Parallelization of BFS Graph Algorithm using CUDA

[1] Chetan D. Pise, [2] Shailendra W. Shende

[1, 2] Department of Information Technology,Yeshwantrao Chavan College of Engineering Nagpur University Nagpur,Maharashtra 441 110, INDIA.

Abstract - Graphs play a very important role in the field of Science and Technology for finding the shortest distance between any two places. This Paper demonstrate the recent technology named as CUDA (Compute Unified Device Architecture) working for BFS Graph Algorithm. There are some Graph algorithms are fundamental to many disciplines and application areas. Large graphs are common in scientific and engineering applications consisting operation on million of vertices and edges[1, 11, 12, 18, 19]. To have faster execution of such operations parallel computation is very essential to reduce overall computation time. Today's Graphics processing units (GPUs) have high computation power. and low price. Compute Unified Device Architecture (CUDA) software interface by NVIDIA, becoming a new programming approach of the general purpose computing on graphics processing units (GPGPU). Massively Multithreaded architecture of a CUDA device makes various threads to run in parallel and hence making optimum use of available computation power of GPU. In this paper we are demonstrating the comparison between different large data sets, which are in term of kernel execution that are carried out on GPU using CUDA. In most of the applications GPU can be used as an inexpensive co-processor. Breadth-first search (BFS) is a core primitive for graph traversal and very much important for graph analysis.

Keywords - BFS, Graph, GPU, CUDA, NVIDIA Compiler(NVCC).

1. Introduction

Parallel programming is a generic concept describing a range of technologies and approaches. However in general it describes a system whereby threads of instruction are executed truly in parallel over a shared or partitioned data source. As part of parallel computing, General Purpose computation on Graphics Processing Units (GPGPU) is a new and active field. The main goal in GPGPU is to find parallel algorithms capable of processing concurrently huge amounts of data over a number of Graphic Processing Units (GPU). GPGPU involves using the advanced parallel Graphics Processing Unit devices now readily available for general purpose parallel programming. Within GPGPU research, implementing graph algorithms is an important sub-field and is the focus of this paper [16, 19]. Graph representations are common in many problem domains including scientific, medical and engineering applications. Fundamental graph operations mostly like breadth-first search, single source shortest path, depth-first search etc., occur frequently in these areas. There may some problems which belong to large graph, consisting millions of vertices. These operations have found applications in various problems like routing analysis, map of the countries, transportation, robotics, VLSI chip layout, network traffic analysis, and plant & facility layout, phylogeny reconstruction, data mining, and can require graphs with millions of vertices [1]. Algorithms become impractical on very large graphs on normal System configurations.

Parallel algorithms can achieve practical times on basic graph operations but at a high hardware cost. Bader et al. [6] use CRAY MTA-2 supercomputer to perform BFS and single pair shortest path on very large graphs. The hardware is too expensive to use such methods. Commodity graphics hardware has become a cost-effective parallel platform to solve many problems. Many problems in the fields of linear algebra [6], computer vision, image processing, signal processing, etc., have benefited from its speed and parallel processing capability. There are lot of graph algorithms which are implemented on GPU. They are, however, severely limited by the memory capacity and architecture of the existing GPUs. GPU clusters have also been used to perform compute intensive tasks, like accurate nuclear explosion simulations, finite element computations, etc.

Authors[2] Swapnil D. Joshi and V. S. Inamdar also discussed about GPU as GPU stands for Graphics Processing Unit and is a single chip processor used primarily for 3D applications. It creates lighting effects and transforms objects every time a 3D scene is redrawn. These are fully mathematically-intensive tasks and would put quite a strain on the expensive CPU. GPU provides high computational power with low costs. More transistors can be devoted for data computation rather than data caching & flow control as in case of CPU. With multiple cores driven by very high bandwidth (memory), today's GPU's offer incredible resources. The G80 series of GPUs from Nvidia also offers an alternate programming model called Compute Unified Device Architecture (CUDA) to the underlying parallel processor. Pawan Harish[1] demonstrates that how CUDA is highly suited for general purpose programming on the GPUs and provides a model close to the PRAM model. The interface uses standard C code with parallel features. A similar programming model also known as Close To Metal (CTM) is provided by ATI/AMD [18].

Pawan Harish et al[1, 18,19] present the implementation of a few fundamental graph algorithms on the Nvidia GPUs using the CUDA model. Specially, we show results on breadth-first search (BFS), all-pairs shortest path (APSP), single-source shortest path (SSSP) algorithms on the GPU. This method is capable of handling large graphs, unlike previous GPU implementations [9]. We can perform BFS on a 10 million vertex random graph with an average degree of 6 in one second and SSSP on it in 1.5 seconds. The times on a scale-free graph of same size is nearly double these. We able to compute APSP on graphs with 30K vertices in about 2 minutes. Due to some constrains of memory on the CUDA device graphs above 12 million vertices with 6 degree per vertex cannot be handled using current GPUs [1, 18, 19].

Author Yanwei Zhao et al[3] says that the prevalent trends in microprocessor architecture has been continually increasing chip-level parallelism. Multi-core CPUs which providing several scalar cores are now commonplace and there is every indication that the trend towards increasing parallelism will continue on towards many-core chips that provide far higher degrees of parallelism.

Applications[7]: CUDA C started in early 2007, a variety of application and industries have enjoyed a great deal of success by choosing to build applications in CUDA C technology. Furthermore, applications running on NVIDIA graphics processors enjoy superior performance per dollar and performance per watt than implementations built exclusively on traditional central processing technologies. Just a few of the ways in which people have put CUDA C and the CUDA Architecture into successful use.

Objective:

- Main Objective is to reduce the time for processing millions of vertices from a large graph.

- Providing a good algorithmic solution for upcoming Challenges using newly area as High Performance computing.

- Instead of using expensive supercomputer to process large data we use a low price GPU(GFORCE -GT630M).

- Lastly shows that how the GPU's are better to use rather than normal CPU's to process large data by using best appropriate algorithm.

2. CPU vs GPU

GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remaining of the code still runs on the CPU. From a user's perspective, parallel applications simply run significantly faster on GPU. A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few number of cores to optimized sequential serial processing while a GPU consists of thousands of smaller and more efficient cores designed for performing multiple tasks simultaneously.

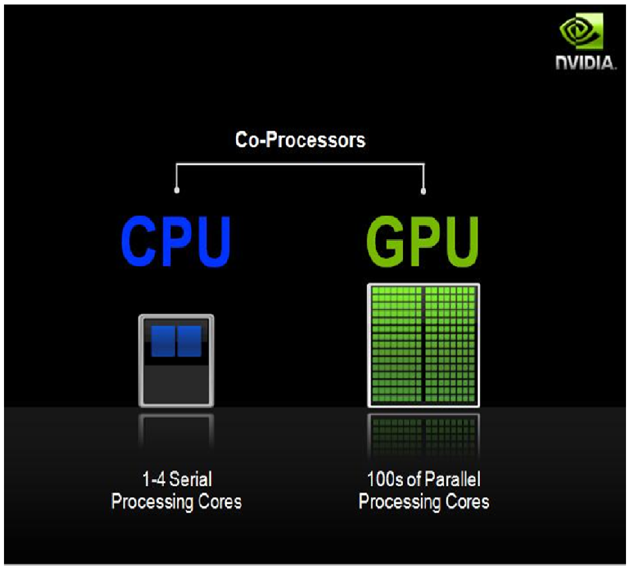
NVIDIA.
Co-Processors
CPU
GPU
1-4 Serial Processing Cores
100s of Parallel Processing Cores

Figure 1: How CPU and GPU [10]

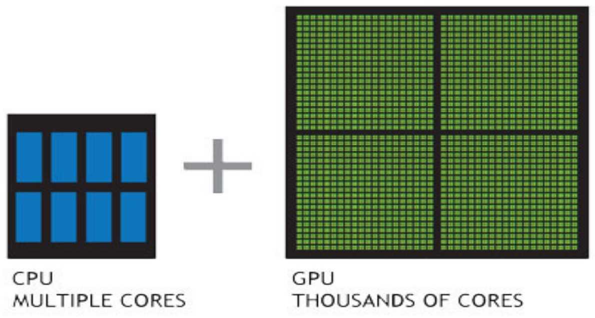

Figure 2: CPU vs GPU [10]

Every GPUs have thousands of cores to process Parallel Processing workloads efficiently and efficiently.

3. GPU and GPGPU

General purpose programming on Graphics processing Units (GPGPU) tries to solve a problem by posing it as a graphics rendering problem. Using GPU we can restrict the range of solutions. A GPGPU solution is designed to follow the general flow of the graphics pipeline (consisting of vertex, geometry and pixel processors) with each iteration of the solution being one rendering pass. The GPU memory layout is also optimized for graphics rendering. This controls/restricts the GPGPU solutions as an optimal data structure may not be available. Creating efficient data structures using the GPU memory model is a challenging problem in itself [1]. GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate engineering scientific and enterprise applications. Pioneered in 2007 by NVIDIA Corporation, GPUs now power energy-efficient data centers in government labs, enterprises, universities and small-and-medium businesses around the world [10, 15, 18].

The program execution flow on CUDA

The CUDA applications include[20] two parts of codes, oneis CPU (Host-End) serial code segment, which achieves the initialization and configuration work, the other one is GPU(Device-End) parallel code segment, of which the kernel function executes as the way of SIMT (Single Instruction Multiple Threads).The CUDA applications can be usually divided into five parts. They are initialization work, data loading, parallel computing, results processing and memory release respectively, just as Fig. 4 shows. The initialization work mainly includes parameters computing, host memory and device memory allocation. Data loading refers to transforming frame data from host memory to device memory. Parallel computing completes the kernel function. Results processing can contain two kinds of work, one is to transform the frame data from device memory to host memory and write back to disk, the other one is to call the APIs such as Direct3D to display images directly. Memory release is responsible for recycling memory resources.

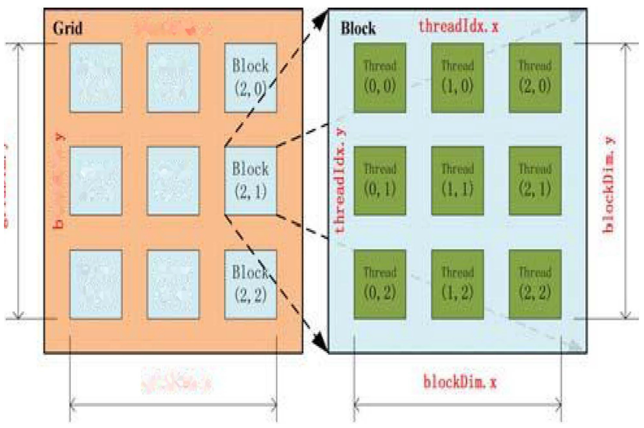

Fig. 3. The thread architecture of the parallel program on GPU[20]

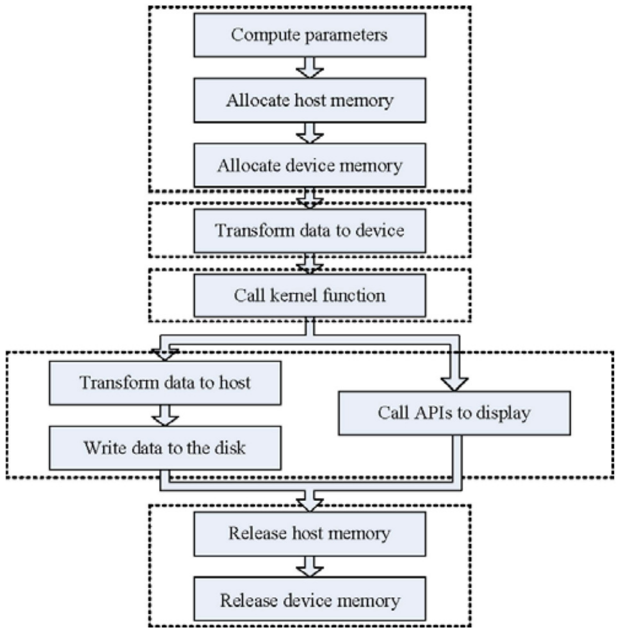

Fig. 4. The program execution flow on CUDA[20]

4. CUDA

Compute Unified Device Architecture (CUDA) is a new software and hardware architecture for issuing and managing Multiple computations on the GPU as a data parallel computing device (SIMD) with the no need of mapping them to a graphics API. CUDA has been developed by Nvidia and to use this architecture requires an Nvidia GPU. It is available for the GTX 6 series , GeForce 8 series, etc GPUs, Tesla Solutions and some Quadro Solutions [2].

a. CUDA Hardware

J. M. Kemp, et al [5] uses CUDA enabled GPUs manufactured by NVIDIA, starting from their G80 range of GPUs and all respective subsequent versions. It is worth noting that an NVIDIA's G80 range as 8800GTX GPU will be used for their project.

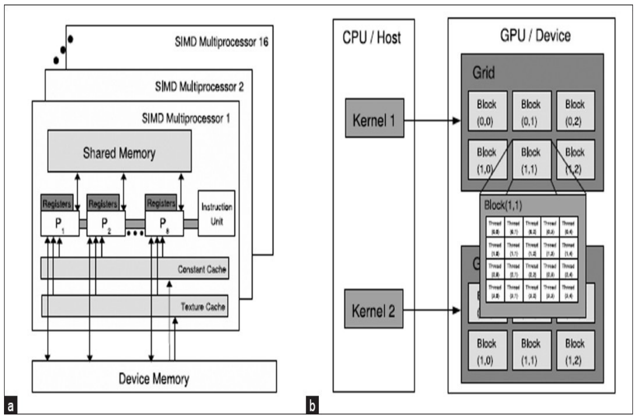

Figure 5: CUDA hardware model

b. CUDA Software

CUDA allows programmers to write functions known as kernels that will be executed on the GPU. A programmer does not see the hardware architecture of the GPU. Instead, they see a number of threads which are organized into blocks. Each thread is executed following the Single Program Multiple Data (SPMD) model. A CUDA programmer can write how many threads are able to define to execute for each kernel. On an NVIDIA 8800GTX, the programmer can define 512 threads per block only.

V. S. Inamdar et al[2] says that CUDA program is organized into a host program, which consisting one or more sequential threads running on the CPU as host and on device like GPU one or more parallel kernels that are suitable for execution for parallel processing. As a software interface, CUDA API is a set of library functions which can be coded as an extension of the C language. A NVIDIA CUDA compiler (NVCC) generates executable code for the CUDA device.

5. Parallel Algorithms

I.A. Stewart and Summerville (2007, p25) et al[5, 13, 16] talks of a layered model approach to the architecture design which "organizes a system into layers, each of which provide a certain set of services". Step by step Layered approach used in this project to design. Each layer can be seen in Figure 6.

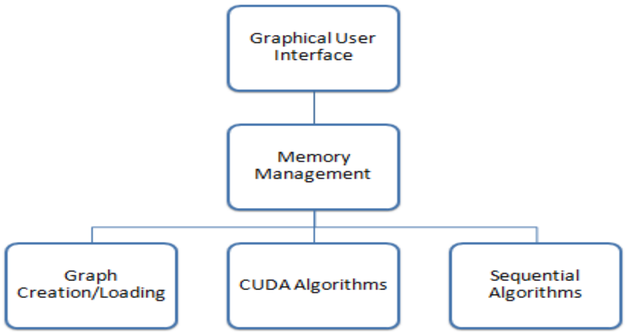

Figure 6: Layered system architecture

BFS

This implementation of BFS using CUDA is based upon the work by Harish & Narayanan [in 1] (2007). However, instead of searching a undirected, weighted graph, this implementation searches an unweighted, undirected graph. The search halt process can be happen as soon as the goal node is found, meaning that BFS will always find the fastest route to the goal node.

For BFS, there is one thread per vertex, demonstrated in Figure . Using a block size of 512, the number of blocks and therefore threads are calculated via the number of vertices in the graph that is to be searched. This shows that there is the potential to have 511 redundant threads that are idle during the kernel execution [5].

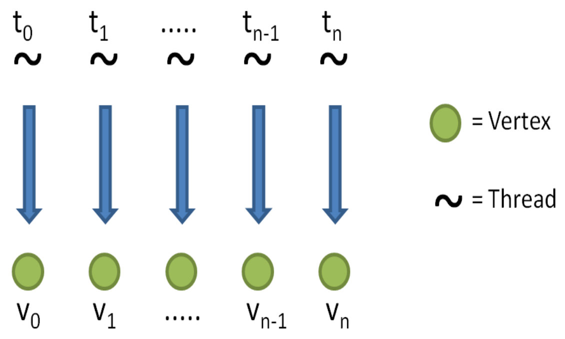

Figure 7: Example mapping of threads to vertices

Author David A. Bader and Kamesh Madduri [6] proposed that Unlike prior parallel approaches to BFS, author says on the MTA-2 we do not consider load balancing or the use of distributed queues for parallelizing BFS.

6. Experimental Results

The following results are worked out using NVIDIA GTX630M consisting 96 cores, and with the technology CUDA 5.5. For 6 Nodes, For 4096 Nodes.

Table 1: Experimental Result

Sr. No.	Graph Data Set (No. of NODES)	Time to Execute in GPU	No. of Times Kernel Execute	Time to Execute in CPU	Speed-Up Factor (CPU/ GPU)
1.	6 Nodes	0.212512 ms	3	3 ms	14.11x
2.	9 Nodes	0.291680 ms	6	2 ms	6.8568x
3.	4096 Nodes	1.192544 ms	8	11.236 second	9000x

7. Conclusion

In this project for parallelization we used NVIDIA Graphics card with 2 Multi-processors. As per this project various serial and parallel algorithms can be develop to compare the results between then and analyze which algorithms are best one for upcoming challenges. In this I used NVIDIA GTX630M for my implementation of BFS algorithm with version CUDA 5.5. I used various similar data sets and find that how many times kernel is going to execute and also find the execution time of such a large data set .

Acknowledgments

Firstly I say thank you to Dr. Pawan Harish[1] sir for their kind support, thank you to my guide Prof. S. W. Shende Sir, Prof. Dharmik Sir(HOD), Thank you to Prof. Jeetendra Tembhurne(CUDA Developer from VNIT) and to all friends who supported me and special thanks to Aniruddha and his friends.

References

[1] Pawan Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA", Center for Visual Information Technology, IIITHyderabad 2008.

[2] Swapnil D. Joshi, Mrs. V. S. Inamdar, "Performance Improvement in Large Graph Algorithms on GPU using CUDA: An Overview", International Journal of Computer Applications (0975 – 8887) Volume 10– No.10, November 2010.

[3] Yanwei Zhao, Zhenlin Cheng, Hui Dong, Jinyun Fang, Liang Li, "Fast Map Projection on CUDA", IGARSS 2011.

[4] Enrico Mastrostefano, Massimo Bernaschi, Massimiliano Fatica, "Large Graph on multi-GPUs", IAC-CNR, Rome, Italy NVIDIA Corporation May 11, 2012.

[5] J. M. Kemp, I. A. Stewart, "Parallel Graph Searching Algorithms using CUDA", Board of Examiners in the School of Engineering and Computing Sciences, Durham University 2011.

[6] David A. Bader, Kamesh Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2", February 26, 2006.

[7] Book: Jason Sanders, Edward Kandrot, "Programming based on CUDA".

[8] Book: CUDA C Programming Guide.

[9] Book: David B. Kirk, Wen-Mei W. Hwu, "Programming Massively Parallel Processors".

[10] http://www.nvidia.com/object/what-is-gpu-computing.html

[11] 9th DIMACS implementation challange – Shortest paths http://www.dis.uniroma1.it/challenge9/download.shtml.

[12] 10th DIMACS Implementation Challenge-Graph Partitioning and Graph Clustering http://www.cc.gatech.edu/dimacs10/index.html

[13] Duane Merrill, Michael Garland, Andrew Grimshaw. "Scalable GPU Graph Traversal" ACM PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.2012

[14] Lijuan Luo, Martin Wong, Wen-mei Hwu, "An Effective GPU Implementation of Breadth-First Search", ACM DAC-10 June 13-18, 2010, Anaheim, California, USA.

[15] Alex Quach, Firas Abuzaid and Justin ChenCUDA, "Implementation of Large Graph Algorithms", CS224W Project Final Report, 10-Dec-2010.

[16] Michael J. Dinneen, Masoud Khosravani and Andrew Probert, "Using OpenCL for Implementing Simple Parallel Graph Algorithms".

[17] Jianlong Zhong, Bingsheng He, "Parallel Graph Processing on Graphics Processors Made Easy", Proceedings of the VLDB Endowment, Vol. 6, No. 12.

[18] Chetan D. Pise, Shailendra W. Shende, “Parallelization of Graph Algorithms on GPU using CUDA”, ICRIET-19 Jan 2014, Bangalore, INDIA.

[19] Chetan D. Pise, Shailendra W. Shende, “NVIDIA Graphics Card for Parallelization of BFS Graph Algorithms using CUDA”, ICRTET’-14, 28th march 2014, Nashik, INDIA, Elsevier Proceeding.

[20] Weiyang Mu , Jing Jin , Hongqi Feng , Qiang Wang Member IEEE, “The Implementation and Performance Analysis of A WMMF Parallel Algorithm on GPU”.

Authors:

Chetan D. Pise received the B.E. degree in Information Technology from KITS Ramtek, Nagpur, INDIA in 2011 and perusing MTech. degree in Information Technology from YCCE, Nagpur in 2013. He worked as a Teaching Assistant at 2-WEEK ISTE DBMS-WORKSHOP conducted by IIT Bombay (21st may to 31st may 2013) in YCCE Nagpur.

Shailendra W. Shende pursuing Phd. from VNIT, Nagpur in the area of Parallel Processing. He worked as a Coordinator at 2-WEEK ISTE DBMS-WORKSHOP conducted by IIT Bombay (21st may to 31st may 2013) in YCCE Nagpur. He now currently working as a Asst. Prof. in Information Technology Department at YCCE Nagpur.