Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Towards accelerating irregular EDA applications with GPUs

Hao Qian*, Yangdong Deng, Bo Wang, Shuai Mu

Institute of Microelectronics, Tsinghua University, Beijing, China

ARTICLE INFO

Article history:
Received 5 October 2010
Received in revised form
1 April 2011
Accepted 27 May 2011
Available online 23 June 2011

Keywords:
GPU
CUDA
Sparse matrix
EDA
Data parallel
Static timing analysis
Sparse matrix vector product
Conjugate gradient
Breadth first search
Survey propagation
RTL simulation
CMB
Message-passing

ABSTRACT

Recently graphic processing units (GPUs) are rising as a new vehicle for high-performance, general purpose computing. It is attractive to unleash the power of GPU for Electronic Design Automation (EDA) computations to cut the design turn-around time of VLSI systems. EDA algorithms, however, generally depend on irregular data structures such as sparse matrix and graphs, which pose major challenges for efficient GPU implementations. In this paper, we propose high-performance GPU implementations for a set of important irregular EDA computing patterns including sparse matrix, graph algorithms and message-passing algorithms. In the sparse matrix domain, we solve a core problem, sparse-matrix vector product (SMVP). On a wide range of EDA problem instances, our SMVP implementation outperforms all prior work and achieves a speedup up to $50\times$ over the CPU baseline implementation. The GPU based SMVP procedure is applied to successfully accelerate two core EDA computing engines, timing analysis and linear system solution. In the graph algorithm domain, we developed a SMVP based formulation to efficiently solve the breadth-first search (BFS) problem on GPUs. We also developed efficient solutions for two message-passing algorithms, survey propagation (SP) based SAT solution and a register-transfer level (RTL) simulation. Our results prove that GPUs have a strong potential to accelerate EDA computing through designing GPU-friendly algorithms and/or re-organizing computing structures of sequential algorithms.

1. Introduction

Integrated circuits (ICs) have become the most complex machine made by the human being. Today IC designers depend on the Electronic Design Automation (EDA) software to properly handle the ever-increasing IC complexity in a timely fashion. The computing demand for EDA software is still fast rising with the advent of 32 nm technology node. For instance, IC designers have to spend a couple of hours to perform a timing analysis on a 10 M-gate design, while a gate level simulation of a full chip could take weeks or even months. Another example is the circuit simulation problem. Given a Giga-Hertz phase-lock loop (PLL) circuit, a transient analysis would need to simulate the circuit for millions of cycles and thus a complete run would take months to finish. However, today's electronic appliances typically have a fixed market window as short as 6 months [1]. Due to the prohibitive CPU time for design implementation and verification, such a tight schedule suggests that only a small portion of the complete solution space can be explored in the design process if the productivity of EDA software cannot scale accordingly. On the other hand, the IC product development cost can reach $100M at 32 nm technology node [2]. Such an overwhelming cost suggests that IC designers have to perform even more intensive verification to minimize the possibility of buggy designs. As a result, future EDA software tools have to deliver even higher computing throughput.

In the history of EDA technology, the constantly increasing processing capability mainly came from the synergy of two forces: (1) development of smarter algorithms and/or more efficient software implementations and (2) scaling of CPU performance. Unfortunately, now the single-CPU performance is relatively saturating. Since the semiconductor process is still offering growing integration capacity, multi-core processors are inevitably becoming the dominant computing resources for EDA applications. It is thus essential to develop parallel solutions for the EDA industry such that the momentum of function increase in VLSI designs can be maintained [3].

Recently, general purpose computing on graphic processing units (GPUs) has become a very important trend of high performance computing [4]. Unlike multi-core CPUs that generally utilized a task level parallelism, graphic processing units (GPUs) exploit 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

* Corresponding author.
E-mail address: cyqh1028@hotmail.com (H. Qian).

a very high computing throughput. For example, NVIDIA's flagship GPU, Fermi, could reach a peak floating-point throughput of 1.5 TFLOPS [5,6]. On workloads with appropriate computing and memory access patterns, GPU could even attain a speedup of over $100\times$.

Accordingly, it is appealing to unleash the computing power of GPU for EDA applications. There are already a few papers, e.g., [7,8], and [9], presenting encouraging results on utilizing GPU to solve specific EDA problems. However, a comprehensive investigation on the foundation of GPU-based EDA computing is still critical, especially 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 hundreds 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 and force-driven placement algorithms, (e.g., [10] and [11]). Based on our experiments on ISPD2006 benchmark circuits [12], such matrices are extremely sparse, where most rows only have 3–5 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 [3] identified major EDA applications and surveyed the underlying computing patterns. Out of the 17 major EDA applications surveyed in [3], 15 applications are built on top of graph algorithms and 4 applications involved sparse matrix computations.

Although it has long been known that sparse matrix operation and graph algorithms possess rich data level parallelism [13], it is extremely challenging to efficiently implement them on GPUs. For example, the computation of sparse-matrix vector product (SMVP) has been widely considered as one tough problem for GPUs and only marginal speedup can be accomplished until recently. In [14], Bell and Garland introduced a novel solution on NVIDIA GPUs for the SMVP problem. Their GPU implementation could attain a throughput of $\sim$10 GFLOPS on problem instances from different engineering domains where relatively long strip of non-zeros exist. However, our experiments using the code released with [15] indicate that the speedup is still limited on the problem instances from EDA applications. As the second example, the GPU implementations for the breadth-first search (BFS) problem proposed in [17] 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 [17] do not have much advantage over their CPU equivalents. The inefficiency for the above two domains 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. Another major hurdle for high performance on GPUs is the poor load balance induced by the irregularity. When performing parallel operations on sparse matrices and graphs, the workload for each basic unit of parallel execution varies dramatically. However, GPU executes computation in a batched manner, where a group of threads would have exactly the same instruction schedule. Therefore, the slowest thread would determine the execution time for this group of threads.

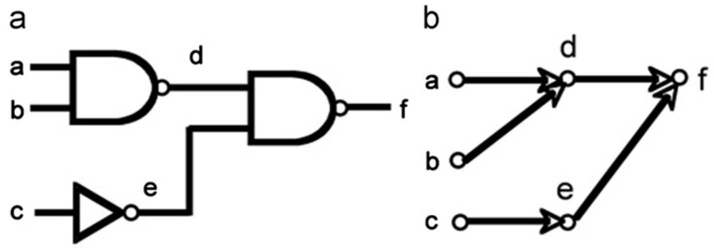
As a first step toward a systematic parallelization of EDA applications and based upon our preliminary work [18], we explore efficient GPU solutions for two typical computing patterns, sparse matrix and graph algorithmic operations. First, we developed an efficient GPU implementation for the sparse matrix vector product (SMVP) problem, which is the central piece of sparse matrix operations. On a wide range of EDA problem instances, the GPU based SMVP implementation could outperform all previously published results and achieve a speedup of up to $\mathbf{50\times}$. We then applied the SMVP procedure to expedite two important EDA computing patterns, circuit delay calculation and conjugate gradient based linear system solution. A speed-up factor of one order of magnitude is observed on both problems. Next we extended our work to the graph algorithm domain by solving the classical breadth-first graph traversal problem through a SMVP based formulation, which is more aligned to the data parallel model of GPUs. In addition, the techniques developed for SMVP can also be used to efficiently accelerate a survey propagation (SP) based SAT solver [19] by a factor of around $\mathbf{20\times}$. We also develop GPU solutions to accelerate another key EDA application, register transfer level (RTL) simulation. Our GPU based RTL simulator can be faster by its CPU counterpart by a factor of over 18. The most important message delivered in this work is that the computing power of GPU can be successfully unleashed through properly re-organizing sequential computing structures and/or re-designing data parallel algorithms.

The remaining of this paper is organized as follows. In Section 2, we review the typical irregular computing patterns found in EDA applications. Next, the hardware architecture of NVIDIA GPUs and the corresponding data parallel programming model are presented in Section 3. In Section 4, we propose efficient techniques to solve the sparse matrix vector product problem on GPUs. The performance of the GPU implementations is also compared with their CPU equivalents. Section 5 discusses how to apply our GPU based techniques to accelerate a series of EDA applications such as static timing analysis and the solution of linear systems for circuit placement. In Section 6 we explain how the sparse-matrix vector product pattern and its underlying techniques can be used to accelerate graph algorithm problem, breadth-first graph traversal. In Section 7, we introduce our GPU based message-passing solutions for a survey propagation based satisfiability (SAT) problem solver and a GPU-accelerated RTL simulator. Finally, we conclude the paper and outline future research directions.

2. Irregular data structures of EDA applications

Many scientific and engineering applications are based on regular data structures such as linear arrays and matrices (i.e., dense matrices). The access of data generally follows a predictable pattern in such data structures. For instance, when computing the product of a matrix and a vector, the data accessing pattern is fixed as long as the dimension of the matrix is known. In addition, a good level of load balance is relatively easy to achieve on a parallel computer. In the matrix vector product example, the workload can be evenly distributed if the computation of one matrix row and the vector is assigned to a single processing element (e.g., a CPU core). Hence, each processing element would need to complete exactly the same amount of job. By taking advantage of the predictability and good load balance, the regular data structures are relatively amenable to parallel computers.

On the other hand, EDA applications are usually based on irregular data structures such as sparse matrix and graphs. The irregularity is determined by the very nature of IC circuit structure. Fig. 1(a) illustrates a simple but commonly found circuit pattern. The circuit can be stored in an EDA database as a graph like the one illustrated in Fig. 1(b). In such a graph, every signal is treated as a node and two nodes share an edge if they are connected through a gate. Such a graph is obviously irregular because the number of edges connected to one node is not uniform across the graph. In addition, the graph is also very sparse in the sense that most nodes only have a small number of edges connecting to neighboring nodes, although a few high-fanout



c

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

d

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

Fig. 1. Sparse matrix and compressed sparse row format. (a) A simple circuit, (b) directed graph corresponding to (a), (c) adjacency matrix corresponding to the graph in (b) and (d) CSR data structures for the matrix in (c).

nets also exist. Besides the graph, a matrix representation is also widely used in many EDA applications requiring numerical analysis. Fig. 1(c) is an adjacency matrix equivalent to the graph in Fig. 1(b). The matrix has a non-zero entry at position *(i, j)* if node *i* and node *j* share an edge. In this example, the value of *(i, j)* is set to 1. It is readily seen that most the matrix entries are zero. It should be noted that there are many different forms of matrices in EDA applications, but the underlying pattern is similar to the adjacency matrix. The sparsity in such a matrix can be utilized to reduce the computation complexity since the zero entries can be skipped. One of most commonly used storage formats is Compressed Sparse Row (CSR) format [20]. Interested readers please refer to [20] for the details as well as other sparse matrix representations. In the CSR format, three 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*, has one entry for each row. The value of *rowptr[k]* is actually the index of the first non-zero entry on row *k* in vector *elem*. In other words, *rowptr[k+1]*–*rowptr[k]* determines the number of non-zero entries on row *k*, while the first non-zero of row *k* appears at position *rowptr[k]* inside vector *elem*. Fig. 1(d) is the CSR representation of the matrix of Fig. 1(c).

The sparse matrix and graphs are prevalent in EDA applications. In [13], 17 major EDA applications were surveyed and it was identified that 4 applications were built on top of sparse matrix operations and 15 applications involved graph manipulations. In addition, the 17 applications do not include technology CAD applications such as device physics and process simulations, which also involve large scale sparse matrices for finite element computations.

Another important irregular pattern is the message-passing mechanism. Such a pattern is typically based on a graph. Nodes in the graph would send messages along the edges. With specifically designed protocols, the message-passing process would converge at a given stage. On the one hand, the message-passing algorithms are friendly to GPU execution because different nodes/edges can be simultaneously handled. On the other hand, message-passing algorithms have to be irregular because of the irregular structure. In this work, we consider two message-passing algorithms, survey propagation based SAT solution and RTL simulation.

The irregular data structures pose significant difficulty for modern computer architectures because of the insufficient data locality and predictability. For instance, the computing throughput for sparse matrices is considerably slower than that for dense matrix, no matter on a sequential or a parallel platform (e.g., [21] and [22]). On parallel machines, a high computing throughput for irregular data structures is ever harder to maintain due to the poor load balance.

3. Overview of CUDA platform

In this work, we use NVIDIA's CUDA platform to accelerate irregular EDA applications. Similar to Intel's x86, CUDA is an instruction set architecture supported by a family of NVIDIA GPU products. In this section, we briefly introduce the essential details of CUDA-enabled hardware and its programming model.

3.1. Hardware architecture

The architecture of NVIDIA's flagship GPU chip, GT200, is illustrated in Fig. 2 (partly adapted from [5]). The main computing resource consists of 240 scalar processors (SPs), evenly distributed into 30 streaming multiprocessors (SM). A single SP has its own execution hardware, but no instruction fetch and decoding capabilities, which would be taken care of by the SM. A SM would fetch instructions and schedule them on its 8 internal SPs. Two special functional units (SFUs) and a double precision unit are also 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 and later series of NVIDIA GPUs. The GPU chip is actually an array of SMs working concurrently. Given proper computing and communication patterns, the peak floating point throughput of GT200 series can reach 700~800 GFLOPS [22].

During GPU computing, data would be stored in video memory, i.e., so-called global 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). In order to hide the long latency, 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 boosts effective memory bandwidth by taking advantage of the parallel architectures of memories. Every SM is equipped with a 16 KB shared memory, which could provide up to 16 single

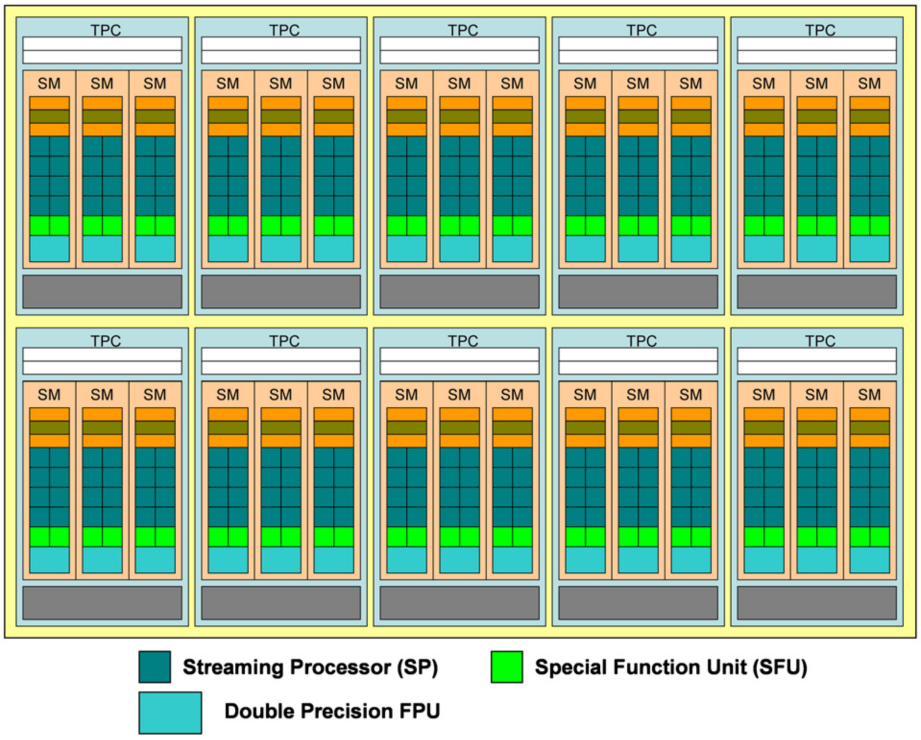


Fig. 2. NVidia GPU architecture.

precision floating numbers of data in one clock cycle. It is a 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. And the new released Fermi architecture [26] GTX480 introduced L2 cache, more configurable (16 KB/48 KB) shared memory/L1 cache and more cores in the SM, continuing to provide increasing power for high performance computing.

3.2. CUDA programming model

Tailored for data parallel computing on NVIDIA GPUs, CUDA [23] provides a parallel programming model, which determines how the parallelism can be conveyed with a programming language. A CUDA program is composed of code segments running on both a CPU and a GPU. Coordinated by CPU, GPU code would be concurrently executed by GPU. A 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 GPU 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. 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. Threads in different blocks could share data through the global memory. The left side of Fig. 3 is an illustration of the hierarchical thread organization.

The 3-tier thread organization is tailored to match the hardware architecture. The corresponding relations are illustrated in Fig. 3(partly adapted from [23]). A thread runs on a given SP when

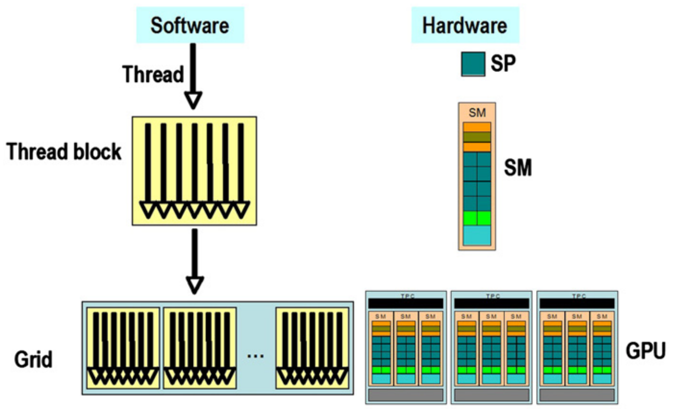


Fig. 3. Parallel program organization in CUDA.

it is active. When a kernel is launched, a thread block would be assigned to a certain SM, while the SM could accept multiple thread blocks as long as the total number of threads is under a threshold. A global controlling unit schedules and maps a grid of thread blocks to SMs on a GPU. During execution, 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.

4. Sparse matrix vector product on GPUs

EDA applications involve many different operations on sparse matrices. After investigating a wide range of such EDA applications including circuit simulation, circuit placement and finite element based layout stress analysis, we find that sparse matrix vector product (SMVP) procedure is often the bottleneck. For example, our conjugate-gradient based linear systems solver [24]

```
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;
    }
}
```

Fig. 4. 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.

In this section, the related work for computing the sparse matrix vector product problem is reviewed in Section 4.1. Then we identify the performance bottleneck of solving SMVP on GPUs and propose a set of efficient techniques accordingly. The experimental results and acceleration factors are reported in Section 4.3.

4.1. Introduction and prior work

Given the CSR sparse matrix format, the sequential code (in standard C language) for computing SMVP is listed in Fig. 4. In this program, 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 entry with index *j* will be multiplied by the vector element with index as *col[j]*.

Starting with the code listed in Fig. 4, a straightforward implementation of SMVP with CUDA would create multiple threads with each computing of a single row. Refs. [14,25] provide details for such an approach. The performance, however, turns out to be unsatisfying. Only a 20% speed-up can be attained on a set of placement problem instances [25]. And there exist tough matrix instances where the straightforward GPU implementation can be slower than its CPU equivalent [25]. A careful analysis reveals two major reasons leading to the inefficiency. First, the memory access cannot be optimally coalesced because one thread (computing one element for the product vector) needs to load a varying number of potentially inconsecutive data words from memory. In fact, over 95% of the memory accesses are non-coalesced for many problems [25]. Secondly, the load balance is poor since most rows only have a handful of non-zeros, while there also exist rows that have hundreds of nonzero. Therefore, the GPU run time is dominated by those less sparse rows. Due to the above two problems, most of the times over 80% of the GPU cores have to idle.

In [14], Bell and Garland proposed a very novel solution to the 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 long strips (>32) of non-zeros, 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. Our experiments indicated that the speed-up is still within $5\times$ for EDA problem instances.

4.2. Efficient CUDA implementations for SMVP

By carefully analyzing the code listed in Fig. 4, 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 rather 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. Considering the difference, the two phases should be organized as two different 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 Fig. 5(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 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 Fig. 5(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. The expanded vector, *v_expanded*, has the same length as *elem[]*. As shown in Fig. 5(b), The value of an entry is set as *v_expanded[elemid]=v[col[elemid]]*. Fig. 5(c) listed the improved implementation with perfectly coalesced memory accesses. 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 certainly incurs a memory overhead, but the order of space complexity would not change and EDA instances are usually very sparse. Experimental results showed that the *product* kernel could be accelerated by one order of magnitude using expanded vectors.

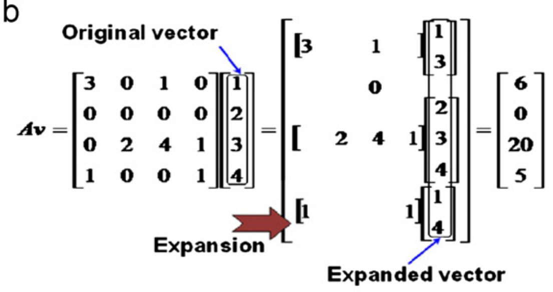
The second kernel, which is designated the *summation* kernel, is responsible of summing the products in one row together and then output it to the product vector. The major bottleneck is again the un-coalesced memory access due to the irregular distribution of non-zeros.

As mentioned in Section 3, a 16 KB shared memory is installed into each streaming multiprocessor. In the case of SMVP, each product created by our first kernel is only used once in the summation process and so there is no need for data sharing

a

```
__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]];
}
```

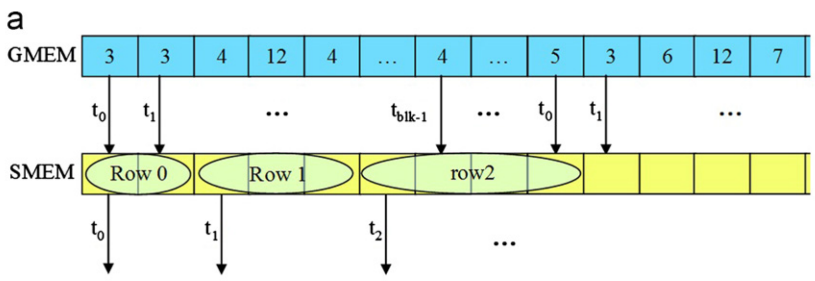
b



c

```
__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];
}
```

Fig. 5. Product kernel. (a) Straightforward implementation, (b) an example of vector expansion and (c) implementation with expanded vector.



b

```
 1 __global__ void summation_kernel(const unsigned int *rowptr, float *middle,
 2                                  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 - num_nz_before) < SHARED_MEM_PER_BLOCK)
27                 sum += cache[ cache_idx];
28             else
29                 sum += middle[ i];
30         }
31         p[row] = sum;
32     }
33 }
```

Fig. 6. Summation kernel. (a) Coalescing through shared memory and (b) source code of summation kernel.

among different threads. However, the shared memory can be used to make the memory access coalesced. The idea is illustrated in Fig. 6(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 memory in a coalesced manner. The summation process could then use data in the shared memory. Of course, the load balance still cannot be perfect in this case. However, the overhead is considerably smaller because only addition operations need to be done.

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 16 KB 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 are 10 non-zeros on a row on average, the hit ratio is still higher than 45%. In addition, in the recently announced new generation NVIDIA GPU, Fermi, a 64 KB L1 cache/shared memory will be installed for every multiprocessor and a 768 KB L2 cache will be available for the whole chip [26]. So our caching mechanism would be more efficient.

The *summation* kernel is listed in Fig. 6(b). 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 [14]. 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.

4.3. Results

We tested our SMVP kernels on a wide range of sparse matrices related to various 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. The CPU implementation of SMVP procedure was carefully optimized and the CPU time is slightly faster than Matlab [27] (for a matrix with 14 M non-zeros, our CPU implementation spends 72 ms on average, while Matlab needs 81 ms).

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 the *B&G* kernel. The source code of *B&G* was downloaded from NVIDIA's CUDA forum [15]. Note that *B&G* kernel was later incorporated into NVIDIA's CUSP package [16].

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. As we can see from the "DATA TRANS PERCENT" column of Tables 3 and 4, the data transfer time is less than 32% compared to 100 times of GPU kernel execution time (obtained by NVIDIA's CUDA profiler). The overhead is acceptable because SMVP generally would be called for hundred or even thousands of times. In fact, the overhead would be less than 4% when the SMVP kernel is called for over 1000 times. Similarly, we do not include the vector expansion time since many times the expansion can be amortized by a large number of iterations. In addition, the GPU expansion kernel is very efficient. It usually finishes within 20% of the execution time of the *product* kernel, while the *summation/warped summation* kernel dominates the execution time in all the experiments. In a typical VLSI design session, it is common to load and prepare design data immediately after launching EDA software, while this overhead is amortized across the large number of succeeding operations.

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.

Table 1
Characteristics of matrix set 1.

Instance	# Rows	# Columns	# Non-zeros	Avg. # non-zeros per row	Description
ns3Da	20,414	20,414	1,679,599	82.3	3D Navier Stokes equation
raefsky3	21,200	21,200	1,488,768	70.2	Turbulence problem
venkat01	62,424	62,424	1,717,792	27.5	2D Euler solver
b18_100K.	100,000	333,740	14,327,592	143.3	Static timing analysis
Yangliu	235,620	235,620	17,563,926	74.5	Finite element based layout stress analysis

Table 2
Characteristics of matrix set 2.

Instance	# Rows	# Columns	# Non-zeros	Avg. # non-zeros per row	Description
Lin	256,000	256,000	1,766,400	6.9	Large sparse Eigenvalue problem
t2em	921,632	921,632	4,590,832	5.0	Electromagnetic problems
ecology1	1,000,000	1,000,000	4,996,000	5.0	Circuit theory applied to animal/gene flow
cont11	1,468,599	1,961,394	5,382,999	3.7	Linear programming
sls	1,748,122	62,729	6,804,304	3.9	Large least-squares problem
G3_circuit	1,585,478	1,585,478	7,660,826	4.8	AMD circuit simulation
thermal2	1,228,045	1,228,045	8,580,313	7.0	FEM, steady state thermal problem
kkt_power	2,063,494	2,063,494	12,771,361	6.2	Optimal power flow, nonlinear optimization
Freescale1	3,428,755	3,428,755	17,052,626	5.0	Freescale circuit simulation

Table 3
SMVP throughput for matrix set 1.

Instance	CPU	*B&G*	Speed-up	*Product+ summation*	Speed-up	*Product+warped summation*	Speed-up	CUSPARSE	Speed-up	Data trans percent (%)
ns3Da	0.51	5.02	9.84	3.79	7.43	18.08	**35.43**	6.79	13.31	5.20
raefsky3	0.53	12.29	23.15	5.41	10.19	14.54	**27.40**	7.29	13.75	5.18
venkat01	0.53	8.13	15.47	5.71	10.87	8.63	**16.43**	4.15	7.83	7.45
Yangliu	0.49	9.08	18.53	5.67	11.57	20.20	**41.22**	7.39	15.08	4.66
b18_100K	0.55	5.14	9.38	5.12	9.34	27.42	**50.03**	7.42	13.49	4.41

Table 4
SMVP throughput for matrix set 2.

Instance	CPU	*B&G*	Speed-up	*Product+ summation*	Speed-up	*Product+ warped summation*	Speed-up	CUSPARSE	Speed-up	Data trans percent (%)
Lin	0.26	1.23	4.81	9.23	**36.04**	1.28	5.01	4.16	16.0	5.74
t2em	0.29	1.54	5.40	12.41	**43.44**	1.68	5.88	5.11	17.62	**31.44**
ecology1	0.24	0.93	3.87	9.03	**37.43**	1.01	4.20	3.15	13.13	25.32
cont11	0.31	1.14	3.63	10.66	**33.84**	1.25	3.95	1.25	4.03	25.67
sls	0.28	1.18	4.26	10.10	**36.49**	1.32	4.77	5.09	16.41	28.44
G3_circuit	0.21	0.91	4.24	8.86	**41.45**	0.99	4.64	3.08	14.67	25.44
thermal2	0.21	1.24	5.81	8.97	**41.89**	1.35	6.31	3.86	18.38	24.87
kkt_power	0.26	1.23	4.77	5.70	**22.01**	1.34	5.16	3.61	13.88	25.22
Freescale1	0.29	1.65	5.76	11.56	**40.37**	1.88	6.57	3.49	12.03	25.09

The fifth 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 [28], while the remaining two are from static timing analysis (explained later in Section 4.1) and finite element based layout stress analysis [29]. All taken from [28], 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.

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 [23]. Columns 2, 3, 5, 7 and 9 report the GFLOPS results of CPU, the *B&G* kernel, the *product+summation* kernels, and the product+warped summation kernels, and cusparseScsrmv kernel (released by NVIDIA's built-in CUSPARSE library [23]), respectively. Columns 4, 6, 8 and 10 collect the corresponding speedup values against the CPU baseline implementation. To rule out the variance of execution time, we conducted 100 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 16 × on all test cases and over 40 × on the two largest matrices. Meanwhile, due to the perfect parallelization of the element-wise multiplication, our implementation also significantly outperforms the original *B&G* kernel.

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 16 × on all test cases and over 40 × on the two largest matrices. Meanwhile, due to the perfect 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 for this set of matrices is realized by the *B&G* kernel with a speedup of around 5 ×. Now our kernels accelerate all test cases by one order of magnitude (over 30 × 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 40 ×.

From Tables 3 and 4, it can be seen that our GPU implementations also outperform NVIDIA's recently release CUSPARSE built-in library.

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 reorganization 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.

5. Applications of GPU based SMVP procedure in EDA computing

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.

5.1. Delay calculation for static timing analysis (STA)

In [30], the authors 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 path matrix can be established by allocating one row to each path and one column to each pin. An entry *(i, j)* of the matrix equals to 1 if pin *j* belongs to path *i*. A delay vector is built by allocating an entry for every pin and the value of an entry is chosen as the delay associated with the specific pin. Under such a set-up, the product vector of the path matrix and the delay vector will naturally have each entry equal to the delay of a path. As a result, the problem of path based delay calculation is simply a direct application of the SMVP procedure.

We tested our SMVP kernels on two largest ITC99 benchmark circuits, b18 and b19 [31]. For each circuit, we created two test cases by randomly choosing 50K and 100K paths, respectively. The delay vector is constructed by using the parameters from a 0.13 μm standard cell library [32]. 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 100 M paths per second, equivalent to a speedup of around 50 × against CPU implementations. Note that the combination of the *product+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 computing density.

Table 5
Characteristics of STA matrices.

Instance	# Rows	# Columns	# Non-zeros	Avg. # non-zeros per row
b18_50K	50,000	333,740	7,057,712	141.2
b18_100K	100,000	333,740	14,327,592	143.3
b19_50K	50,000	673,144	5,562,170	111.2
b19_100K	100,000	673,144	11,332,042	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**

5.2. 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. [10] and [11]). In fact, more than 90% of the CPU time of the global placement process would be spent on solving linear systems using a Conjugate Gradient (CG) method [33]. In a CG solver, generally over 90% CPU time is consumed by the SMVP procedure according to our experiences on a force-driven placer [24]. In other words, a faster CG solver would be fundamental to expedite many EDA applications.

We implemented a CG solver on top of our SMVP kernels. The pseudo code for the CG method is listed in Fig. 7. The matrix M is a preconditioning matrix used to improve the convergence speed. We used a diagonal preconditioning matrix in this work and so a parallelization can be trivially achieved by perform a division on all vector entries simultaneously. Put it in another way, the $Mz^{(i-1)}=r^{(i-1)}$ operation would not really incur a matrix-vector multiplication. Other preconditioning matrices can be more complex, but generally can be similarly handled in parallel. 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 a factor of 50–100 $\times$ on GPUs.

```
compute $r^{(0)} = b - Ax^{(0)}$ for an initial guess $x^{(0)}$
loop while(not convergent)
    solve $Mz^{(i-1)} = r^{(i-1)}$
    $\rho_{i-1} = r^{(i-1)T} \cdot z^{(i-1)}$
    if $i = 1$
        $p^{(1)} = z^{(0)}$
    else
        $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
        $p^{(i)} = z^{(i-1)} + \beta_{i-1} \cdot p^{(i-1)}$
    end if
    $q^{(i)} = A\, p^{(i)}$
    $\alpha_i = \rho_{i-1}/( p^{(i)T} \cdot q^{(i)})$
    $x^{(i)} = x^{(i-1)} + \alpha_i \cdot p^{(i)}$
    $r^{(i)} = r^{(i-1)} - \alpha_i \cdot q^{(i)}$
end loop
```

Fig. 7. Preconditioned conjugate gradient method.

We tested our CG solver on 7 ISPD06 placement benchmark circuits [12]. The first 3 columns of Table 7 show the statistics of these circuits. We then use a hybrid approach defined in [34] 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 3–5, the *product+summation* kernels deliver the best GFLOPS values as expected. The solution time can be reduced by a factor of around 20 $\times$ on GPU's. For the largest test case with 2.5 M cells, the GPU implementations could be 28 times faster.

6. Graph algorithmic patterns on GPUs

In the previous 2 sections, we introduce efficient techniques to accelerate the SMVP pattern and its related applications on GPUs. In this section, we tackle another category of irregular computing pattern, graph algorithm operations. As illustrated in Fig. 8, the sparse matrix is closely related to many graph problems arising from EDA applications. We show that graph algorithmic patterns can be efficiently accelerated either through converting them into an equivalent sparse matrix based formulation or taking advantage of techniques developed for SMVP pattern.

Garland [25] already showed how to compute the shortest path using SMVP. Our experiments proved that on average a 15 $\times$ 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 levelized 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.

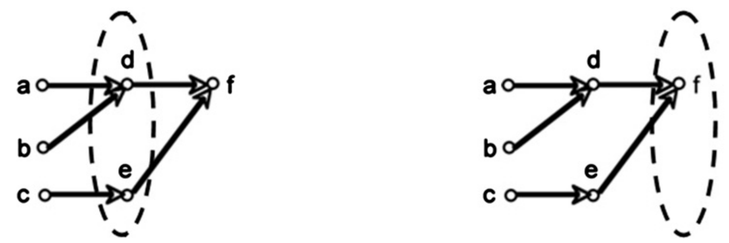
In EDA applications, a circuit is usually represented as a direct graph as illustrated in Fig. 8. A breadth-first traversal would start at the input nodes and then proceed toward the output nodes. However, a direct manipulation of a graph proves to be difficult

Table 7
Characteristics of placement matrices.

Instance	# Cells	# Nets	# Rows	# Columns	# Non-zeros	Avg. # non-zeros per row
new_blue1	330,474	228,901	398,096	398,096	1,595,758	4.0
new_blue2	441,516	465,219	529,672	529,672	2,340,198	4.4
new_blue3	494,011	552,199	561,819	561,819	1,875,296	3.3
new_blue4	646,139	637,051	784,321	784,321	3,152,532	4.0
new_blue5	1,233,058	1,284,251	1,447,872	1,447,872	6,287,726	4.3
new_blue6	1,255,039	1,288,443	1,521,703	1,521,703	6,287,132	4.1
new_blue7	2,507,954	2,636,820	3,068,717	3,068,717	15,120,230	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



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

Fig. 8. BFS through sparse matrix vector product.

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	333,740	869.45	3.84E+07	67.22	4.96E+08	**12.93**
b18_1	315,683	802.28	3.93E+07	63.02	5.01E+08	**12.73**
b19	673,144	1795.15	3.75E+07	130.43	5.16E+08	**13.76**
b19_1	638,283	1685.31	3.79E+07	120.83	5.28E+08	**13.95**

for an efficient GPU implementation [17]. Accordingly, we propose a sparse matrix based solution for the BFS traversal to leverage the efficiency of our SMVP kernels.

6.1. Implementation

Here the graphs are represented as an adjacency matrix, A, as described in Section 2. An entry (i, j) of the adjacency matrix is equal to 1 if node i has a directed edge toward node j. Then the BFS can be realized through a matrix vector product operation. The computation can be based on a SMVP kernel since the adjacency matrix is sparse. Here we also need a vector, x, to record the vertices just reached. Every entry of x corresponds to a vertex in the graph and an entry is set to 1 if it is on the wavefront of the BFS traversal. Then the product of A^T and x would create a new vector, which also has its each entry corresponding to a vertex. The essential point it that an entry of A^Tx has a non-zero value if and only if the matching vertex is connected to the previous wavefront through an edge. This procedure can be repeated by multiplying A^T with the current product vector again and again until all nodes have been visited. A nice feature of this approach is that 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 [35].

Fig. 8 illustrates the breadth first traversal process using the matrix vector product formulation. The dotted circle represents vertices reached after one expansion. We begin expansion from primary input nodes a, b and c. So the first three entries of vector, x, is set to 1 initially. After the first expansion, or equivalently the first product, vertices d and e are reached, and their entries in the product vector have a non-zero value. The entry for vertex d equals to 2, which means there are two paths leading to d. Then the product vector can be multiplied by A^T again. This time vertex f would be reached and there are 3 paths that converge at this vertex.

To check if all nodes are traversed, we need two extra kernels, both already having very efficient GPU implementations. The first kernel assigns one thread for each vertex to check if the 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 [13] and we use the efficient implementation provided in CUDA SDK [23].

6.2. Experimental results

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 [31]. The timing graphs are constructed by treating each pin as a vertex and each gate or wire as a directed edge (please refer to [35] for details). The baseline CPU implementation also uses the sparse matrix formulation, which is faster than the traditional queue based BFS procedure (e.g. the classic implementation outlined in [35]). 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 a single fan-out. Accordingly, the *B&G* kernel and the *product+warped summation* kernels perform very badly on these test cases, where a marginally $2\times$ speedup can be achieved against the CPU baseline. So we only reports results collected by using the *product+summation* kernel. And the speed-up is relatively low compared to before as well. However, the throughput of the GPU implementation is approaching 500 M vertices per second, which used to be achievable only on supercomputers.

7. Message-passing algorithm on GPUs

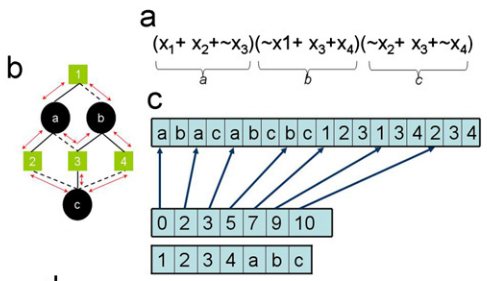
Message-passing is increasingly becoming a major computing pattern [37]. Although it is also based on a graph structure, the basic computing pattern is quite different from the graph algorithm because generally a message-passing algorithm could correctly finish at any order of message processing. Such a feature makes it inherently proper for parallel execution. In this section, we use GPU to accelerate two message-passing algorithms, survey propagation based SAT solution and asynchronous RTL simulation.

7.1. Survey propagation based SAT solution on GPUs

The SAT, or Boolean Satisfiability problem, is one of the best known computer science problems [36]. Besides its theoretic importance, SAT solvers are also the workhorse of modern formal verification tools [38]. It has been deployed in a wide range of verification applications including equivalence checking (e.g., [39]), property checking (e.g., [40]), assertion-based verification (e.g., [41]), and design intent coverage (e.g., [42]).

The SAT problem is generally defined in the conjunctive normal form (CNF) format, which is a formula linking clauses together by AND ($\wedge$) operators. A clause is a disjunction of literals. A literal can be either a variable or its negation. An example of CNF is shown in Fig. 9(a). The solution to a SAT problem is an assignment of variables (TRUE or FALSE) such that the formula is TRUE, or to prove that such an assignment does not exist. If an assignment could be found to make a formula equal to TRUE, the formula is satisfiable, otherwise it is unsatisfiable.

Modern SAT solvers can be roughly classified into two categories, DPLL styled deterministic solvers and iterative stochastic solvers [43]. Modern DPLL solvers have been made extremely efficient on structured problem instances and could provide complete solutions. However, they often have difficulty to handle random instances [44]. Besides, the inherent recursive structure and backtrack mechanism made such solvers less amenable to GPUs. On the other hand, statistic optimization based solvers are relatively easy to be implemented on GPUs because local searches could generally be performed in parallel. Among different statistical optimization based SAT solvers, Survey propagation, receives great success at solving hard instances of SAT problems and is considered as one of the latest major breakthroughs in stochastic SAT solvers [19].



a
(x1+ x2+~x3)(~x1+ x3+x4)(~x2+ x3+~x4)
a
b
c
b
1
a
b
2
3
4
c
c
a b a c a b c b c 1 2 3 1 3 4 2 3 4
0 2 3 5 7 9 10
1 2 3 4 a b c

d

$$\pi^u(t) = \left[1 - \prod_u (1-\eta^u)\right] \prod_s (1-\eta^s)$$

$$\pi^s(t) = \left[1 - \prod_s (1-\eta^s)\right] \prod_u (1-\eta^u)$$

$$\pi^0(t) = \prod_0 (1-\eta^0)$$

e

$$\eta^{t+1} = \prod \left[\frac{\pi^u(t)}{\pi^u(t)+\pi^s(t)+\pi^0(t)}\right]$$

Fig. 9. Solving SAT with the survey propagation algorithm, (a) a SAT problem instance, (b) factor graph for (a), (c) data structure for storing the factor graph of (b) for the GPU implementation, (d) message from variable to clause and (e) message from clause to node.

Survey propagation is a relatively new incomplete algorithm that has its origin in statistical physics. It is actually derived from the cavity method developed for the study of spin glasses [20]. It uses a graph data structure, i.e., factor graph, to represent a CNF formula. The factor graph corresponding to Fig. 9(a) is illustrated in Fig. 9(b), where variables are represented as circular nodes with integer labels, and clauses are encoded in square vertices with alphabet labels. If a variable or its negation shows up in a clause, the variable and clause nodes are connected by an edge. An edge drawn in solid line means the variable appears as a positive variable, while a dashed line indicates the variable appears as its negation.

The survey propagation algorithm is based on a message passing mechanism. A survey datum η associated to every literal node. The initial values of η are generated randomly in the interval *(0, 1)*. Then the η values are passed to the neighboring clause nodes. Upon receiving η values from neighboring variable nodes, every clause node calculates three values according to the formulas in Fig. 9(d). The above 3 values are again sent back to the variable nodes as messages. Each variable node now updates η according to the formula in Fig. 9(e). Then the messages pass back to the clause. This continues until all η being converged, and using these converged η, values of some variables could be fixed so that the hard SAT problem would be simplified and become extremely trivial. Please refer to [19] for the details of the SP algorithm.

7.1.1. Data parallel implementation

The survey propagation algorithm can be parallelized in a rather straightforward manner. The original CPU implementation [45] adopts a random sequence of clauses to update the message from variables to clause and sequentially update the message from clause to variables in this clause in the sequence of variable appearance. Experiments show that the ordering of messages passing does not affect the convergence behavior. In other words, for either clauses or variables, the messages can be updated concurrently. Meanwhile, when the number of clauses and variables are large, the computing density can be rather high. Such properties make this algorithm very friendly to GPUs.

In the GPU implementation, the factor graph is stored as an adjacency list [36]. Here we need two arrays as illustrated in Fig. 9(c). One array, *adj*, records all the edges in the graph. The other array, *nodePtr*, has an entry for each node to book-keep the starting position of its first edge in the previous array. In Fig. 9(c), the data structure of the exemplar factor graph is also illustrated. It can be seen that the storage format is actually a simplified version of the compressed row storage (CRS) format, especially if we consider the similarities between *nodePtr* and *rowPtr* arrays as well as *col* and *adj* arrays. We still need a few other arrays to store type of nodes, π and η values, and so on.

To implement SP on GPUs, we construct two kernels to update π and η values, respectively. Both kernels would deploy one thread for one clause (variable). When updating π *(η)*, a thread needs to read the neighboring variables (clause) nodes from array *adj* and perform relevant computations. In other words, the operations are very similar to the row operations in the SMVP implementation. Accordingly, we can adapt the row-wise summation kernels to the SP context. The shared memory based coalescing mechanism is also very useful here, and we only need to replace the original summation operation to the operations defined by the formulas listed in Fig. 9(d) and (e).

7.1.2. Experimental results

We use a 2.66 GHz Intel Core2 Duo E6750 with 4 GB RAM and a NVIDIA GTX280 GPU. The CPU version of implementation of the survey propagation algorithm is distributed on the website of the author [45].

First we used three 18,000-variable benchmarks from the random large 3-SAT section of SAT competition 2009 [46]. These problem instances have 18,000 variables and 75,600 clauses, with the ratio of the number of clauses to variables 4.2, which belong to the hardest 3-SAT problems. In fact, we tested a few leading-edge, deterministic solvers (prize-winners of recent SAT competitions) on the problems, and on average it takes over 10 min to attack such hard instances. The sequential survey propagation algorithm already outperforms the deterministic solvers by a factor of 3. So in Table 10, we compare the experimental results between the sequential and GPU implementations of SP. The GPU implementations could deliver a speed up of around 18 $\times$.

We also tested a given set of random hard (again the ratio of number of clauses over variables is 4.2) 3-SAT instances. We tested the problem instances until with the number of variables being up to 900,000 and reach the memory limitation of the NVIDIA GTX280 graphics card. In the experiments, we fixed 1% variables at each round of iteration. Table 11 shows the results. Our GPU implementation is around 20 $\times$ faster than its CPU equivalent. In a previous work, the GPU implementation realized a 9 $\times$ speed-up on an NVIDIA GTX 7900 GPU [47]. Our work distinguishes from [47] in two aspects. First, we excised the CUDA general purpose programming model instead of a graphics API. Second, [47] used a single kernel to handle the message updating process, but our work exploits two kernels to take advantage of the varying parallelism inherent at the variable and clause updating stages.

7.2. Parallel RTL simulation on GPUs

As mentioned before, verification is taking a dominant portion of a typical IC design process. Among various verification methods for digital circuits, logic simulation is the most fundamental means [57]. With the constantly increasing IC complexity, logic simulation can be extremely time-consuming. Logic simulation can be performed at multiple abstraction levels. A few recent works proposed effective GPU solutions for gate level simulation (e.g., [50–52]) and SystemC based system level simulation [53]. On the other hand, there has been no work reported on GPU accelerated register-transfer level simulation, although RTL simulation is the most frequently used verification method. Such a lack of progress is mainly due to the extreme irregularity inherent in RTL simulation workload. In this section, we propose a GPU based parallel RTL simulator. To the best of the authors' knowledge, it is world's first work to apply GPUs.in RTL simulation.

Logic simulation algorithms can be classified into two approaches, oblivious simulation and event driven simulation. The former would evaluate all the processes at the each time point no matter whether a process needs to change its state, while the latter would only simulate those processes that receive new input signals. The event-driven approach tends to be more efficient because generally only a small portion of a circuit has activities. From an implementation point of view, a logic simulator can be interpretive and compiled-code based. Interpretive simulation uses a centralized scheduler to manipulate events, while the compiled code simulation transforms the circuit elements to executable code to be directly run on a CPU. In this work, we take an event driven, compiled code approach. The input circuit is captured in Verilog hardware description language [58] and then source-to-source translated equivalent CUDA code for parallel execution.

There are many different parallel logic simulation algorithms. The fundamental approach is to simultaneously execute events happening at the same time, but such events usually could provide sufficient parallelism. Inspired by [52], in this work we take an asynchronous simulation protocol, Chandy–Misra–Bryant (i.e., CMB) algorithm [54,55]. The CMB protocol allows a circuit element to move forward its evaluation as long as its next state can be determined. A recent work [52] proved that such a protocol could extract more parallelism to better exploit the computing power of GPUs.

7.2.1. Implementation

Fig. 10 shows an example circuit described at register transfer level. At register transfer level, circuit behavior is embedded in processes, which would be executed concurrently. Note that the RTL description is similar to the gate level circuit representation shown in Fig. 1, but with the gates replaced by the processes. The RTL simulation is inherently parallel because the processes naturally run concurrently.

GPU execution follows a SIMD manner in the sense that every 32 threads in a warp would always follow exactly the same instruction schedule. For gate simulation, even though different gates have distinct logic functions, they can be handled by looking

Table 10
Runtime of random benchmarks from SAT 2009 competition.

Instance	CPU(s)	GPU(s)	Speed-up
1	363.56	20.00	**18.2** $\times$
2	343.10	19.04	**18.0** $\times$
3	346.67	19.40	**17.9** $\times$

Table 11
Results of random benchmark.

# Variables	CPU(s)	GPU(s)	Speed-up
50,000	45.37	2.32	**19.6** $\times$
100,000	116.34	5.55	**21.0** $\times$
300,000	404.69	19.57	**20.7** $\times$
500,000	710.76	34.31	**20.7** $\times$
900,000	1415.29	66.97	**21.1** $\times$

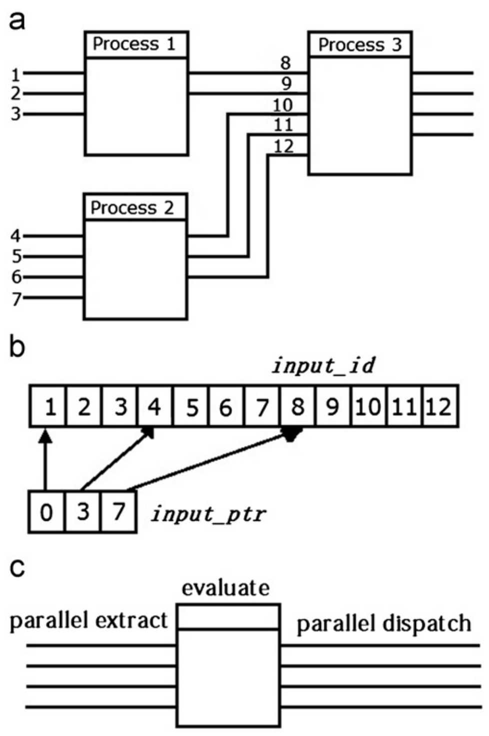


Fig. 10. Structure of RTL simulation, (a) basic structure, (b) input data structure and (c) program flow.

up a truth table so that all threads could have identical program execution behavior. In the case of RTL simulation, the situations are much more complicated because the RTL processes generally have very diverse internal behaviors and would need an impractically large truth table. In other words, RTL simulation belongs to the category of task-parallel applications [56]. If the RTL processes are directly mapped to different threads, the load balance has to be poor due to the divergent program execution paths.

Although SIMD parallelism is not sufficient in RTL simulation, it must be realized that GPUs are also highly multi-threaded. In fact, GPUs have dedicated thread context switching hardware. When a warp needs to wait for memory response, the warp would be suspended and another ready warp would start to run in the next cycle. As a result, we assign every process to a distinctive warp to avoid the divergence. Inside the warp, we use the 32 threads to extract the inputs and distribute the outputs for the process simultaneously Fig. 10(c), but only one single thread for evaluating the process itself. In other words, we take a hybrid parallel scheme by combining task-level and data level parallelism.

We adopted a data structure similar to the CSR format for sparse matrices to store the input pins (and output pins) as illustrated in Fig. 10(b). As shown in Fig. 10(c), our CMB based simulation flow comprises three stages, which are implemented as 3 kernels. The first kernel extracts the input values of the processes, the second is to evaluate every RTL process, and the last is to distribute the output values. The 1st and 3rd kernels use one thread to deal with a signal pin, while the 2nd one deploys a warp to handle a unique process.

The process behavior can be arbitrarily complex. Following the spirit of the compiled-code simulation, we directly translate the Verilog code inside processes into CUDA by taking advantage of the similarity between Verilog and C languages. However, two major differences between Verilog and CUDA have to resolve: (1) Verilog has two types of assignments, blocking and non-blocking [58]; and (2) CUDA does not directly support inter-block

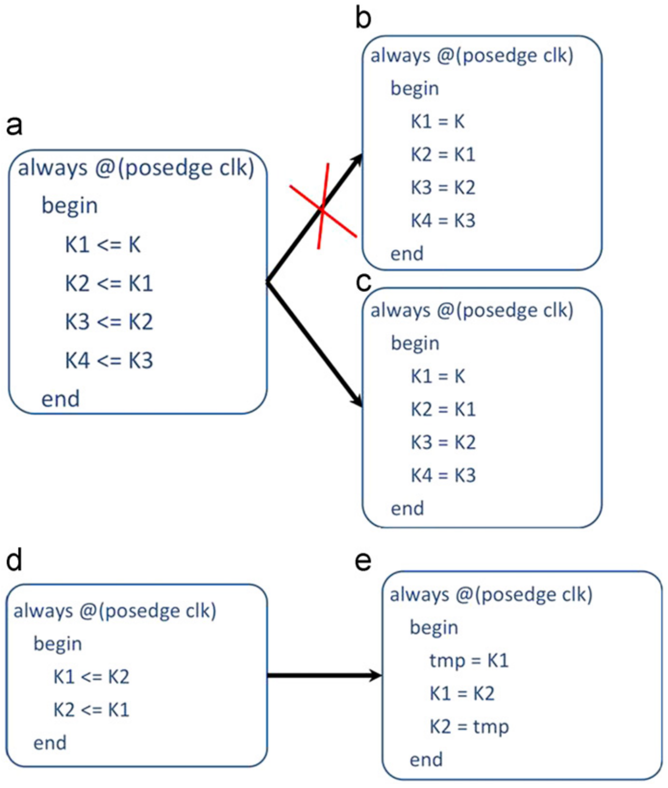


Fig. 11. Non-blocking to blocking transform.

Table 12
Runtime of RTL simulation.

Benchmark	CPU(s)	CPU+GPU(s)	Speed-up
Adder	258.47	13.18	**19.6 ×**
Mux	83.32	4.56	**18.3 ×**
ASIC	178.66	5.06	**35.3 ×**

synchronization. To deal with the first problem, we use dependency sorting to transform all non-blocking assignments to blocking ones because CUDA could only handle such statements. An example is shown in Fig. 11(a), where the *always* block (i.e., a process in Verilog) implies four flip-flops after synthesis. If we directly change the non-blocking statements to blocking ones as exemplified in Fig. 11(b), the behavior would be significantly different because all the assignments would be executed sequentially. The corresponding hardware would consist of a single flip-flop. On the other hand, if we perform a dependency sorting in which an assignment would be placed before another assignment if the former has its right-hand side appearing in at the left-hand side of the latter. In case of value swap or other dependency that cannot be processed in a straightforward manner, intermediate temporary variables would be introduced. This is illustrated by the transformation of the *always* block listed in Fig. 11(d) to the one in Fig. 11(e). The second problem, fortunately, could be automatically solved through the CMB protocol with the help of kernel launch with little overhead.

7.2.2. *Experimental results*

In this section, we present the preliminary results of our GPU accelerated RTL simulator. The benchmarks used in this work include a full-adder array, a multiplex channel and an ASIC built on top of the DES core taken from Opencores [59]. Currently, the Verilog to CUDA translation is half-automatic and only a core Verilog subset is supported. The experimental results were collected on a NVIDIA Tesla C2050 GPU, while the CPU reference simulator is a commercial one, Mentor Graphics' ModelSim SE 6.6d [60], running on an Intel Core i3 2.93 GHz based PC with a Wndows7 OS. Each benchmark was simulated for 100,000 cycles under randomized input patterns. Table 12 summarizes the simulation results. On the three benchmarks, we observed a speed-up of over 18 × . And we observed that the simulator behaves better on compute- intensive benchmarks (e.g., 35 × speed-up for ASIC), where each process has more computing activities. The major bottleneck is the reading and writing to the global memory. For a process with a lower computing density, the CPU time for evaluating the process would be negligible when compared with the time for memory accesses.

8. Conclusion and future work

Modern GPUs have become powerful computing platforms at an affordable price. It is thus appealing to accelerate the time consuming EDA applications with GPUs so as to shorten the VLSI design turn-around time. 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 three important irregular EDA computing patterns, the sparse-matrix vector product, graph theoretic problems and message passing algorithms. We first proposed an efficient data parallel the SMVP implementation, which could accelerate many computing intensive EDA applications by an order of magnitude. Then we adapt the SMVP procedure to address irregular graph algorithmic problems. By introducing a new formulation of breadth-first search using sparse matrix, the breadth-first search operation can be efficiently solved on GPUs with a speedup of over 10×. We also developed efficient GPU solutions for two message-passing algorithms, survey propagation and parallel event-driven logic simulation. Our work establishes that with proper transformation of computational structures and re-designing of algorithms, GPUs could serve as a powerful platform to solve irregular EDA problems.

In the future, we are going to extend our work in several directions to continue the effort of using GPU to expedite the IC design verification process. First, we will investigate the feasibility of GPU for massively parallel SPICE simulation with a direct method [48]. To address the strong data dependency in the LU factorization process hindering an efficient GPU implementation, we are currently exploring a good reordering of the matrix as well as the computing flow to allow high parallelism. Another appealing approach is the so-called asynchronous relaxation method [49]. Secondly, we are exploring the potential of a hybrid SAT solver that orchestrates both CPU and GPU to solve SAT problems in a concurrent manner. One key feature is that the CPU and GPU solvers would exchange information to expedite the overall solution process. Finally, we will continue to optimize the GPU accelerated RTL simulator so as to automate the Verilog to CUDA translation process.

Acknowledgment

This work was partially supported by a faculty initialization funding from the Tsinghua University and NVIDIA Professor Partnership Awards. We acknowledge hardware donation from NVIDIA. We thank Dr. N. Bell and Dr. M. Garland for sharing their code and paper on the sparse matrix vector product problem. We are also indebted to Prof. Chun Zhang and Dr. Hua Li for their generous help in setting up our simulation environment.

References

[1] S.R. Maeng, Embedded computer systems, Lecture 1 (2009) 〈http://camars.kaist.ac.kr/~maeng/cs310/embedded09.htm〉.
[2] Y. Deng, High-performance embedded computing with many-core processors, in: Proceedings of the Tsinghua-Leuven Workshop, 2010.
[3] B. Catanzaro, K. Keutzer, B. Su, Parallelizing CAD: a timely research agenda for EDA, in: Proceeding of the 45th Annual Design Automation Conference, 2008, pp. 12–17.
[4] D. Blythe, Rise of the graphics processor, Proceeding of IEEE 96 (5) (2008) 761–778.
[5] NVIDIA, Fermi GPU Whitepaper.
[6] Tech Report, Nvidia's Fermi GPU Architecture Revealed, techreport.com/articles.x/17670.
[7] Z. Feng, P. Li, Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms, in: Proceeding of the International Conference on Computer-Aided Design, 2008, pp. 647–654.
[8] K. Gulati, S.P. Khatri, Towards Acceleration of fault simulation using graphics processing units, in: Proceedings of the 45th Annual Design Automation Conference, 2008, pp. 822–827.
[9] K. Gulati, J.F. Croix, S.P. Khatri, R. Shastry, Fast circuit simulation on graphics processing units, in: Proceedings of the 2009 Asia and South Pacific Design Automation Conference, 2009, pp. 403–408.
[10] J.M. Keinhans, G. Sigl, F.M. Johannes, K.J. Antreich, Gordian: VLSI placement by quadratic programming and slicing optimization, IEEE Transactions on CAD 10 (3) (1991) 356–365.
[11] H. Eisenmann, F.M. Johannes, Generic global placement and floorplanning, in: Proceedings of the 35th Annual Design Automation Conference, 1998, pp. 269–274.
[12] G.J. Nam, J.C. Alpert, P.G. Villarrubia, ISPD 2005/2006 Placement Benchmarks. Modern Circuit Placement, Springer, US, 2007 (Chapter 1).
[13] G.E. Blelloch, Vector Models for Data-Parallel Computing, MIT Press, 1990.
[14] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, vol. 18, 2009.
[15] N. Bell, Sparse Matrix-Vector Multiplication on CUDA, 〈http://forums.nvidia.com/index.php?showtopic=83825&st=0〉, 2008.
[16] 〈http://code.google.com/p/cusp-library/〉.
[17] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: Proceeding of the 14th International Conference on High Performance Computing, 2007, pp. 197–208.
[18] Y. Deng, B. Wang, S. Mu, Taming irregular EDA applications on GPUs, in: Proceeding of the 2009 International Conference on Computer-Aided Design, 2009, pp. 539–546.
[19] A. Braunstein, M. Mezard, R. Zecchina, Survey propagation: an algorithm for satisfiability, Random Structures and Algorithms 27 (2) (2005) 201–226.
[20] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, 2000.
[21] A. Ghuloum, et al., Ct: Flexible Parallel Programming for Terascale Architectures, Intel Whitepaper, 2007.
[22] V. Volkov, J.W. Demmel, LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs, Technical Report No. UCB/EECS-2008-49, EECS Department, University of California, Berkeley, 2008.
[23] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 3.2, 2010.
[24] Y. Deng, S. Mu, The potential of GPUs for VLSI physical design automation, in: Proceedings of the 9th International Conference on Solid-State and Integrated-Circuit Technology, 2008, pp. 2272–2275.
[25] M. Garland, Sparse matrix computations on manycore GPU's, in: Proceedings of the 45th Design Automation Conference, 2008, pp. 2–6.
[26] NVIDIA WHITEPAPER, NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}, 2009.
[27] MATHWORKS,Matlab V7.4. 〈http://www.mathworks.com〉, 2007.
[28] T. Davis, University of Florida Sparse Matrix Collection 〈http://www.cise.ufl.edu/research/sparse/matrices〉, 2009.
[29] J. Xue, et al., Layout-dependent STI stress analysis and stress-aware RF/analog circuit design optimization, in: Proceedings of the 2009 International Conference on Computer Aided Design, 2009, pp. 521–528.
[30] A. Ramalingam, G.J. Nam, A.K. Singh, M. Orshansky, S.R. Nassif, D.Z. Pan, An accurate sparse matrix based framework for statistical static timing analysis, in: Proceedings of the 2006 International Conference on Computer-Aided Design, 2006, pp. 231–236.
[31] ITC BENCHMARKS (2nd release). Available from: 〈http://www.cad.polito.it/tools/itc99.html〉, 1999.
[32] SMIC, 0.13 um Low Leakage Cadence PDK. 〈http://www.smics.com/website/cnVersion/DS/SMIC-PDK.htm〉, 2009.
[33] R. Barret, et al., Templates for the Solution of Linear Systems, second ed., SIAM, 1994.
[34] N. Viswanathan, C. Chu, FastPlace: efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model, in: Proceedings of the 2004 International Symposium on Physical Design, 2005, pp. 26–33.
[35] S. Sapatnekar, Timing, first ed., Springer, 2004 (Chapter 5).
[36] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press, 2001.
[37] B. Frey, Graphical Models for Machine Learning and Digital Communication, MIT Press, 1998.
[38] M.R. Prasad, A. Biere, A. Gupta, A Survey of recent advances in SAT-based formal verification, International Journal on Software Tools for Technology Transfer 7 (2) (2005) 156–173.
[39] E. Goldberg, M. Prasad, R. Brayton, Using SAT for combinational equivalence checking, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2001, pp. 114–121.
[40] P. Dasgupta, A Roadmap for Formal Property Verification, first ed., Springer, 2006.
[41] H.D. Foster, A.C. Krolnik, D.J. Lacey, Assertion-Based Design, second ed., Springer, 2004.
[42] A. Sinha, et al., Design intent coverage revisited, ACM Transactions on Design Automation of Electronic Systems 14 (1) (2009) 9.
[43] A. Biere, M. Heule, H. Van Maaren, T. Walsh, Handbook of Satisfiability, IOS Press, 2009.
[44] S. Pilarski, G. Hu, Speeding up SAT for EDA, in: Proceeding of the Conference on Design, Automation and Test in Europe, 2002, pp. 1081.
[45] R. Zecchina, Survey Propagation Code Release, 〈http://users.ictp.it/~zecchina/SP〉, 2005.
[46] SAT COMPETITION, 3-SAT Benchmark, 〈http://www.satcompetition.org/2009〉, 2009.
[47] P. Manolios, Y. Zhang, Implementing survey propagation on graphics processing units, in: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, 2006.
[48] T. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006.
[49] G.M. Baudet, Asynchronous iterative methods for multiprocessors, Journal of the ACM 25 (2) (1978) 226–244.
[50] D. Chatterjee, A. DeOrio, V. Bertacco, GCS: high-performance gate-level simulation with GPUs, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2009, pp. 1332–1337.
[51] D. Chatterjee, A. DeOrio, V. Bertacco, Event-driven gate-level simulation with GPUs, in: Proceedings of the 46th annual Design Automation Conference, 2009, pp. 557–562.
[52] Bo D. Wang, Yuhao Zhu, Yangdong Deng, Distributed time, conservative parallel logic simulation on GPUs, in: Proceedings of the 47th annual Design Automation Conference, 2010, pp. 761–766.
[53] M. Nanjundappa, et al., SCGPSim: a fast systemC simulator on GPUs, in: Proceedings of the 15th Asia and South Pacific Design Automation Conference, 2010, pp.149–154.

[54] R. Bryant, Simulation of packet communications architec-ture computer system, MIT-LCS-TR-188, MIT, 1977.
[55] K.M. Chandy, J. Misra, V. Holmes, Distributed simula-tion of networks, Computer Networks 3 (1979) 105–113.
[56] The OpenCL Specification V1.1, 2011.
[57] P. Rashinkar, P. Paterson, L. Singh, System-on-a-Chip Verification: Methodology and Techniques, first ed., Springer, 2000.
[58] IEEE, The IEEE Verilog 1364-2001 Standard.
[59] Opencores.org, 〈http://opencores.com/project/des/overview〉.
[60] Mentor Graphics, 〈http://model.com/〉.

Hao Qian is a Master candidate of Institute of Microelectronics, Tsinghua University, Beijing, China. He received his Bachelor degree from School of Microelectronics and Solid-State Electronics University of Electronic Science and Technology of China. His research interests include GPU architecture and high performance computing.

Yangdong Deng is an Associate Professor of Institute of Microelectronics at Tsinghua University, Beijing, China. His research interests include parallel electronic design automation (EDA) algorithms, electronic system level (ESL) design, and parallel program optimization. He initialized the effort of using GPU to accelerate the time-consume VLSI design verification process, which has been the bottleneck of modern VLSI development. His works achieved the best results in GPU based sparse-matrix operations, logic simulation, and software routers. His research is supported by the EDA Key Project of China Ministry of Science and Technology, Tsinghua-Intel Center of Advanced Mobile Computing Technology, Intel University Programs, NVidia Professor Partnership Awards, and others. He also leads the systems modeling team of the Tsinghua-Intel Center of Advanced Mobile Computing Technology.

He received his Ph.D. degree in Electrical and Computer Engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2006. He received his ME and BE degrees in Electronic Department from Tsinghua University, Beijing, in 1998 and 1995, respectively. Before joining Tsinghua University, he was a consulting technical staff of Magma Design Automation, San Jose, USA.

Bo Wang is currently a Ph.D. candidate of Electrical Engineering Department, Stanford University. He received his B.S. degree from Tsinghua University in Electrical Engineering. His research interests include parallel and distributed computing. He has been doing research on multi-processors and GPUs.

Shuai Mu is a Ph.D. candidate of Institute of Microelectronics, Tsinghua University, Beijing, China. He received his Bachelor degree from Institute of Microelectronics, Tsinghua University in 2008. He also received the award of outstanding undergraduates in Tsinghua University in 2008. Currently, his research interests include parallel algorithm implementation for EDA on GPU, architecture design and optimization based on GPGPU. He has published 4 papers on international conference on parallel computing as the first author.