International Journal of Computer Applications (0975 – 8887)
Volume 10– No.10, November 2010

Performance Improvement in Large Graph Algorithms on GPU using CUDA: An Overview

Swapnil D. Joshi
Student
Department of Computer Engineering and IT,
College of Engineering, Pune, MS, India

Mrs. V. S. Inamdar
Assistant Professor
Department of Computer Engineering and IT,
College of Engineering, Pune, MS, India

ABSTRACT

The basic operations on the graphs with millions of vertices are common in various applications. To have faster execution of such operations is very essential to reduce overall computation time. Today’s Graphics processing units (GPUs) have high computation power and low price. This device can be treated as an array of Single Instruction Multiple Data (SIMD) processors using CUDA software interface by Nvidia. 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 case of graph algorithms, vertices of the graphs are processed in parallel by mapping them to various threads on device. By making thousands of threads to run in parallel, computation time required for these algorithms is drastically decreased as compared to their CPU implementation.

We studied different parallel algorithms for Breadth first search, all pairs shortest path that are carried out on GPU using CUDA and make their comparative study with respect to execution time, data structure used, input data etc. In the paper, we presented overview of various parallel methods carried out on GPU using its multithreaded architecture for BFS, APSP by various authors.

General Terms

Parallel computing, Graph Algorithms, SIMD architecture, GPU, CUDA.

Keywords

BSP mode, Level Synchronization.

1. INTRODUCTION

1.1 Need of Performance Improvement in large graph algorithms

Graphs are very popular data representations in various fields including scientific and engineering domains. In some problems large graphs with millions of vertices are to be processed. These operations have found applications in various problems like map of the countries, routing analysis, transportation, robotics, VLSI chip layout, network traffic analysis, data mining, and plant & facility layout etc. Basic operations on the graphs such as Breadth first search, all pairs shortest path, max flow/min cut algorithm plays important role in such problems. Sequential methods for such graph operations are available but they are not that efficient with respect to computing time and use of available resources [1]. It is always essential to have faster execution of such operations to reduce overall complexity of whole problem [4]. Also, to store the graph with millions of vertices in a file and processing of such a large file in efficient manner is a challenging task. For this purpose, various data structures are studied and compared for their performance with respect to space complexity.

1.2 Graphics Processing Unit (GPU)

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 mathematically-intensive tasks, which otherwise, would put quite a strain on the CPU. Lifting this burden from the CPU frees up cycles that can be used for other jobs. 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 memory bandwidth, today's GPUs offer incredible resources for both graphics and non-graphics processing.

1.3 Compute Unified Device Architecture (CUDA)

CUDA stands for Compute Unified Device Architecture and is a new hardware and software architecture for computation on GPU. This architecture is by Nvidia and it makes use of maximum of the computation power provided by GPU by deploying massive multithreading. Also, GPU and CUDA can be used in association to design and implement any general purpose application on GPU, thus making it GPGPU (General purpose graphics processing unit). CUDA provides an API that’s an extension to the C programming language for a minimum learning curve. It also provides general DRAM memory addressing for more programming flexibility i.e. both scatter and gather memory operations. It features a parallel data cache or on-chip shared memory with very fast general read and writes access, that threads use to share data with each other [2]. GPU and CUDA collectively used to achieve parallelism in great sense.

The overview of the graph algorithms on GPU using CUDA presented here is organized as follows. Section 2 is about the various possible representations of the graphs. The details of CUDA hardware and software models are described in section 3. Then types of graphs used in experiments and some terminologies related to the topic are covered. Section 6 talks about the Parallel implementation of the breadth first search. Subsequent sections contain overview of parallel single source shortest path and all pairs shortest path.

2. GRAPH REPRESENTATION ON CUDA DEVICE

It is very important to have efficient way of storing the complex graphs, as it involves millions of vertices. We have various data structures those can be used to store such graphs. J. Hyvonen et al. [5] have studied these data structures in depth for CPU. It is very much useful in case of sparse representations.

In case of GPU, it does not allow the use of user defined data structures very well. So it is very challenging job to represent the data as an efficient data structure [6]. With the help of CUDA, GPU can be programmed with better data representations as CUDA treats memory as a general array and hence can support efficient data structures. For the graph G (V, E), adjacency matrix is popular data structure as it is very simple to represent and to understand. However in case of large graphs with millions of vertices, it is not the good choice as space requirement is O(V^2). Also in case of sparse graphs, it has various entries zero. Adjacency list is a better choice for storing the graphs. Vibhav Vinit et al.[7] have suggested the use of packed adjacency list representation for the graphs. It consists of two lists for vertex and the edges in the graph. Every vertex in the vertex list points to its starting edge list in the packed adjacency list of edges. This uses feature of CUDA as it supports uneven array size. This approach is very efficient with respect to space complexity as it requires just O(V+E) memory size.

3. COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

Compute Unified Device Architecture (CUDA) is a new software and hardware architecture for issuing and managing computations on the GPU as a data parallel computing device (SIMD) without the 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 GeForce 8 series GPUs, Tesla Solutions and some Quadro Solutions.

3.1 Hardware Model

CUDA Device is collection of various multiprocessors with m processors each (figure 1). Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD). It has its own shared memory which is common to all the processors inside it. The processors within multiprocessors have set of 32-bit registers, texture and constant memory caches. Texture and constant caches are read only cached memory space and texture cache is optimized for texture fetching operations. These multiprocessors communicate with each other through the device memory, which is available to all processors of the multiprocessors.

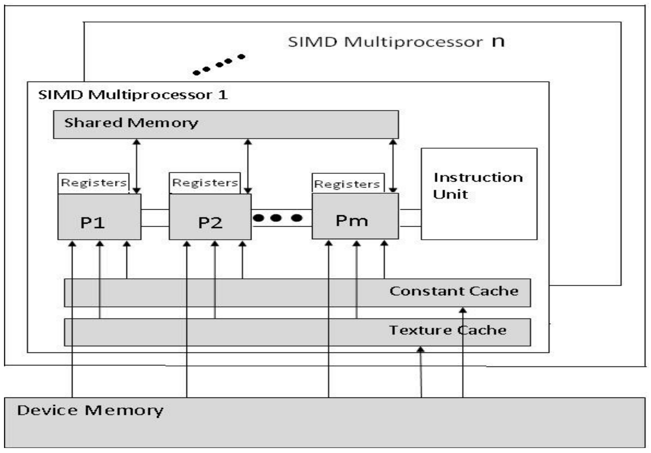

Figure 1: CUDA Hardware Model

3.2 Programming Model

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

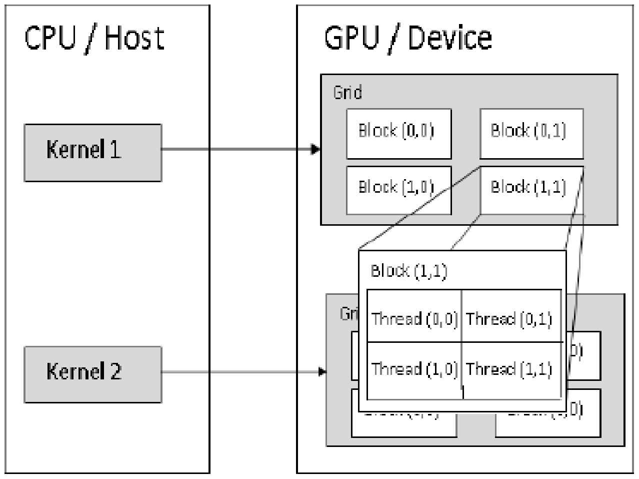

Figure 2: CUDA Software Model

As a software interface, CUDA API is a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. For the programmer, the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU, 32 on present GPUs. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) is mapped to a multiprocessor at a given time (figure 2). A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses.

Multiple blocks can be assigned to a multiprocessor and their execution is time-shared. A single computation on a device generates a number of blocks. A collection of all blocks in a single computation is called a grid. All threads of the blocks mapped to a multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel. GPU is a co-processor to the CPU and needs to be initiated by the CPU.

3.3 Extensions to C programming language

CUDA supports various programming languages like C, C++, JAVA with JCUDA. It provides some extensions to program on CUDA device i. e. Kernel code. Some examples include:

- Function type qualifiers to specify whether a function is executed on the device or the host and whether it is callable from the device or the host. E.g. _device_, _global_, _host_
- Variable type qualifiers to specify the memory space in which a variable resides. E.g. _device_, _constant_, _shared_.
- A directive to be given while the execution of a kernel and which specifies the execution model viz. the grid and block dimensions.
- Variables to specify grid and block dimensions and block and thread IDs. E.g. griddim, threadIDx, blockdim, blockIDx, warpsize.

Also, CUDA provides some additional libraries such as following.

- The CUBLAS library: CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA driver. It provides functions to create matrix and vector objects in GPU memory and process them; uploading the final result in host memory in the end.
- The CUFFT library: CUFFT provides a simple interface for computing parallel Fast Fourier Transforms on the GPU.

4. TYPES OF GRAPHS

There are three major categories of the graphs. Experiments on these graphs are carried out. Random Graphs are the graphs in which there is not much difference in the degrees of the vertices in the graph as well as large number of vertices have similar degrees. A slight variation from the average degree results in a drastic decrease in number of such vertices in the graph. In case of R-MAT / Scale Free graphs, a large number of vertices have small degree with a few vertices having large degree. This model best approximates large graphs found in real world. For these graphs, it is difficult to predict the load on processors at particular time and processing activities in two iterations may vary greatly. Due to its small degree distribution over most vertices and uneven degree distribution these graphs expand slowly in each iteration and exhibit uneven load balancing on the threads. Therefore these graphs have poor performance even after applying multithreading as compared to the other graphs. SSCA#2 graphs are made up of random sized cliques of vertices with a hierarchical distribution of edges between cliques based on a distance metric. It is explained by D A Badar et al [9]. There is one more type of graph called grid graphs. In grid graphs each node has fixed number of neighbors. Therefore the number of nodes traversed in current level is almost same as those in previous one.

5. SOME BASIC TERMINOLOGIES

- Heterogeneous programming: The Compute Unified Device Architecture (CUDA) from Nvidia presents a heterogeneous programming model where the parallel hardware can be used in conjunction with the CPU. This provides good control over sequential flow of execution which was absent from the earlier GPGPU. Serial code executes on the host while parallel code executes on the device.
- Host and Skeleton: CPU is known as Host. Also, code to be run on Host as skeleton code. Basically these are the sequential steps necessary for synchronization of threads and performed on CPU.
- Device and Kernel: The steps which are specific to the algorithm we are constructing are to be run on GPU and is called device code or Kernel.
- Synchronization of the threads: The CUDA hardware can be seen as a multicore/manycore co-processor in a bulk synchronous parallel mode when used in conjunction with the CPU. As we know, there are thousands of vertices to be processed at a time, so it is very essential to have synchronization between the threads running in parallel.
- Bulk synchronous parallel model (BSP): CUDA hardware is used in conjunction with CPU. Synchronization of the threads is achieved with CPU deciding the barrier for synchronization. Concurrent computation takes place on each processing element asynchronously. Processing elements exchange the data between them if necessary. Each thread waits for all other threads to finish achieving synchronization.

6. PARALLEL BREADTH FIRST SEARCH

The Breadth first search (BFS) has tremendous applications in various areas. These include image processing, space searching, network analysis, graph partitioning, automatic theorem proving etc. The BFS problem is, given an undirected, unweighted graph G(V,E) and a source vertex S, find the minimum number of edges needed to reach every vertex V in G from source vertex S. The best time complexity reported for sequential algorithm is O(V+E).

6.1 BSP mode and Level Synchronization

P. Harish et al. [11] solve the problem using concept of level synchronization, in which all the vertices at particular level are processed in parallel. It assigns a thread to every vertex. Concurrent computation takes place at vertices of current level and all threads waits for other threads at that level to finish, treating CUDA device as bulk synchronous parallel model. They maintained one global cost array Ca of size V, which contains the number of edges needed to reach that vertex from source vertex. Also, frontier array F_a contains the vertices at the current level. Vertices those are present in frontier array updates the costs of their neighbor vertices with cost of itself plus one. Vibhav et al. [7] have used vertex compaction process with the help of prefix sum i. e. deploying threads only for those vertices which are active. At particular time, only small number of vertices may be active. Vertex compaction is very useful for removing unnecessary threads. They carried out experiments on various types of graphs and compared the results with the best sequential implementation of BFS.

It shows lower performance on low degree graphs. In such case, parallelism is achieved at small extent i. e. very small number of vertices is processed at a time in linear graphs. Due to linear nature of R-MAT graphs, expansion of frontier is very slow at every level and it results in low multithreading. Also, uneven load balancing is one reason for poor performance of such graphs. Authors report 5 times speed up in case of R-MAT graphs. In case of random graphs and SSCA#2 graphs, expansion of frontier is somewhat uniform and enough number of threads is deployed at every level. This results in speed up of nearly 15 times over their CPU implementation [7].

6.2 Lockstep BFS for grid graphs

The method mentioned in 6.1 utilizes an array of flags of length V, as number of nodes traversed at each depth level is arbitrary and not depends upon the number of nodes in previous level. On the other hand, there is one type of graphs in which every vertex has fixed number of neighbors, called grid graphs. During graph traversal, single direction is traversed at a time. Such traversal guarantees that particular vertex is traversed from a single vertex only. Also, number of nodes traversed in certain depth level is nearly same as that in its preceding depth level. Mohamed Hussein et al. [16] have implemented this method, called Lockstep BFS for grid graphs. They concluded when traversing nodes at depth level k from nodes at depth level k-1, applying the lockstep BFS traversal technique allows us to use an array of flags whose size is equal to the number of nodes in level k-1, which is much smaller than V, total number of vertices in the graph.

7. PARALLEL SINGLE SOURCE SHORTEST PATH

The single source shortest path (SSSP) problem is, given weighted graph G(V, E, W) with all weights positive, find the smallest combined weights of the edges between other vertices in the graph and given source vertex. Dijkstra's sequential algorithm requires computation time of O (VlogV+E).

Parallel implementations of SSSP using CUDA are carried out by some researchers. P. Harish et. al. [11] and Vibhav et. al [7] reports good speed up over CPU counterpart of SSSP. Unlike BFS, level synchronization is not possible in case of SSSP due to the reason that cost may change later on discovering less weighted path between the vertices. Vibhav et al. [7] demonstrate multithreading to great extent. They maintains two arrays, one Boolean array as Execution mask M_a which contains the vertices currently getting processed and cost array C_a which holds the smallest weight of the path between source vertex and other vertices. The vertices present in M_a updates the cost of all its neighbors. Here two costs are compared, one of which is the current cost of the neighbor and other is cost of current plus the weight between the current and neighbor vertices. The minimum of these two is appended to cost array C_a. The whole procedure ends when execution mask gets empty or no further changes in cost array.

During this process, simultaneous updating may take place at same location in C_a. This problem is resolved by maintaining the alternate array C_{ua} and using atomic functions provided by CUDA. These functions are used to resolve the concurrent writes by allowing only one thread to write at certain place at a time.

P. Harish et. al. [11] consider the graphs with degree per vertex 6-7 and weights ranging from 1-10 and report speed up of 70 times over CPU version of SSSP. Scale free graphs have large degree at some vertices, which results in more lookups to the device memory and hence computation time gets increased. Due to this, like BFS, SSSP also shows lower performance for scale free graphs than random graphs. Vibhav et al [7] experimented on bigger graphs with average degree 12 and weights up to 100 on GTX 280 GPU. As compared to boost sequential implementation of SSSP, they gain speed up of 20. R-MAT graphs perform badly as compared to other types of graphs. Vertex compaction process is used here also and observed 40% improvement in computation time than normal parallel version in case of R-MAT graphs.

8. PARALLEL ALL PAIRS SHORTEST PATH

In all pairs shortest path problem (APSP), given an weighted graph G(V, E, W) with positive weights, aim is to find out least weighted path from every vertex to every other vertex. Floyd-Warshall's, the well known APSP algorithm requires $O(V^3)$ computing time and $O(V^2)$ space. Due to this large space requirement, it is not feasible to handle the large graphs with millions of vertex on GPU as memory size restrictions.

8.1 APSP using SSSP

This approach suggests implementing APSP by running Single source shortest path (SSSP) for every vertex. P. Harish et al. [11] have carried out this approach. They reported that SSSP requires O (V) space. They considered large graphs with average degree of vertex as 6-7, maximum degree as thousand and average weights 1-10 in magnitude. This SSSP implementation carried out on GPU NVIDIA Geforce 8800 GTX is 70 times faster than its CPU counterpart. Also, for random graphs SSSP timings are comparable to those for BFS. They conclude that FW algorithm on GPU requires single O(V) operation looping over $O(V^2)$ threads which creates extra overhead for context switching the threads whereas their approach of APSP using SSSP requires only O(V) threads.

8.2 Tiled FW algorithm

The problem of restrictions on the graph size due to available memory is solved by Katz et al. [12]. Their approach handles graph size larger than on-board memory available to the GPU by breaking the graphs in nontrivial on-chip shared memory cache efficient manner to increase performance and is shared memory cache efficient. They implemented blocked (tiled) formulation of the algorithm. The basic idea is to revise original FW algorithm into a hierarchically parallel method that can be distributed, in parallel, across multiple processors on the GPU and further on multiple GPUs. Matrix is partitioned in sub blocks of equal size and processed. This technique provides 60-130X speedup over a standard CPU solution $O(V^3)$. The implementation of this method on NVIDIA QUADRO FX 5600 is 5-6.5 times faster than previous approach by P. Harish et al. [11].

8.3 Matrix multiplication method

Vibhav et al [7] have used adjacency matrix for cache efficient graph representation. The two methods are designed and implemented for APSP and they reported better results than any of the previous work for the algorithm. Like Katz et al. sub matrices of a matrix are processed but in different ways. This technique provides two new ideas as follows:

- Streaming blocks: It is assumed that CPU memory is large enough for storing large graphs. Adjacency matrix present in the CPU memory is divided into the rectangular row and column sub matrices. These are streamed into the host global memory.
- Lazy minimum evaluation: For sparse graphs the connections are few and the other entries of the adjacency matrix are infinity. For the entry infinity, all operations on that can be skipped without missing correctness and hence skipping all paths involving a non-existent edge. It is reported that this method results in speed up of 2 to 3 times.

As compared to Katz et al. this matrix based method proves to be 2-4 times faster for larger graphs.

8.4 Gaussian Elimination based method

The Gaussian elimination based APSP by Buluc et al [13] based on idea of splitting each APSP step recursively into 2 APSPs involving graphs of half the size. The base case is when there are 16 or fewer vertices, Floyd’s algorithm is applied. It is fastest among the all approaches for APSP. However, introducing the Lazy minimum evaluation to that approach provides further speed up of 2-3 times, according to Vibhav et. Al [7].

9. CONCLUSION AND FUTURE WORK

In the paper we presented overview of the graph algorithms like BFS, APSP those are implemented on GPU using CUDA in parallel. For storing the input graph and the results of the algorithms, it is very important to use efficient data structure. As compared to the CPU implementation of such graphs, GPU implementation achieves very great speed up. It is very important how programmers make optimum use of multithreading that can be possible on CUDA device.

We think it is also useful to print the subsequent vertices those found in the shortest path for any algorithm. Also, sometimes there is need of first n shortest paths in the graph i. e. first shortest, second shortest etc. In future, we will be working on these two requirements in graph algorithms.

10. REFERENCES

[1] The Ninth DIMACS implementation challenge on shortest pathshttp://www.dis.uniroma1.it/ challenge9/

[2] NVIDIA. NVIDIA CUDA Programming Guide 2.0

[3] http://www.nvidia.com/object/cudadevelop.html/www.gpgpu.org

[4] S. Nagendran, M. Ashrafulla, J. Bagga, 2008 “Assessment of SIMD programming on graphics card”.

[5] J. Hyvonen, J. Saramaki, and K. Kaski, 2008 Efficient data structures for sparse network representation. Int. J. Comput. Math., 85(8):1219-1233.

[6] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, 2006 Memory model for scientific algorithms on graphics processors. In SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 89.

[7] Vibhav Vineet and P. J. Narayanan, 2009 “Large graph algorithms for massively multithreaded architecture”

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, 2004 R-MAT: A recursive model for graph mining. In In SIAM International Conference on Data Mining.

[9] D. A. Bader and K. Madduri, 2006 Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA- 2. In ICPP, pages 523-530

[10] Paulius Micikevicius, 2004 General parallel computation on commodity graphics hardware: Case study with the all pairs shortest paths problem. In PDPTA, pages 1359–1365.

[11] P. Harish and P. J. Narayanan, 2007 Accelerating Large Graph Algorithms on the GPU Using CUDA. In HiPC, volume 4873 of Lecture Notes in Computer Science, pages 197-208,

[12] G. J. Katz and J. T. Kider, Jr. , 2008 All pairs shortest-paths for large graphs on the GPU. In GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pages 47-55.

[13] A. Buluc, J. R. Gilbert, and C. Budak, 2008 Gaussian Elimination Based Algorithms on the GPU. Technical report, November.

[14] D. A. Bader and K. Madduri, 2006 GTgraph: A Synthetic Graph Generator Suite. Technical report.

[15] P. J. Narayanan, 1993 Processor Autonomy on SIMD Architectures. In International Conference on Supercomputing, pages 127-136.

[16] M. Hussein, A. Varshney, and L. Davis, 2007 On Implementing Graph Cuts on CUDA. In First Workshop on General Purpose Processing on Graphics Processing Units.