Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 12, December 2015, pg.103 – 106

Review Paper on Optimised and Accelerated Parallel Graph Algorithm on GPGPU

C.P. Mogal[1], Prof. C.R. Barde[2]

[1]PG Student, Dept of Computer Engg, R.H. Sapat College of Engineering, Nashik, India

[2]Assistant Professor, Dept of Computer Engg, R.H. Sapat College of Engineering, Nashik, India

[1] mogalcp.31@gmail.com; [2] erchandubarde@gmail.com

Abstract— Graph algorithms FORM fundamental to many disciplines and are common in scientific and engineering applications. The need for high computation power and low price results into areas such as Graphics processing units (GPU). It largely consists of Single Instruction Multiple Data. GPU's for best suited for regular data parallel algorithms. This paper deals with how to use GP-GPUs efficiently for graph algorithms and efficient GPU implementations for the various problems thus faced. The algorithms focus on minimising irregularity at both algorithmic and implementation level. It also deals with analysis of all pair shortest path algorithm by performing on different memories of GPU.

Keywords— Graph, Algorithm, GP-GPU, All Pair Shortest Path, Single Instruction Multiple Data

I. INTRODUCTION

Today, hundreds of applications are GPU-accelerated and the number is growing. Many practical domains like bioinformatics, computational chemistry, medical imaging, machine learning, Defence, space searching, network analysis, film making and animation etc. uses large graphs having a millions of vertices are difficult to process. Using high-end computers, practical-time implementations are reported but these supercomputers have limited access. Efficient performance of those applications requires fast implementation of graph processing. Hence Graphics Processing Units (GPUs) of today having a high computational power of accelerating capacity are deployed. The NVIDIA GPU can be worked as a SIMD processor array with the CUDA programming model. In this system, graph algorithms such as Breadth-First Search, All Pair shortest path and traveling salesmen problem are performed on GPU capabilities and hence improve performance of graph processing for better efficiency these applications.

All Pair shortest path (APSP) algorithm can be stated as, given weight graph and in that find smallest path between each pair of vertices. Travelling Salesmen Problem (TSP) stated as, given graph G and source S, find the minimum cost tour length where all vertices are visited at least once. These graph algorithms are used in many practical applications as a fundamental tool. BFS algorithm can be used in data mining for fast data retrieving and TSP algorithm can be used in GPS system. Literature survey reveals that there are many approaches to accelerate graph algorithms. Fast implementation of sequential graph algorithm [1] exists but algorithm become impractical for very large graph. Graph operations performed by using parallel algorithm which executed in a practical time, but required high hardware cost. Bader et. Al [2] perform graph algorithm by using CRAY supercomputer. Though this method is fast, the hardware used in them is very expensive.

Now a day's adoption of GPGPU (General-Purpose computation on Graphics Processing Unit) is increased in many applications [3]. To accelerate various graphs processing GPGPU has been used. While those, GPU-based solution give a significant performance improvement over CPU-based implementation.

In this paper we proposed the graph algorithms which are Breadth-First Search and Single Source Shortest Path through use of GPU. Efficient approaches are proposed for

1) Reducing data transfer rate CPU to GPU
2) Reducing access of global memory by using shared memory.
3) Mapping of neighbouring solutions to threads of GPU using thread control function
4) Memory management

II. LITERATURE REVIEW

A. Medusa Framework for Graph Processing

Medusa greatly simplifies implementation of GPGPU programs for graph processing, with many fewer lines of source code written by developers. The optimization techniques significantly improve the performance of the runtime system, making its performance comparable with or better than manually tuned GPU graph operations. Framework simplifies programming graph processing algorithms on the GPU. That framework provides sequential interference to developer. It provides six device code APIs for developer to write GPU graph processing algorithm. [5]

Medusa hides a GPU specific programming details with a small set of system provided APIs. Medusa front end automatically transforms definition device code APIs and user defined data structure into compatible CUDA kernels. Medusa storage component allow developers to initialize the graph structure through the use of system APIs like Add Edge and Add Vertex then Medusa runtime component which is responsible for executing the user-defined APIs in parallel on GPU. Medusa BFS method, the Medusa applies L threads to vertex has L edges. This results, Medusa incurs more memory accesses. Their experimental result shows that on large diameter graph the performance of Medusa-based algorithm is reduced.

B. Pawan Harish Proposed algorithms

Pawan Harish [4] presents 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

C. Various Algorithm and Optimization Techniques

SR NO	TITLE	TECHNIQUE	NODES	OPTIMIZATION /NOT	CUDA/OPENCL
I	GPu Computing for parallel LSM Algorithm	Effective LSM Technique	1300	Optimization used for data transfer thread control	Cuda & Open cl
II	Medusa Graph Algorithms	Sequential Interface framework which convert serial to parallel code	1000	Fine grain API	CUDA
III	High performance gpu accelerating local optimization in TSP	Two option search algorithm	3038	Optimization in access patters	CUDA/OPENCL

IV	Accelerating large graph algorithm on gpu using cuda	Thread level parallelism	250K	No Optimisation	CUDA
V	Accelerating cuda graph algorithm maximum wrap	Novel virtual wrap centric approach	107631	Optimization as dynamic workload distribution	CUDA

TABLE I
LITERATURE SURVEY ON LEVEL OF OPTIMIZATION AND TECHNIQUE USED

III.HARD WARE AND SOFTWARE SPECIFICATIONS

A. Languages: CUDA

B. Software Description Interface

A single machine with CUDA capable GPU, Ubuntu and NVIDIA CUDA Toolkit is required for running the application.

C. Operating Environment.

Development is done in CUDA. CUDA (Compute Unified Device Architecture) was first proposed by NVIDIA in 2007, offering versions for both the Windows and Linux operating systems, while version 2.0 contained support for MacOS X also. CUDA was designed to be of use in both professional and casual graphics cards, having similarities with the OpenCL framework and with Microsoft Direct Compute alternative.

The Programming Paradigm: CUDA SDK uses an extended C language that allows the user to program using the CUDA architecture. A user defined C function that is executed in the GPU is called a kernel. A set of parallel threads, which are organized into thread blocks and grids of thread blocks, execute the kernel concurrently .The programmer specifies the number of times the kernel has to be executed by specifying the number of threads in the program. Each thread executes one instance of the kernel. So, if the user specifies the number of threads as N, the kernel will be executed N times by N different threads.CUDA follows a Single Instruction Multiple Thread (SIMT) programming mode.

GPU Thread Architecture The massive parallelism in the CUDA programming model is achieved through its multi-threaded architecture. This thread parallelism allows the programmer to partition the problem into coarse sub problems that can be processed in parallel by blocks of threads, and each sub problem is further divided into finer pieces that can be solved cooperatively in parallel by all threads within a block. The CUDA threads are organized into a two-level hierarchy using unique coordinates called block JD and thread JD. Each of these threads can be independently identified within the kernel using its unique identifier represented by the built-in variable blockldx and threadJdx. The programmer can configure the number of threads required in a thread block, with a maximum of 1024 threads per block. An instance of the kernel is executed by each of these threads.

CUDA Thread Organization: A group of 32 threads with consecutive thread IDs is called a Warp, which is the unit of thread scheduling in SMs. The Kepler architecture supports 16 SMs each of which can track a total of 48 warps simultaneously.

CUDA Memory Organization a large portion of computing time on the device is spent on data movement, especially the reading of data into the individual threads. Since there are hundreds of arithmetic units on the GPU, the memory bandwidth of the computer chip is often the bottleneck or major time consumer. With four types of memory available on the GPU, picking the right memory type and utilizing it correctly is crucial for maximum speed. The four types of device memory are

- Global Memory: Global Memory By far the largest amount of memory on the device is read-and-write global memory: 500 megabytes. Although far slower than other memory types, it is relatively constraint free and the easiest to use.

 Shared Memory: The shared memory is read-and-write memory resides physically on the GPU. It is placed as opposed to off-chip DRAM, so it is much faster than global memory. Threads are one block that allows accessing shared memory. But threads in other block don't have access to share memory in different block. This type of memory provides an excellent speedup because threads in one block communicate with each other and use share memory. But synchronization is necessary between threads.

 Constant Memory: Constant memory is read-only memory that does not change over the course of kernel execution. Constant memory is cached consecutive reads of the same address.

 Texture Memory: Texture memory is global memory. The difference between global memory and texture memory is that, the texture memory is accessed through a dedicated read- only cache, and this cache performs filtering that carried out linear floating point operations during read process. The cache, however, is different to a normal cache, in that it is optimized for spatial locality and not locality in memory

IV. CONCLUSIONS

There is a huge room for improvement present work on Graph processing algorithm and optimizing them. Every time a new GPU card is released with improved computational features, the horizon further advances. System provides a parallel approach for graph algorithms that are Breadth First Search, All Pair Shortest Path. These applications can be ported to multiple GPU devices that will run in parallel. In this system, optimized BFS algorithms achieve parallelism through edges-centric approach and achieve better execution time. As the number of GPU cards used increases, a proportional speed up of the application is expected. The process is expected to produce enormous speed as all the costly computations can be offloaded to the GPU and Overlapping kernel execution & data transfer By using overlap kernel execution with data transfers we can increase speed of execution.

ACKNOWLEDGEMENT

The author wish to thank GES'S R.H.Sapat College of Engineering And Research Centre Nashik, HOD of Computer Department, Guide and All PG Staff for supporting and motivating for this work.

REFERENCES

[1] Jun-Dong Cho, Salil Raje, and Majid Sarrafzadeh, Fast approximation algorithms on maxcut, k-coloring, and k-color ordering for vlsi applications, IEEE Transactions on Computers, 47(11):12531266, 1998.
[2] David A. Bader and Kamesh Madduri, Designing multithreaded algorithms for breadth- first search and st-connectivity on MTA-2, In ICPP, pages 523530, 2006.
[3] J.D. Owens, D. Luebke, N.K. Govindaraju, M. Harris, J. Kruger, A survey of generalpurpose computation on graphics hardware, in Proc. Eurographics, State Art Rep., 2005, pp. 21-51.
[4] P. Harish and P.J. Narayanan, ''Accelerating Large Graph Algorithms on the GPU Using CUDA,'' in Proc. HiPC, 2007, pp. 197-208.
[5] J. Zhong and B. He. An overview of medusa: simplified graph processing on GPUS. In PPoPP, pages 283–284, 2012.
[6] L. Luo, M. Wong, and W.-M. Hwu," An Effective GPU Implementation of Breadth-First Search," in Proc. DAC, pp. 52-55, 2010.
[7] S. Hong, S.K. Kim, T. Oguntebi, and K. Olukotun," Accelerating CUDA Graph Algorithms at Maximum Warp," in Proc. PPoPP, pp. 267-276 2011.
[8] Jianlong Zhong and Bingsheng He,"Medusa: Simplified Graph Processing on GPUs".IEEE Transaction on parallel and distributed system, Vol. 25, NO. 6, JUNE 2014.
[9] CUDA Zone. Official webpage of the nvidia cuda api. Website, http://www.nvidia.com/object/cuda home.html.
[10] Vibhav Vineet and P. J. Narayanan,"Large graph algorithms for massively multithreaded architecture" in Proc. HiPC, 2009.
[11] BADER, D. A., AND CONG, G. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). Journal of Parallel and Distributed Computing 65, 9 (2005), 994–1006
[12] NVIDIA, C. Cuda: Compute unified device architecture programming guide. Technical report, NVIDIA, 2014
[13] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. Introduction to algorithms, 1990.
[14] VINEET, V., AND NARAYANAN, P. J. CUDA Cuts: Fast Graph Cuts on the GPU. In Proceedings of the CVPR Workshop on Visual Computer Vision on GPUs (2008).

