGISTER MEMORY:*

Data hold within the register memory is only visible to the thread that wrote it and lasts just for the lifetime of that thread. In most cases, accessing a register consumes zero clock cycles per instruction. However, delays will occur because of read after write dependencies and bank conflicts. The latency of read after write dependencies is roughly a 24 clock cycles. For newer CUDA devices that have 32 cores per multiprocessor, it may take up to 768 threads to completely hide latency.

B. *SHARED MEMORY:*

Data keep in shared memory is visible to all or any threads inside that block and lasts for the duration of the block. This is invaluable as a result of this kind of sort memory permits for threads to communicate and share data between one other.

C. *CONSTANT MEMORY:*

Constant memory won’t be used here as a results of their helpful for less than a specific form of applications. Constant memory is used for data that may be not change over the course of kernel execution and is read only. Using constant instead of global memory will reduce the desired memory bandwidth. However, this performance gain will only be realised when a warp of threads read identical location. Similar to constant memory.

D. *TEXTURE MEMORY:*

Texture memory is another sort of read-only memory on the device. Once all reads in a very warp are physically adjacent, using texture memory will cut back memory traffic and can increase the performance compared to a global memory.

E. *GLOBAL MEMORY:*

Data stored in Global Memory is visible to all or any threads within the application and lasts for the duration of the host allocation. It is read and write memory. It is slow and not cached and requires sequential and aligned 16 byte reads and writes to be fast

Table. 1. Shows the detail difference between a CUDA memory types as per their locations as on chip or off chip, its Size, its latency and what types of access they have.

Memory	Location	Size	Latency	Access
Register	On-chip	16384 32-bits registers per SM	0	R/W
Shared Memory	On-Chip	16KB per SM	0	R/W
Constant	On-Chip	64KB	0	R
Texture	On-Chip	1GB	>100 cycles	R
Global	Off-Chip	1GB	>100 cycles	R/W

Table. 1. CUDA memory types

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

VII. EXPERIMENTAL SETUP

The Software Development Kit or the SDK may be a good way to learn a few about a CUDA, anyone will compile the examples and can learn how the toolkit works. The SDK is available at the NVIDA's website and can be downloaded by any aspiring programmers who wants to learn about a CUDA programming. Anyone who has some basic knowledge about C programming can begin a CUDA programming very quickly. No prior knowledge of graphics programming is required to write CUDA codes. CUDA is derived from C with some modifications that enabled it to run on GPU. CUDA is a C for GPU.

A. *To start developing a CUDA applications do the following steps:*

1. Install visual studios as an environment for CUDA programming.
2. Install specific NVIDIA GPU drivers according to GPU model and also install the CUDA SDK.
3. Write a program code according to a normal C/C++ programming constructs.
4. Now, change the written program into the CUDA parallel code by using the library functions provided by the SDK. The library functions are used to copy data from host to device, change execution from CPU to GPU and Vice versa, copy data from device to host.

B. *The Basics of CUDA code implementation as:*

1. Allocate CPU memory.
2. Allocate same amount of GPU memory using library function "CudaMalloc".
3. Take data input in CPU memory.
4. Copy data into GPU memory using library function CudaMemCpy with parameter as (CudaMemcpyHostToDevice).
5. Perform processing in GPU memory using kernel calls. Kernel calls are a way to transfer control from CPU to GPU; they also specify the number of grids, blocks and threads i.e. Parallelism is required for your program.
6. Copy final data in CPU memory using library function CudaMemCpy with parameter as (CudaMemcpyHostToDevice).
7. Free the GPU memory or other threads using library function Cudafree.

VIII. ADVANTAGES AND DISADVANTAGES

A. Advantages:

1. The kernel calls in CUDA are written in simple C like languages. So, a programmer's task is get simpler.
2. Kernel code has full pointer support.
3. Supports C++ constructs.
4. Fairly simple integration API.
5. Better fully GPU accelerated libraries currently available.
6. CUDA can be used for a large number of languages.
7. The programmer has a lot of help available in the form of documentation and sample codes for different platforms.
8. A programmer can use CUDA's visual profiler, a tool used for performance analysis.
9. Updates are more regular.
10. Has been on the market much longer.

B. Disadvantages:

1. Restricted to only a NVIDIA's GPU.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

2. If CUDA accelerated hardware is not present, error is reported and task is aborted. There is no CPU backup in this scenario.

3. It is difficult to start programming in CUDA, because it requires setting up of the environment in which CUDA enabled programs can run. For example, The NVCC compiler has to be incorporated into build.

IX. APPLICATIONS

CUDA provides a various benefit for many applications such as:

A. *Fast video Transcoding:*
Transcoding could be a very common, and highly advance and complex procedure that simply involves trillions of parallel computations, many of are floating point operations. Applications such as FormatFactory have been created which is harness the raw computing power of GPUs in order to transcode video much faster than ever before.

B. *Video Enhancement:*
Complicated video enhancement techniques often need an enormous amount of computations. As an example, there are lots of algorithm that can upscale a movie by using information from frames surrounding the current frame. This involves too many computations for a CPU to handle in a real time.

C. *Computational Sciences:*
In the raw field of computational sciences, CUDA is extremely advantageous. As an example, it is currently possible to use CUDA with MATLAB, which may increase computations by an excellent quantity. Different common tasks such as computing eigenvalues, or a SVD decompositions, or other different matrix mathematics will use CUDA in order to speed up the calculations.

D. *Fluid Dynamics:*
Fluid dynamics simulations have also been created. These simulations need a large number of calculations, and are useful for wing design, and other engineering tasks.

X. CONCLUSION AND FUTURE WORK

The GPUs are gaining more popularity within the scientific computing community due to their high parallel processing capability and easy availability, and are becoming the preferred choice of the programmers due to the support offered for programming by models such as CUDA.

The future of parallel computing, is clearly very much in the hands of NVIDA's CUDA Architecture. So, NVIDIA still has a lot of challenges to meet to make a CUDA stick, since whereas technologically it's undeniably a success, the task now is to convince developers that it's a credible platform and that doesn't look like it will be easy for NVIDIA.

A. Accelerated rendering of 3D graphics
B. Accelerated interconversion of Video file formats.
C. Accelerated encryption, decryption and compression.
D. Mining cryptocurrencies.

XI. ACKNOWLEDGEMENT

We thank for our guide to inspire us for this Paper and to guide us for this paper, and the special thanks for the entire NVIDIA's team that brings the CUDA to market.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

REFERENCES

1. NVIDIA Corporation "What is GPU Computing?", http://www.nvidia.com/object/GPU_Computing.html
2. Wikipedia, "CUDA", http://en.wikipedia.org/wiki/CUDA
3. J. Nickolls and W.J. Dally, "The GPU Computing Era", IEEE Micro, vol. 30, no. 2, 2010, pp. 56-69.
4. See http://gpgpu.org.
5. Anthony Lippert – "NVIDIA GPU Architecture for General Purpose Computing", http://cs.wm.edu/~kempler/cs654/slides/nvidia.pdf.
6. David Kirk/NVIDIA and Wen-Mei Hwu, "CUDA Programming Model", 2006-2008.
7. Mathew Guidry, Charles McClendon "Parallel Programing with CUDA".
8. NVIDIA Corporation, "NVIDIA CUDA Programming Guide", Version 7.0, (September 2015).
9. Supercomputing blog "Practical Applications for CUDA", http://supercomputingblog.com/cuda/practical-applications-for-cuda.
10. D.goddeke, R. Strozodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, S. Tureka, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput. 33 (Nov. 2007) 685-699.

BIOGRAPHY

Prasad C. Shinde
Student, Department of Computer Engineering, BVCOEL, Lavale, Pune

Tejaswi P. Kadam
Student, Department of Computer Engineering, BVCOEL, Lavale, Pune

Rushikesh Mahadik
Student, Department of Computer Engineering, BVCOEL, Lavale, Pune

Prof. Kumkum Bala
Professor, Department of Computer Engineering, BVCOEL, Lavale, Pune