H-BFT: A Fast Breadth-First Traversal Algorithm for Sparse Graphs and its GPU Implementation

Dinali R. Dabarera, Himesh Karunarathna, Erandika Harshani and Roshan G. Ragel
Department of Computer Engineering
Faculty of Engineering
University of Peradeniya, Sri Lanka
Email:gdrdabarera@gmail.com, himeshsameera@gmail.com, harshanierandikanr@gmail.com, ragelrg@gmail.com

***Abstract*—With Moore's law in effect, as the complexity of digital electronic circuits increases, the amount of time spent 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 fulfil 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.**

In this paper, we present a new simple, fast and parallelizable BFT algorithm for sparse graphs, 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 180x and 25x for the SMVP-BFT and H-BFT respectively.

I. Introduction

With the doubling of transistors in the electronic circuits in every two years, the complexity of digital electronic circuits also increases day by day. This is one of the greatest challenges faced by the EDA (Electronic Design Automation) Industry. An EDA tool is a software, 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 the circuits. There are four major steps in Electronic Design Automation. They are placement, routing, optimization and post silicon validation. These steps are intrinsically difficult. For such situations, 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 [1]. In this paper, we mainly focus on one of the most common graph algorithms, Breath-First Search, which is useful in EDA as mentioned in [1]. With the increase of the number of gates in circuits, the time taken for the circuit graph traversals also increases. This problem has become the main bottleneck in EDA [2].

There are many methods of increasing performance of EDA graph traversals. Some of them are hardware accelerations [3], use of server farms [4] and high performance computers, which costs a lot for a single EDA tool user. Therefore, use of GPU (Graphical Processing Unit) is the cheapest method of improving performance of EDA tools, which means even a circuit designer can use the graphic card of his laptop to improve the designing process. Due to the SIMD architecture of these GPUs, modern GPUs are more efficient at manipulating computer graphics, image processing and parallel computations than CPUs. At present, a GPU is found in a personal computer as a video card or embedded into its motherboard [5]. The word "GPU" was popularized by NVIDIA Corporation. With the introduction of GeForce 8 Series by NVIDIA, GPUs have become the main mode of computations against CPU [5]. It also has become a field of research called General Purpose Computing on GPU (GPGPU). The latest release of NVIDIA, "Kepler" has the worlds fastest, most efficient HPC architecture which helps in high performance computing as mentioned in [6]. There are many latest features such as unified memory architecture and dynamic parallelism that newly added to the Kepler architecture. Through this paper we will provide a better and a faster solution for the Breath-First Traversal with the use of these new features.

The rest of the paper is organized as follows: In Section II, we discuss the related work and in Section III we present the background. Then in Section IV, we describe our algorithm, the H-BFS with it's design and the implementation on both CPU and GPU. In Section V, we provide our experimental results and it's analysis. Finally we conclude in Section VI.

II. Related Work

Bell et al. had studied different data structures and algorithms for sparse matrix vector product implementation on the CUDA platform in [7], in 2008. In their study, it is mentioned that, grid-based matrix structure has occupied a performance of 36 GFLOP/s in single precision and 16 GFLOP/s in double precision on a GeForce GTX 280 GPU, while an unstructured finite-element matrix structure has occupied a performance in excess of 15 GFLOP/s and 10 GFLOP/s in single and double precision respectively.

As in [8], Bell et al. had done another research work on implementing sparse matrix vector multiplication process on throughput oriented processors. They 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 lot of computations. They also conclude that in order to effectively utilize the GPU resources, the kernels needed to have a fine-grain parallelism.

Deng et al. have done a research on increasing performance of EDA tool algorithms on GPU [9], which also describes some important irregular EDA computing patterns of Sparse Matrix Vector Product (SMVP). They had considerably ac-

978-1-5090-6132-7/16/$31.00 2016 IEEE

celerated a SMVP based formulation of Breadth-First Search (BFS) using CUDA to get a speedup up of 10X.

One of the main GPU based EDA tool provider Rocketick, which was recently acquired by Cadence has introduced a product called RocketSim in 2011 [10]. It is a software based solution, which is installed on standard servers and accelerates leading simulators such as incisive, VCS and ModelSim. RocketSim solves the functional verification bottlenecks in chip designing by offloading most time-consuming calculations to an ultra-fast GPU based engine. It 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.

A Breadth-First Search parallelization focused on fine grain task management is described on a research done by Merrill et al. in 2011 [11]. It has achieved an asymptotically optimal O(|V |+ |E |) work complexity. Busato et al. had implemented an efficient algorithm for Breadth First Search using Kepler GPU architectures in [12]. Their implementation of BFS has achieved an optimum work complexity.

Luo et al. have proposed an effective implementation of Breadth-First Search on GPU in [13]. They have used a hierarchical queue management technique and a three-layer kernel arrangement strategy. The experimental results have achieved up to 10 times speedup over a classical fast CPU implementation. This method is mostly suitable for accelerating sparse graphs, which are widely seen in the field of EDA.

From a survey [14] done by Deng 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 the use of some GPU libraries such as CUBLAS [15], MAGMA[16] and CULA [16]. 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.

Harish et al. have worked on accelerating large graph algorithms on GPU in [17]. This provides a faster solution 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.

According to all these, we can see that many have tried to improve the performance of Breadth-First Search algorithm which is in the form of Sparse Matrix Vector Product using GPU.

III. Background - Breadth First Traversal using Sparse Matrix Vector Product

Breadth-First Search is an algorithm 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. Breadth-First Search is the main graph traversal algorithm which has become the basis for many higher level analysis graph algorithms according to [11]. As in [9], 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. As many research work has been done on the Sparse Matrix Vector Product (SMVP) Implementation of Breadth-First Traversal (BFT), we built our reference model with the SMVP Implementation of BFT.

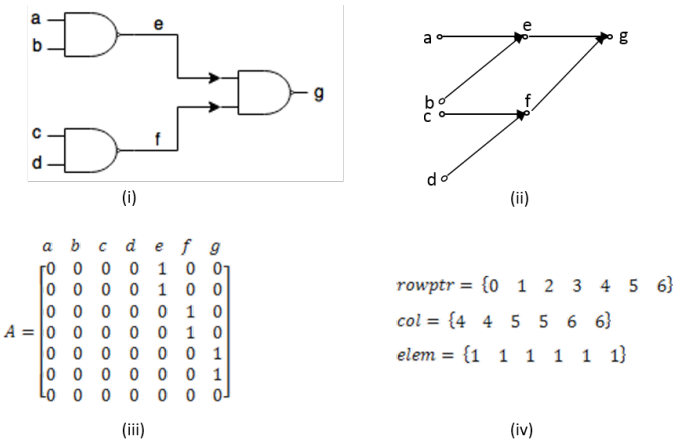

Fig. 1. Sparse Matrix and Compressed Row Format, i) Simple circuit, ii) Directed graph corresponds to i, iii) Sparse matrix of ii, iv) Compressed Row Format of iii

A. Basic Design of SMVP-BFT

EDA tools are used to design large electronic gate circuits. The circuits that are used by EDA tools can be easily interpreted as graphs, and these graphs can be stored as sparse matrices as shown in Fig. 1. There are about six ways of representing a sparse matrix [18]. They are Compressed Sparse Row (CSR) Format, Compressed Sparse Column (CSC) Format, Coordinate (COO) Format, Diagonal Format, ELLPACK format and Packet format. These formats are mainly used to save extra memory allocation for zero elements.

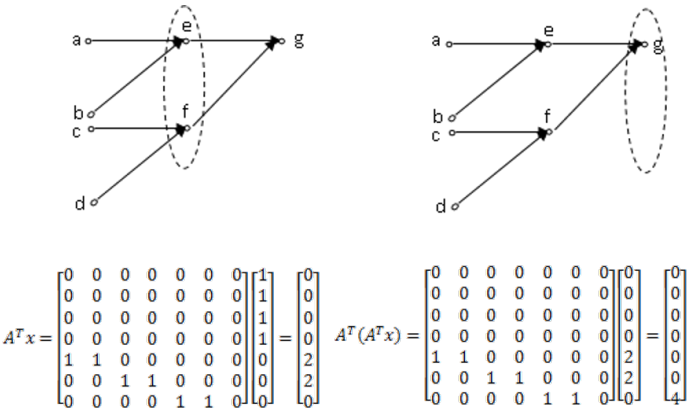

Fig. 2. BFT using Sparse Matrix Vector Multiplication

According to the related work in Section II, as shown in Fig. 2 we got to know that many have tried the Sparse Matrix Vector Product (SMVP) Implementation, in different ways. It clearly depicts how the Breadth First Search is used to traverse through the circuit. A^T is the transpose matrix of A, which is the adjacency matrix of the circuit graph and x is a vector which is the inputs to the circuit. This can have one for input pins and zero for other pins. In this example shown in Fig. 2, all three inputs were considered to be 1 and others to be

0. Each time when A^T is multiplied with the vector of the current state, the vector of the next state can be taken as the output. We can continue this iterative process until the output vector becomes a zero vector, which is our stopping condition. Therefore, through this method, Breadth-First Traversal can be easily achieved, since the output of each iteration will give the nodes/pin which are high in the same level. From this we can clearly identify the states of a electronic circuit at each timing intervals. Furthermore, this is a level by level traversal through the graph circuit using SMVP. If the number of intermediate levels of the graph is L, the SMVP-BFT can be expressed as $A^{'}..(A^{'}(A^{'} * x))$, L times until the output vector becomes a zero vector, here $A^{'} = A^T$.

```
int  *smvp_Mul(int *yHost,int *xHost, int *cscValHost,
  *cscRowPtrAHostPtr, int *cscColIndexAHost,int n){
  int k=0,i=0;
  for (i = 0; i < n; i++){

  int row_start= cscRowPtrAHostPtr[i];
  int row_end= cscRowPtrAHostPtr[i + 1];
  for (k= row_start; k< row_end; k++){
       yHost[i] +=
          cscValHost[k] * [cscColIndexAHost[k]];
  }
 }
return yHost;
}
```

Listing 1: SMVP multiplication in C

B. Implementation of SMVP-BFT

We implemented the above algorithm in two ways as follows as CPU implementation and GPU implementation. The CPU version is a single threaded implementation. But for the GPU version we have used dynamic parallelism feature of Kepler GPU architecture [6]. The inputs to the SMVP-BFT are the edges of the graph and the list of input vertices to the graph circuit.

1) Serial Implementation: In the serial implementation we load the data of the edges of the graph and the input vertices into four integer vectors called $cscValHost$, $cscRowPtrAHostPtr$, $cscColIndexAHost$ and $xHost$ and do the iterative multiplication as shown in Listing 1.

In order to stop this iterative function $smvp\_Mul$, each time we are checking the output vector to verify whether it is a zero vector or not. If it is a zero vector we stop the multiplication process since we have achieved the final state.

```
__global__ void parentKernel( int * n,double *csrValA,
int *csrRowPtrA,const int *csrColIndA, int  *x,
int  *y, int * valC ){
  *valC=1;
  int i;
  int count =0;
  while(*valC){
     *valC=0;
     SMVP<<<ceil(*n/256.0),256>>>
          (n,csrValA,csrRowPtrA,csrColIndA,x,y);
     cudaDeviceSynchronize();

     count = count +1;
     checkStatus<<<ceil(*n/256.0),256>>>
                          (x,y,valC,n);
     cudaDeviceSynchronize();
     }
}
```

Listing 2: SMVP-BFT parent kernel where Dynamic Parallelism called.

2) Parallel GPU Implementation: Here we have used the same SMVP-BFT method with the use of GPU threads. From the CUDA occupancy calculator [19], we calculated that one block should contain 256 threads and the number of blocks depends on the number of vertices, n. Therefore, the number of blocks needed is $ceil(n/256.0)$. Calling GPU kernels from CPU in each iteration cause a lot of overhead and slow down the execution process. To avoid this overhead, NVIDIA has introduced dynamic parallelism in the Kepler architecture [6]. As shown in Listing 2, we use this new feature and created a parent kernel which is launched by a single thread in a single block. It launches other multiplication kernels in a loop.

As in Listing 3, a separate kernel was called inside the parent kernel to do the SMVP using GPU threads . Since this kernel is called inside the device itself, it make use of dynamic parallelism, which gives less overhead than calling from CPU.

```
__global__ void SMVP( int*  n, double *csrValA,
  int *csrRowPtrA,int *csrColIndA,int  *x, int  *y ){

  int tid= (blockDim.x * blockIdx.x ) +
  threadIdx.x;
  if(tid<*n){
       int k;
       y[tid]=0;
       for(k=csrRowPtrA[tid];k < csrRowPtrA[tid+1];k++){

         y[tid]= y[tid]+
                  ((int)csrValA[k]*x[csrColIndA[k]]);
       }
 }  }
```

Listing 3: SMVP Multiplication Kernel

IV. 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.

A. 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. 3.

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 4, we go through the GraphList (the array of Edge structures) one by one and check for the level

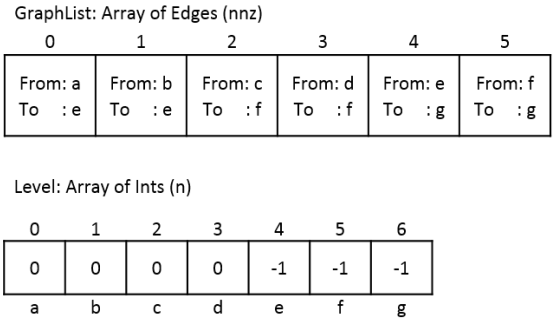

Fig. 3. GraphList and level variables for the example circuit in Fig.1

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 4: Traversal in H-BFT

B. 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 5 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 5: parent kernel of H-BFT in CUDA which uses dynamic parallelism

As in Listing 6, $BreadthFirstSearch$ kernel is launched with nnz number of threads on GPU. The maximum block size of 256 is selected from the CUDA occupancy calculator [19] 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 5, 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 6: BreadthFirstSearch kernel of H-BFT in CUDA

V. Experimental Results and Analysis

A. Experimental Setup

The CPU implementation of both SMVP-BFT algorithms and H-BFT, which were single threaded versions were tested on a server with a 6th gen Intel i7 (6700K) processor and 32GB RAM. Their GPU implementations were tested on an NVIDIA Kepler K40 GPU card having 2880 CUDA cores and 12GB of dedicated RAM.

The graphs that we generated consist of constant number of vertices (n) and different number of edges (nnz). The EDA tools work normally work with really large circuits which has millions of gates. Therefore, we can achieve better performance by using dynamic parallelism on large circuit graphs. Due to this, we chose n as 10^6 and nnz in the range of 10^7 to 10^8 at constant intervals. We generated three sets of graphs, which fulfills three different datasets such as dataset A, dataset B and dataset C. In dataset A, the edges in the GraphList are in order of their levels from the input vertices as shown in Fig. 3 . Meanwhile in dataset B, the edges of the GraphList are in the reverse order, which means the edges of last level to the edges of the first. In the dataset C, the edges in the GraphList are placed in a random order without considering the levels of the graph.

B. GPU Speedup compared to CPU

We tested both the algorithms (SMVP-BFT and H-BFT) on the above test bench with the three sets of data. We executed one algorithm on single dataset for three times and got the average execution time since their standard deviation is very small.

1) SMVP-BFT Algorithm: According to the results we got, for all types of datasets it gave the similar execution time for a particular 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. 4 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.

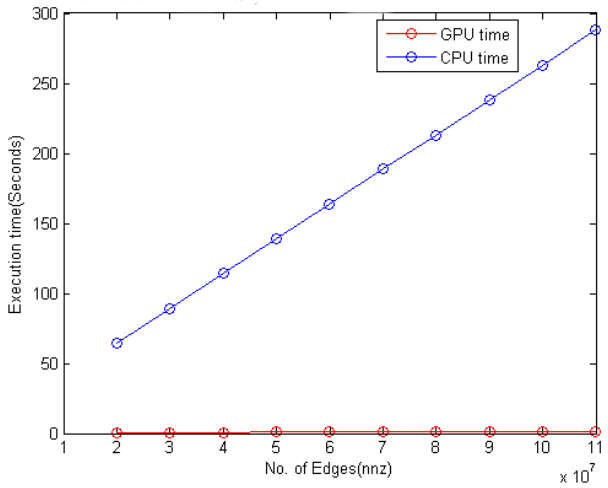

Fig. 4. Average Execution time of SMVP-BFT

From the Fig. 5, it can be seen that SMVP-BFT CUDA version has achieved a maximum speedup of 180X over its CPU implementation. Here speedup is the ratio between execution time on GPU and execution time on CPU. The reason for the speed up should mainly be attributed to the dynamic parallelism we used in our GPU implementation.

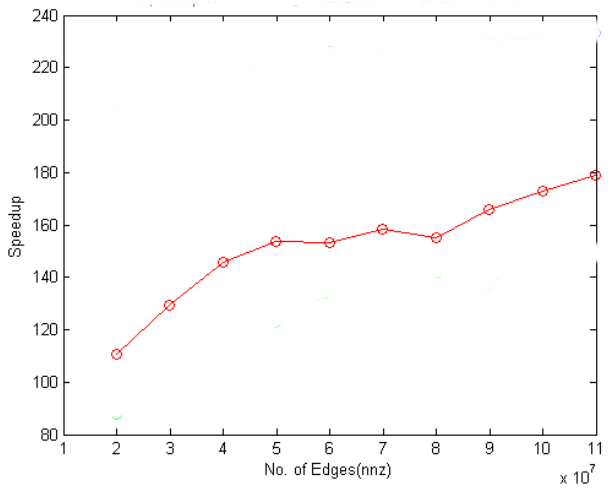

Fig. 5. Average Speedup of SMVP-BFT

2) H-BFT: Fig. 6 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 is getting larger, the time taken by the CPU to execute a serial H-BFT also increases. However, since the edges are in order, the CPU version will complete the graph traversal in a single iteration, while the GPU version takes more than one. According to Fig. 7, 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 at last. Therefore, to fill the "Level" array, both the CPU and the GPU versions take more than one iterations. As seen in Fig. 8, in dataset C the gap between CPU and GPU is constant, meanwhile with the increase of nnz the CPU time increases.

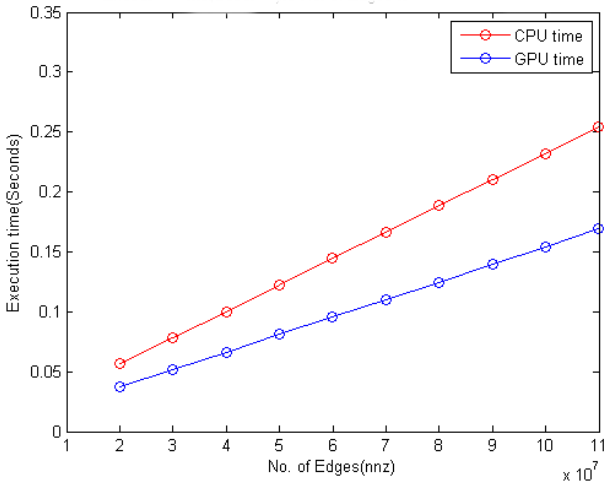

Fig. 6. Execution time of H-BFT for dataset A

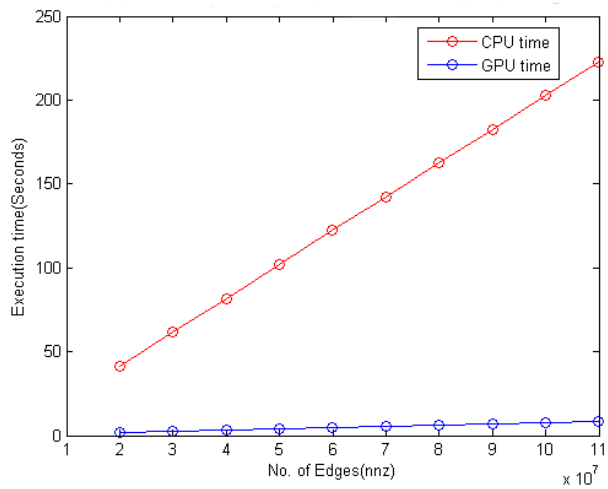

Fig. 7. Execution time of H-BFT for dataset B

When considering the speedup of GPU as depicts in Fig. 9, for dataset B, H-BFT has achieved the highest speedup of 25X 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.

3) SMVP-BFT vs. H-BFT: Table I shows the execution time taken by all the algorithms when nnz=10^8. This results is tabulated to compare the performance between the improved

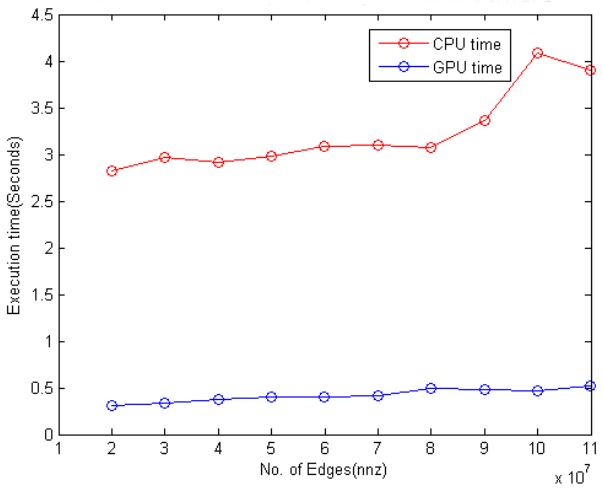

Fig. 8. Execution time of H-BFT for dataset C

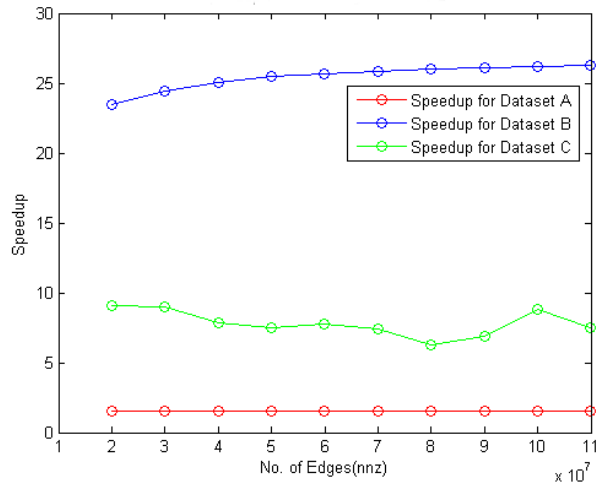

Fig. 9. Speedup of H-BFT for all datasets

SMVP-BFT and our new H-BFT implementations. In the CPU, H-BFT is faster than the SMVP-BFT. For the dataset considered, on CPU, for the average case, H-BFT is about 75X faster than the improved version of the state of the art SMVP-BFT. For the same dataset, on GPU, H-BFT is about 3X faster than the improved version of the state of the art SMVP-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 I
EXECUTION TIME TAKEN BY ALL THE ALGORITHMS WHEN NNZ=10^8

VI. CONCLUSION

In this paper, we have taken the state of the art Breadth First Traversal (BFT) implementation on GPU and improved its performance by using dynamic parallelism. Further, we have proposed a new algorithm (we call it H-BFT), which is much faster. For the dataset we have selected, we have shown that the H-BFT CPU version is about 75X faster on average compared to the best-known CPU implementation of the BFT. Further, the GPU implementation of H-BFT is shown to be 3X faster than its CPU version. The H-BFT's speedup in the CPU is achieved via the careful selection of the right data structures for the sparse graph representation and in the GPU is achieved by the dynamic parallelism feature available in the modern GPUs.

REFERENCES

[1] Y. Deng and S. Mu, "Electronic design automation with graphic processors: A survey," *Foundations and Trends in Electronic Design Automation*, vol. 7, no. 12, pp. 1–176, 2013. [Online]. Available: http://dx.doi.org/10.1561/1000000028
[2] T. Karn, S. Rawat, D. Kirkpatrick, R. Roy, G. S. Spirakis, N. Sherwani, and C. Peterson, "Eda challenges facing future microprocessor design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1498–1506, 2000.
[3] K. Gulati and S. P. Khatri, *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*, 1st ed. Springer Publishing Company, Incorporated, 2010.
[4] TechTarget. (1999-2016) server farm (web farm, web server farm). [Online]. Available: http://whatis.techtarget.com/definition/server-farm-Web-farm-Web-server-farm
[5] N. Corporation. (2016) Introducing the new nvidia pascal architecture. [Online]. Available: http://www.nvidia.com/object/gpu-architecture.html
[6] ——. (2016) Kepler the world's fastest, most efficient hpc architecture. [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html
[7] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.
[8] ——, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.
[9] Y. S. Deng, B. D. Wang, and S. Mu, "Taming irregular eda applications on gpus," in *Proceedings of the 2009 International Conference on Computer-Aided Design*. ACM, 2009, pp. 539–546.
[10] Rocketic. Rocketsim. [Online]. Available: http://www.rocketick.com/rocketsim/rocketsim
[11] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable gpu graph traversal," *ACM Transactions on Parallel Computing*, vol. 1, no. 2, p. 14, 2015.
[12] F. Busato and N. Bombieri, "Bfs-4k: an efficient implementation of bfs for kepler gpu architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1826–1838, 2015.
[13] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th design automation conference*. ACM, 2010, pp. 52–55.
[14] Y. Deng and S. Mu, *Electronic Design Automation with Graphic Processors: A Survey*. Now Publishers Incorporated, 2013.
[15] N. Corporation. (2016) cublas. [Online]. Available: https://developer.nvidia.com/cublas
[16] ——. (2016) Magma. [Online]. Available: https://developer.nvidia.com/magma
[17] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High performance computing–HiPC 2007*. Springer, 2007, pp. 197–208.
[18] N. Corporation. (2007-2015) cusparse- cuda toolkit documentation. [Online]. Available: http://docs.nvidia.com/cuda/cusparse/#matrix-formats
[19] lxkarthi. (2007) cuda-calculator. [Online]. Available: https://github.com/lxkarthi/cuda-calculator