University of Peradeniya

Department of Computer Engineering

CO421: Final Year Project I

Accelerating EDA Algorithms Using GPU

Authors
Dinali Dabarera(E/11/064)
Erandika Harshani (E/11/145)

Supervisor
Dr. Roshan Ragel

May 26, 2016

Abstract

With the time, the complexity of electronic circuits increases day by day. Therefore, it is important to improve the performance of EDA Algorithms to fulfil the future requirements of the EDA tool industry. Out of many algorithms, Breadth First Search (BFS) is the most commonly used algorithm to traverse the nodes/gates of the electronic circuit. Nvidia has introduced latest libraries and functionalities which increase the performance of computations done on sparse matrices.

In this project, we use latest Nvidia functionalities to increase the performance of EDA algorithms including Breadth First Search (BFS), which has become a challenging task due to sparse matrices and other limitations.

Contents

List of Figures

List of Tables

1. Introduction

It is observed that, as stated in Moores law, for past few years the number of transistor in electronic circuits has approximately doubled in every two years[20]. Therefore, the complexity of electronic circuits increases day by day. This has become a great challenge for EDA industry. EDA which stands for Electronic Design Automation is a highly important and expensive industry in the world. There are four major steps in Electronic Design Automation. They are placement which is the most essential step, routing which is the most crucial step in Integrated Circuit (IC) designing, power optimization which is the main use of EDA tools to optimize (reduce) the power consumption of a digital design and post silicon validation which is the final step of EDA design flow. Today there are more than twenty EDA companies all around the world which fulfills all the needs of electronic design automation. Some of the famous companies are Synopsys, Cadence, Ansys and EasyEDA.

1.1. EDA Tools

EDA tool is software tool which is used for designing electronic circuits and systems such as printed circuit boards and integrated circuits. EDA tools are mainly used by chip designers and circuit designers to design and analyze entire semiconductor chips. These EDAs consists of block diagram editors, embedded software development IDEs, system level designing tools, etc.

1.2. EDA Algorithms

Many EDA problems are intrinsically difficult. For such problems, certain heuristic algorithms can be applied to find an acceptable solution first. But since the data of EDA tools are in the form of graphs, many graph algorithms are also used in EDA tools. According to [12], following are some of the heuristic (algorithms that use common sense to solve problems) and graph algorithms which are used in EDA tools.

Heuristic algorithms:
1. Greedy algorithms - eg. Traveling salesman problem.
2. Dynamic programming [10].
3. Branch and bound - A general technique for improving the searching process.

Graph algorithms:
1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. Topological Sort
4. Shortest and longest path algorithms (Dijkstras algorithm, Bellman-Ford algorithm)
5. Minimum spanning tree
6. Maximum flow and minimum cut (The Ford-Fulkerson method and the Edmonds-Karp algorithm)

1.3. Nvidia GPU

There are many methods of increasing performance of EDA algorithms. Some of them are use of cloud computing, use of server farms and use of high performance computers. These methods are really expensive and need a lot of hardware. Therefore, use of GPU, the Graphical Processing Unit is the cheapest method of improving performance of EDA tools, which means even a simple circuit designer can use his laptops graphic card to improve the designing process.

Modern GPUs are more efficient at manipulating computer graphics, image processing and parallel computations. The parallel structure of these GPUs makes them more effective than general CPU. At present GPU can be present in a personal computer as a video card or embedded into its motherboard or on the CPU die. The word GPU was popularized by Nvidia. With the introduction of GeForce 8 Series by Nvidia, GPUs have become the main mode of computations against CPU. It also has become a field of research called General Purpose Computing on GPU which is called GPGPU. Some of the fields are machine learning, big data mining, image processing, linear algebra, and statistics.

The latest release of Nvidia, Kepler has the worlds fastest most efficient HPC architecture which helps in high performance computing. There are many latest features that have added to the Kepler architecture. Following are some of the features in Kepler architecture:

Innovative streaming multiprocessor design which allows greater percentage of space to be applied to processing cores than control logic.

1. Dynamic parallelism which allows accelerating all nested loops by dynamically spawning new threads in GPU on its own without going back to CPU.
2. Hyper Q which slashes CPU idle time by allowing multiple CPU cores to simultaneously utilize a single Kepler GPU.

For further details, please refer Nvidia official website [22].

1.4. The Project and Its Objectives

The research works on GPU-accelerated EDA are done at three different design stages. They are system level, RTL level and gate level. In typical IC designing process simulation based verification has already becomes the bottleneck. Until now there has been dedicated considerable amount of research efforts to accelerate various EDA tools with GPUs [12].

The most common form of data structure that satisfy EDA algorithms is the graph data structure. When the complexity of digital circuit increases, the number of nodes of the graph data structure also increases from millions to billions. But the number of connections are much more less than the number of nodes. Sparse matrices are mainly used to store this kind of graph data structure.

Handling sparse matrices was really challenging on GPU. Sparse matrices contains lot of zero elements than non-zero elements. Therefore, it causes problems when copying data from CPU to GPU. Because it wastes lot of memory. But at present this problem has been handled by GPU with the use of shared memory and different formats of representations of sparse matrices. The CuSparse library which has specially created by Nvidia for handling sparse matrices in GPU. Although these problems were solved, still EDA tools performance has only increased up to 30 times[25]. Therefore, in order to handle the future requirements of performance in EDA tools should be increased.

Through this project our main objective is to utilize the latest features of Nvidia GPU and optimize the EDA algorithms such as BFS and DFS, in order to achieve greater performance than the state of the art.

2. Background

2.1. Related work

One of the main GPU based logic simulation provider Rocketick has produced a product called RocketSim in 2011 [25]. It is a software based solution which is installed on standard servers and accelerates leading simulators such as incisive, VCS and ModelSim and rapid compilation. RocketSim also solves the simulators bottleneck problem by offloading most time-consuming calculations to an ultra-fast GPU based engine. Other than a hardware based accelerators, RocketSim runs alongside the existing test bench, eliminating ramp-up time while providing precise results. It also supports very large complex designs that include more than billion logic gates. RocketSim solves functional verification bottlenecks and has achieved 10X faster verilog simulations for highly complex designs.

Yangdong et al. Had done a research on Taming irregular EDA applications on GPU [28], that describes high performance GPU implementations of two important irregular EDA computing patterns such as graph traversal and Sparse Matrix Vector Product (SMVP). In this, they had considerably accelerated a SMVP based formulation of Breadth-First Search (BFS) using CUDA to get a speedup up to 10X and presented the experimental results. Finally they had implemented a graph traversal problem based on SMVP procedure.

From some of the surveys on Electronic Design Automation with Graphic Processors [9] done by YangDong et al. Gives a detailed description about GPU architecture with its programming model and the essential design patterns for EDA computing. It shows useful information on use of some GPU libraries such as CUBLAS [21], MAGMA[23] and CULA [23]. It also gives a brief description on some of the algorithms such as Breadth-First Search, Shortest Path, Minimum Spanning Tree, Map Reduce and Dynamic Programing.

A Breadth-First Search parallelization focused on fine grain task management is described on a research done by Duane et al. In 2011 [18]. It has achieved an asymptotically optimal O(|V |+ |E |) work complexity. Nathan et al. Had studied on Efficient Sparse Matrix Vector Multiplication [1] in 2008. This has clearly described the data structures and algorithms for Sparse Matrix Vector Multiplication that are efficiently implemented on the platform for the fine-grained parallel architecture of the GPU.

Gunjan et al. Have described a new approach for graph algorithms on GPU using CUDA [26] in 2013. This gives a detailed parallel implementations of two basic graph algorithms such as Breadth-First Search and Dijkstras single source shortest path algorithm by using a new approach called edge base kernel execution on GPU and the performance analysis of the implementation on different types of graphs. It also gives a brief description on CUDA basics along with GPU architecture.

John et al. From cadence design systems did a research on Introduction to GPU programming for EDA[5] in 2009 which clearly states the GPU architecture and the tools available to utilize this valuable resource. This mainly discuss two main GPU programming languages such as CUDA and OpenCL than can be used to harness the power of GPU and the applicability of the GPU to EDA-specific problems.

Sangamithra et al. Have proposed a novel floor planning algorithm based on simulated annealing on GPU in [13]. Simulated annealing (SA) has been the most commonly used method for the fixed-outline floor planning problem. Compute intensive algorithms like fault simulation, power grid simulation and event-driven logic simulation have been successfully mapped to GPU platforms to obtain significant speedups. Compared to the sequential algorithms, these proposed techniques achieved 6-16X speedup for a range of MCNC and GSRC benchmarks with a better accurate solution.

Michael et al. Have implemented an efficient Sparse Matrix Vector Multiplication on GPU in [1]. They have discussed data structures, sparse matrix formats such as diagonal format, ELLPACK format, Compressed Sparse Row Format, Coordinate format, packet format and Hybrid format, and algorithms for Sparse matrix vector multiplication that can be efficiently implemented for the fine-grained architecture of the GPU .

Wong et al. Have found out an effective implementation of Breadth-First Search on GPU in [17]. This has used a hierarchical queue management technique and a three-layer kernel arrangement strategy. The experimental results have achieved up to 10 times speedup over the classical fast CPU implementation. This method is mostly suitable for accelerating sparse and near-regular graphs which are widely seen in the field of EDA.

Narayan et al. Have done a research on accelerating large graph algorithms on GPU in [14]. This also gives an overview of general purpose programming (GPGPU) and fundamental algorithms using GPU programming model on large graphs. This also provides fast solutions for Breadth-First Search, Single Source Shortest Path and All-Pairs Shortest Path on very large graphs at very high speeds using a GPU instead of expensive supercomputers.

Federico et al. Have found out an efficient implementation of Bellman-Ford algorithms for Kepler GPU architectures in [3]. They summarize the basic concepts on CUDA, Kepler, Maxwell, GPUs and Bellman-Fords algorithms. They also state that some important points to optimize many graph characteristics in order to give a clear overview on how they impact on the H-BF work efficiency.

Chethan et al. Have presented a clear picture on parallelization of graphing algorithms using CUDA in [24]. They have developed various types of serial and parallel implementations and compared the performances of each algorithms on both GPU and CPU. They mainly talk about Breadth-First Search and prove that it is better to implement on GPU rather than on CPU in order to get a better performance.

Nathan et al. Had done another research work on Implementing Sparse Matrix Vector Multiplication on Throughput oriented processors in [2]. They also have addressed different types of sparse matrices and verified their specific uses. According to their findings vector approach of sparse matrices ensures contiguous memory access but lead to waste of time due to lack of non-zero elements and DIA and ELL techniques are well suited to matrices obtained from structured grids. They also conclude that in order to effectively utilize the GPU resources, the kernels needed to have a fine-grain parallelism.

Al-Kawam et al. Had done their work on GPU implementation of Timberwolf Placement Algorithm in [16]. This gives further details on full implementation of Timber Wolf algorithms on GPU and demonstrate how GPU is used to accelerate the performance of EDA tools. These algorithms have been implemented using C language, on a Xeon workstation. Through harnessing the power of GPUs, they have achieved a higher degree of performance.

Busato et al. Had implemented an efficient algorithms for Breadth First Search using Kepler GPU architectures in [4]. Their implementation of BFS has achieved an asymptotically optimum work complexity. They also propose few most efficient implementations of Breadth First Search in their research work.

Furthermore, Table 2.1 gives a brief summary of graph algorithms described in research papers which are mentioned above. It clearly shows that many research papers mentioned Breadth-First Search more frequently than any other graph algorithms.

Moreover, a short summary of Breadth-First Search implementations that were done in previous research works were shown in the Table 2.2. This clearly show that Breadth-First Search is one of the most common graph traversal algorithms and the main building block for a wide range of graph applications.

BFS Algorithm	**SMVP Algorithm**	**Bellman-Ford Algorithm**	**Floor Planing Algorithms**	**All Pair Shortest Path Algorithm**	**Single Source Shortest Path Algorithm**
High Performance and Sacalable GPU Graph Traversal. [18]	Taming Irregular EDA Applications on GPUs.[8]	An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures.[3]	Optimizing Simulated Annealing on GPU: A Case Study with IC Floor planning.[13]	Accelerating large graph algorithms on the GPU using CUDA.[14]	Accelerating large graph algorithms on the GPU using CUDA.[14]
Taming Irregular EDA Applications on GPUs.[8]	New Approach for Graph Algorithms on GPU using CUDA.[26]				
An Effective GPU Implementation of Breadth-First Search.[17]	Efficient Sparse Matrix-Vector Multiplication on CUDA.[1]				
New Approach for Graph Algorithms on GPU using CUDA.[26]	Optimizing Simulated Annealing on GPU: A Case Study with IC Floorplanning.[13]				
Accelerating large graph algorithms on the GPU using CUDA.[14]					
BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures.[4]					

Table 2.1.: Summary of Research papers.

Topics	Abstract	BFS Implementation Method	Results
High Performance and Scalable GPU Graph Traversal.[18]	Parallelization focused on task management.	Using fine grained, bulk asynchronous parallelism	A asymptotic optimal O(|V |+ |E |) work complexity and high performance on real world graphs.
Taming Irregular EDA Applications on GPUs. [8]	High performance GPU implementation of graph traversal using Sparse Matrix vector Product.	Using a new formulation of BFS with Sparse Matrix Vector Product($A^T * x$)	Speed up over 10X
An Effective GPU Implementation of Breadth-First Search. [17]	A new GPU implementation of BFS using proper managements of queues and kernels.	Using hierarchical queue management techniques and three layer kernel arrangement strategies	Speed up of 10X
New Approach for Graph Algorithms on GPU using CUDA. [26]	A parallel implementation of graph operations such as Dijkstra's and BFS algorithm on various types of graphs	Using edge based kernel execution on GPU	Greater degree of parallelism achieved when mapping threads with edges rather than nodes
BFS-4K [4]	A comparison between the efficient BFS implementations on GPU.	Using advanced features of NVIDIA GPU Kepler architecture.	An asymptotic optimal work complexity

Table 2.2.: Summary of Breadth-First Search Implementation Researches.

3. Breadth-First Search

Breadth-First Search is an algorithms that can be used to traverse a graph data structure by starting from the root node and visiting the neighboring nodes first before moving to the next level neighbors. According to [18] Breadth-First Search is the main graph traversal algorithm which has become the basis for many higher level analysis graph algorithms. This BFS can be implemented in a recursive way as well as non-recursive way. There were more than five research papers and one journal which explained BFS or Breadth First Search and how it can be used to optimize the performance of EDA algorithms.

EDA tools mainly contain lot of graph algorithms as they are working with electronic circuits. In these tools, according to [28], BFS is mainly used to fulfill two different kinds of applications. They are levelized logic simulation and finding critical path in block based timing analysis. The circuits which are used by EDA tools can be easily interpreted as graphs, and these graphs can be stored as sparse matrices. There are about six ways of representing a sparse matrix. They are Compressed Sparse Row Format, Compressed Sparse Column Format, Coordinate Format, Diagonal Format, ELLPACK format and Packet format. These formats are mainly used to save extra memory allocation for zero elements.

Figure 3.1 clearly shows how a simple gate circuit can be shown as directed timing graph and how a directed graph can be shown as a sparse matrix in the form of Compressed Row format. In this figure A(i,j) = 1 if and only if I is directed to j and also A(I,j) can have a non-zero element to represent a connection from cell I to j. In timing graph analysis every pin is treated as a node on the graph. When comparing the Figure 3.1 c and d, it is clear that the memory that is needed to store the graph data is less in d than c. Therefore without much difficulty the memory that used for storage can be saved.

These graphs also can be stored as adjacency list of edges in order to save storage memory in the main memory as shown in the Figure 3.2. But when dealing with GPU, for past few years before the arrival of Kepler architecture, people tend to use sparse matrix data formats in order to store these graphs than adjacency lists or Linked List because copying data from CPU to GPU in the form of linked Lists cause illegal memory address accessing problems. Today the Kepler architecture of modern GPU has many new features such as unified memory and dynamic parallelism which helps to use Linked list data structures in graph traversal algorithms without much difficulty.

Furthermore, through out next few chapters we discuss, how Breadth-First Search can be implemented using Sparse Matrix Vector Multiplications with the help of Csparse, C

Figure 3.1.: Sparse Matrix and Compressed Row Format, a) Simple circuit, b) Directed graph corresponds to a, c) Sparse matrix of b, d) Compressed Row Format of c (Taken from [8])

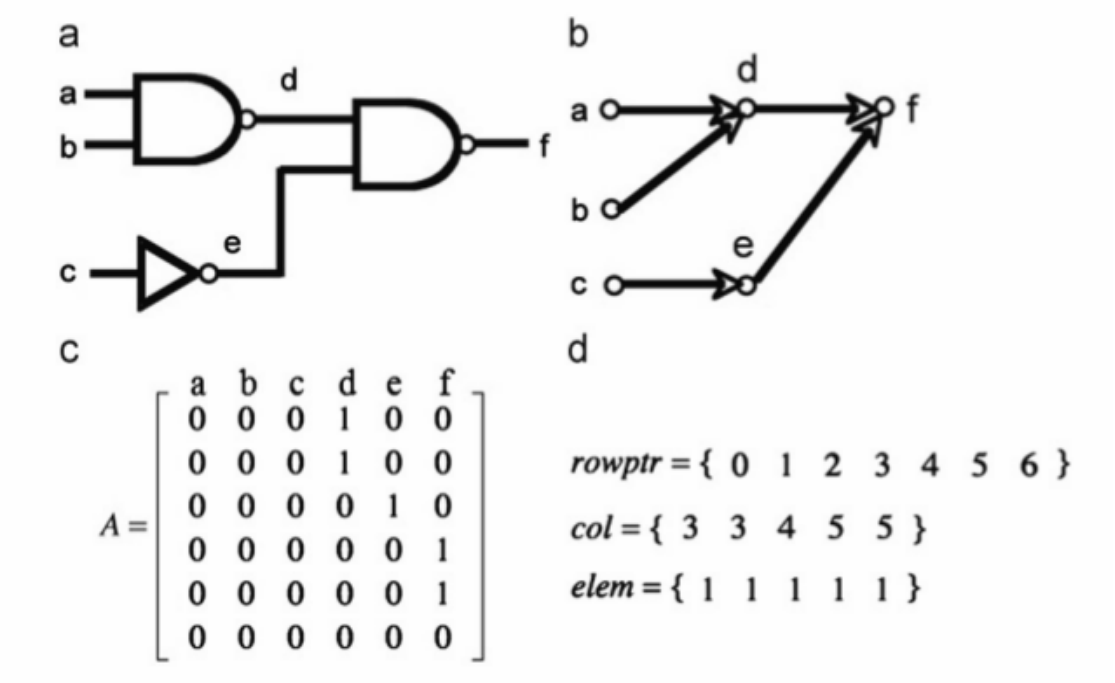
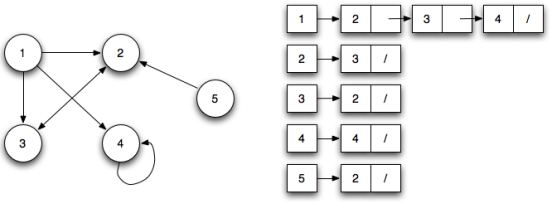
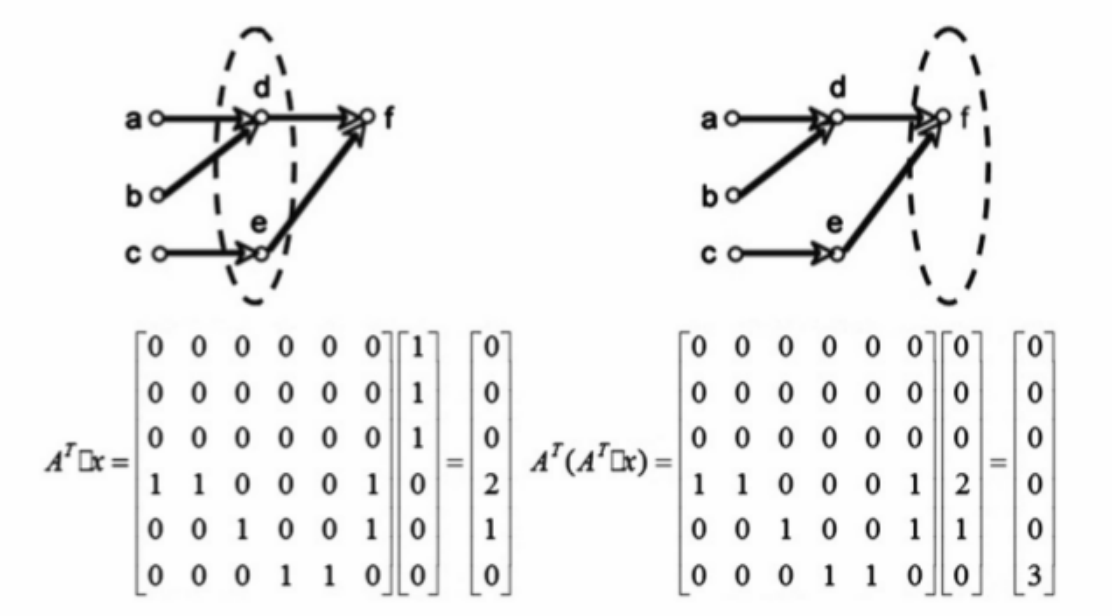


Figure 3.2.: Adjacency List representation of a graph



library and CuSparse, CUDA library and using Linked List data structure with the use of Dynamic parallelism and unified memory features of Kepler GPU architecture. How Dynamic parallelism will helps us in Adjacency List Implementation is discussed later in our future works.

Figure 3.3.: BFS using Sparse Matrix Vector Multiplication (Taken from [8])



3.1. Breadth-First Search using Sparse Matrix Vector Multiplication(SMVP-BFS)

In the Figure 3.3, it clearly shows how the Breadth First Search is used to traverse through the circuit shown above in the Figure 3.1. A^T is the transpose matrix of A and x is the input vector to the circuit. This input vector x(i, 0) can have any non-zero element for input pins and in this condition all three inputs were considered to be 1. Each time when A^T is multiplied with the vector of the current state, the vector of the next state can be taken as shown above. Therefore through this method, Breadth-First Search can be easily achieved. If the number of intermediate levels of the graph is n, the Breadth-First Search can be expressed as here $A^{'} = A^T$.

Sparse matrices are common in scientific applications and they contain more zero elements than non-zero elements. It is critical to store only non-zero elements in order to save space and running time. A most common operation on sparse matrices is to multiply them by a dense vector which was used by this Breadth-First Search as the main operation. The output was given by the dot product of each sparse row with the dense vector. If n is the number of non-zero elements in the row, the depth of the computation will be $O(logn)$ which is the depth of the sum.

In this project, we have implemented the above Breadth-First Search using two special sparse matrix libraries called Csparse and CuSparse separately in both C and CUDA.

4. CPU Implementation of SMVP-BFS

4.1. Csparse

Csparse is basically a C library, which has been developed by Thimothy Davis in order to compute the primary mathematical computations on sparse matrices. The main format of sparse matrix used in this library is compressed column format of sparse matrices. Csparse is free software that is licensed under GNU Lesser General Public License and achieves few main goals of the developer. The source code of Csparse with some demo codes can be found here in [6], in order to get a better understanding on Csparse.

4.2. SMVP-BFS Implementation in Csparse

CSparse library has all the most common linear algebra computations of sparse matrices. Out of them, we have used matrix multiplication and transpose in our BFS implementation. Furthermore, we have used the same method that described above in chapter 3 to implement the Breadth-First Search using CSparse library.

$T = cs\_load(file);$ $\quad H = cs\_load(fp);$
$A = cs\_triplet(T);$ $\quad cs\_spfree(T);$
$G = cs\_triplet(H);$ $\quad cs\_spfree(H);$
$AT = cs\_transpose(A, 1);$
$C = cs\_multiply(AT, G);$
$N = C;$
$for(i = 0; i < b; i++)\{$
$\quad N = cs\_multiply(AT, N);$
$\}$

Algorithm 1: BFS using CSparse

The sparse matrix of the electronic circuit graph and the vector x is inputted as two files, with three columns of data which represents the x coordinate, y coordinate and the value of the non-zero element ass shown in the Algorithm 1. This two files are loaded in to a matrix and a vector using the CSparse library. This loaded matrix and the vector were converted in to compressed column format and the matrix A was transposed to AT. Finally the Breadth-First Search traversal stages were taken by getting the multiplication of AT and the vector G as shown in the above algorithm.

The code relevant to this whole Breadth-First Search process is attached as APPENDIX B.1 in Appendices.

5. GPU Implementation of SMVP-BFS

5.1. CuSparse

CuSparse is a CUDA library that includes basic linear algebra that handles sparse matrix type data structures in different forms. This is mainly developed on top of NVIDIA CUDA and available in the form of C and C++. All the functions have categorized under 4 sections such as level1, level2, level3 and conversion. While assuming all input data to be on GPU memory, CuSparse does all the computations inside the device (GPU) memory. CuSparse especially requires hardware compatibility of at least 2.0 or higher. Further details on syntax and samples can be taken from NVIDIA CUDA toolkit Documentation available in [7].

5.2. SMVP-BFS Implementation in CuSparse

As mentioned above in Chapter 3, Breadth-First Search was implemented using CuSparse library of NVIDIA, in order to traverse all intermediate states of an electronic circuit. According to NVIDIA CUDA toolkit documentation for CuSparse, all the linear algebraic operations such as matrix and vector multiplication and addition and conversions of sparse matrix formats has implemented in the library itself.

As we use the same input data used in CSparse implementation, we had to convert into compressed row format and need to do some configurations before used in Breadth-First Search as the shown by Algorithms 2.

$cusparseOperation\_ttrans =$
$CUSPARSE\_OPERATION\_NON\_TRANSPOSE$;
$cusparseIndexBase\_tidxBase = CUSPARSE\_INDEX\_BASE\_ZERO$;
$cusparseAction\_tcopyValues = CUSPARSE\_ACTION\_NUMERIC$;
$cusparseHand\_thandle$;
$cusparseCreate(\&handle)$;
$cusparseMatDescr\_tdescrA$;
$cusparseCreateMatDescr(\&descrA)$;
$cusparseStatus\_tcusparseXcoo2csr(cusparseHandle\_thandle, constint *$
$cooRowInd, intnnz, intm, int * csrRowPtr, cusparseIndexBase\_tidxBase)$;
Algorithm 2: Conversion of Coordinate format to Compressed Row format

This CuSparse implementation of Breadth-First Search was done in two methods with the help of new features of NVIDIA GPU such as unified memory and dynamic parallelism.

5.2.1. Non-Unified Memory Implementation of SMVP-BFS

In this Non-Unified Memory implementation method the program looks like a general GPU, CUDA program.The memory for matrices which taken as co-ordinate format and compressed row format was allocated in the CPU first and then copied to the GPU memory. After doing the transposing and compressed column format matrix vector multiplication in several steps, final result was copied back from GPU to CPU.

$cudaMalloc((void**)\&cooxIndexCuda, nnzsizeof(int));$
$cudaMalloc((void**)\&cooyIndexCuda, nnzsizeof(int));$
$cudaMalloc((void**)\&cooValCuda, nnzsizeof(double));$
$cudaMalloc((void**)\&csrRowPtrACudaPtr, nsizeof(int));$
$cudaMalloc((void**)\&cscColIndexACuda, nnzsizeof(int));$
$cudaMalloc((void**)\&cscValACuda, nnzsizeof(double));$
$cudaMalloc((void**)\&cscRowPtrACudaPtr, nsizeof(int));$
$cudaMalloc((void**)\&xCuda, nsizeof(double));$
$cudaMalloc((void**)\&yCuda, nsizeof(double));$
$cudaMemcpy(cooxIndexCuda, cooxIndexHostPtr, sizeof(int) \times nnz, cudaMemcpyHostToDevice);$
$cudaMemcpy(cooyIndexCuda, cooyIndexHostPtr, sizeof(int) \times nnz, cudaMemcpyHostToDevice);$
$cudaMemcpy(cooValCuda, cooValHostPtr, sizeof(double) \times nnz, cudaMemcpyHostToDevice);$
$cudaMemcpy(xCuda, xHost, sizeof(double)n, cudaMemcpyHostToDevice);$

$cusparseStatus\_tcusparseDcsr2csc(cusparseHandle\_thandle, intm, intn, intnnz, constdouble* csrVal, constint * csrRowPtr, constint * csrColInd, double * cscVal, int * cscRowInd, int * cscColPtr, cusparseAction\_tcopyValues, cusparseIndexBase\_tidxBase);$
for *(i=0;i<NumTimes;i++)* **do**
  $cusparseStatus\_tcusparseDcsrmv(cusparseHandle\_thandle, cusparseOperation\_ttransA, intm, intn, in$
  $alpha, constcusparseMatDescr\_tdescrA, constdouble * csrValA, constint * csrRowPtrA, constint * csrColIndA, constdouble * x, constdouble * beta, double * y)$
end
$cudaMemcpy(yHost, xCuda, sizeof(double)n, cudaMemcpyDeviceToHost);$

Algorithm 3: Non-Unified Memory Implementation of SMVP-BFS

According to the above Algorithm 3, *cudaMalloc* is used to allocate memory on GPU, and *cudaMemcpy* was used to copy the data from CPU to GPU. In order get the transpose of the matrix, the compressed row format of the matrix was converted to compressed column format using *cusparseDcsr2csc*. The $A^T x$ multiplication was done using *cusparseDcsrmv* and the final result was copied back to the CPU from device memory.

The GPU code in Appendix B.2, shows the complete CUDA program for the Breadth-First Search which uses Non-Unified Memory Implementation.

5.2.2. Unified Memory Implementation of SMVP-BFS

Unified memory is one of the main features in the Kepler architecture of NVIDIA GPU. That simply means, there is a common memory that can be accessed by both CPU and GPU at the same time. This reduces the time taken to coping data from CPU to device and device to CPU, while increasing the performance of the Breadth-First Search.

$cudaMallocManaged(\&cooxIndex, nnzsizeof(int))$;
cudaMallocManaged(&cooyIndex,nnzsizeof(int));
$cudaMallocManaged(\&cooVal, nnzsizeof(double))$;cudaMallocManaged(&x,n ×
sizeof(double)); $cudaMallocManaged(\&csrRowPtr, nsizeof(int))$;
$cudaMallocManaged(\&cscColIndex, nnzsizeof(int))$;
$cudaMallocManaged(\&cscVal, nnzsizeof(double))$;
$cudaMallocManaged(\&cscRowPtr, nsizeof(int))$;

$cusparseStatus\_tcusparseDcsr2csc(cusparseHandle\_thandle, intm, intn, intnnz, constdouble*$
$csrVal, constint * csrRowPtr, constint * csrColInd, double * cscVal, int *$
$cscRowInd, int *$
$cscColPtr, cusparseAction\_tcopyValues, cusparseIndexBase\_tidxBase)$;
for *(i=0;i<NumTimes;i++)* **do**
  $cusparseStatus\_tcusparseDcsrmv(cusparseHandle\_thandle, cusparseOperation\_ttransA, i$
  $alpha, constcusparseMatDescr\_tdescrA, constdouble * csrValA, constint *$
  $csrRowPtrA, constint * csrColIndA, constdouble * x, constdouble *$
  $beta, double * x)$
end

Algorithm 4: Unified Memory Implementation of SMVP-BFS

According to the Algorithm 4 above, it is clear that first the memory is allocated in the unified memory instead of CPU memory or GPU memory. Next the transpose process and the compressed column sparse matrix-vector multiplication was done as shown in the algorithm. The complete CUDA code for the Breadth-First Search with unified memory is given in the Appendix B.3 of Appendices.

6. Experimental Setup

6.1. Types of GPUs Tested

6.1.1. NVIDIA Tesla C2075

Tesla C2075 has 448 CUDA cores and has a single precision floating point performance. It consists of 6GB GDDR5 dedicated memory with 448 cores and 21504 threads. It has more performance than GeForce 820M and supports many CUDA libraries without much difficulty.The Breadth-First Search SMVP without unified memory was run on this and Figure 6.1 depicts view of this Nvidia Tesla C2075 graphic card.

Figure 6.1.: Nvidia Tesla C2075

6.1.2. NVIDIA Tesla K40

This is the worlds fastest accelerator graphic card that gives up to 10x performance than CPU. It consists of 2880 cores and 30720 threads with 12GB of CPU accelerated memory process over larger data sets. This card is commonly used in big data analytic. Tesla K40 or Kepler card contains some special features such as unified memory and dynamic parallelism in order to increase the performance. We tested the code Breadth-First Search with unified memory on this GPU card and Figure 6.2 shows the Kepler cards front view.

Figure 6.2.: Nvidia Tesla K40

6.2. Generating Data-set

6.2.1. Benchmarks

A benchmark in simple words is a point which is taken as a reference in order to measure something. Similarly in circuit world, we tend to use some benchmark circuits in order to test our circuit designs and circuit algorithms. Many universities such as University of Florida [11], University of Michigan [19] and ITC [15] have freely shared their benchmark circuit data-sets and sparse matrix data-sets to use them in future researches.

6.2.2. Method of using Benchmarks

The benchmarks of sparse matrices which we found were in the Coordinate format and we used them directly for our testing purposes. But they didnt fulfill our requirement as they were not real electronic digital circuits. While searching for new benchmarks, we played around with what we have in order to check the performance of our written codes in different GPU architectures as shown in the results.

As in order to check the performance on GPU, the data-set should be much larger than the total number of threads in the GPU cores. Therefore we created our own sparse matrices data-sets in the form of Coordinate format which has non zero elements from 10^2 to 10^8 in multiples of 1000 using a shell script and which has a common matrix size of (nxn), where $n = 10,000$. Appendix B.4 will give the full code that we used to generate the above data set for testing purposes.

6.3. Method of getting results

The Breadth-First Search which was written using CSparse and CuSparse were tested in three different times in the same machine which has 6th Gen Intel i7 (6700K) at 4.0 GHZ and a 32 GB DDR3 RAM at 2133 MHz and two GPU cards(NVIDIA Tesla K40 and Tesla C2075).

A script was run to get 3 outputs for same data-set and got the average of them as the execution time taken for each data-set. Since the standard deviation of each output was considerably really small, a single result was taken at the end as the total execution time for the Breadth-First Search.

This experiment was run on 3 set-ups such as,
- CPU - 6th Gen Intel i7 (6700K) at 4.0 GHZ and a 32 GB DDR3 RAM at 2133 MHz with singled threaded implementation
- Tesla K40 - NVIDIA Tesla K40 GPU card with 2880 cores / 30720 threads, 12GB of memory which run on above CPU.
- Tesla C2075 - NVIDIA Tesla C2075 GPU card with 448 cores / 21504 threads, 6GB of memory which run on above CPU.

7. Results

The sparse matrix data-sets in the form of Co-ordinate format has a constant size of $n*n$ where $n = 10^4$. The number of non zero elements (NNZ) that it has, varied from 10^3 to $523 * 10^3$ with a variation of 10^3. All these data was tested on 3 set-ups as mentioned in section 6.3 in Chapter 6.
When getting the results we don't change the sparse matrix size(n) and keep it constant while changing only NNZ throughout the experiment. Its because, only the number or non-zero elements of the matrix matters for the sparse matrix vector multiplication in compressed row format, not its size. More information on sparse matrix vector multiplication can be obtain from [27].

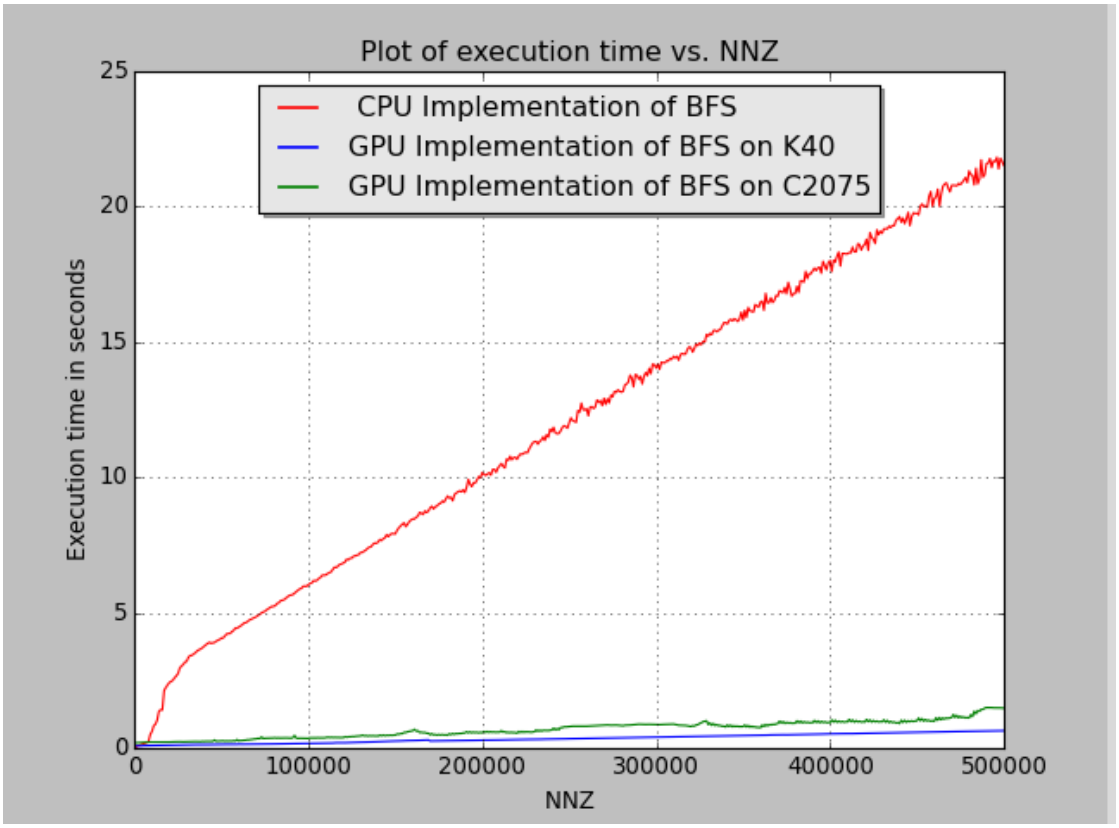
7.1. CPU vs. GPU

For this section we tested Breadth-First Search which was implemented using sparse matrix library called CSparse and CuSparse as mentioned in the above sections 4.2 and 5.2.1 in all three test set-up.

The Figure 7.1 depicts how the execution time varied with the number of non-zero elements(NNZ) in the data-set of the co-ordinate format of the created sparse matrix. The CPU time linearly increases with a large gradient while, the GPU execution time increases linearly with a very small gradient compared to CPU.

Since the size of the matrix is constant where $n = 10^4$, for a data set which is in matrix format, the execution time should be constant for all the sizes of non-zero elements. But since we have used co-ordinate format of sparse matrices for the storage of graph data, the execution time increased linearly with the non-zero elements or the sparsity.

Figure 7.1.: Execution time Vs NNZ



7.2. Tesla K40 vs. Tesla C2075

Tesla K40 and Tesla C2075 are two GPU cards which run on the CPU with 6th Gen Intel i7 (6700K) at 4.0 GHZ and a 32 GB DDR3 RAM at 2133 MHz.The GPU Tesla K40 card has 30,720 threads while Tesla C2075 card has 21,504 threads. As we have use two sparse matrix libraries to implement the matrix multiplication in SMVP-BFS, we assume a optimum thread utilization has done by the libraries itself. Therefore, due to the increase of 9216 threads and 2432 cores the GPU Tesla K40 has less execution time than the GPU Tesla C2075 as shown in the Figure 7.2.

When considering the speed up of the above test case, the Figure 7.3 clearly shows that the Breadth-First Search which implemented on Tesla K40 with non-unified memory has higher speed up of average 35X than the CPU version of Breadth-First Search implemented using CSparse.The Tesla C2075 also has a speed up more than 15X over the CPU implementation. According to this it is clear that Tesla K40 has a better performance than Tesla C2075 over CPU.

Figure 7.2.: Execution Time Vs NNZ of GPUs

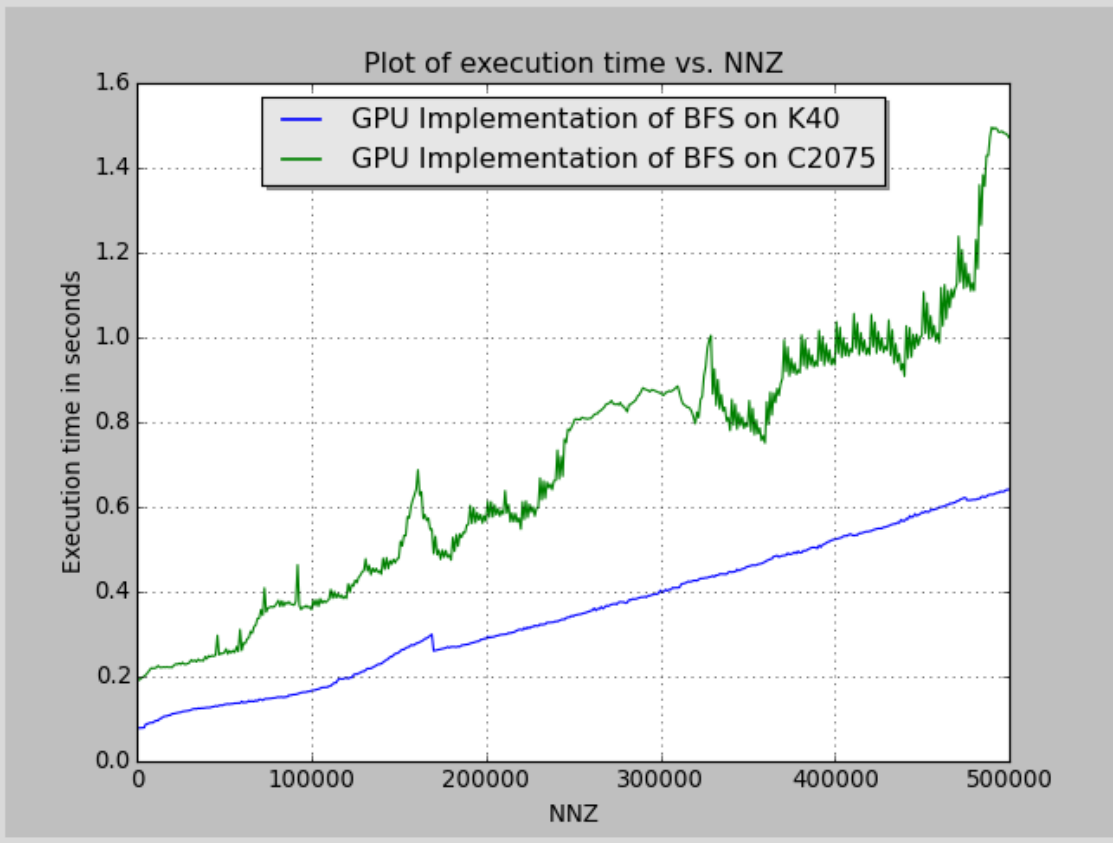
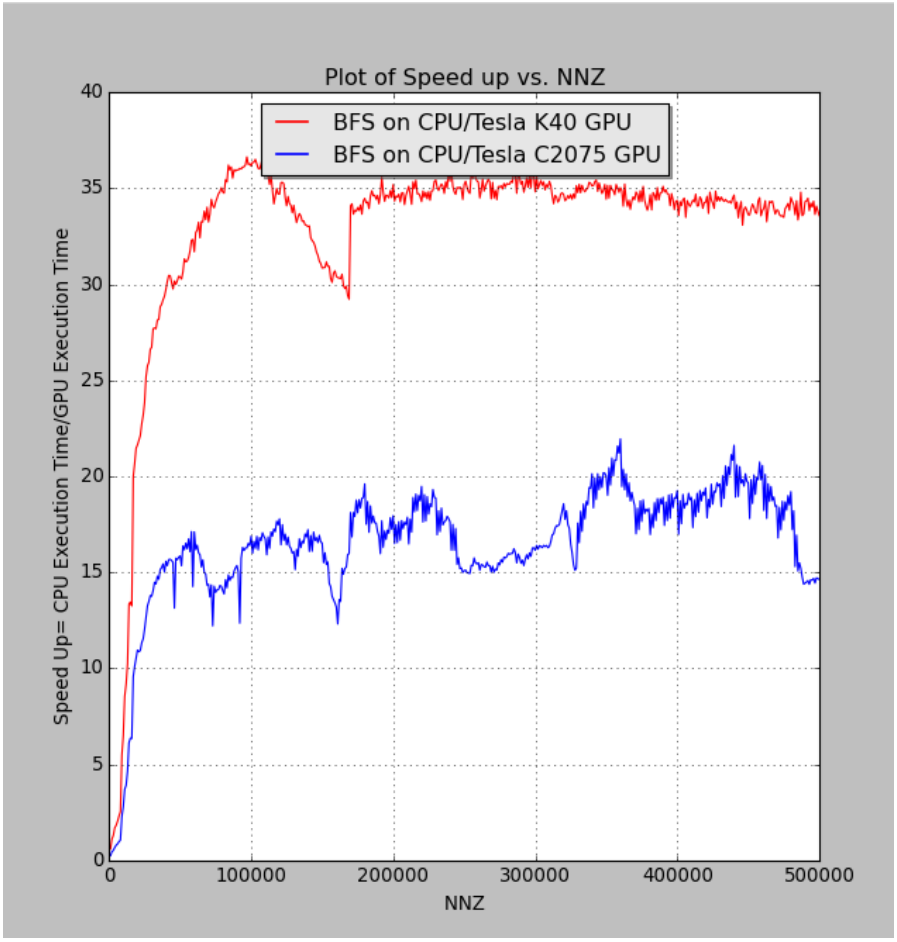


Figure 7.3.: Speed-up Vs NNZ of GPUs



7.3. Unified Memory vs. Non-Unified Memory

Unified memory is one of the important feature in the new NVIDIA Kepler GPU architecture. With this there is a common memory space that can be accessed by both the CPU and the device. For this section we tested the Breadth First Search Implementation mention above in section 5.2.1 and 5.2.2 on the Tesla K40 GPU in the machine with 6th Gen Intel i7 (6700K) at 4.0 GHZ and a 32 GB DDR3 RAM at 2133 MHz

Figure 7.4.: Execution Time Vs NNZ of Unified and Non-Unified Memory

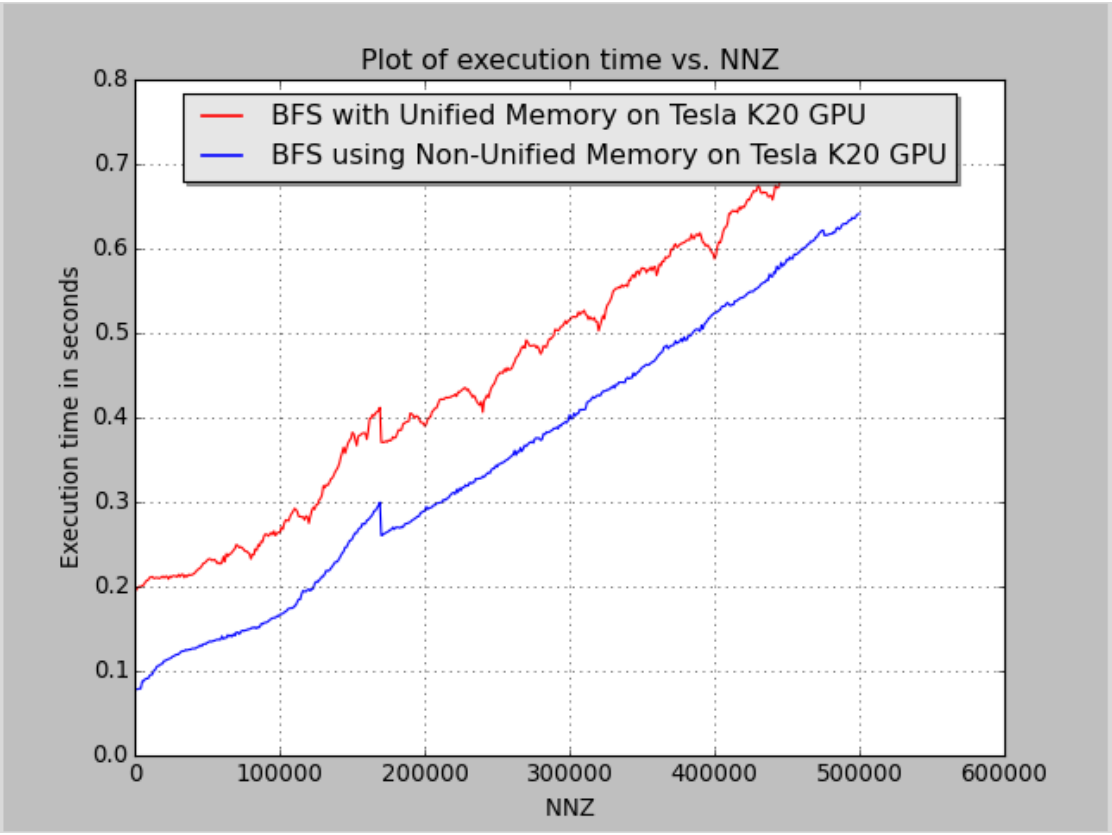


Figure 7.4 clearly depicts that Breadth-First Search implementation using Unified memory is slower than the Non-Unified Memory Implementation of Breadth-First Search. This also shows that unified memory does not effect much on the Breadth-First Search which uses Sparse Matrix Vector Multiplication.

8. Conclusion

According to the results we got, it is clearly seen that we can achieve a better performance about 35X speed-up in the Breadth-First Search approach with the use of CuSparse library and the Non-Unified memory implementation on Tesla K40 GPU than Tesla C2075 GPU. It is also clear that Unified memory of Tesla K40 does not effect for the sparse matrix multiplication in the SMVP-BFS.

8.1. Future work

- Implement Linked-List/Adjacency List form of Breadth-First Search with the use of Unified memory and Dynamic parallelism of GPU.
- Compare its performance with the CPU BFS implementation and SMVP-BFS implementation.

Bibliography

[1] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[2] Nathan Bell and Michael Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 18.

[3] Federico Busato and Nicola Bombieri. "An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures". In: ().

[4] Federico Busato and Nicola Bombieri. "BFS-4K: an efficient implementation of BFS for kepler GPU architectures". In: *Parallel and Distributed Systems, IEEE Transactions on* 26.7 (2015), pp. 1826–1838.

[5] John F Croix and Sunil P Khatri. "Introduction to GPU programming for EDA". In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. ACM. 2009, pp. 276–280.

[6] *Csparse*. URL: `http://people.sc.fsu.edu/~jburkardt/c_src/csparse/csparse.html` (visited on 03/25/2016).

[7] *CuSparse Toolkit Documentation*. URL: `http://docs.nvidia.com/cuda/cusparse/#axzz46L4qUu6F` (visited on 03/25/2016).

[8] Yangdong Steve Deng, Bo David Wang, and Shuai Mu. "Taming irregular EDA applications on GPUs". In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. ACM. 2009, pp. 539–546.

[9] Yangdong Deng and Shuai Mu. *Electronic Design Automation with Graphic Processors: A Survey*. Now Publishers Incorporated, 2013.

[10] *Dynamic programming tutorial*. URL: `https://www.codechef.com/wiki/tutorial-dynamic-programming` (visited on 03/25/2016).

[11] *Florida Sparse Matrix Benchmark collection*. URL: `https://www.cise.ufl.edu/research/sparse/matrices/` (visited on 03/25/2016).

[12] *GPU for Enhancing the EDA Tools*. URL: `https://drive.google.com/open?id=0B0Twwzq1zJrfWEI1RkxHOGlFYzQ` (visited on 03/25/2016).

[13] Yiding Han, Sanghamitra Roy, and Koushik Chakraborty. "Optimizing simulated annealing on GPU: A case study with IC floorplanning". In: *Quality Electronic Design (ISQED), 2011 12th International Symposium on*. IEEE. 2011, pp. 1–7.

[14] Pawan Harish and PJ Narayanan. "Accelerating large graph algorithms on the GPU using CUDA". In: *High performance computing–HiPC 2007*. Springer, 2007, pp. 197–208.

[15] *ITC99 benchmark dataset*. URL: `http://www.cad.polito.it/downloads/tools/itc99.html` (visited on 03/25/2016).

[16] Ahmad Al-Kawam and Haidar M Harmanani. "A Parallel GPU Implementation of the Timber Wolf Placement Algorithm". In: *Information Technology-New Generations (ITNG), 2015 12th International Conference on*. IEEE. 2015, pp. 792–795.

[17] Lijuan Luo, Martin Wong, and Wen-mei Hwu. "An effective GPU implementation of breadth-first search". In: *Proceedings of the 47th design automation conference*. ACM. 2010, pp. 52–55.

[18] Duane Merrill, Michael Garland, and Andrew Grimshaw. "High-Performance and Scalable GPU Graph Traversal". In: *ACM Transactions on Parallel Computing* 1.2 (2015), p. 14.

[19] *Michigan Sparse Matrix Benchmark dataset*. URL: `http://web.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html` (visited on 03/25/2016).

[20] *Moore's law wikipedia page*. URL: `https://en.wikipedia.org/wiki/Moore's_law` (visited on 03/25/2016).

[21] *Nvidia documentation for CUBLAS*. URL: `http://docs.nvidia.com/cuda/cublas/#axzz46L4qUu6F` (visited on 04/22/2016).

[22] *NVIDIA Kepler*. URL: `http://www.nvidia.com/object/nvidiakepler.html` (visited on 03/25/2016).

[23] *Nvidia MAGMA Developer*. URL: `https://developer.nvidia.com/magma` (visited on 04/22/2016).

[24] Chetan D Pise and Shailendra W Shende. "Parallelization of BFS Graph Algorithm using CUDA". In: ().

[25] *RocketSim*. URL: `http://www.rocketick.com/rocketsim/rocketsim` (visited on 03/25/2016).

[26] Gunjan Singla, Amrita Tiwari, and Dhirendra Pratap Singh. "New approach for graph algorithms on GPU using CUDA". In: *International Journal of Computer Applications* 72.18 (2013).

[27] *Sparse matrix vector multiplication*. URL: `http://www.cs.cmu.edu/~scandal/cacm/node9.html` (visited on 05/26/2016).

[28] *Taming Irregular EDA Applications on GPUs*. URL: `http://ieeexplore.ieee.org/` (visited on 03/25/2016).

A. Glossary

Optional glossary of technical terms.

B. Project artifacts

B.1. Breadth-First Search code implemented using Csparse

```
string title = "This is a Unicode  in the sky"
/*
Defined as π = lim_{n→∞} P_n/d where P is the perimeter
of an n-sided regular polygon circumscribing a
circle of diameter d.
*/
const double pi = 3.1415926535
  # include <stdlib.h>
  # include <limits.h>
  # include <math.h>
  # include <stdio.h>
  # include <time.h>
  # include "csparse.h"

  int main ( void )
  {
      cs *A;
      cs *AT;
      cs *C;
      cs *D;
      cs *Eye;
      cs *H;
      cs *G;
      cs *N;
      int i;
      int m,b;
      cs *T;
      FILE *file;
      file = fopen("dataset/dataset3.st","r");
      printf("Enter the No of steps:");
      scanf("%d",&b);
      b=b-1;
```

```
33
34          FILE *fp;
35          fp = fopen("dataset/x3.st","r");
36
37          T = cs_load ( file );//loading the triplet matrix from the dataset file.
38          H = cs_load ( fp );//loading the triplet matrix from the input x file.
39
40
41          A = cs_triplet ( T );//getting compressed column form of matrix T
42
43          cs_spfree ( T );//clearing T
44
45          G = cs_triplet ( H );
46          cs_spfree ( H );
47
48          clock_t begin, end;
49          double time_spent;
50
51          begin = clock();//starting the clock
52
53          AT = cs_transpose ( A, 1 );//getting transpose of matrix A(AT=A')
54
55          m = A->m;//m=number of rows of A
56
57          T = cs_spalloc ( m, m, m, 1, 1 );//creating triplet identity matrix
58
59          for ( i = 0; i < m; i++ )
60          {
61              cs_entry ( T, i, i, 1 );
62          }
63
64          Eye = cs_triplet ( T );
65          cs_spfree ( T );
66
67          C = cs_multiply (  AT,G);//computing A'*G; A'->transpose of input dataset f
68          N=C;
69          for(i=0;i<b;i++){//repeating no of steps
70              N = cs_multiply (  AT,N);
71
72          }
73
74          end = clock();//taking end time of above opration
75          cs_print ( N, 0 );//printing final result
76
```

```
77          cs_spfree ( A );//clearing spaces
78          cs_spfree ( AT );
79          cs_spfree ( C );
80          cs_spfree ( Eye );
81          cs_spfree ( G );
82          cs_spfree ( N );
83
84          printf ( "\n" );
85
86          time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
87
88          printf("the time taken for cSparse Execution :%lf Seconds. \n",time_spent);
89
90          return ( 0 );
91      }
```

B.2. Breadth-First Search code implemented using CuSparse for Tesla GPU architecture with non-unified memory.

```
1               /***
2               *Read the Matrix file and do the BFS =A'(A'.x)
3               *
4               ***/
5
6               #include <stdio.h>
7               #include <stdlib.h>
8               #include <err.h>
9               #include <cuda_runtime.h>
10              #include "helpers.cuh"
11              #include "cusparse.h"
12
13
14
15
16
17              void checkStatus(cusparseStatus_t status){
18
19                      if(CUSPARSE_STATUS_SUCCESS==status){
20                              printf("the operation completed successfully \n");
21                      }else if (CUSPARSE_STATUS_NOT_INITIALIZED==status){
```

```
                printf("CUSPARSE_STATUS_NOT_INITIALIZED \n");
            }else if(CUSPARSE_STATUS_ALLOC_FAILED==status){
                printf("CUSPARSE_STATUS_ALLOC_FAILED\n");
            }else if(CUSPARSE_STATUS_INVALID_VALUE==status){
                printf("CUSPARSE_STATUS_INVALID_VALUE\n");
            }else if(CUSPARSE_STATUS_ARCH_MISMATCH==status){
                printf("CUSPARSE_STATUS_ARCH_MISMATCH\n");
            }else if(CUSPARSE_STATUS_EXECUTION_FAILED==status){
                printf("CUSPARSE_STATUS_EXECUTION_FAILED\n");
            }else{
                printf("CUSPARSE_STATUS_INTERNAL_ERROR\n");
            }


}
int findMaxColNo(int array[] ){
    int k;
    int max = INT_MIN;

    for(k=0;k<(sizeof(array)/sizeof(int));k++){
        if(max<array[k]){
            max=array[k];
        }
    }
    printf("return max %d\n",max+1 );
    return max+1;

}
int main(){
   /*
   Read Data from file

   */


     //char line[256];
     int k=0;
     int noOfRows,noOfCols,nnz;
     int      n, i;
     int NumTimes=2;


     FILE* fileNew = fopen("data/dataset1.txt", "r");
```

```
66
67                  fscanf(fileNew, "%d %d %d",&noOfRows, &noOfCols, &nnz);
68
69                int  xIndex[nnz];//= {0,0,1,2,2};
70                int yIndex[nnz];//= {0,2,1,1,2};
71                double  Val[nnz];// = {1.0,1.0,1.0,1.0,1.0};
72                double X[noOfCols];
73
74                printf("No fo rows %d, No of Cols %d, nnz %d \n",noOfRows,noOfCols,nnz);    //
75
76
77                while (k<nnz) {
78                    /* note that fgets don't strip the terminating \n, checking its
79                       presence would allow to handle lines longer that sizeof(line) */
80                       fscanf(fileNew, "%d %d %lf",&xIndex[k], &yIndex[k], &Val[k]);
81                       k++;
82
83                }
84                /* may check feof here to make a difference between eof and io failure -- netu
85                   timeout for instance */
86
87
88
89                fclose(fileNew);
90                n= noOfRows;
91                printf("No of cols %d\n",n);
92
93               FILE* fileXVector = fopen("data/x1.txt", "r");
94                int h=0; int n1,n2;
95                while(h<nnz){
96                      fscanf(fileXVector, "%d %d %lf",&n1, &n2, &X[h]);
97                      h++;
98
99                }
100               fclose(fileXVector);
101
102               cusparseStatus_t status1,status2,status3,status4,status5,status6;
103
104           /************  Host Variables  ***************/
105           //coo host variables
106               int * cooxIndexHostPtr=(int *)malloc(sizeof(int)*nnz);
107               int * cooyIndexHostPtr=(int *)malloc(sizeof(int)*nnz);
108               double * cooValHostPtr=(double *)malloc(sizeof(double)*nnz);
109
```

```
110             //input host matrix varibles
111                 double * xHost=(double *)malloc(sizeof(double)*n);
112             //output host matrix variables
113                 double * yHost=(double *)malloc(sizeof(double)*n);
114 
115             /***********   GPU variables***********/
116             //coo gpu matrix variables
117                 int * cooxIndexCuda=(int *)malloc(sizeof(int)*nnz);
118                 int * cooyIndexCuda=(int *)malloc(sizeof(int)*nnz);
119                 double * cooValCuda=(double *)malloc(sizeof(double)*nnz);
120 
121             //csr gpu matrix variables
122 
123                 int *    csrRowPtrACudaPtr;
124 
125             //csc gpu matrix variables
126                 int *    cscColIndexACuda;
127                 int *    cscRowPtrACudaPtr;
128                 double * cscValACuda;
129             //gpu input matrix
130                 double * xCuda;
131                 double * tmpCuda;
132             //gpu output matrix
133                 double * yCuda;
134 
135                 const double alpha = 1.0;
136                 const double beta = 0.0;
137 
138                 printf("testing example\n");
139 
140              /*****************Here m should equal to n ************************
141 
142              /*************Asign  arrays to pointers***************/
143 
144              for(i=0;i<nnz;i++){
145                     cooxIndexHostPtr[i]=xIndex[i];
146                     cooyIndexHostPtr[i]=yIndex[i];
147                     cooValHostPtr[i]=Val[i];
148                   //  printf("the input data line %d ,%d ,%lf \n",cooxIndexHostPtr[
149                 }
150 
151 
152               for(i=0;i<n;i++){
153                   xHost[i]=X[i];
```

```
154             }
155
156
157              printf("\n");
158
159              /******************************** CUDA stuff starts here ****************
160              cudaDeviceProp prop;
161                  cudaGetDeviceProperties(&prop, 1);
162                  cudaSetDevice(1);
163                  fprintf(stderr,"Device name: %s\n", prop.name);
164
165              //start measuring time
166              cudaEvent_t start,stop;
167              float elapsedtime=0.0;
168              cudaEventCreate(&start);
169              cudaEventRecord(start,0);
170
171
172          //malloc space on GPU
173
174               cudaMalloc((void**)&cooxIndexCuda,nnz*sizeof(int));
175               cudaMalloc((void**)&cooyIndexCuda,nnz*sizeof(int));
176               cudaMalloc((void**)&cooValCuda,nnz*sizeof(double));
177               cudaMalloc((void**)&csrRowPtrACudaPtr,  n*sizeof(int));
178               cudaMalloc((void**)&cscColIndexACuda,nnz*sizeof(int));
179               cudaMalloc((void**)&cscValACuda,nnz*sizeof(double));
180               cudaMalloc((void**)&cscRowPtrACudaPtr,  n*sizeof(int));
181               cudaMalloc((void**)&xCuda,n*sizeof(double));
182               cudaMalloc((void**)&yCuda,n*sizeof(double));
183               cudaMalloc((void**)&tmpCuda,n*sizeof(double));
184
185               checkCuda(cudaMemcpy(cooxIndexCuda,cooxIndexHostPtr,sizeof(int)*nnz,cudaMemcp
186               checkCuda(cudaMemcpy(cooyIndexCuda,cooyIndexHostPtr,sizeof(int)*nnz,cudaMemcp
187               checkCuda(cudaMemcpy(cooValCuda,cooValHostPtr,sizeof(double)*nnz,cudaMemcpyHo
188               checkCuda(cudaMemcpy(xCuda,xHost,sizeof(double)*n,cudaMemcpyHostToDevice));
189
190
191             /*************************Cu Sparse part **********************/
192
193          //define the matrix features
194
195              cusparseOperation_t trans= CUSPARSE_OPERATION_NON_TRANSPOSE;
196              cusparseIndexBase_t idxBase = CUSPARSE_INDEX_BASE_ZERO;
197              cusparseAction_t copyValues= CUSPARSE_ACTION_NUMERIC;
```

```
198                 cusparseHandle_t handle;
199                 status4=cusparseCreate(&handle);  checkStatus(status4);
200
201                 cusparseMatDescr_t descrA ;
202                 status5 =cusparseCreateMatDescr(&descrA); checkStatus(status5);
203
204             /**
205             cusparseStatus_t
206             cusparseXcoo2csr(cusparseHandle_t handle, const int *cooRowInd,
207                              int nnz, int m, int *csrRowPtr, cusparseIndexBase_t id
208
209
210             Read more at: http://docs.nvidia.com/cuda/cusparse/index.html#ixzz46Adj
211             Follow us: @GPUComputing on Twitter | NVIDIA on Facebook
212             **/
213
214             status6= cusparseXcoo2csr(handle,cooxIndexCuda,nnz,n,csrRowPtrACudaPtr,
215             checkStatus(status6);
216
217             /*
218               cusparseStatus_t
219             cusparseDcsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
220                              const double *csrVal, const int *csrRowPtr,
221                              const int *csrColInd, double           *cscVal,
222                              int *cscRowInd, int *cscColPtr,
223                              cusparseAction_t copyValues,
224                              cusparseIndexBase_t idxBase)
225
226
227             Read more at: http://docs.nvidia.com/cuda/cusparse/index.html#ixzz456mk
228             Follow us: @GPUComputing on Twitter | NVIDIA on Facebook
229             */
230
231
232                              status1 =   cusparseDcsr2csc(handle,n,n,nnz,cooValCuda
233
234                             checkStatus(status1);
235
236
237
238                 /*
239                 cusparseStatus_t
240             cusparseDcsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
241                            int m, int n, int nnz, const double          *alpha,
```

```
242                         const cusparseMatDescr_t descrA,
243                         const double          *csrValA,
244                         const int *csrRowPtrA, const int *csrColIndA,
245                         const double          *x, const double          *beta,
246                         double          *y)
247
248             */
249
250
251             for(i=0;i<NumTimes;i++){
252                    status2=cusparseDcsrmv(handle,trans,n,n,nnz,&alpha,descrA,cscValACuda,c
253                    checkStatus(status2);
254
255                    status3=cusparseDcsrmv(handle,trans,n,n,nnz,&alpha,descrA,cscValACuda,c
256                    checkStatus(status3);
257
258                     checkCuda(cudaMemcpy(xCuda,yCuda,sizeof(int)*nnz,cudaMemcpyDeviceToDev
259              }
260              checkCuda(cudaMemcpy(yHost,yCuda,sizeof(double)*n,cudaMemcpyDeviceToHost));
261
262
263
264              //free the cuda memory
265             cudaFree(cooxIndexCuda);
266             cudaFree(cooyIndexCuda);
267             cudaFree(cooValCuda);
268             cudaFree(csrRowPtrACudaPtr);
269             cudaFree(cscColIndexACuda);
270             cudaFree(cscValACuda);
271             cudaFree(cscRowPtrACudaPtr);
272             cudaFree(xCuda);
273             cudaFree(yCuda);
274             cudaFree(tmpCuda);
275
276
277             //end measuring time
278             cudaEventCreate(&stop);
279             cudaEventRecord(stop,0);
280             cudaEventSynchronize(stop);
281             cudaEventElapsedTime(&elapsedtime,start,stop);
282
283             //print the output yHost
284
285          //print the answer : CUBLAS gives the answer placed in column major order
```

```
286                 printf("Answer : \n");
287                 for(i=0;i<n;i++){
288                         printf("%f \n",yHost[i]);
289
290                 }
291
292
293                 //print the time spent to stderr
294
295                 fprintf(stderr,"Time spent for operation on cusparse CUDA(Including
296
297
298                     free(cooxIndexHostPtr);
299                     free(cooyIndexHostPtr);
300                     free(cooValHostPtr);
301                     free(xHost);
302                     free(yHost);
303
304                     return 0;
305             }
```

B.3. Breadth-First Search code implemented using CuSparse for Kepler GPU architecture with unified memory.

```
1           /***
2           * Read the Matrix file and do the BFS =A'(A'.x)
3           *
4           ***/
5
6           #include <stdio.h>
7           #include <stdlib.h>
8           #include <err.h>
9           #include <cuda_runtime.h>
10          #include "helpers.cuh"
11          #include "cusparse.h"
12
13
14
15
16
```

```
17          void checkStatus(cusparseStatus_t status){
18
19                  if(CUSPARSE_STATUS_SUCCESS==status){
20                          printf("the operation completed successfully \n");
21                  }else if (CUSPARSE_STATUS_NOT_INITIALIZED==status){
22                          printf("CUSPARSE_STATUS_NOT_INITIALIZED \n");
23                  }else if(CUSPARSE_STATUS_ALLOC_FAILED==status){
24                          printf("CUSPARSE_STATUS_ALLOC_FAILED\n");
25                  }else if(CUSPARSE_STATUS_INVALID_VALUE==status){
26                          printf("CUSPARSE_STATUS_INVALID_VALUE\n");
27                  }else if(CUSPARSE_STATUS_ARCH_MISMATCH==status){
28                          printf("CUSPARSE_STATUS_ARCH_MISMATCH\n");
29                  }else if(CUSPARSE_STATUS_EXECUTION_FAILED==status){
30                          printf("CUSPARSE_STATUS_EXECUTION_FAILED\n");
31                  }else{
32                          printf("CUSPARSE_STATUS_INTERNAL_ERROR\n");
33                  }
34
35
36          }
37          int findMaxColNo(int array[] ){
38              int k;
39              int max = INT_MIN;
40
41              for(k=0;k<(sizeof(array)/sizeof(int));k++){
42                  if(max<array[k]){
43                      max=array[k];
44                  }
45              }
46              printf("return max %d\n",max+1 );
47              return max+1;
48
49          }
50          int main(){
51             /*
52             Read Data from file
53
54             */
55
56
57
58               //char line[256];
59               int k=0;
60               int noOfRows,noOfCols,nnz;
```

```
61              int      n, i;
62
63              int NumTimes=2;
64
65              FILE* fileNew = fopen("data/dataset1.txt", "r");
66
67              fscanf(fileNew, "%d %d %d",&noOfRows, &noOfCols, &nnz);
68
69             int  xIndex[nnz];//= {0,0,1,2,2};
70             int yIndex[nnz];//= {0,2,1,1,2};
71             double  Val[nnz];// = {1.0,1.0,1.0,1.0,1.0};
72             double X[noOfCols];
73             int n1,n2;
74             printf("No fo rows %d, No of Cols %d, nnz %d \n",noOfRows,noOfCols,nnz)
75
76
77             while (k<nnz) {
78                 /* note that fgets don't strip the terminating \n, checking its
79                    presence would allow to handle lines longer that sizeof(line) */
80                    fscanf(fileNew, "%d %d %lf",&xIndex[k], &yIndex[k], &Val[k]);
81                    k++;
82
83             }
84             /* may check feof here to make a difference between eof and io failure
85                timeout for instance */
86
87
88
89             fclose(fileNew);
90             n= noOfRows;
91             printf("No of cols %d\n",n);
92
93             FILE* fileXVector = fopen("data/x1.txt", "r");
94             int h=0;
95             while(h<nnz){
96                   fscanf(fileXVector, "%d %d %lf",&n1, &n2, &X[h]);
97                   h++;
98
99             }
100            fclose(fileXVector);
101
102            cusparseStatus_t status1,status2,status3,status4,status5,status6;
103
104        /************  Host Variables  ***************/
```

```
105         //coo host variables
106             int * cooxIndex;
107             int * cooyIndex;
108             double * cooVal;
109 
110         //input host matrix varibles
111             double * x;
112         //output host matrix variables
113             double * y;
114 
115         /***********   GPU variables***********/
116         //csr gpu matrix variables
117 
118             int *    csrRowPtr;
119 
120         //csc gpu matrix variables
121             int *    cscColIndex;
122             int *    cscRowPtr;
123             double * cscVal;
124         //gpu input matrix
125 
126             double * tmpVector;
127 
128             const double alpha = 1.0;
129                  const double beta = 0.0;
130 
131                  printf("testing example\n");
132 
133 
134         /***************Select device*********************************************/
135 
136                 cudaDeviceProp prop;
137                 cudaGetDeviceProperties(&prop, 0);
138                 cudaSetDevice(0);
139                 fprintf(stderr,"Device name: %s\n", prop.name);
140 
141 
142 
143          /****************CudaMallocManaged***********************************/
144 
145 
146          cudaMallocManaged(&cooxIndex,nnz*sizeof(int));
147          checkCudaError();
148          cudaMallocManaged(&cooyIndex,nnz*sizeof(int));
```

```
149         checkCudaError();
150         cudaMallocManaged(&cooVal,nnz*sizeof(double));
151         checkCudaError();
152         cudaMallocManaged(&x,n*sizeof(double));
153         checkCudaError();
154         cudaMallocManaged(&y,n*sizeof(double));
155         checkCudaError();
156         cudaMallocManaged(&tmpVector,n*sizeof(double));
157         checkCudaError();
158         cudaMallocManaged(&csrRowPtr,n*sizeof(int));
159         checkCudaError();
160         cudaMallocManaged(&cscColIndex,nnz*sizeof(int));
161         checkCudaError();
162         cudaMallocManaged(&cscVal,nnz*sizeof(double));
163         checkCudaError();
164         cudaMallocManaged(&cscRowPtr,n*sizeof(int));
165         checkCudaError();
166
167
168         /*************Asign  arrays to pointers***************/
169
170         for(i=0;i<nnz;i++){
171                cooxIndex[i]=xIndex[i];
172                cooyIndex[i]=yIndex[i];
173                cooVal[i]=Val[i];
174              //  printf("the input data line %d ,%d ,%lf \n",cooxIndexHostPtr[i],c
175            }
176
177
178           for(i=0;i<n;i++){
179               x[i]=X[i];
180          }
181
182
183            /********************************* CUDA stuff starts here ************
184
185
186            //start measuring time
187            cudaEvent_t start,stop;
188            float elapsedtime=0.0;
189            cudaEventCreate(&start);
190            cudaEventRecord(start,0);
191
192
```

```
193
194
195          /*************************Cu Sparse part **********************/
196
197         //define the matrix features
198
199             cusparseOperation_t trans= CUSPARSE_OPERATION_NON_TRANSPOSE;
200             cusparseIndexBase_t idxBase = CUSPARSE_INDEX_BASE_ZERO;
201             cusparseAction_t copyValues= CUSPARSE_ACTION_NUMERIC;
202             cusparseHandle_t handle;
203             status4=cusparseCreate(&handle);  checkStatus(status4);
204
205             cusparseMatDescr_t descrA ;
206             status5 =cusparseCreateMatDescr(&descrA); checkStatus(status5);
207
208         /**
209         cusparseStatus_t
210         cusparseXcoo2csr(cusparseHandle_t handle, const int *cooRowInd,
211                         int nnz, int m, int *csrRowPtr, cusparseIndexBase_t idxBase)
212
213
214         Read more at: http://docs.nvidia.com/cuda/cusparse/index.html#ixzz46AdjmqnI
215         Follow us: @GPUComputing on Twitter | NVIDIA on Facebook
216         **/
217
218         status6= cusparseXcoo2csr(handle,cooxIndex,nnz,n,csrRowPtr,idxBase);
219     checkStatus(status6);
220
221     /*
222       cusparseStatus_t
223     cusparseDcsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
224                     const double *csrVal, const int *csrRowPtr,
225                     const int *csrColInd, double           *cscVal,
226                     int *cscRowInd, int *cscColPtr,
227                     cusparseAction_t copyValues,
228                     cusparseIndexBase_t idxBase)
229
230
231     Read more at: http://docs.nvidia.com/cuda/cusparse/index.html#ixzz456mkjYNe
232     Follow us: @GPUComputing on Twitter | NVIDIA on Facebook
233     */
234
235
236         status1 =   cusparseDcsr2csc(handle,n,n,nnz,cooVal,csrRowPtr,cooyIndex,cscVal,cscColIr
```

```
237
238       checkStatus(status1);
239
240
241
242         /*
243         cusparseStatus_t
244     cusparseDcsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
245                    int m, int n, int nnz, const double          *alpha,
246                    const cusparseMatDescr_t descrA,
247                    const double          *csrValA,
248                    const int *csrRowPtrA, const int *csrColIndA,
249                    const double          *x, const double          *beta,
250                    double          *y)
251
252         */
253
254         for(i=0;i<NumTimes;i++){
255                status2=cusparseDcsrmv(handle,trans,n,n,nnz,&alpha,descrA,cscVal,csc
256                checkStatus(status2);
257
258                status3=cusparseDcsrmv(handle,trans,n,n,nnz,&alpha,descrA,cscVal,csc
259                checkStatus(status3);
260                cudaMemcpy(x,y,sizeof(int)*nnz,cudaMemcpyDeviceToDevice);
261
262         }
263         //end measuring time
264         cudaEventCreate(&stop);
265         cudaEventRecord(stop,0);
266         cudaEventSynchronize(stop);
267         cudaEventElapsedTime(&elapsedtime,start,stop);
268
269
270
271
272
273         //print the output yHost
274
275
276         printf("Answer : \n");
277         for(i=0;i<n;i++){
278                  printf("%f \n",y[i]);
279
280         }
```

```
281
282
283
284         //free the cuda memory
285         cudaFree(cooxIndex);
286         cudaFree(cooyIndex);
287         cudaFree(cooVal);
288         cudaFree(csrRowPtr);
289         cudaFree(cscColIndex);
290         cudaFree(cscVal);
291         cudaFree(cscRowPtr);
292         cudaFree(x);
293         cudaFree(y);
294         cudaFree(tmpVector);
295
296         //print the time spent to stderr
297
298         fprintf(stderr,"Time spent for operation on cusparse CUDA(Including memory allocation
299
300
301             return 0;
302     }
```

B.4. C code to make Coordinate format Sparse matrix.

```
1         #include<stdio.h>
2
3         int main(){
4         long long int n=10000;
5         long long int N=n*n;
6         long long int nnz=100; //nnz
7         long int i;
8         long long int in, im;
9
10        FILE * dataFile =fopen("data1.txt", "w");
11
12
13        im = 0;
14        fprintf(dataFile,"%lld %lld %lld\n",n ,n , nnz );
15        for (in = 0; in < N && im < nnz; ++in) {
16        long long int rn = N - in;
```

```
17          long long int rm = nnz - im;
18          if (rand() % rn < rm) {
19          long long int num=in+0;
20          long long int x= num/n;
21                   long long int y=num%n;
22              /*Write data to the file*/
23          fprintf(dataFile,"%lld %lld 1.0\n",x ,y);
24          }
25          }
26
27          fclose(dataFile);
28          return 0;
29          }
```