Taming Irregular EDA Applications on GPUs

Yangdong (Steve) Deng
Institute of Microelectronics
Tsinghua University
Beijing, 100084, China
86-10-62771733
dengyd@tsinghua.edu.cn

Bo David Wang
Institute of Microelectronics
Tsinghua University
Beijing, 100084, China
86-10-62784684
david.b.wang@gmail.com

Shuai Mu
Institute of Microelectronics
Tsinghua University
Beijing, 100084, China
86-10-62784684
mus04ster@gmail.com

ABSTRACT

Recently general purpose computing on graphic processing units (GPUs) is rising as an exciting new trend in high-performance computing. Thus it is appealing to study the potential of GPU for Electronic Design Automation (EDA) applications. However, EDA generally involves irregular data structures such as sparse matrix and graph operations, which pose significant challenges for efficient GPU implementations. In this paper, we propose high-performance GPU implementations for two important irregular EDA computing patterns, Sparse-Matrix Vector Product (SMVP) and graph traversal. On a wide range of EDA problem instances, our SMVP implementations outperform all published work and achieve a speedup of one order of magnitude over the CPU baseline. Upon such a basis, both timing analysis and linear system solution can be considerably accelerated. We also introduce a SMVP based formulation for Breadth-First Search and observe considerable speedup on GPU implementations. Our results suggest that the power of GPU computing can be successfully unleashed through designing GPU-friendly algorithms and/or re-organizing computing structures of current algorithms.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD). D.1.3 [**Concurrent Programming**]: Parallel Programming.

General Terms

Algorithms, Design, Languages.

Keywords

GPU, CUDA, EDA, sparse matrix, sparse-matrix vector product, static timing analysis, graph algorithms, breadth-first search, conjugate gradient, placement, data parallel computing.

1. INTRODUCTION

Due to the relative saturation of single-CPU performance, multi-core processors are inevitably becoming the dominant computing resources for EDA applications. Recently, general purpose computing on graphic processing units (GPGPU) has become a very important trend of high performance computing [1]. Unlike multi-core CPUs that generally exploit task level parallelism, graphic processing units (GPUs) utilize a data parallel programming model. Upon receiving a workload, a GPU would launch tens of thousands of fine-grain threads concurrently, with each thread executing the same program but on a different data set. Modern GPUs could deliver a very high computing throughput. For example, NVidia's flagship GPU, GT280, could reach a peak floating-point throughput higher than the latest CPU by a factor of 30. On workloads with appropriate computing and memory access patterns, GPU could even attain a speedup of over 100X. Meanwhile, GPU programming has been made much more accessible to non-graphic programmers with the introduction of NVidia's Compute Unified Device Architecture (CUDA) technology [2].

Like many other software communities, Electronic Design Automation (EDA) also needs an overhaul of parallelization so as to keep pace with the ever increasing VLSI complexity. It is thus appealing to unleash the computing power of GPU for EDA applications. There are already a few papers [e.g. 3-5] presenting encouraging results on utilizing GPU to solve specific EDA problems. However, a comprehensive evaluation on the potential of GPUs for EDA computing is still needed, because EDA applications mainly depend on irregular data structures that are less amenable to GPUs.

The irregular data access patterns are determined by the very nature of VLSI circuits. In a typical gate level netlist, while most gates would only connect to a small but non-fixed number of neighboring cells, certain gates could have hundred of fan-outs. Hence, the resultant data structures encoding the netlist have to be irregular. One example is the connection matrix required by the quadratic placement and force-driven placement [e.g. 6 and 7]. Based on our experiments on ISPD2006 benchmark circuits [8], such matrices are extremely sparse, where most rows only have 3 to 5 non-zeros and a very small number of rows having hundreds or thousands of non-zeros.

Major irregular EDA computing patterns include sparse matrix manipulations and graph algorithms. In fact, the authors of [9] identified major EDA applications and surveyed the underlying computing patterns. Out of the 17 major EDA applications investigated in [9], 15 applications are built on top of graph algorithms and 4 applications involved sparse matrix computations. In addition, the 17 applications do not include device physics and process simulations, which also involve large scale sparse matrices for finite element computations.

Although it has long been known that sparse matrix operation and graph algorithms possess sufficient data level parallelism [10], it's extremely challenging to efficiently implement them on GPUs. The reason is largely due to GPU's design philosophy, which is to devote most die area on computing resources but little on caches.

For irregular applications where the memory access patterns are unpredictable, GPUs would have difficulty to hide memory latency and maintain good load balance. For example, the computation of sparse-matrix vector product (SMVP) has also been widely considered as one tough problem for GPUs and only marginal speedup can be accomplished until recently. Bell and Garland introduced a very novel solution [11] on NVidia GPUs to address the data irregularity. A throughput of ~10 GFLOPS can be achieved on problems from different engineering domains. However, our experiments using the code released with [11] indicate that the speedup is still limited on EDA problem instances. As the second example, the GPU implementations for the Breadth-First Search (BFS) problem proposed in [12] could only achieve a good speedup on randomly created graphs where the number of edges on a node is well bounded. For graphs extracted from real-world applications, the GPU implementations in [12] do not have much advantage over their CPU equivalents.

As a first step toward a systematic parallelization of EDA applications, we explore efficient GPU solutions for irregular EDA applications. Of course, the two topics of sparse matrix and graph traversal still cover too broad a scope. So in this work we focus on SMVP and BFS, which are typical problems from the above two categories. We developed efficient SMVP implementations using NVidia's CUDA technology. We also identified key combinations of techniques to adapt to problems with varying internal structures. On a wide range of EDA problem instances, we could achieve a speedup up to 50X, which outperforms all published work. We then confirmed that our SMVP kernels could expedite 2 fundamental EDA applications, namely static timing analysis and conjugate gradient based linear system solution, by one order of magnitude. Next we extended our work to graph traversal problems by proposing a SMVP based formulation for the breadth-first search problem, which is more aligned to GPU's data parallel model. Our work proved that through properly re-organizing computing structures and/or re-designing algorithms we can efficiently exploit GPU's power for irregular EDA applications.

The rest of this paper is organized as follows. In section 2, we review the hardware architecture of NVidia GPUs and the corresponding data parallel programming model. In section 3, we introduce our CUDA implementation for the SMVP problem and then present the experimental results. Next we discuss how to directly apply the SMVP kernels to solve two EDA applications, static timing analysis and circuit placement. Section 5 covers graph traversal problems based on the SMVP procedure. Finally we conclude the paper and outline future research directions.

2. OVERVIEW OF CUDA PLATFORM

In this work, we use NVidia's CUDA platform to develop GPU implementations. Here we briefly introduce the important details of CUDA hardware and software.

2.1 Hardware Architecture

The architecture of NVidia's latest flagship GPU chip, GT200, is illustrated in Figure 1. The main computing resource consists of 240 streaming processors (SPs), evenly distributed into 10 streaming multiprocessors (SM). A single SP has its own execution hardware, but no instruction fetch and decoding capabilities, which wold be taken care of by the SM. A SM would fetch instructions and schedule them on its 8 internal SPs. Two special functional units (SFU) and a double precision unit are also

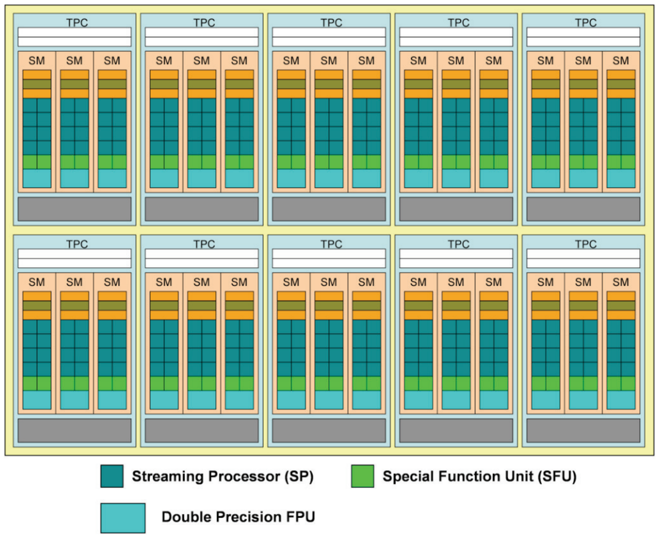


Figure 1. NVidia GPU architecture

installed inside each SM. The SFUs have dedicated logic for mathematical functions, while the double precision unit is a new feature only belonging to G200 series GPUs. The GPU chip is actually an array of SMs working concurrently. Given proper computing and data accessing patterns, the peak floating point throughput of GT200 series can reach 700~800GFLOPS.

During GPU computing, data would be stored inside the so-called global memory, i.e. video memory, integrated on the graphic card. Although the memory bus could deliver a high bandwidth, the latency of accessing the global memory is still in the range of 400~800 cycles (GPU core clock). Today's GPUs are enhanced with a memory coalescing mechanism such that accesses to adjacent memory addresses by neighboring processing elements could be combined into a single operation. Such a blocked accessing mechanism significantly boost effective memory bandwidth by taking advantage of the parallel architectures of memories. Every SM is equipped with a 16KB shared memory, which could provide up to 16 4-byte words of data in one clock cycle. It is completely software-controlled cache so that frequently used data can be placed close to the computing resource without suffering many times of global memory latency. Programmers could also use the shared memory to make memory access coalesced. Later we will show this technique is essential to improve the efficiency of handling irregular data structures.

2.2 CUDA Programming Model

CUDA is a platform technology developed by NVidia for data parallel computing. A CUDA program is composed of codes running on both CPU and GPU. The GPU code would be concurrently executed by GPU as coordinated by CPU. The function called by CPU but executed on GPU is called a kernel. One CUDA program could have multiple kernels.

According to the CUDA model, a GPGPU application could launch up to tens of thousands of threads, with each running the same program on different data sets. A thread is the minimum unit of parallel execution and the internal code runs sequentially. A number of threads are organized into thread blocks in a 1-D, 2-D, or 3-D manner. The arrangement should match the problem structure so as to simplify programming. The threads inside a block could exchange data through the shared memory and synchronize with one another. A kernel is composed of a grid of thread blocks arranged as a 1-D or 2-D array.

During GPU computing, thread blocks are assigned to SMs. Here one thread block could be allocated to one and only one SM, but one SM could accept several thread blocks. A SM groups every 32 threads into a warp. The threads in a warp have the same instruction execution schedule. Since a SM has 8 SPs, 1 warp of 32 threads finishes an instruction in every four clock cycles.

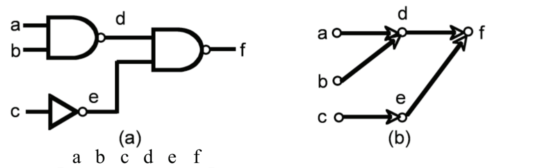
3. SPARSE MATRIX VECTOR PRODUCT

3.1 Introduction

A matrix is sparse when only a small percentage of its elements have a non-zero value. One example is the adjacency matrix for a VLSI netlist used by many EDA applications. As illustrated in Figure 2, the matrix entry *(i, j)* represents the connection from cell *i* to cell *j*. An entry is a non-zero only when there's a net connecting cells *i* and *j*. Obviously, a cell usually has only a few connected cells in a whole netlist and thus the matrix tend to be very sparse.

One of most commonly used storage format is Compressed Sparse Row (CSR) format [13]. Interested readers please refer to [13] for other formats. In CSR format, 3 vectors are required to represent a sparse matrix. Vector *col* records the column index of each non-zero, while vector *elem* stores the non-zero values. The third vector, *rowptr*, keeps track the location of each row's 1st nonzero in vector *elem*.

We investigated a series of sparse matrix based EDA applications including circuit simulation, circuit placement and finite element based layout stress analysis. Our observation is that sparse matrix vector product (SMVP) procedure is often the bottleneck. For instance, our conjugate-gradient [14] solver for linear systems



$$A = \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

(c)

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

(d)

Figure 2. Sparse matrix and compressed row storage format (a) A simple netlist, (b) Directed graph corresponding to (a), (c) Adjacency matrix corresponding to (b), (d) CSR data structures for the matrix in (c)

```
void smvp_serial(unsigned int *rowptr, unsigned int *col,
      float *elem, const unsigned int num_rows, float *v, float *p)
{
    for(unsigned int row=0; row<num_rows; ++row) {
        unsigned int row_begin = rowptr[row];
        unsigned int row_end = rowptr[row+1];
        float sum = 0.0;
        for(unsigned int j= row_begin; j< row_end; ++j)
            sum += elem[j] * v[col[j]];
        p[row] = sum;
    }
}
```

Figure 3. Computing sparse matrix vector product

spends more than 90% of CPU time in SMVP computing. Accordingly, we believe that SMVP problem is one of the key irregular patterns that need to be evaluated on GPU.

3.2 Efficient CUDA Implementations for SMVP

3.2.1 Introduction and Prior Work

The C code for SMVP is listed in Figure 3. Every round of the outer loop generates one element of the product vector. The inner loop traverses all non-zero elements in one row. Using the CSR format, a given non-zero with index *j* will be multiplied by the vector element with index *col[j]*.

Starting with the code listed in Figure 3, a straightforward implementation of SMVP with CUDA would create multiple threads with each computing of a single row. References [11] and [15] provide details for such an approach. The performance, however, turns out to be unsatisfying. A careful analysis reveals two major reasons leading to significant inefficiency. First, the memory access cannot be coalesced because one thread (computing one element for the product vector) needs to load varying number of data words from memory. Secondly, the load balance is poor since there could be hundreds of nonzero elements in some rows, while most other rows have only a few. Therefore, the GPU run time is dominated by those less sparse rows. In fact, the above problems make SMVP problem extremely challenging. And there exist tough matrix instances where the straightforward GPU implementation can be slower than its CPU equivalent [14].

In [11], Bell and Garland proposed a very novel solution to the CSR based SMVP problem. This work uses a warp, i.e. 32 threads scheduled and executed as a batch, to process one row. The advantages are two-fold: 1) Memory accesses can be coalesced because the 32 continuous threads in one warp work together to fetch the non-zeros on one row; and 2) For those rows with many non-zeros (>32), the workloads can be distributed to multiple streaming processors and thus the load balance can be improved. This approach is extremely efficient for those matrices with long strips of non-zeros and a throughput of over 10 GFLOPS can be achieved. Nevertheless, the implementation is less efficient for problem instances arising from EDA applications, where most rows have only several non-zeros.

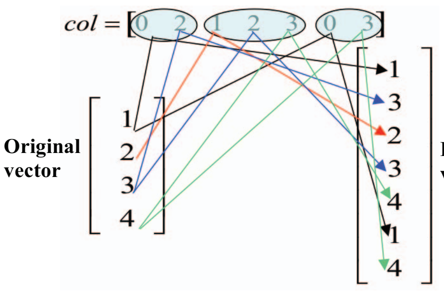
3.2.2 Our New Approach

By carefully analyzing the code listed in Figure 3, we realized that the SMVP problem actually consists of two phases with different available parallelism. In the first phase, every non-zero matrix element must be multiplied by a corresponding vector element. From this point of view, the multiplication operations are fully regular. In the second phase, we calculate the sum of the products on each row. Here the number of summations per row is determined by the distributions of non-zeros and thus cannot be regular for general cases. The two phases can be organized as two succeeding GPU kernels.

The first kernel, designated as the **product kernel**, can be implemented in a straightforward manner by assigning one multiplication to each thread. The CUDA kernel code is listed in Figure 4(a), where the products of each pair of matrix and vector elements are stored in array *middle[]*. Potentially all such multiplications can be computed in parallel on an ideal GPU with

```
__global__ void product_kernel(const unsigned int *col, const float *elem,
const unsigned int num_nz, const float *v, float *middle){
    unsigned int elemid = blockIdx.x * blockDim.x + threadIdx.x;
    if( elemid < num_nz)
        middle[elemid] = elem[elemid] * v[col[elemid]];
}
```

(a) Straightforward implementation



(b) Example of vector expansion

```
_global__ void expanded_product_kernel(const float *elem, const unsigned int
num_nz, const float *v_expanded, float *middle){
    unsigned int elemid = blockIdx.x * blockDim.x + threadIdx.x;
    if( elemid < num_nz)
        middle[ elemid] = elem[ elemid] * v_expanded[ elemid];
}
```

(c) Implementation with expanded vector

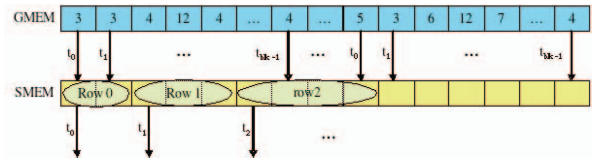
Figure 4. Product kernel

virtually unlimited number of threads. And note that the computation loads are also perfectly balanced.

Here the difficulty lies in the fetching of vector elements from the global memory. As shown in Figure 4(a), the load operation, i.e. *v[col[elemid]]*, cannot be coalesced generally because the value of *col[elemid]* can be arbitrary. Accordingly, we perform an expansion operation to vector *v* as illustrated in Figure 4(b). The expanded vector, *v_expanded*, has the same length as *elem[]* with *v_expanded[elemid]* = *v[col[elemid]]*. Figure 4(c) listed the improved implementation with completely coalesced memory access. The expansion operation can be efficiently implemented in CUDA and our experiments indicated that it takes less than 20% of the execution of the product kernel. In addition, many applications allow the expanded vector to be reused for many times. The expansion of course incurs a memory overhead, but the order of space complexity would not change and EDA instances are usually very sparse. Experiment results showed that the product kernel could be accelerated by one order of magnitude using expanded vectors.

The second kernel, which is called the **summation kernel**, is responsible of summing the products in one row together and then output it to the product vector. The major bottleneck is again the un-coalesced memory access due to the irregular distribution of non-zeros.

As mentioned in Section 2, a 16KB shared memory is installed into each streaming multiprocessor. In the case of SMVP, each product create by our first kernel is only used once in the summation process and so there's no need for data sharing among different threads. However, the shared memory can be used to make the memory access coalesced. The idea is illustrated in Figure 5(a), where GMEM stands for global memory and SMEM represents shared memory. We can use threads in one block to load the products created by the product kernel into shared



(a) Coalescing through shared memory

```
1  __global__ void summation_kernel( const unsigned int *rowptr,
2      float *middle, const unsigned int num_row, const unsigned int num_nz,
3      const unsigned int num_loads_per_thread, float *p)
4  {
5    __shared__ float cache[SHARED_MEM_PER_BLOCK];
6    __shared__ unsigned int num_nz_before;
7    unsigned int thread_begin = blockIdx.x * blockDim.x;
8    if(threadIdx.x == 0)
9        num_nz_before = rowptr[thread_begin]/16*16;
10   __syncthreads();
11   unsigned int elemid, cache_idx;
12   for( int i = 0; i < num_loads_per_thread; i++){
13       cache_idx = i * NUM_THREAD_PER_BLOCK + threadIdx.x;
14       elemid = num_nz_before + cache_idx;
15       if( cache_idx < SHARED_MEM_PER_BLOCK && elemid < num_nz)
16           cache[cache_idx] = middle[elemid];
17   }
18   __syncthreads();
19
20   unsigned int row = thread_begin + threadIdx.x;
21   if ( row < num_row){
22       float sum = 0.0;
23       unsigned int row_begin = rowptr[row];
24       unsigned int row_end = rowptr[row + 1];
25       for ( unsigned int i = row_begin; i < row_end; i++){
26               if( i >= num_nz_before && (cache_idx = i –
27                   num_nz_before) < SHARED_MEM_PER_BLOCK)
28                   sum += cache[ cache_idx];
29               else
30                   sum += middle[ i];
31       }
32       p[row] = sum;
33   }
   }
```

(b) Source code of summation kernel

Figure 5. Summation kernel

memory in a coalesced manner. The summation process could then use data in the shared memory.

Of course, due to the limit of shared memory size, not all data can be cached. Since we have 768 threads running on one streaming multiprocessor, every thread could load 5 or 6 floating point numbers (4 bytes each) into the 16KB shared memory. We developed a CPU based shared memory simulator to evaluate the effectiveness of the above technique. The results proved that the "hit ratio" (probability of required data in the shared memory) could be higher than 90% for a wide range of matrix instances when the average number of non-zeros per row is less than 6. When there're 10 non-zeros on a row on average, the hit ratio is still higher than 45%.

The summation kernel is listed in Figure 5. On line 9, we derive the index of the elements required by the 1st thread in the current thread block. The index is then aligned to be a multiple of 16, which is required by the coalescing rules. The code on lines from 12 to 17 coordinates threads in one block work together to load data into shared memory. The remaining code performs the summation. Before an adding operation, the code on line 26 checks if the data is already cached in the shared memory. If it's not cached, a global memory access is still needed.

In the rest of the paper, the combination of the above two kernels is designated as the **product + summation kernels**. When the average number of non-zeros per row is larger than 10, the summation kernel becomes less efficient. Under such a circumstance, we switch to another summation kernel, designated as the **warped summation kernel**, using a technique inspired by [12]. The idea is to use a warp of 32 threads to perform the summation process. Since the multiplication process is already conducted in parallel, the combination of the product kernel and the warped summation kernel is more efficient. We name such a combination as the **product + warped summation kernels**.

3.3 Results

We tested our SMVP kernels on a wide range of sparse matrices (not necessarily from EDA applications) to verify the effectiveness of our techniques. All experiments are performed on a Linux PC with a 3.33-GHz, Core 2 Duo processor, an NVidia GTX280 graphic card and 4-GB RAM. All programs are implemented with CUDA release 2.1 [16]. The CPU implementation of SMVP procedure was carefully optimized and the CPU time is slightly faster than Matlab [17] (for a matrix with 14M non-zeros, our CPU implementation spends 72ms on average, while Matlab needs 81ms).

Table 1. Characteristics of matrix set 1

Inst.	# rows	# columns	# non-zeros	avg. # non-zeros per row	description
ns3Da	20414	20414	1679599	82.3	3D Navier Stokes equation
raefsky3	21200	21200	1488768	70.2	Turbulence problem
venkat01	62424	62424	1717792	27.5	2D Euler solver
b18_100K.	100000	333740	14327592	143.3	Static timing analysis
Yangliu	235620	235620	17563926	74.5	Finite element based layout stress analysis

Table 2. Characteristics of matrix set 2

Inst.	# rows	# columns	# non-zeros	avg. # non-zeros per row	description
Lin	256000	256000	1766400	6.9	Large sparse Eigenvalue problem
t2em	921632	921632	4590832	5.0	Electromagnetic problems
ecology1	1000000	1000000	4996000	5.0	Circuit theory applied to animal/gene flow
cont11	1468599	1961394	5382999	3.7	Linear programming
sls	1748122	62729	6804304	3.9	Large least-squares problem
G3_circuit	1585478	1585478	7660826	4.8	AMD circuit simulation
thermal2	1228045	1228045	8580313	7.0	FEM, steady state thermal problem
kkt_power	2063494	2063494	12771361	6.2	Optimal power flow, nonlinear optimization
Freescale1	3428755	3428755	17052626	5.0	Freescale circuit simulation

We reported GLFOPS of GPU implementations as well as their speedup against their CPU equivalents. To make a comprehensive comparison, we also include the results created by Bell and Garland's kernel, designated as **B&G kernel**. The source code downloaded from NVidia's CUDA forum [18] uses texture memory to store the vector to be multiplied because texture memory is cached. However, the texture memory is read-only and thus cannot be applied to the cases where the vector to be multiplied needs to be iteratively updated. So in our experiments we removed the texture access code in B&G, but it should be noted that both B&G and our kernels could enjoy another 30% performance improvement with the texture memory.

When computing throughput, we do not include the data transfer time between CPU and GPU because generally the data will be re-used for many times. For example, the same matrix could be multiplied by around 1000 times in the case of finite element based layout stress analysis, while a linear system solver would iterative over the same matrix for over 80 times. Similarly, we do not include the vector expansion time since the expansion kernel usually finishes within 20% of the execution time of the product kernel. In addition, the summation/warped summation kernel dominates the execution time in all the experiments and the product kernel only consumes 5-20% of the total time.

We used two sets of test cases with radically different numbers of non-zeros per row. The characteristics of these matrices are listed in Tables 1 and 2, both ordered by the number of non-zeros. The 5th column reports the average number of non-zeros in each row. The matrices in the first set have relatively more non-zeros on each row. Among these, the first 3 matrices are randomly picked from University of Florida Sparse Matrix Collection [19], while the remaining two are from static timing analysis (explained later in section 4.1) and finite element based layout stress analysis [20]. All taken from [19], the matrices in the second set are either directly created by EDA programs or by applications closely related to EDA. Determined by the problem nature as discussed in the previous section, these matrix instances have rather few non-zeros in each row.

Table 3. SMVP throughput for matrix set 1

Instance	CPU	B&G	speed-up	product+ summation	speed-up	product +warped summation	speed-up
ns3Da	0.51	5.02	9.84	3.79	7.43	18.08	**35.43**
raefsky3	0.53	12.29	23.15	5.41	10.19	14.54	**27.40**
venkat01	0.53	8.13	15.47	5.71	10.87	8.63	**16.43**
Yangliu	0.49	9.08	18.53	5.67	11.57	20.20	**41.22**
b18_100K	0.55	5.14	9.38	5.12	9.34	27.42	**50.03**

Table 4. SMVP throughput for matrix set 2

Instance	CPU	B&G	speed-up	product+ summation	speed-up	product +warped summation	speed-up
Lin	0.26	1.23	4.81	9.23	**36.04**	1.28	5.01
t2em	0.29	1.54	5.40	12.41	**43.44**	1.68	5.88
ecology1	0.24	0.93	3.87	9.03	**37.43**	1.01	4.20
cont11	0.31	1.14	3.63	10.66	**33.84**	1.25	3.95
sls	0.28	1.18	4.26	10.10	**36.49**	1.32	4.77
G3_circuit	0.21	0.91	4.24	8.86	**41.45**	0.99	4.64
thermal2	0.21	1.24	5.81	8.97	**41.89**	1.35	6.31
kkt_power	0.26	1.23	4.77	5.70	**22.01**	1.34	5.16
Freescale1	0.29	1.65	5.76	11.56	**40.37**	1.88	6.57

In Tables 3 and 4 we report both the throughput and the speedup of GPU against the CPU baseline implementations. The throughput is measured in Giga FLoating Operation Per Second (GFLOPS). During a SMVP operation, each non-zero element incurs one floating multiplication and one floating addition. So the total number of floating operations is just two times the number of non-zeros. Then the throughput can be computed by measuring execution time with utility functions provided in CUDA SDK [16]. Columns 2, 3, 5, and 7 report the GFLOPS results of CPU, the B&G kernel, the product + summation kernels, and the product + warped summation kernels, respectively. Columns 4, 6, and 8 collect the corresponding speedup values against the CPU baseline implementation. To rule out the variance of execution time, we conducted 1000 runs for each dataset and report average time of one run.

For matrix set 1, the product + warped summation kernels perform the best because the workloads of different threads are better balanced. Such an implementation could achieve a speedup of over **15X** on all test cases and over **40X** on the two largest matrices. Meanwhile, due to the fully parallelization of the element-wise multiplication, our implementation also significantly outperforms the original B&G kernel.

For matrix set 2, the product + summation kernels have a considerable performance advantage. Before this work, the best SMVP throughput is realized by the B&G kernel with a speedup of around 5X. Now our kernels accelerate all test cases by **one order of magnitude (over 30X for 8 out of 9 cases)**. For the circuit and thermal simulation test cases (i.e. G3_circuit, thermal2 and Freescale), the performance of our SMVP implementation can be improved by a factor of over **40X**.

Now the efficiency of our SMVP kernels has been established. It should be noted that our GPU implementations can be embedded into any EDA applications involving large scale sparse matrix operations. Meanwhile, our techniques of data re-organization can be integrated into a compilation framework for automatic parallel optimizations. It also bears mentioning that the data parallel model is complement to the task-level parallel and distributed computing models. Accordingly, GPU computing could provide a new dimension of parallelization for future EDA software.

4. DIRECT APPLICATIONS OF THE SMVP KERNEL

In this section, we show how to use the SMVP kernels to solve 2 commonly used EDA applications, static timing analysis and linear system solution for circuit placement.

4.1 Static Timing Analysis (STA)

In [21], Ramalingam et al. introduced an efficient procedure to derive timing analysis through SMVP. Given a netlist, the timing graph is constructed by treating every pin as a node. There is an edge between 2 nodes if they are connected through a gate or a wire. Then by enumerating all paths that can be sensitized, a matrix can be established by allocating one row to each path and one column to each pin. A matrix entry *(i, j)* equals to 1 if pin *j* belongs to path *i*. A delay vector is built by allocating an entry for every pin and assigning the value as the associated delay. Path based timing analysis can be carried out by a SMVP procedure on the above matrix and vector.

We tested our SMVP kernels on two largest ITC99 benchmark circuits, b18 and b19 [22]. For each circuit, we created two test cases with 50K and 100K paths randomly picked up. The delay vector is constructed by using the parameters from a 0.13um standard cell library [23]. Table 5 describes the statistics of these matrices and Table 6 lists the throughput in terms of number of paths per second of STA. Since the average number of non-zeros on each row is relatively high, the product + summation kernels perform best and achieve a throughput of more than 100M paths per second, equivalent to a speedup of around 50X against CPU implementations. Note that the combination of the product and warped summation kernels significantly outperforms the original B&G kernel. Our timing analysis engine can be extended to handle statistical timing and an even higher speedup can be expected due to the larger data volume.

Table 5. Characteristics of STA matrices

Instance	# rows	# columns	# non-zeros	avg. # non-zeros per row
b18_50K	50000	333740	7057712	141.2
b18_100K	100000	333740	14327592	143.3
b19_50K	50000	673144	5562170	111.2
b19_100K	100000	673144	11332042	113.3

Table 6. STA throughputs (#paths per second) via SMVP

Instance	CPU	B&G	speed-up	product + summation	speed-up	product + warped summation	speed-up
b18_50K	1.95E+06	1.78E+07	9.14	1.80E+07	9.23	1.02E+08	**52.33**
b18_100K	1.92E+06	1.79E+07	9.34	1.79E+07	9.31	9.57E+07	**49.82**
b19_50K	2.46E+06	1.98E+07	8.05	2.34E+07	9.51	1.10E+08	**44.64**
b19_100K	2.42E+06	2.01E+07	8.28	2.37E+07	9.80	1.14E+08	**47.00**

4.2 Linear System Solution for Circuit Placement

Many EDA applications involve solution of large scale linear systems. One typical application is the analytical placement (e.g. [5, 6]), where more than 60% of the CPU time is spent on solving linear systems using a Conjugate Gradient (CG) solver [24]. According to our experiences on a force-driven placer [14], over 90% CPU time of a CG solver is consumed by the SMVP procedure. Accordingly, we implemented a CG solver on top of our SMVP kernels. Besides the SMVP procedure, a CG solver also needs SAXPY (sum of $Alpha \times x + y$, where $Alpha$ is a scalar and x and y are vectors), inner product, and reduction kernels, which all can be accelerated by 50-100X on GPU's.

We tested our CG solver on 7 ISPD06 placement benchmark circuits [8]. The first 3 columns of Table 7 show the statistics of these circuits. We then use a hybrid approach defined in [25] to construct the connection matrix. The characteristics of the sparse matrices are listed in the last 4 columns of Table 7. The throughput results for solving a linear system (i.e. equivalent to one round of global placement) are collected in Table 8. Since the number of non-zero per row is in the range of 3 to 5, the product + summation kernels deliver the best GFLOPS values as expected. The solution time can be reduced by a factor of around 20X on GPU's. For the largest test case with 2.5M cells, the GPU implementations could be 28 times faster.

Table 7. Characteristics of placement matrices

Instance	# cells	# nets	# rows	# columns	# non-zeros	avg. # non-zeros per row
new_blue1	330474	228901	398096	398096	1595758	4.0
new_blue2	441516	465219	529672	529672	2340198	4.4
new_blue3	494011	552199	561819	561819	1875296	3.3
new_blue4	646139	637051	784321	784321	3152532	4.0
new_blue5	1233058	1284251	1447872	1447872	6287726	4.3
new_blue6	1255039	1288443	1521703	1521703	6287132	4.1
new_blue7	2507954	2636820	3068717	3068717	15120230	4.9

Table 8. Throughputs of CG (GFLOPS)

Instance	CPU	B&G	speed-up	product+ summation	speed-up	product+ warped summation	speed-up
new_blue1	0.26	1.27	4.80	4.84	**18.32**	1.35	5.12
new_blue2	0.28	1.40	5.09	5.63	**20.43**	1.50	5.44
new_blue3	0.27	1.08	4.07	5.84	**21.99**	1.12	4.23
new_blue4	0.28	1.30	4.65	5.08	**18.18**	1.39	4.97
new_blue5	0.25	1.39	5.45	6.02	**23.68**	1.51	5.92
new_blue6	0.26	1.33	5.04	6.44	**24.32**	1.43	5.40
new_blue7	0.25	1.57	6.16	7.33	**28.79**	1.71	6.74

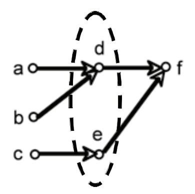
5. BREADTH FIRST SEARCH (BFS)

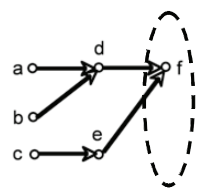
Graph algorithms, which also tend to be irregular, constitute another family of core EDA algorithms. As illustrated in Figure 2, the sparse matrix is closely related to many graph problems arising from EDA applications. Accordingly, our SMVP kernel can be deployed to accelerate the solution of many graph algorithms.

Garland already showed how to compute the shortest path using SMVP [14]. Our experiments proved that a 15X speedup can be achieved with our SMVP kernels. Similar to the shortest path problem, Breadth First Search (BFS) is also widely used by EDA applications. Moreover, BFS exhibits a similar computing pattern to many EDA applications such as logic simulation and block based timing analysis. For instance, during logic simulation a transition at an output pin would trigger events on neighboring gates. Such a pattern can be exactly captured in a BFS process.

We use the adjacency matrix illustrated in Figure 2 as an example. The circuit is represented as a direct graph. An entry *(i, j)* is equal to 1 if node *i* has a directed edge toward node *j*. Figure 6 illustrates the breadth first traversal process. The dotted circle represents vertices reached after one expansion. We begin expansion from primary input nodes *a*, *b*, and *c*, which are recorded in a vector, *x*, by setting the corresponding entries to 1. The product vector of A^Tx has one unique entry for one vertex and a non-zero indicates that the corresponding vertex has been reached. This procedure can be repeated by multiplying A^T with the product again and again until all nodes have been visited. A nice feature of this approach is the value of an entry in the product vector reflects how many paths lead to the corresponding graph node. Such information is necessary for the critical path method (CPM) used in block based timing analyzers [26].

To check if all nodes are traversed, we need two extra kernels, with both already having very efficient GPU implementations. The first kernel assigns one thread for each vertex to check if the





$$A^T \bullet x = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 1 \\ 0 \end{bmatrix} \quad A^T(A^T \bullet x) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \end{bmatrix}$$

Figure 6. BFS through sparse matrix vector product

corresponding entry in the product vector is set to non-zero after SMVP. If the condition satisfies, a thread will write a 1 to a book-keeping vector, on which the second kernel then performs a logic and operation. The second kernel is actually a parallel reduction [10] and we use the efficient implementation provided in CUDA SDK [16].

Table 9. Characteristics of connection matrices and BFS results

Instance	#nodes	CPU time (ms)	throughput (#vertices /s)	GPU time (ms) product + summation	throughput (#vertices/s)	speed-up
b18	333740	869.45	3.84E+07	67.22	4.96E+08	**12.93**
b18_1	315683	802.28	3.93E+07	63.02	5.01E+08	**12.73**
b19	673144	1795.15	3.75E+07	130.43	5.16E+08	**13.76**
b19_1	638283	1685.31	3.79E+07	120.83	5.28E+08	**13.95**

The results of BFS are summarized in Table 9. The input matrices are created from the timing graph of the 4 largest ITC benchmark circuits [22]. The timing graphs are constructed by treating each pin as a vertex and each gate or wire as a directed edge (please refer to [26] for details). The baseline CPU implementation also uses the sparse matrix formulation, which is faster then the traditional queue based BFS procedure (e.g. the classic implementation outlined in [27]). The average number of non-zeros on each row is very low (i.e. between 1 and 2 for all test cases), because most pins only drive 1 fan-out. Accordingly, the B&G kernel and the product + warped summation kernels perform very badly on these test cases, where a marginally 2X speedup can be achieved against the CPU baseline. So we only reports results collected by using the product + summation kernel. The throughput of the GPU implementation is approaching 500M vertices per second, which used to be achievable only on supercomputers.

6. CONCLUSION AND FUTURE WORK

Modern GPUs are delivering considerable computing power. The irregular data access patterns inherent to EDA applications, however, pose noteworthy challenges for efficient GPU implementations. In this work, we propose effective GPU based techniques to address two important irregular EDA computing patterns, the sparse-matrix vector product problem and the breadth-first graph traversal problem. We first introduce efficient SMVP implementations, which could accelerate many computing intensive EDA applications by an order of magnitude. Then we adapt the SMVP procedure to address irregular graph traversal problems. By introducing a new formulation of breadth-first search using sparse matrix, the BFS operation can be efficiently solved on GPUs with a speedup of over 10X. The above work establishes that with proper data structure transformation and algorithm re-designing, GPUs will serve as a powerful platform to solve irregular EDA problems.

In the future, we are going to extend our work in several directions. First, we will investigate applying our new BFS formulation to solve the logic/RTL simulation problem. Second, it's worth exploring the feasibility of GPU for solving SPICE simulation using a direct method [28]. Another important extension is to address the time-consuming formal verification problems.

7. ACKNOWLEDGMENTS

This work was partially supported by a faculty initialization funding from Tsinghua University and NVidia Professor Partnership Awards. We also acknowledge hardware donation from NVidia. We thank N. Bell and M. Garland for sharing their code and paper on the sparse matrix vector product problem.

8. REFERENCES

[1] Blythe, D. 2008. Rise of the graphics processor. Proceeding of IEEE, Vol. 96, No. 5, 761– 778, May, 2008.

[2] NVidia. 2008. CUDA programming guide.

[3] Feng, Z. and Li, P. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. In Proc. of Int'l Conf' on Computer Aided Design.

[4] Gulati, K. and Khatri, S. P. 2008. Towards acceleration of fault simulation using graphics processing units. In Proc. of. ACM IEEE Design Automation Conf.

[5] Gulati, K., Croix, J. F., Khatri, S. P., and Shastry, R. 2009. Fast circuit simulation on graphics processing units. in Proc. of Conf' on Asia and South Pacific Design Automation.

[6] Kleinhans, G. Sigl, F. Johannes, and Antreich, K. 1991. Gordian: VLSI placement by quadratic programming and slicing optimization. IEEE Trans. CAD, vol. 10, no.3, March 1991.

[7] Eisenmann, H. and Johannes, F. M. 1998. Generic global placement and floorplanning. In Proc of Design automation Conf., 269-274.

[8] Nam, G.-J., Alpert, J. C., and Villarrubia, P. G. 2007. ISPD 2005/2006 placement benchmarks. Modern Circuit Placement. Ch. 1. Springer US.

[9] Catanzaro, B., Keutzer, K., and Su, B.-Y. 2008. Parallelizing CAD: a timely research agenda for EDA. In Proc. Design Automation Conf., 12-17.

[10] Blelloch, G. E. 1990. Vector models for data-parallel computing. MIT Press.

[11] Bell, N. and Garland, M. 2008. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report. NVR-2008-004.

[12] Harish, P. and Narayanan, P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA. In Proc. Of High Performance Computing – HiPC. 197-208.

[13] Saad, Y. 2000. Iterative methods for sparse linear systems. SIAM.

[14] Deng, Y. and Mu, S. 2008. The potential of GPUs for VLSI physical design automation. In Proc. of International Conference on Solid-State and Integrated-Circuit Technology.

[15] Garland, M. 2008. Sparse matrix computations on manycore GPU's. In Proc. of Design Automation Conference, Jun. 2008, pp. 2-6.

[16] NVidia, 2009. CUDA Programming Guide, CUDA Driver, Toolkit, and SDK code samples. http://www.nvidia.com/object/cuda_get.html.

[17] MathWorks. 2007. Matlab V7.4. http://www.mathworks.com/.

[18] Bell, N. 2008. Sparse Matrix-Vector Multiplication on CUDA. http://forums.nvidia.com/index.php?showtopic=83825&st=0

[19] Davis, T. University of Florida Sparse Matrix Collection. Available online: http://www.cise.ufl.edu/research/sparse/matrices/.

[20] Xue, J., et. al. 2009. Layout Dependent STI Stress Analysis and Stress-Aware RF/Analog Circuit Design Optomization. In Proc. of ACM/IEEE Int'l Conf. on Computer-Aided Design.

[21] Ramalingam, A., Nam, G.-J., Singh, A. K., Orshansky, M., Nassif, S. R., and Pan, D. Z. 2006. An accurate sparse matrix based framework for statistical static timing analysis. In Proc. of the 2006 Int'l Conf. on Computer-Aided Design, [doi>10.1145/1233501.1233547]

[22] ITC'99 Benchmarks (2nd release), Available online: http://www.cad.polito.it/tools/itc99.html.

[23] SMIC, 0.13um Low Leakage Cadence PDK. http://www.smics.com/website/cnVersion/DS/SMIC-PDK.htm.

[24] Barret, R. et al, 1994. Templates for the Solution of Linear Systems, 2nd Edition, SIAM.

[25] Viswanathan, N. and Chu, C. C.-N. 2005. FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. IEEE Trans. Computer-Aided Design, Vol. 24, No. 5, 722-733, 2005.

[26] Sapatnekar, S. 2004. Timing. Springer; 1 edition. Ch. 5.

[27] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. 2001. Introduction to algorithms. 2nd Edition. MIT Press.

[28] David, T. 2006. Direct methods for sparse linear systems. SIAM.