University of Peradeniya

Department of Computer Engineering

CO425: Final Year Project II

A Faster GPU Implementation Of Breadth-First Traversal For EDA Graph Circuits

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

Supervisor
Dr. Roshan Ragel

October 5, 2016

Abstract

With Moore's law in effect, as the complexity of digital electronic circuits increases, the amount of time spend by the electronic design automation (EDA) tools to design such circuits also increases. It brings us to the point, where we need to improve the performance of EDA algorithms to fulfill the present and the future requirements of the EDA industry. Out of many algorithms used by these tools, Breadth First Traversal (BFT) is one of the most commonly used algorithms to traverse the gates of electronic circuits.

Therefore, we present a simple, fast and parallelizable BFT algorithm named, H-BFT. We show that the CPU implementation of H-BFT is about 75x faster than the CPU implementation of the state of the art, the Sparse Matrix Vector Product (SMVP) based BFT. Further, with the new features introduced by NVIDIA in their GPUs, we have accelerated both the state of the art SMVP based BFT implementation and our new H-BFT implementation. The best speedups we achieved via these accelerations are 150x and 10x for the SMVP-BFT and H-BFT respectively.

Contents

List of Figures

List of Tables

1. Summary

In CO421, our main objective was to understand the available EDA graph algorithms on GPU. From literature review we found out that the Breadth-First Search using Sparse Matrix Vector Product was the most common implementation and research work done by many researchers till 2015. With the introduction of Kepler GPU card by NVIDIA the direction changed. But in Semester 7, what we focused on was, implementation of Breadth-First Traversal using Sparse Matrix Vector Product with the prevailing sparse matrix libraries called cSprase and CuSparse on two different CPU cards Tesla K40 and Tesla C2075 .

According to the results we got in CO 421, it is clearly seen that we can achieve a better performance about 35X speed-up in the Breadth-First Traversal 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.

2. 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.

2.1. 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.

2.2. 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)

2.3. 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].

2.4. 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 [6] 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 such as dynamic parallelism [9] and optimize a commonly used EDA algorithm called Breadth-First Traversal, in order to achieve greater performance than the state of the art. Also through this research we implemented a new algorithm for Breadth-First Traversal called "H-BFT" which is simple and faster on both CPU and GPU.

3. Background

3.1. 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 [8] 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 3.1 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 3.2. 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 Alg rithm**
High Performance and Sacalable GPU Graph Traversal. [18]	Taming Irregular EDA Applications on GPUs.[7]	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 gra algorithms the GPU us CUDA.[14]
Taming Irregular EDA Applications on GPUs.[7]	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 3.1.: 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. [7]	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 3.2.: Summary of Breadth-First Search Implementation Researches.

4. Breadth-First Search

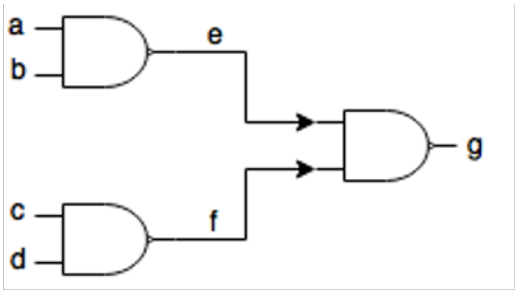
Breadth-First Search is an algorithms that can be used to traverse a graph data structure by starting from the root node and visiting the neighboring nodes first before moving to the next level neighbors. According to [18] Breadth-First Search is the main graph traversal algorithm which has become the 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.

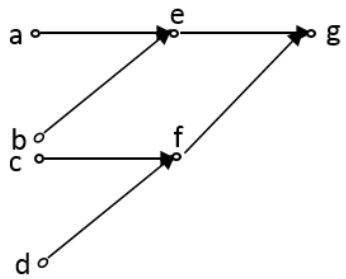
Figure 3.1 clearly shows how a simple gate circuit can be shown as directed timing graph and how a directed graph can be shown as a sparse matrix in the form of Compressed Row format. In this figure A(i,j) = 1 if and only if i is directed to j and also A(i,j) can have a non-zero element to represent 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



(i)



(ii)

$$A = \begin{array}{c} \begin{matrix} a & b & c & d & e & f & g \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(iii)

$$rowptr = \{0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6\}$$

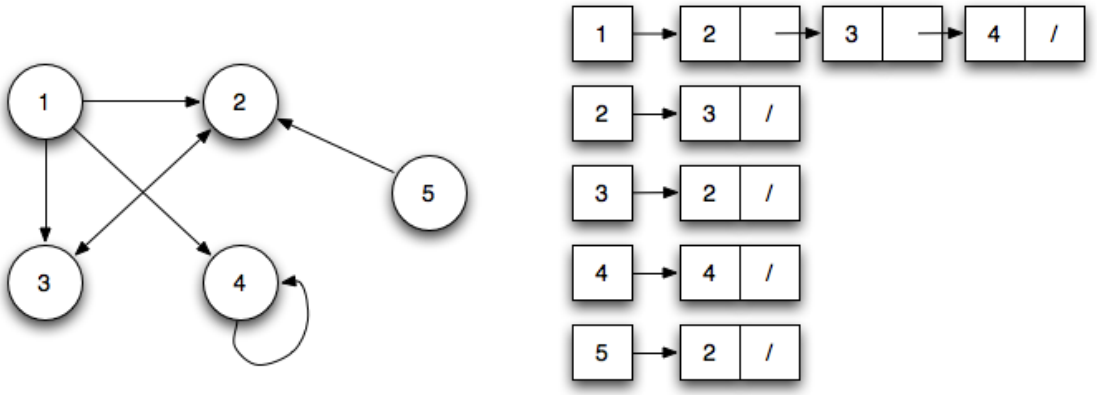
$$col = \{4 \quad 4 \quad 5 \quad 5 \quad 6 \quad 6\}$$

$$elem = \{1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1\}$$

(iv)

Figure 4.1.: Sparse Matrix and Compressed Row Format, i) Simple circuit, ii) Directed graph corresponds to i, iii) Sparse matrix of ii, iv) Compressed Row Format of iii

Figure 4.2.: 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.

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

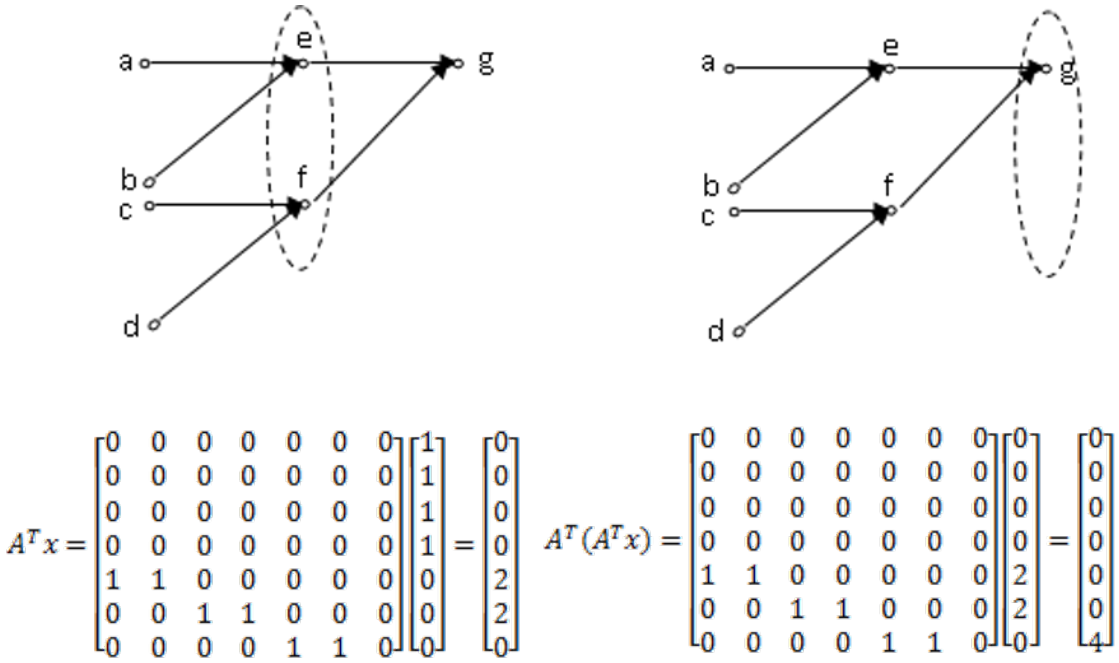


Figure 4.3.: BFT using Sparse Matrix Vector Multiplication

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$A^{'}..(A^{'}(A^{'} * x))$, $n$times, 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 done the GPU implementation of the above Breadth-First Search using dynamic parallelism for Kepler GPU card.

4.1.1. CPU Implementation of SMVP-BFS

This implementation was a single threaded implementation that was written using C. Once the graph data is taken as three arrays of a sparse matrix in the compressed column format and the inputs to the circuit was taken as a one single vector, the compressed row matrix vector multiplication was done in a loop while checking for a output zero vector as shown in Algorithm 2.

$while(checkStatus(tmpHost, n))$
$tmpHost =$
$results(tmpHost, xHost, cscValHost, cscRowPtrAHostPtr, cscColIndexAHost, n);$
$for(i = 0; i < n; i++)$
$xHost[i] = tmpHost[i];$
end
end

Algorithm 1: Multiplication loop

$for(i = 0; i < n; i++)\ yHost[i] = 0;$
$introw_start = cscRowPtrAHostPtr[i];$
$introw_end = cscRowPtrAHostPtr[i + 1];\ for(k = row_start; k < row_end; k++)$
$yHost[i] = yHost[i] + (cscValHost[k] * xHost[cscColIndexAHost[k]]);\ end$
end

Algorithm 2: Compressed Row Vector Multiplication

4.1.2. GPU Implementation of SMVP-BFS

GPU implementation of SMVP-BFS is a multi-threaded implementation which calls 10^6 number of threads once, in a single multiplication kernel call. Calling device functions again and again from the host causes lot of overhead. Therefore, dynamic parallelism on GPU has used in order to minimize this overhead. In this scenario, a parent kernel is called from the host and this parent kernel will call, all the other iterative multiplication kernels from the device itself. After finishing all the multiplication processes, finally the

output is copied back to the Host from the Device. Till then there is no intervention from the Host to the Device.

$*lev = 1;$
$while(*lev == 1)\ BreadthFirstSearch <<< ceil(*edges/256.0), 256 >>>$
$(adjacencyList, vertices, level, lev, edges);\ cudaDeviceSynchronize();$
$isLevelFilled <<< ceil(*vertices/256.0), 256 >>>> (level, vertices, lev);$
$cudaDeviceSynchronize();\ end$

Algorithm 3: Parent Kernel where dynamic parallelism used

The same algorithm which is used in CPU implementation is used in GPU implementation as well. Algorithm 3 shows the method of calling dynamic parallelism inside the parent kernel. Furthermore, the Appendix B.3 will provide the in detail code for the GPU implementation where dynamic parallelism is used

5. H BFT- Our Breadth-First Traversal

Even though we use GPU, we believe that the Sparse Matrix Vector Product is a expensive computational method, since sparse matrix is represented by three integer arrays, the multiplication process is difficult than the general matrix-vector multiplication. Therefore, in this paper we are introducing a fast and simple algorithm H-BFT for the circuit graph traversal. This is a Breadth-First Traversal with multiple input vertices which traverses the circuit level by level.

5.1. Design and Implementation of H-BFT on CPU

There are two main important variables involved in this algorithm. One of them is "GraphList" which is an array of structures called "Edge" of size nnz, here nnz is the number of edges in the graph. This "Edge" is a data structure that uses two integer values to store the source and the destination vertices of an edge of a graph. The other variable is called "level" which is an array of integers of size n, the number of vertices in the graph. The "level" array is used to store the levels of each vertex while traversing. These two main variables are clearly depicts in the Fig. 5.1.

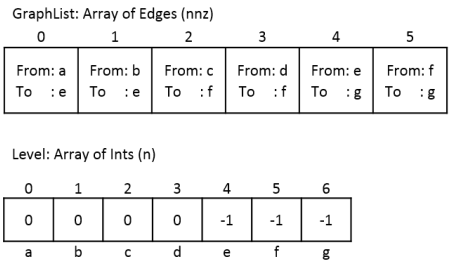


Figure 5.1.: GraphList and level variables for the example circuit in Fig.4.1

T The graph data, which is input to the H-BFS is transferred to the "GraphList". Initially the "level" array is filled with -1 values in order to indicate that these vertices are not visited. Since we know the input vertices, we make the levels of those vertices to zero in order to indicate that these vertices are inputs to the circuit.

Our CPU version is a single threaded implementation. In this case, as in Listing 1, we go through the GraphList (the array of Edge structures) one by one and check for the level value of "From" vertex and the "to" vertex. If the level value of the "From" vertex

is a non-negative value and the level value of the "to" vertex is a negative value, then the level value of the "to" vertex is updated. We continue this process, until the level array is completely filled with non-negative values.

```
struct Edge element;
for(k=0;k<*edges;k++){
        element = adjacencyList[k];
        if ((level[element.from]>=0)&&
          (level[element.to]==-1)){
                level[element.to] =
                 level[element.from]+1;
      }
}
```

Listing 1: Traversal in H-BFT

5.2. Design and Implementation of H-BFT on GPU

For the GPU implementation, we used the same data structures that is used in serial implementation. Furthermore, as shown in Listing 2 we use dynamic parallelism using GPU threads. From the CPU the *parentKernel* is called using a single thread in a single block. This parent kernel is responsible for the Breadth First Traversal happening in the parallel H-BFT algorithm. This kernel will launch the children kernels *BreadthFirstSearch* and *isLevelFilled* repetitively till the whole level array fills with non-negative values.

```
__global__ void parentKernel(struct Edge *
adjacencyList, int * vertices, int * level,
               int * lev, int * edges){
*lev=1; int kb=0;

while(*lev){
    kb = kb+1;
    BreadthFirstSearch<<<ceil(*edges/256.0),256>>>
         (adjacencyList,vertices,level,lev,edges);
    cudaDeviceSynchronize();
    isLevelFilled<<<ceil(*vertices/256.0),256>>>
                        (level,vertices,lev);
    cudaDeviceSynchronize();
}}
```

Listing 2: parent kernel of H-BFT in CUDA which uses dynamic parallelism

As in Listing 3, *BreadthFirstSearch* kernel is launched with nnz number of threads on GPU. The maximum block size of 256 is selected from the CUDA occupancy calculator [**CalOcupancy**] in order to achieve better performance. Therefore, the number of blocks, which is needed to launch nnz threads is $ceil(nnz/256.0)$. A single thread will go through a single Edge element in the graph dataset and check for the level values of it's "From"

and "To" elements whether it is non-negative and if level value of "To" is a negative value, update the level value of "To" element similar to the CPU version. If it is non-negative, that thread will idle without doing any update. The next iteration is called only if all the threads finished their work in the current iteration. The threads will wait till one level is finished because the current level updates depends on the previous level updates. Therefore, for this process as shown in Listing 2, we used *cudaDeviceSynchronize*() after every kernel call.

```
__global__ void BreadthFirstSearch(
   struct Edge * adjacencyList, int * vertices,
   int * level, int * lev, int * edges ){

   int tid = (blockDim.x * blockIdx.x ) +
                  threadIdx.x;
   *lev = 0;
   if(tid<*edges){

     struct Edge element =
              adjacencyList[tid];
  if (level[element.from]>=0 and
           level[element.to]==-1){
     level[element.to] =
              level[element.from]+1;
  }  }}
```

Listing 3: BreadthFirstSearch kernel of H-BFT in CUDA

6. Experimental Setup

6.1. Types of GPU Tested

6.1.1. 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 5.1 shows the Kepler cards front view.

Figure 6.1.: 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 10^7 to 10^8 and which has a common matrix size of (nxn), where $n = 10^6$. The input vertices to the circuit is a constant which 500 in every case. The above graph data sets again converted into three forms of graphs. Such as Best case where edges of the graph is in order as shown in figure 5.1, worst case where edges are in reverse order and average case where a combination of both.

6.3. Method of getting results

The Breadth-First Search which was implemented on CPU and GPU were tested in two 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 3 sets of data-sets 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.

7. Results

The sparse matrix data-sets in the form of Co-ordinate format has a constant size of $n * n$ where $n = 10^6$. The number of non zero elements (NNZ) that it has, varied from 10^7 to 10^8 with a variation of 10^7. All these data was tested on 2 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

7.1.1. SMVP-BFT Algorithm

According to the results we got, for all types of datasets it gave the similar execution times with the nnz. Because once the dataset is loaded to a sparse matrix, which is in the compressed column format, it is the same multiplication process that takes place in the SMVP-BFT. Fig. 7.1 clearly shows that with the increase of nnz, the execution time also increases because the sparsity of the graph become decreasing and increase the number of computations that should be done.

From the Fig. 7.2, it can be seen that SMVP-BFT CUDA version has achieved a maximum speedup of 150X over its CPU implementation.

7.1.2. H-BFT Algorithm

Fig. 7.4. depicts the average execution time taken by the H-BFT to run dataset A. It clearly shows that the GPU version is faster than the CPU version. Because, when the size of the dataset getting bigger the time taken by the CPU to execute a serial H-BFT also increases. But since the edges are in order, the CPU version will complete the graph traversal in one iteration, while the GPU version takes more than one. According to Fig. 7.5, it is clear that in dataset B the gap between the execution time of CPU and GPU is increasing. When the GraphList is in reverse order, when we are traversing through the "Edge" element by element, since they are in reverse order the input vertices are visited

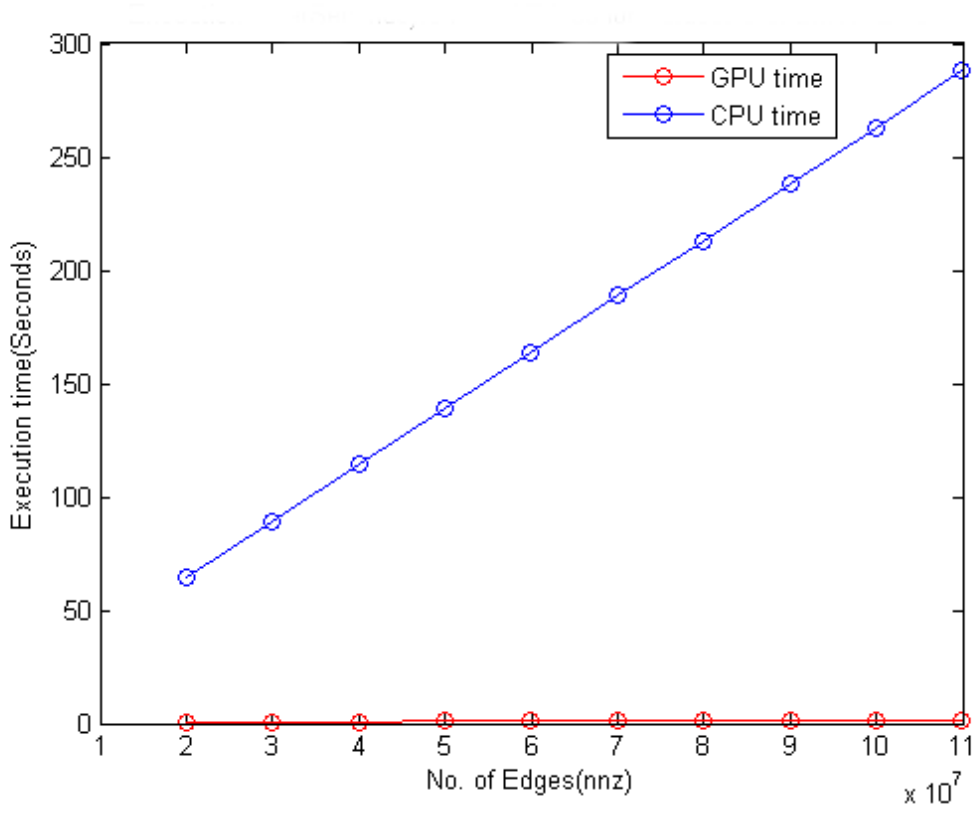


Figure 7.1.: Execution time Vs NNZ for SMVP-BFS

at last. Therefore, to fill the "Level" array, CPU version takes more than one iterations as well as in GPU version. As seen in Fig.7.3, in the dataset C the gap between CPU and GPU is constant, meanwhile with the increase of nnz the CPU time increases.

When considering the speedup of GPU as depicts in Fig. 7.6, for dataset B, H-BFT has achieved the highest speedup of 10X over the CPU version. While the least speedup is achieved by the GPU is for dataset A, which is 2X. Furthermore, dataset C achieves around 10X speedup over the CPU version.

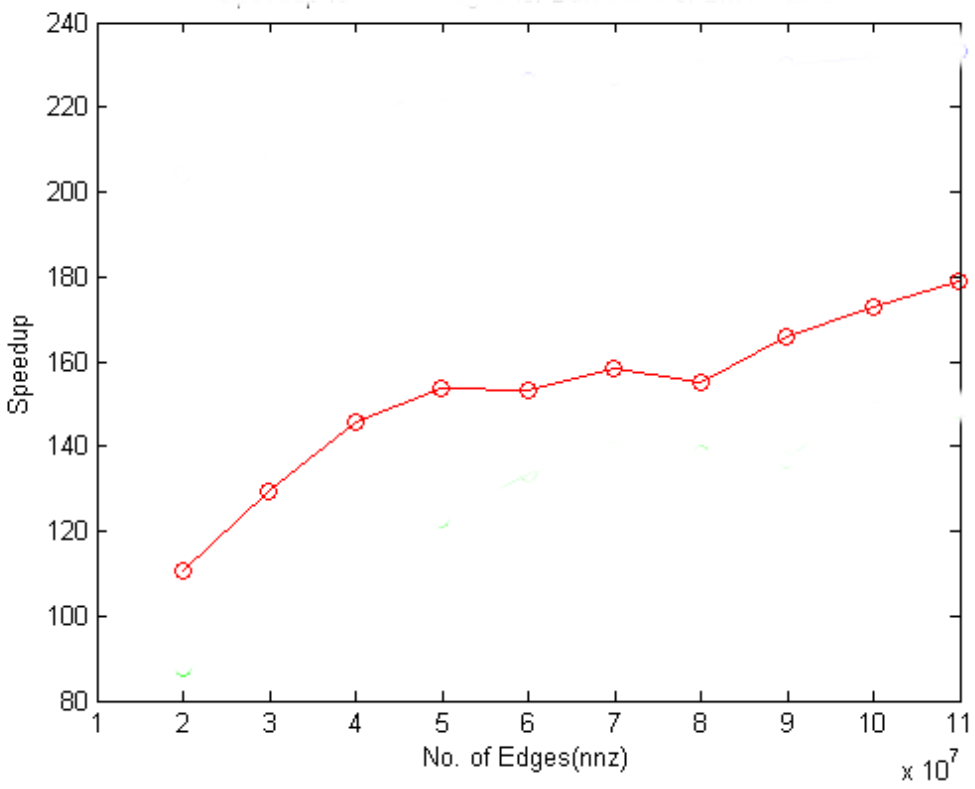


Figure 7.2.: Speedup for SMVP-BFS

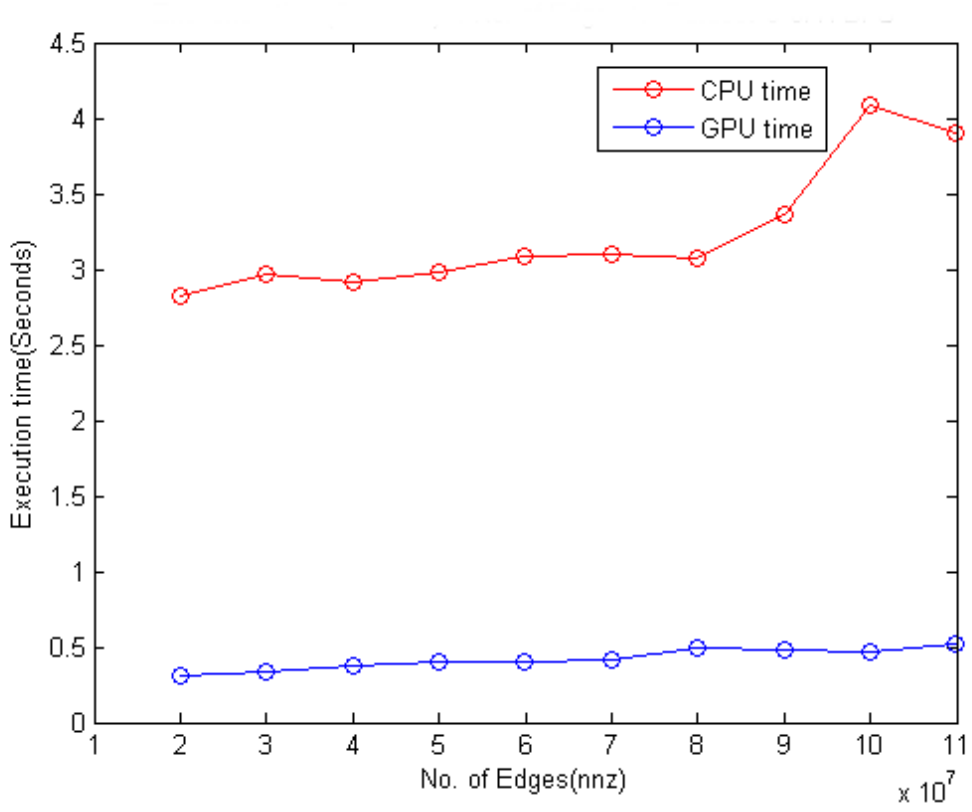


Figure 7.3.: Execution Time Vs NNZ of H-BFT for Average case data sets

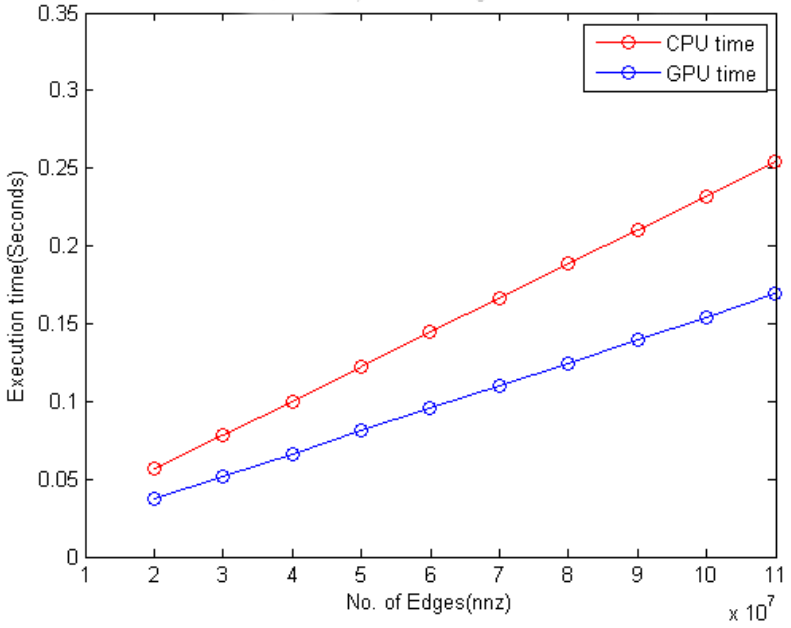


Figure 7.4.: Execution Time Vs NNZ of H-BFT for Best case data sets

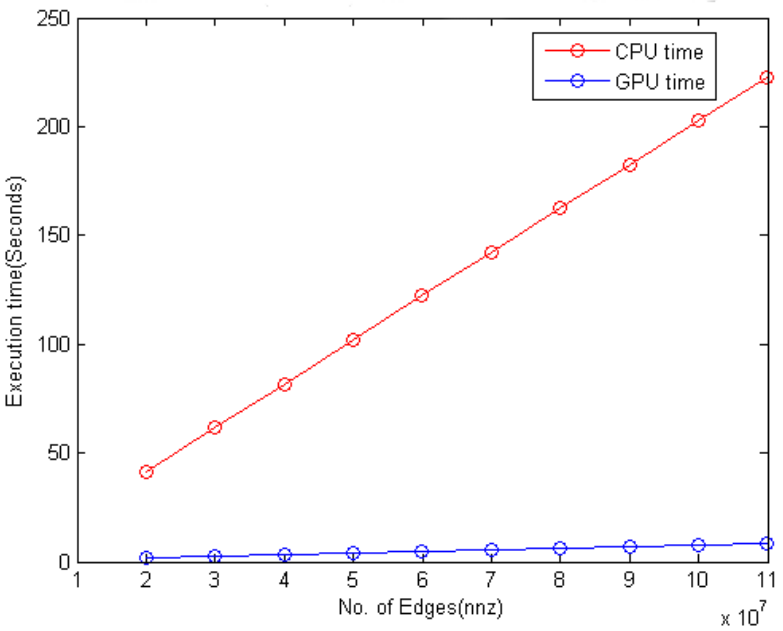


Figure 7.5.: Execution Time Vs NNZ of H-BFT for Worst case data sets

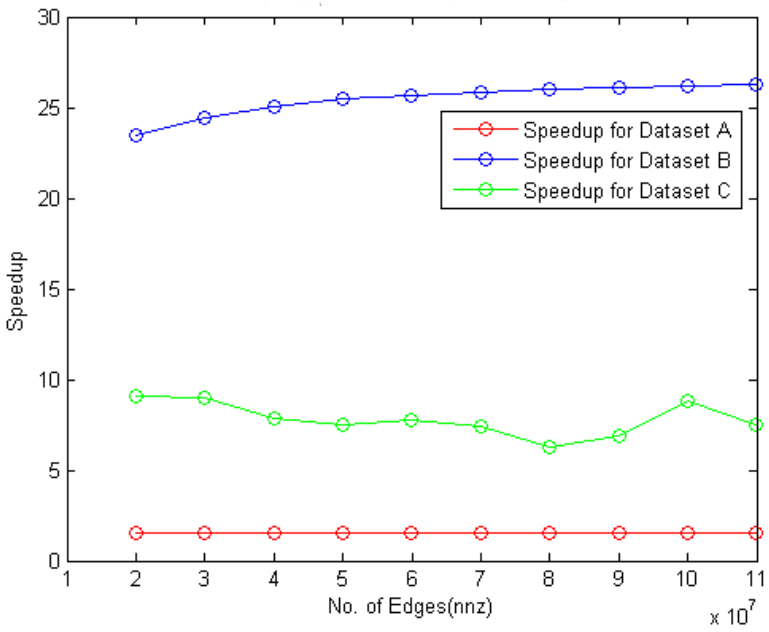


Figure 7.6.: Speed-up Vs NNZ of H-BFT for all the data sets

7.2. SMVP-BFT vs. H-BFT

Table 7.1 shows the execution time taken by all the algorithms when nnz=10^8. Out of the CPU versions of SMVP-BFT and H-BFT, for all the three types of datasets (A,B,C) the H-BFT is faster than the SMVP-BFT. Meanwhile for the GPU versions of SMVP-BFT and H-BFT, for the datasets A and C , H-BFT is faster than SMVP-BFT. But for the dataset B, SMVP-BFT is faster than H-BFT.

Algorithm	Execution Time (s)
H-BFT CPU : dataset A	0.255
H-BFT GPU : dataset A	0.169
H-BFT CPU : dataset B	222.945
H-BFT GPU : dataset B	8.496
H-BFT CPU : dataset C	3.907
H-BFT GPU : dataset C	0.521
SMVP-BFT CPU : average All datasets	287.808
SMVP-BFT GPU : average All datasets	1.610

Table 7.1.: Execution time taken by all the algorithms when nnz=10^8

8. Conclusion

In this report, we presented a comparison of two algorithms, which can be used for Breadth-First Traversal of large circuit graphs. One is SMVP-BFT, which is implemented using dynamic parallelism on GPU and the other is our newly implemented H-BFT using dynamic parallelism on GPU. Furthermore, with evidence it is proven that our solution, H-BFT is much more faster on CPU as well as on GPU (except for dataset B) than the SMVP-BFT. We believe our contribution will help to increase the performance of EDA tools where BFT is used.

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

[7] 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.

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

[9] *Dynamic Parallelism on GPU*. URL: `https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/` (visited on 07/25/2016).

[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. SMVP-Breadth-First Search implementation using dynamic parallelism

```
  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
 /***
ead the Matrix file and do the BFS =A'(A'.x)

*/

nclude <stdio.h>
nclude <stdlib.h>
nclude <err.h>
nclude <cuda_runtime.h>
nclude "helpers.cuh"
nclude "cusparse.h"


global__ void  checkStatus(int * x,int * y, int * val, int * n ){


  int j = (blockDim.x * blockIdx.x ) + threadIdx.x;

   if(j<*n){
      x[j] =y[j];

      if(y[j]!=0 && *val==0){
            *val = 1;
      }else{
```

```
32
33          *val=0;
34      }
35
36  }
37
38
39
40
41
42 global__ void SMVP( int*  n,
43               double *csrValA,
44               int *csrRowPtrA, const int *csrColIndA,
45              int  *x, int  *y ){
46
47    int tid= (blockDim.x * blockIdx.x ) + threadIdx.x;
48    if(tid<*n){
49             int k;
50             y[tid]=0;
51             for(k=csrRowPtrA[tid]; k < csrRowPtrA[tid+1];k++){
52
53                y[tid]= y[tid]+ ((int)csrValA[k]*x[csrColIndA[k]]);
54
55             }
56
57    }
58
59
60
61
62
63
64 global__ void parentKernel( int * n,
65               double *csrValA,
66               int *csrRowPtrA, const int *csrColIndA,
67              int  *x, int  *y, int * valC ){
68
69
70 *valC=1;
71 int i;
72
73   while(*valC){
74       *valC=0;
75
```

```
76
77        SMVP<<<ceil(*n/256.0),256>>>(n,csrValA,csrRowPtrA,csrColIndA,x,y);
78        cudaDeviceSynchronize();
79
80 heckStatus<<<ceil(*n/256.0),256>>>(x,y,valC,n);
81        cudaDeviceSynchronize();
82
83    }
84
85
86
87
88
89
90
91
92
93 id checkStatus(cusparseStatus_t status){
94
95        if(CUSPARSE_STATUS_SUCCESS==status){
96                printf("the operation completed successfully \n");
97        }else if (CUSPARSE_STATUS_NOT_INITIALIZED==status){
98                printf("CUSPARSE_STATUS_NOT_INITIALIZED \n");
99        }else if(CUSPARSE_STATUS_ALLOC_FAILED==status){
100               printf("CUSPARSE_STATUS_ALLOC_FAILED\n");
101       }else if(CUSPARSE_STATUS_INVALID_VALUE==status){
102               printf("CUSPARSE_STATUS_INVALID_VALUE\n");
103       }else if(CUSPARSE_STATUS_ARCH_MISMATCH==status){
104               printf("CUSPARSE_STATUS_ARCH_MISMATCH\n");
105       }else if(CUSPARSE_STATUS_EXECUTION_FAILED==status){
106               printf("CUSPARSE_STATUS_EXECUTION_FAILED\n");
107       }else{
108               printf("CUSPARSE_STATUS_INTERNAL_ERROR\n");
109       }
110
111
112
113
114
115
116 t main(int arg,char** args){
117
118    int device=0;
119    int * finalLevel = (int*) malloc(sizeof(int));
```

```
120
121
122     int k=0;
123     int noOfRows,noOfCols,nnz;
124     int       n, i;
125
126
127
128
129     FILE* fileNew = fopen(args[1], "r");
130
131 Get values for nnz and n*/
132     fscanf(fileNew, "%d",finalLevel);
133     fscanf(fileNew, "%d %d %d",&noOfRows, &noOfCols, &nnz);
134
135     n= noOfRows;
136    // printf("No of cols %d\n",n);
137
138    cusparseStatus_t status1,status4,status5,status6;
139
140 ***********  Host Variables  ***************/
141 coo host variables
142    int * cooxIndexHostPtr=(int *)malloc(sizeof(int)*nnz);
143    int * cooyIndexHostPtr=(int *)malloc(sizeof(int)*nnz);
144    double * cooValHostPtr=(double *)malloc(sizeof(double)*nnz);
145
146 input host matrix varibles
147    int * xHost=(int *)malloc(sizeof(int)*n);
148
149
150 output host matrix variables
151    int * yHost=(int *)malloc(sizeof(int)*n);
152
153    int valHost=0;
154
155 **********   GPU variables***********/
156 coo gpu matrix variables
157    int * cooxIndexCuda=(int *)malloc(sizeof(int)*nnz);
158    int * cooyIndexCuda=(int *)malloc(sizeof(int)*nnz);
159    double * cooValCuda=(double*)malloc(sizeof(double)*nnz);
160
161 csr gpu matrix variables
162
163    int *     csrRowPtrACudaPtr;
```

```
164
165 csc gpu matrix variables
166   int *    cscColIndexACuda;
167   int *    cscRowPtrACudaPtr;
168   double * cscValACuda;
169 gpu input matrix
170   int * xCuda;
171   int * tmpCuda;
172 gpu output matrix
173   int * yCuda;
174   int * nCuda;
175   int * valCuda;
176
177 /   const double alpha = 1.0;
178 /   const double beta = 0.0;
179
180
181    int val;
182   while (k<nnz) {
183
184           fscanf(fileNew, "%d %d %d",&cooxIndexHostPtr[k], &cooyIndexHostPtr[k],&val);
185           cooValHostPtr[k] =(double)val;
186           k++;
187
188   }
189   /* may check feof here to make a difference between eof and io failure -- network
190      timeout for instance */
191
192
193
194   fclose(fileNew);
195
196   for (i = 0; i < n; ++i) {
197
198               xHost[i]=0;
199
200        }
201
202 t  startArrayCount;
203 ILE * vectorFile= fopen(args[2],"r");
204     fscanf(vectorFile,"%d",&startArrayCount);
205
206   //unified memory allocation for input vertice vector
207
```

```
208
209
210      int tempVal;
211   //asign values for input vector
212     for(i=0;i<startArrayCount;i++){
213
214                fscanf(vectorFile,"%d",&tempVal);
215                xHost[tempVal]=1 ;
216
217               // printf("%d ,",tempVal);
218       }
219
220
221
222
223 ********************************* CUDA stuff starts here ************************
224 udaDeviceProp prop;
225       cudaGetDeviceProperties(&prop, device);
226       cudaSetDevice(device);
227       fprintf(stderr,"Device name: %s\n", prop.name);
228
229 /start measuring time
230 udaEvent_t start,stop;
231 loat elapsedtime=0.0;
232 udaEventCreate(&start);
233 udaEventRecord(start,0);
234
235
236 malloc space on GPU
237
238    cudaMalloc((void**)&cooxIndexCuda,nnz*sizeof(int));
239    cudaMalloc((void**)&cooyIndexCuda,nnz*sizeof(int));
240    cudaMalloc((void**)&cooValCuda,nnz*sizeof(double));
241    cudaMalloc((void**)&csrRowPtrACudaPtr,  n*sizeof(int));
242    cudaMalloc((void**)&cscColIndexACuda,nnz*sizeof(int));
243    cudaMalloc((void**)&cscValACuda,nnz*sizeof(double));
244    cudaMalloc((void**)&cscRowPtrACudaPtr,  n*sizeof(int));
245    cudaMalloc((void**)&xCuda,n*sizeof(int));
246    cudaMalloc((void**)&yCuda,n*sizeof(int));
247    cudaMalloc((void**)&tmpCuda,n*sizeof(int));
248    cudaMalloc((void**)&nCuda,sizeof(int));
249    cudaMalloc((void**)&valCuda,sizeof(int));
250
251    checkCuda(cudaMemcpy(cooxIndexCuda,cooxIndexHostPtr,sizeof(int)*nnz,cudaMemcpyHo
```

```
252    checkCuda(cudaMemcpy(cooyIndexCuda,cooyIndexHostPtr,sizeof(int)*nnz,cudaMemcpyHostToDevice)
253    checkCuda(cudaMemcpy(cooValCuda,cooValHostPtr,sizeof(double)*nnz,cudaMemcpyHostToDevice));
254    checkCuda(cudaMemcpy(xCuda,xHost,sizeof(int)*n,cudaMemcpyHostToDevice));
255    checkCuda(cudaMemcpy(nCuda,&n,sizeof(int),cudaMemcpyHostToDevice));
256    checkCuda(cudaMemcpy(valCuda,&valHost,sizeof(int),cudaMemcpyHostToDevice));
257
258
259  /*************************Cu Sparse part **********************/
260
261 define the matrix features
262
263   cusparseOperation_t trans= CUSPARSE_OPERATION_NON_TRANSPOSE;
264   cusparseIndexBase_t idxBase = CUSPARSE_INDEX_BASE_ZERO;
265   cusparseAction_t copyValues= CUSPARSE_ACTION_NUMERIC;
266   cusparseHandle_t handle;
267   status4=cusparseCreate(&handle);//  checkStatus(status4);
268
269   cusparseMatDescr_t descrA ;
270   status5 =cusparseCreateMatDescr(&descrA); //checkStatus(status5);
271
272 *
273 sparseStatus_t
274 sparseXcoo2csr(cusparseHandle_t handle, const int *cooRowInd,
275                int nnz, int m, int *csrRowPtr, cusparseIndexBase_t idxBase)
276
277
278 ad more at: http://docs.nvidia.com/cuda/cusparse/index.html#ixzz46AdjmqnI
279 llow us: @GPUComputing on Twitter | NVIDIA on Facebook
280 /
281
282 atus6= cusparseXcoo2csr(handle,cooxIndexCuda,nnz,n,csrRowPtrACudaPtr,idxBase);
283 checkStatus(status6);
284
285
286 cusparseStatus_t
287 sparseDcsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
288                const double *csrVal, const int *csrRowPtr,
289                const int *csrColInd, double           *cscVal,
290                int *cscRowInd, int *cscColPtr,
291                cusparseAction_t copyValues,
292                cusparseIndexBase_t idxBase)
293
294
295 ad more at: http://docs.nvidia.com/cuda/cusparse/index.html#ixzz456mkjYNe
```

```
llow us: @GPUComputing on Twitter | NVIDIA on Facebook



               status1 =   cusparseDcsr2csc(handle,n,n,nnz,cooValCuda,csrRowPtrACud

              //checkStatus(status1);



  /*
  cusparseStatus_t
sparseDcsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
             int m, int n, int nnz, const double          *alpha,
             const cusparseMatDescr_t descrA,
             const double          *csrValA,
             const int *csrRowPtrA, const int *csrColIndA,
             const double          *x, const double          *beta,
             double          *y)



global__ void parentKernel( int * n,
              double *csrValA,
              int *csrRowPtrA, const int *csrColIndA,
             int  *x, int  *y, int * valC ){


             cusparseDcsrmv(handle,trans,n,n,nnz,&alpha,descrA,cscValACuda,cscRowPt
                checkCuda(cudaMemcpy(xHost,xCuda,sizeof(double)*n,cudaMemcpyDeviceT

  */



  parentKernel<<<1,1>>>(nCuda,cscValACuda,cscRowPtrACudaPtr, cscColIndexACuda,xCuda
  cudaDeviceSynchronize();
  checkCuda(cudaMemcpy(yHost,xCuda,sizeof(int)*n,cudaMemcpyDeviceToHost));



       //free the cuda memory
     cudaFree(cooxIndexCuda);
     cudaFree(cooyIndexCuda);
```

```
340       cudaFree(cooValCuda);
341       cudaFree(csrRowPtrACudaPtr);
342       cudaFree(cscColIndexACuda);
343       cudaFree(cscValACuda);
344       cudaFree(cscRowPtrACudaPtr);
345       cudaFree(xCuda);
346       cudaFree(yCuda);
347       cudaFree(tmpCuda);
348
349
350       //end measuring time
351       cudaEventCreate(&stop);
352       cudaEventRecord(stop,0);
353       cudaEventSynchronize(stop);
354       cudaEventElapsedTime(&elapsedtime,start,stop);
355
356       //print the output yHost
357
358   //print the answer : CUBLAS gives the answer placed in column major order
359       //printf("Answer : \n");
360
361       /*for(i=0;i<n;i++){
362               printf("%f ",yHost[i]);
363
364       }  */
365
366
367       //print the time spent to stderr
368
369       printf("%d, %d , %1.5f \n",n, nnz,elapsedtime/(float)1000);
370
371
372
373       free(cooxIndexHostPtr);
374       free(cooyIndexHostPtr);
375       free(cooValHostPtr);
376       free(xHost);
377       free(yHost);
378
379       return 0;
380
```

B.2. CPU implementation of H-BFT algorithm in C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>


struct Edge{

   int from ;
   int to;
};



int isLevelFilled(int * level, int * vertices){
     int i;
     for(i=0;i<*vertices;i++){
        if(level[i]==-1){
            return 1;

        }

     }

     return 0;


}

void BreadthFirstSearch(struct Edge * adjacencyList,int * level,  int * edg
        int k;
        struct Edge element;
        for(k=0;k<*edges;k++){
                 element = adjacencyList[k];
                   if ((level[element.from]>=0)&&(level[element.to]==-1)){
                          level[element.to] = level[element.from]+1;
                   }

        }

```

```
41
42          }
43
44          int main(int arg,char** args){
45
46
47
48
49
50
51               int noOfRows;
52               int   i;
53               int v1,v2;
54               int count = 1;
55
56               int finalLevel;
57           //----------------------unit host variables----------==========================
58
59
60
61
62
63
64              int * vertices=(int *) malloc(sizeof(int));
65              int * edges=(int *) malloc(sizeof(int));
66              int * startArrayCount = (int *) malloc(sizeof(int));
67
68          //-----------------------------------------------------------------
69               FILE* fileNew = fopen(args[1], "r");
70
71               fscanf(fileNew, "%d",&finalLevel);
72               fscanf(fileNew, "%d %d %d",&noOfRows, vertices, edges);
73              // printf("No fo rows %d, No of Cols %d, nnz %d \n",noOfRows,*vertices,*edges);
74
75
76
77
78             int * level= (int *)malloc(sizeof(int)*(*vertices));
79
80             struct Edge * edgeList =(struct Edge * )malloc(sizeof(struct Edge)*(*edges));
81
82          //----------------------------------------creating host graph structure array
83
84
```

```
85          for (i = 0; i < *vertices; ++i) {
86                      level[i] = -1;
87          }
88          int val;
89           for (i = 0; i < *edges; ++i) {
90
91                       fscanf(fileNew, "%d %d %d",&v1, &v2, &val);
92
93                      // Adding edge v1 --> v2
94                       edgeList[i].from = v1;
95                       edgeList[i].to = v2;
96
97           }
98
99
100
101        //---------------------------------------input nodes...
102
103          FILE * vectorFile= fopen(args[2],"r");
104          fscanf(vectorFile,"%d",startArrayCount);
105
106        //---------------------------------host input vector-----==================
107
108             int tempVal;
109
110             for(i=0;i<*startArrayCount;i++){
111
112                       fscanf(vectorFile,"%d",&tempVal);
113                       level[tempVal]=0;
114
115               }
116
117            //starting the clock
118               clock_t begin, end;
119               double time_spent;
120                         begin = clock();
121
122          //---------------------------------test print Graph and input nodes----
123
124
125          while(isLevelFilled(level,vertices)){
126              BreadthFirstSearch(edgeList,level,edges);
127              count ++;
128          }
```

```
129
130
131
132         //------------------------------------------------------------------------------
133
134
135
136
137             end = clock();
138             //taking end time of above opration
139
140
141          time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
142
143          printf("%d, %d, %lf \n",*vertices,*edges,time_spent);
144
145          free(level);
146          free(edgeList);
147          free(vertices);
148          free(edges);
149          free(startArrayCount);
150
151         return 0;
152
153     }
```

B.3. GPU implementation of H-BFT using dynamic parallelism

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "helpers.cuh"
4
5
6   struct Edge{
7
8      int from ;
9      int to;
10  };
11
12
13
```

```
14    __global__ void isLevelFilled(int * level, int * vertices, int * lev){
15
16         int j = (blockDim.x * blockIdx.x ) + threadIdx.x;
17
18            if(level[j]==-1 && *lev==0){
19                *lev=1;
20
21            }
22
23
24
25
26    }
27
28
29    __global__ void BreadthFirstSearch( struct Edge * adjacencyList, int * vertices
30
31                int tid = (blockDim.x * blockIdx.x ) + threadIdx.x;
32                *lev = 0;
33                if(tid<*edges){
34
35                     struct Edge element = adjacencyList[tid];
36                      if (level[element.from]>=0 and level[element.to]==-1){
37                              level[element.to] = level[element.from]+1;
38                      }
39
40                }
41
42
43    }
44
45
46    __global__ void parentKenel(struct Edge * adjacencyList, int * vertices, int *
47
48           *lev=1;
49
50            int kb=0;
51             while(*lev){
52              //  printf("kernel itteratin %d \n",kb);
53                kb = kb+1;
54                 BreadthFirstSearch<<<ceil(*edges/256.0),256>>> (adjacencyList,vert
55                 cudaDeviceSynchronize();
56                 isLevelFilled<<<ceil(*vertices/256.0),256>>>(level,vertices,lev);
57                 cudaDeviceSynchronize();
```

```
58              }
59              printf("kernel itteratin %d \n",kb);
60
61
62
63      }
64
65        int max_array(int a[], int num_elements){
66           int i, max=-1;
67           for (i=0; i<num_elements; i++){
68             if (a[i]>max) {
69                max=a[i];
70             }
71           }
72           return(max);
73        }
74
75        int main(int arg,char** args){
76
77        //cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
78        cudaDeviceSetCacheConfig(cudaFuncCachePreferShared);
79
80                int device=0;
81             cudaDeviceProp prop;
82                cudaGetDeviceProperties(&prop, device);
83                cudaSetDevice(device);
84                fprintf(stderr,"Device name: %s\n", prop.name);
85
86
87
88
89
90              int noOfRows;
91              int   i;
92              int v1,v2;
93           int finalLevel;
94            int * Hvertices=(int *) malloc(sizeof(int));
95            int * Hedges=(int *) malloc(sizeof(int));
96            int * Hlev =(int *)malloc(sizeof(int));
97            int * HstartArrayCount = (int *) malloc(sizeof(int));
98
99        //-------------------------------------------------------------
100            FILE* fileNew = fopen(args[1], "r");
101
```

```
102             fscanf(fileNew, "%d",&finalLevel);
103             fscanf(fileNew, "%d %d %d",&noOfRows, Hvertices, Hedges);
104            // printf("No fo rows %d, No of Cols %d, nnz %d \n",noOfRows,*Hvertices,*
105
106
107
108
109         //************  Host Variables  ***************==================================
110
111            int * Hlevel= (int *)malloc(sizeof(int)*(*Hvertices));
112           struct Edge * HedgeList =(struct Edge * )malloc(sizeof(struct Edge)*(*Hedge
113
114         //-----------------------------------------creating host graph structure arr
115
116           *Hlev = 0;
117
118           for (i = 0; i < *Hvertices; ++i) {
119
120
121                       Hlevel[i] = -1;
122
123
124
125           }
126             int val;
127            for (i = 0; i < *Hedges; ++i) {
128
129                        fscanf(fileNew, "%d %d %d",&v1, &v2, &val);
130
131                       // Adding edge v1 --> v2
132                        HedgeList[i].from = v1;
133                        HedgeList[i].to = v2;
134
135            }
136
137
138
139         //---------------------------------------input nodes...
140
141            FILE * vectorFile= fopen(args[2],"r");
142            fscanf(vectorFile,"%d",HstartArrayCount);
143
144         //--------------------------------host input vector-----==================
145                int tempVal;
```

```
146             for(i=0;i<*HstartArrayCount;i++){
147
148                     fscanf(vectorFile,"%d",&tempVal);
149
150                     Hlevel[tempVal]=0;
151
152               }
153
154
155         /*****************************************.........Start Timimg .........**********
156                cudaEvent_t start,stop;
157             float elapsedtime;
158             cudaEventCreate(&start);
159             cudaEventRecord(start,0);
160
161
162
163         //-----------------------------------test print Graph and input nodes-----------
164
165
166             int * Dvertices;
167             int * Dedges;
168             int * Dlev ;
169             int * DstartArrayCount ;
170             int * Dlevel;
171             struct Edge * DedgeList ;
172
173
174
175         //-------------------------------------allocate memory in the GPU device
176             checkCuda(cudaMalloc((void **)&Dvertices,sizeof(int)));
177             checkCuda(cudaMalloc((void **)&Dedges,sizeof(int)));
178             checkCuda(cudaMalloc((void **)&Dlev,sizeof(int)));
179             checkCuda(cudaMalloc((void **)&DstartArrayCount,sizeof(int)));
180             checkCuda(cudaMalloc((void **)&Dlevel,sizeof(int)*(*Hvertices)));
181             checkCuda(cudaMalloc((void **)&DedgeList,sizeof(struct Edge)* (*Hedges)));
182
183         //-------------------------------------cpy memory from device to host
184             checkCuda(cudaMemcpyAsync(Dvertices,Hvertices,sizeof(int),cudaMemcpyHostToDevice));
185             checkCuda(cudaMemcpyAsync(Dedges,Hedges,sizeof(int),cudaMemcpyHostToDevice));
186             checkCuda(cudaMemcpyAsync(Dlev,Hlev,sizeof(int),cudaMemcpyHostToDevice));
187             checkCuda(cudaMemcpyAsync(DstartArrayCount,HstartArrayCount,sizeof(int),cudaMemcpyHo
188             checkCuda(cudaMemcpyAsync(Dlevel,Hlevel,sizeof(int)*(*Hvertices),cudaMemcpyHostToDev
189             checkCuda(cudaMemcpyAsync(DedgeList,HedgeList,sizeof(struct Edge)*(*Hedges),cudaMemc
```

```
190
191         //-----------------------------------------------kernel calls starts here...
192                 printf("before the kernel \n");
193
194             parentKenel<<<1,1>>>(DedgeList,Dvertices,Dlevel,Dlev,Dedges);
195             cudaDeviceSynchronize();
196             checkCudaError();
197                 checkCuda(cudaMemcpy(Hlevel,Dlevel,sizeof(int)*(*Hvertices),cudaMemcp
198
199
200
201
202
203
204
205         //---------------------------------------free memory device---------
206
207                 cudaFree(Dvertices);
208                 cudaFree(Dedges);
209                 cudaFree(Dlev);
210                 cudaFree(DstartArrayCount );
211                 cudaFree(Dlevel);
212                 cudaFree(DedgeList);
213
214                 cudaEventCreate(&stop);
215             cudaEventRecord(stop,0);
216             cudaEventSynchronize(stop);
217             cudaEventElapsedTime(&elapsedtime,start,stop);
218             //fprintf(stderr,"Time spent for kernel : %.10f seconds\n",elapsedtime/(f
219
220                printf("%d, %d, %.8f \n",*Hvertices, *Hedges,elapsedtime/(float)1000);
221
222         //---------------------------------------free cpu memory--------------------
223
224                 free(Hvertices);
225                 free(Hedges);
226                 free(Hlev);
227                 free(HstartArrayCount );
228                 free(Hlevel);
229                 free(HedgeList);
230
231
232
233                 return 0;
```

```
234         }
```