# Fast Circuit Simulation on Graphics Processing Units

Kanupriya Gulati†, John F. Croix‡, Sunil P. Khatri† and Rahm Shastry‡
† Department of ECE, Texas A&M University, College Station, TX
‡ Nascentric, Inc., Austin, TX

*Abstract*— **SPICE based circuit simulation is a traditional workhorse in the VLSI design process. Given the pivotal role of SPICE in the IC design flow, there has been significant interest in accelerating SPICE. Since a large fraction (on average 75%) of the SPICE runtime is spent in evaluating transistor model equations, a significant speedup can be availed if these evaluations are accelerated. This paper reports on our early efforts to accelerate transistor model evaluations using a Graphics Processing Unit (GPU). We have integrated this accelerator with a commercial fast SPICE tool. Our experiments demonstrate that significant speedups (2.36× on average) can be obtained. The asymptotic speedup that can be obtained is about 4×. We demonstrate that with circuits consisting of as few as about 1000 transistors, speedups in the neighborhood of this asymptotic value can be obtained. By utilizing the recently announced (but not currently available) quad GPU systems, this speedup could be enhanced further, especially for larger designs.**

## I. Introduction

SPICE [1] is the de facto industry standard for circuit level simulation of VLSI integrated circuits. SPICE simulation is typically infeasible for designs larger than 20,000 devices. With the rapidly decreasing minimum feature sizes of devices, the number of devices on a single chip has significantly increased. As a result, it becomes critically important to run SPICE on larger portions of the design to validate electrical and timing behavior before tape-out. Further, process variations increasingly impact the electrical behavior of a design. This is often tackled by performing Monte Carlo SPICE simulations, requiring significant computing and time resources.

As a result, there is a significant motivation to speed up SPICE simulations without losing accuracy. In this paper, we present an approach to accelerate the computationally intensive component of SPICE, by exploiting the parallelism available in graphics processing units (GPUs). In particular, our approach parallelizes and accelerates the transistor model evaluation in SPICE, for BSIM3 [2] models. Our benchmarking shows that BSIM3 model evaluations comprise about 75% of the SPICE runtime. By accelerating this portion of SPICE, therefore, a speedup of up to 4× can be obtained in theory. Our results show that in practice, our approach can obtain a speedup of about 2.36× on average, and as high as 3.07× for one design. *The significance of this is further underscored by the fact that our approach is implemented and integrated in a commercial SPICE accelerator tool, which presents significant speed gains over traditional SPICE implementations, even without GPU based acceleration.*

The SPICE algorithm and its variants simulate the nonlinear time-varying behavior of a design, by employing the following key procedures:

- Formulation of circuit equations using Modified Nodal Analysis [3] (MNA) or Sparse Tableau Analysis [4] (STA).
- Evaluating the time-varying behavior of the design using numerical integration techniques, applied to the non-linear circuit model
- Solving the non-linear circuit model using Newton-Raphson (NR) based iterations.
- Solving a linear system of equations in the inner loop of the engine.

The main time-consuming computation in SPICE is the evaluation of device model equations in different iterations of the above flow. In fact, our profiling experiments, using BSIM3 models, show that on average 75% of the SPICE runtime is spent in performing these evaluations. This is because these evaluations are performed for each device, and possibly repeated for each time step, until the convergence of the NR based non-linear equation solver. The total number of such evaluations can easily run into the billions. Therefore, the speed of the device model evaluation code is a significant determinant of the speed of the overall SPICE simulator [3]. For more accurate models like BSIM4 [5], which account for additional electrical behaviors of deep sub-micron (DSM) processes, the relative runtimes of model evaluations are even higher, and thus the asymptotic speedup that can be obtained by accelerating these evaluations is more than 4×.

This paper focuses on the acceleration of SPICE by performing the transistor model evaluations on the GPU. An industrial design could require several thousand device model evaluations for a given time step. These evaluations are independent. In other words the device model computation requires that the same set of model equations be evaluated, possibly several thousand times, for different devices with no dependency on each other. This property of the device model evaluations matches well with the single instruction multiple data (SIMD) computational paradigm that GPUs implement. Our current implementation handles BSIM3 models. However, using the approach described in the paper, we can easily handle BSIM4 models, or a combination of different models.

GPUs are designed as graphics accelerators for image manipulations, 3D rendering operations, etc., for graphical displays in laptop and desktop computer systems. These graphics acceleration tasks require that the same operations are performed independently on independent regions of display data. Therefore, GPUs are designed to operate in a SIMD fashion, which is a natural computational paradigm for graphical display manipulation tasks. General purpose computations using GPUs have been actively explored [6], [7], [8] in recent times. The significant increase in the number of scientific communities exploring GPUs to implement their computationally intensive algorithms is a testament to the tremendous parallelism that is inherent in the GPU platform. The peak computational and memory bandwidths of recent GPU are quite staggering in contrast to high-end CPUs. Further, the development of open-source programming tools and languages for interfacing with the GPU platforms, along with continuously evolving GPU architectures, has further fueled the growth of general purpose GPU (GPGPU) applications.

Inexpensive GPUs with large memories, large memory bandwidths and high degrees of parallelism are available off the shelf. As observed in [9], the theoretical performance of the GPU has grown from 50 Gflops for the NV40 GPU in 2004 to more than 500 Gflops for G80 GPU (which is used in the GeForce 8800 GTX graphic card) in 2007/8. This high computing power mainly arises due to a heavily pipelined and highly parallel architecture, with extremely high memory bandwidths. GPU memory bandwidths have grown from 42 GB/s for the ATI Radeon X1800XT to 86.4 GB/s for the NVIDIA GeForce 8800 GTX GPU. In contrast the theoretical performance of a 3 GHz Pentium4 CPU is 12 Gflops, with a memory bandwidth of 6 GB/s to main memory.

Our device model evaluation engine is implemented in the Compute Unified Device Architecture (CUDA), which is an open-source programming and interfacing tool provided by NVIDIA for programming their GPU devices. The GPU device used for our implementation and benchmarking is the NVIDIA GeForce 8800 GTS.

Performing the evaluation of device model equations for several thousand devices is a natural match for capabilities of the GPU. This is because such an application can exploit the extreme memory bandwidths of the GPU, as well as the presence of several computation elements on the GPU. To the best of the authors' knowledge, this paper is the first to accelerate circuit simulation on a GPU platform.

The key contributions of this paper are:

- We *exploit the match between parallel device model evaluation and the capabilities of a GPU, a SIMD-based device*. This enables us to harness the computational power of GPUs to accelerate device model evaluations.
- Our implementation caters to the key features required to obtain maximal speedup on a GPU.
  - The different threads, which perform device model evaluations, are implemented so that there are no data or control dependencies between threads.
  - All device model evaluation threads compute identical instructions, but on different data, which exploits the SIMD architecture of the GPU.
  - The values of the device parameters required for evaluating the model equations are obtained using a device memory lookup, thus exploiting the extremely large memory bandwidth of GPUs.
- Our device model evaluation is implemented in a manner which is aware of the specific constraints of the GPU platform such as the use of (cached) texture memory for table lookup, memory coalescing for device memory accesses and the balancing of hardware resources used by different threads. This helps maximize the speedup obtained.
- Our approach is integrated into a commercial circuit simulation tool OmegaSIM [10]. A CPU-only implementation of OmegaSIM is on average 10× to 1000× faster than SPICE (and about 10× faster than other fast SPICE implementations). With the device model evaluation performed using our GPU based implementation, OmegaSIM is sped up by an additional factor of 2.36×, on average.

The remainder of this paper is organized as follows. Some previous work in circuit simulation has been described in Section II. Section III details the architecture of the GPU device and the CUDA programming tool. Section IV details our approach for implementing device model evaluation on a GPU. In Section V we present results from experiments which were conducted by implementing our approach and integrating it in OmegaSIM. We conclude in Section VI.

## II. Previous Work

Several fast SPICE implementations depend upon hierarchical isomorphism to increase performance [11], [12], [13]. In other words they extract hierarchical similarities in the design, and avoid redundant simulations. This approach works well for regular designs such as memories, which exhibit a high degree of repetitive hierarchical structure. However, it is less successful for random logic or other designs without repetitive structures. This approach is not efficient for simulating the post place-and-routed design, since back-annotated capacitances vary significantly so that repetitive blocks of hierarchy can no longer be considered to be identical in terms of their electrical behavior. Our approach parallelizes device model evaluations at each timestep, and hence exhibits a healthy speedup regardless of the regularity (or lack thereof) in a circuit. As such, our approach is orthogonal to the hierarchical isomorphism based techniques.

A transistor level engine targeted for interconnect analysis is proposed in [14]. It makes use of the successive chord (SC) integration method (as opposed to NR iterations) and a table-lookup model for $I_{ds}$ currents. The approach re-uses LU factorization results across multiple timesteps and input stimuli. As noted by the authors, the SC method does not provide all desired convergence properties of the NR method for general analog simulation analysis. In contrast, our approach speeds up device model evaluation for arbitrary circuits in a classical SPICE framework, due to its robustness and industry-wide popularity. Our early experiments demonstrate that model evaluation comprises the majority (∼75%) of the total circuit simulation runtime. Our approach is orthogonal to the non-linear system solution approach, and can thus be used in tandem with the approach of [14] if desired.

The approach of [15] proposed speeding up device model evaluation by using the PACE [16] distributed memory multiprocessor system, with a four-processor cluster. They targeted transient analysis in ADVICE, an AT&T circuit simulation program similar to SPICE, which is available commercially. Our approach, in contrast, exploits the parallelism available in an off-the-shelf GPU for speeding up device model evaluations. Further, their experimental results discuss the speedup obtained for device model evaluation (*alone*) for small example circuits to be about 3.6×. Our results speed up device model evaluation by 30-40× on average. The speedup obtained using our approach for the *entire* SPICE simulation is 2.36× on average. Further, their target multiprocessor system requires the user to perform load balancing upfront. The CUDA architecture and its instruction scheduler (which handles the GPU memory accesses) together abstract the problem of load balancing away from the user. Also, the thread scheduler on the GPU periodically switches between processors to efficiently and dynamically balance their computational resources, without user intervention.

The authors of [17] proposed speeding up circuit simulation using a shared memory multiprocessor system, namely the Alliant FX/8 with a six-processor cluster. They too target transient analysis in ADVICE, but concentrate on two routines – i) an implicit numerical integration scheme to solve the time-varying non-linear system, and ii) a modified approach for solving the set of non-linear equations. In contrast, our approach uses a commercial off-the-shelf GPU to accelerate only the device model evaluations, by exploiting the SIMD computing paradigm of the GPU. During numerical integration, the authors perform device model evaluation by device type. In other words, all resistors are evaluated at once, then all capacitors are evaluated followed by MOSFETs etc. In order to avoid potential conflicts due to parallel writes, the authors make use of locks for consistency. Our implementation faces no such issues, since all writes are automatically synchronized by the scheduler and are thus conflict free. Therefore, we obtain significantly higher speedups. The experimental results of [17] indicate a speedup for *device model evaluation* of about 1-6×. Our results demonstrate speedups for device model evaluation of about 30-40×. They do not report runtimes or speedup obtained for the entire circuit simulation. We improve the runtime for the complete circuit simulation by 2.36× on average.

The commercial tool we used for integrating our implementation of GPU based device model evaluation is OmegaSIM [10]. OmegaSIM's core is a multi-engine, current-based architecture with multi-threading capabilities. Details about the OmegaSIM architecture are not pertinent to this paper, since we implement only the device model evaluations on the GPU.

The application of GPUs for general purpose computations has been actively explored [6], [7], [8]. In the VLSI domain, the approach of [18] accelerates fault simulation by using GPUs. However, to the best of the authors' knowledge, the use of GPUs for accelerating circuit simulation is non-existent.

## III. GPU Architecture

In this section, we discuss the architectural aspects of the NVIDIA GeForce 8800 GTS GPU device used in our experiments. We discuss the hardware model, memory model and the programming model for this device. This discussion is provided to help the reader to better understand the specifics of our implementation of a device model evaluation engine on the GPU. Further details of the GPU architecture can be found in [19], [20].

### A. Hardware Model

The GeForce 8800 GTS architecture has 16 multiprocessors per chip and 8 processors (ALUs) per multiprocessor. During any clock cycle, all the processors of a multiprocessor execute the same instruction, but may operate on different data. There is no mechanism to communicate *between* the different multiprocessors. In other words, no native synchronization primitives exist to enable communication between multiprocessors. We next describe the memory organization of the device.

### B. Memory Model

The memory model of NVIDIA GTS 8800 is shown in Figure 1. Each multiprocessor has on-chip memory of the following four types [19], [20]:

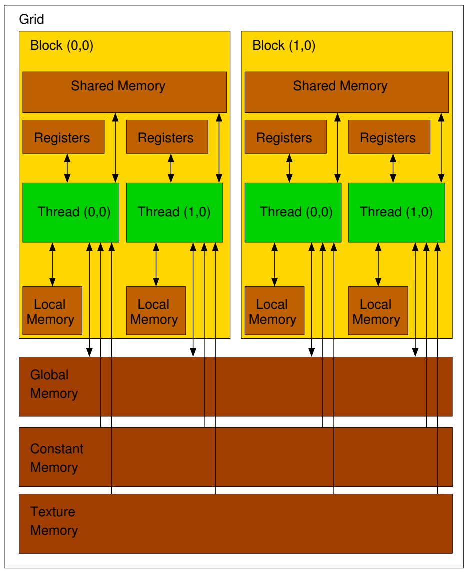

Fig. 1. Memory Model of the NVIDIA GeForce GTS 8800

- One set of local 32-bit *registers* per processor. The total number of registers per multiprocessor is 8192.
- A *shared memory* that is shared by all the processors of a multiprocessor. The size of this shared memory per multiprocessor is 16 KB and it is organized into 16 banks.
- A read-only *constant cache* that is shared by all the processors in a multiprocessor, which speeds up reads from the constant memory space. The size of the constant cache is 8 KB per multiprocessor.
- A read-only *texture cache* that is shared by all the processors in a multiprocessor, which speeds up reads from the texture memory space. The texture cache size is 8 KB per multiprocessor.

The local and global memory spaces are implemented as read-write regions of the device memory and are not cached. These memories are optimized for different uses. The local memory of a processor is used for storing data structures declared in the instructions executed on that processor.

The pool of shared memory within each multiprocessor is accessible to all its processors. Each block of shared memory represents 16 banks of single-ported SRAM. Each bank has 1KB of storage and a bandwidth of 32 bits per clock cycle. Furthermore, since there are 16 multiprocessors on a GeForce 8800 GTS, this results in a total storage of 256KB per multiprocessor. For all practical purposes, this memory can be seen as a logical and highly flexible extension of the local memory. However, if two or more access requests are made to the same bank, a *bank conflict* results. In this case, the conflict is resolved by granting accesses in a serial fashion. Thus, shared memory must be accessed in a fashion such that bank conflicts are minimized.

Global memory is read/write memory that is *not* cached. A single floating point value read from (or written to) global memory can take 400 to 600 clock cycles. Much of this global memory latency can be hidden by if there are sufficient arithmetic instructions that can be issued while waiting for the global memory access to complete. Since the global memory is not cached, access patterns can dramatically change the amount of time spent in waiting for global memory accesses. Thus, coalesced accesses of 32-bit, 64-bit, or 128-bit quantities should be performed in order to increase the throughput and to maximize the bus bandwidth utilization.

The texture cache is optimized for spatial locality. In other words, if instructions that are executed in parallel read texture addresses that are close together, then the texture cache can be optimally utilized. A texture fetch costs one memory read from device memory on a cache miss, otherwise it costs one read from the texture cache. Device memory reads through *texture fetching* routines (provided in CUDA for accessing texture memory) present several benefits over reads from global or constant memory.

A constant memory fetch costs one memory read from device memory only on a cache miss, otherwise it costs one read from the constant cache. The memory bandwidth is best utilized when *all* instructions that are executed in parallel access the same address of the constant memory. We next discuss the GPU programming and interfacing tool.

## C. Programming Model

In this project, the CUDA (Compute Unified Device Architecture) is used to interface to the GPU device. CUDA is a new hardware *and* software architecture for issuing and managing computations on the GPU as a data-parallel computing device, without the explicit need of mapping them to a traditional graphics API [19], [20]. CUDA was released in early 2007.

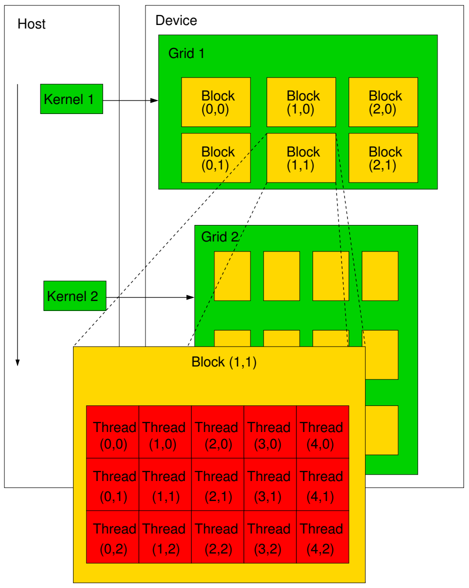

Fig. 2. CUDA Programming Model

CUDA's programming model is summarized in Figure 2. When programmed through CUDA, the GPU is viewed as a compute device capable of executing a large number of threads in parallel. Threads are the atomic units of parallel computation, and the code they execute is called a kernel. The GPU device operates as a coprocessor to the main CPU, or host. Data-parallel, compute-intensive portions of applications running on the host can be off-loaded onto the GPU device. Such a portion is compiled into the instruction set of the GPU device and the resulting program, called a kernel, is downloaded to the device.

A thread block (equivalently referred to as a block) is a batch of threads that can cooperate by efficiently sharing data through some fast shared memory and synchronize their execution to coordinate memory accesses. Users can specify *synchronization points* in the kernel. Threads in a block are suspended until they all reach the synchronization point. Threads are grouped in warps, which are further grouped in blocks. Threads have identity numbers *threadIDs* which can be viewed as a one, two or three dimensional value. All the warps comprising a block are guaranteed to run on the same multiprocessor, and can thus take advantage of shared memory and local synchronization. Each warp contains the same number of threads, called the warp size, and is executed in a SIMD fashion; a *thread scheduler* periodically switches between warps. In case of the GeForce 8800 GTS, the warp size is 32. Thread blocks have restrictions on the *maximum* number of threads in them. In a GeForce 8800 GTS, the maximum number of threads in a thread block is 512. However, the number of threads in a thread block, *dimblock*, is decided by the programmer, who must ensure that i) the maximum number of threads allowed in the block is 512 ii) *dimblock* a multiple of the warp size.

A thread block can be executed by a single multiprocessor. However, blocks of same dimensionality (i.e. orientation of the threads in them) and size (i.e number of threads in them) that execute the same kernel can be batched together into a *grid* of blocks. The number of blocks in a grid is referred to as *dimgrid*. A grid of thread blocks is executed on the device by executing one or more blocks on each multiprocessor using time slicing. However, at a given time, at the most 768 threads can be active in a single multiprocessor. When deciding the *dimblock* and *dimgrid* values, the restriction on the number of registers being used in a single multiprocessor (i.e. 8192) has to be carefully monitored. If this limit is exceeded, the kernel will fail to launch.

The GeForce 8800's synchronization paradigm is *local* to a thread block, and is very efficient. However, threads belonging to different thread blocks (even in the same grid) cannot synchronize.

We next discuss details of our implementation of device model evaluation on the 8800 GPU.

## IV. Our Approach

The SPICE [21], [1] algorithm simulates the nonlinear time-varying behavior of a circuit using the following steps.

- First, the circuit equations are formulated using Modified Nodal Analysis (MNA). This is typically done by *stamping* the MNA matrix based on the types of devices included in the SPICE netlist, as well as their connectivity.
- The time-varying behavior of the design is solved using numerical integration techniques applied to the non-linear circuit model. Typically, the trapezoidal method of numerical integration is used, although the Gear method may be optionally used. Both these methods are implicit methods and are highly stable.
- The non-linear circuit model is solved using Newton-Raphson (NR) based iterations. In each iteration, a linear system of equations needs to be solved. During the linearization step, device model equations need to be evaluated, to populate the linear system of equations.
- Solving a linear system of equations forms the inner loop of the SPICE engine.

We profiled the SPICE code to find the fraction of time that is spent performing device model evaluations, on several circuits. These profiling experiments, which were performed using OmegaSIM, showed that on average 75% of the total simulation runtime is spent in performing device model evaluations for industrial designs. As an example, for the design *Industry_1*, which performs the functionality of a Linear Feedback Shift Register (LFSR), 74.9% of the time was spent in BSIM3 device model evaluations. The *Industry_1* design had 324 devices, and required $1.86\times10^7$ BSIM3 device model evaluations over the entire simulation.

We note that the device model evaluations can be performed in parallel, since they need to be evaluated for every device. Further, these evaluations are possibly repeated (with different input data) for each time step until the convergence of the NR based non-linear equation solver. Therefore, billions of these evaluations could be required for a complete simulation. Also, these computations are independent of each other, exposing significant parallelism for medium to large sized designs. The speed of execution of the device model evaluation code, therefore, significantly determines the speed of the overall SPICE simulator. Since the GPU platform allows significant parallelism, it forms an ideal candidate platform for speeding up model evaluations. Since device model evaluations consume about 75% of the runtime of a CPU based SPICE engine, we can obtain an asymptotic maximum speedup of 4$\times$ if these computations are parallelized. This is in accordance with Amdahl's Law [22], which states that the overall algorithm performance is limited by the portion that is not parallelizable. In the sequel we discuss the implementation of only the device model evaluation portion of the entire SPICE flow.

Our implementation is integrated into an industrial accelerated SPICE tool called OmegaSIM. Note that OmegaSIM, running in a CPU-only mode, obtains significant speedup over competing SPICE offerings. Our implementation, after integration into OmegaSIM results in a CPU+GPU implementation which is 2.36$\times$ faster on average than the CPU-only version of OmegaSIM.

### A. Parallelizing BSIM3 Model Computations on a GPU

Our implementation supports BSIM3 models. In this section, we make several observations about the careful engineering required in order to parallelize BSIM3 device model computations on a GPU. These ideas are implemented in our approach, and together help us achieve the significant speedup in BSIM3 model computations. Note that BSIM4 device model computations can be parallelized in a similar manner.

*1) Inlining if-then-else Code:* The BSIM3 model evaluation code consist of several *if-then-else* statements, with a maximum nesting depth of 4. This code does not contain any *while* or *for* loops. The input to the BSIM3 device model evaluation routine is a number of device parameters, some of which are unchanged during the model evaluation (these parameters are referred to as *runtime parameters*), while others are computed during the model evaluation. The key task is to perform these computations on a GPU, which has a SIMD computation model. For instance, a code fragment such as

```
Codefragment1()
if(cond){ CODE-A; }
else{ CODE-B; }
```

would be converted into the following code fragment for execution on the GPU.

```
Codefragment2()
CODE-A;
CODE-B;
if(cond){ return result of CODE-A; }
else{ return result of CODE-B; }
```

As mentioned, the programming paradigm of a GPU is the Single Instruction Multiple Data (SIMD) model, wherein all threads must compute identical instructions, but on different data. The different threads being computed in parallel should have no data or control dependency among them, to obtain maximal speedup. GPUs also have an extremely large memory bandwidth, which allows multiple memory accesses to be performed in parallel. The SIMD paradigm is thus an appropriate match for performing several device model evaluations in parallel. Our code (restructured as shown in *Codefragment2()*) can be executed in a SIMD fashion on a GPU, with all kernels executing the same instruction in lock-step, but on different data. Of course, this code fragment requires the GPU to perform more instructions than is the case with the original code fragment. However, the large degree of parallelism on the GPU overcomes this disadvantage, and yields impressive speedups, as we will see in the sequel. The above conversion is handled by the CUDA compiler.

*2) Partitioning the BSIM3 Code into Kernels:* The key task in implementing the BSIM3 device model evaluations on the GPU is the partitioning of the BSIM3 code into smaller fragments, with each fragment being implemented as a GPU kernel.

In the limit, we could implement the entire BSIM3 code in a single kernel, which includes all the device model evaluations required for a BSIM3 model card. However, this would not allow us to execute a sufficiently large number of kernels in parallel. This is because of the limitation on the hardware resources available for every multiprocessor on the GPU. In particular, the limitation applies to registers and shared memory. As mentioned earlier, the maximum number of registers for a multiprocessor is 8192. Also, the maximum amount of shared memory for a multiprocessor is 16 KB. If any of these resources are exceeded, additional kernels cannot be run. Therefore, if we had a kernel with 4000 registers, then no more than 2 kernels can be issued in parallel (even if the amount of shared memory used by these 2 kernels is much less than 16 KB). In order to achieve maximal speedup, the GPU code needs to be implemented in a manner that hides memory access latencies, by issuing hundreds of threads at once. In case a single thread (which implements all the device model evaluations) is launched, it will not leave sufficient hardware resources to instantiate a sufficient number of additional threads to execute the same kernel (on different data). As a result, the latency of accessing off-chip memory will not be hidden in such a scenario. To avert this, the device model evaluation code needs to be partitioned into smaller kernels. These kernels are of an appropriate size such that a large number of them can be issued without depleting the registers or shared memory of the multiprocessor. If, on the other hand, the kernels are too small, then large amounts of data transfer will be required from one kernel to another (this is done via global memory). The data that is written by kernel $k$, and needs to be read by kernel $k+j$, will be stored in global memory. If the kernels are extremely small, a large amount of data will have to be written and read to/from global memory, hampering the performance. Hence, in the other extreme case of very small kernels, we may run into a performance bottleneck as well.

Therefore, keeping in mind the limited hardware resources (in terms of registers and shared memory), and the global memory latency and bandwidth constraints, the device model evaluations are partitioned into appropriately sized kernels which maximize parallelism and minimize the global memory access latency. Satisfying both these constraints for a kernel is important in order to maximally exploit the speedup obtained on the GPU.

Our approach for partitioning the BSIM3 code into kernels first finds the control and dataflow graph (CDFG) of the BSIM3 code. Then we find the disconnected components of this graph, which form a set $D$. For each component $d \in D$, we partition the code of $d$ into smaller kernels as appropriate. The partitioning is performed such that the number of variables that are written by kernel $k$ and read by kernel $k+j$, are minimized. This minimizes the number of global memory accesses. Also, the number of registers $R$ used by each kernel is minimized, since the total number of threads that can be issued in parallel on a single multiprocessor is $8192/R$, rounded down to the nearest multiple of 32, as required by the 8800 architecture. The number of threads issued in parallel cannot exceed 768 for a single multiprocessor.

*3) Efficient use of GPU Memory Model:* In order to obtain maximum speedup of the BSIM3 model evaluation code, the different forms of GPU memory need to be carefully utilized. In this section, we discuss the approach taken in this regard.

- **Global Memory**
  At a coarse analysis level, the device model evaluations in a circuit simulator are divided into
  - Creating a DC model for the device, given the operating voltages at the device terminals.
  - Calculating the different output values that are part of the BSIM3 device evaluation code. These are the values that are returned by the BSIM3 device evaluation code, to the calling routine.

  In order to minimize the data transfers from GPU (device) to CPU (host), the results of the set of kernels that compute the DC model parameters are stored in global memory and are not returned back to the host. Next, when the kernels which calculate the values that need to be returned by the BSIM3 model evaluation routine are executed, they can read (or write) the global memory to fetch the DC model parameters. GPUs also have an extremely large memory bandwidth as discussed earlier, which allows multiple memory accesses to the global memory to be performed in parallel, and their latencies to be hidden.
- **Texture Memory**
  In our implementation, the values of the parameters (referred to as *runtime parameters*) required for performing device model evaluations are stored in the texture memory, and are accessed by performing a texture memory lookup. Using the texture memory (as opposed to global, shared or constant memory) has the following advantages:
  - Texture memory of a GPU device is cached as opposed to shared or global memory. Hence we can exploit the benefits obtained from the cached texture memory lookups.
  - Texture memory accesses do not have coalescing constraints as required in case of global memory accesses, making the runtime parameters lookup efficient.
  - In case of multiple look-ups performed in parallel, shared memory accesses might lead to bank conflicts and thus impede the potential speedup.
  - Constant memory accesses in the GPU, as discussed in Section III, are optimal when all lookups occur at the same memory location. This is typically not the case in parallel device model evaluation.
  - The CUDA programming environment has built-in texture fetching routines which are extremely efficient.

  Note that the allocation and loading of the texture memory requires non-zero time, but this cost is easily amortized since several thousand lookups are typically performed using the same runtime parameter data.
- **Constant Memory**
  Our implementation makes efficient use of the constant memory for storing physical constants such as $\pi$, $\varepsilon_o$, etc., required for device model evaluations. Constant memory is cached, and thus, performing several million device model evaluations in the entire circuit simulation flow allows us to exploit the advantage of a cached constant memory. Since the processors in any multiprocessor of the GPU operate in a SIMD fashion, all lookups for the constant parameters occur at the same memory location at the same time. This results in the most optimal usage of constant memory.

*4) Thread Scheduler and Code Statistics:* Once the threads are issued to the GPU, an in-built hardware scheduler performs the scheduling of these threads.

The blocks that are processed by one multiprocessor in one batch are referred to as active. Each active block is split into SIMD groups of threads called warps. Each of these warps contain the same number of threads (warp size) and are executed by the multiprocessor in a SIMD fashion. Active warps (i.e. all the warps from all the active blocks) are *time-sliced* – a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources.

The statistics for our implementation of the BSIM3 device model evaluation code are reported next. The warp size for a NVIDIA 8800 device is 32. Further, the pool of registers available for the threads in a single multiprocessor is equal to 8192. In our implementation, the *dimblock* size is 32 threads. The average number of registers used by a single kernel in our implementation is around 12. A register count limit of 8192 allows 640 threads of the possible 768 threads in a single multiprocessor to be issued, thus having an occupancy of about 83.33% on average. The multiprocessor occupancy is calculated using the *occupancy calculator worksheet* provided with CUDA. The number of registers used per kernel, and the shared memory per block are obtained using the CUDA compiler (*nvcc*) with the *'-cubin'* option.

## V. Experiments

Our device model evaluation engine is implemented and integrated in a commercial SPICE accelerator tool OmegaSIM [10]. In this section, we present details of our experimental setup and results.

In all our experiments, the CPU used was an Intel Core 2 Quad Core (4 processor) machine, running at 2.40 GHz with 4 GB RAM. The GPU card used for our experiments is the NVIDIA GeForce 8800 GTS with 512 MB RAM, operating at 675 MHz.

We first profiled the circuit simulation code. Over several examples, we found that about 75% of the runtime, is consumed by BSIM3 device model evaluations. For the design *Industrial_1*, the code profiling is as follows:

- BSIM3 device model evaluations = 74.9%
- Non-accelerated portion of OmegaSIM code = 24.1%

Thus, by accelerating the BSIM3 device evaluation code, we can asymptotically obtain around 4× speedup for circuit simulation.

Table I compares our approach of implementing the device model evaluation on the GPU to the device model evaluation on the CPU in terms of runtime. Column 1 reports the number of evaluations performed. Columns 2 and 3 report the GPU runtimes (wall clock), in ms, for evaluating the device model equations and the data transfers (CPU $\rightarrow$ GPU as well as GPU $\rightarrow$ CPU), respectively. In particular, the data transfers include transferring the *runtime parameters* and the operating voltages at the device terminal (for all evaluations) from CPU to GPU. The data transfers from the GPU to CPU include the outputs of the BSIM3 device model evaluation code. Column 4 reports the total (processing+transfer) GPU runtimes (in ms). The CPU runtimes (in ms) are reported in Column 5 and the speedup obtained is reported in Column 6.

| # Evaluations | GPU runtimes (ms) | | | CPU runtime (ms) | Speedup |
|---|---|---|---|---|---|
| | Processing | Transfer | Total | | |
| 1M | 81.17 | 196.48 | 277.65 | 8975.63 | 32.33 × |
| 2M | 184.91 | 258.79 | 443.7 | 18086.29 | 40.76 × |

TABLE I
SPEEDUP FOR BSIM3 EVALUATION

Table II compares the runtime of AuSIM (which is OmegaSIM with our approach integrated, and runs partly on GPU and partly on CPU) against the original OmegaSIM (operating on the CPU alone). Columns 1 and 2 report the circuit name and the number of transistors in the circuit, respectively. The number of evaluations required for full circuit simulation is reported in Column 3. Columns 4 and 5 report the CPU-alone and GPU+GPU runtimes (in seconds), respectively. The speedups are reported in Column 6. The circuits Industrial_1, Industrial_2 and Industrial_3 perform the functionality of an LFSR. Circuits Buf_1, Buf_2 and Buf_3 are buffer insertion instances for buses of 3 different sizes. Circuits ClockTree_1 and ClockTree_2 are symmetrical H-tree clock distribution networks. These results show that an average speedup of 2.36× can be achieved over a variety of circuits. Also, note that with an increase in the number of transistors in the circuit, the speedup obtained is higher. This is because the GPU memory latencies can be better hidden when more device evaluations are issued in parallel.

| Ckt Name | # Trans. | Total # Eval. | OmegaSIM (s) | AuSIM (s) | SpeedUp |
|---|---|---|---|---|---|
| | | | CPU-alone | GPU+CPU | |
| Industrial_1 | 324 | $1.86\times10^7$ | 49.96 | 34.06 | 1.47 × |
| Industrial_2 | 1098 | $2.62\times10^9$ | 118.69 | 38.65 | 3.07 × |
| Industrial_3 | 1098 | $4.30\times10^8$ | 725.35 | 281.5 | 2.58 × |
| Buf_1 | 500 | $1.62\times10^7$ | 27.45 | 20.26 | 1.35 × |
| Buf_2 | 1000 | $5.22\times10^7$ | 111.5 | 48.19 | 2.31 × |
| Buf_3 | 2000 | $2.13\times10^8$ | 486.6 | 164.96 | 2.95 × |
| ClockTree_1 | 1922 | $1.86\times10^8$ | 345.69 | 132.59 | 2.61 × |
| ClockTree_2 | 7682 | $1.92\times10^8$ | 458.98 | 182.88 | 2.51 × |
| Avg | | | | | 2.36 × |

TABLE II
SPEEDUP FOR CIRCUIT SIMULATION

The NVIDIA 8800 GPU device supports IEEE 754 single precision floating point operations. However, the BSIM3 model code uses IEEE 754 double precision floating point computations. We first modified all the double precision computations in the BSIM3 code into single precision before modifying it for use on the GPU. We determined the error that was incurred in this process. We found that the accuracy obtained by our GPU-based implementation of device model evaluation (using single precision floating point) is extremely close to that of a CPU-based double precision floating point implementation. In particular, we computed the error over $10^6$ device model evaluations, and found that the maximum absolute error was $9.0\times10^{-22}$ Ampheres, and the average error was $2.88\times10^{-26}$ Ampheres. The relative average error was $4.8\times10^{-5}$. NVIDIA has announced the availability of GPU devices which will support double precision floating point operations in the near future. Such devices will further improve the accuracy of our approach.

Figure 3 and 4 show the voltage plots obtained for *Industrial_2* and *Industrial_3* circuits, obtained by running AuSIM and comparing it with SPICE. Notice that the plots completely overlap.

## VI. CONCLUSIONS

Given the key role of SPICE in the design process, there has been significant interest in accelerating SPICE. A large fraction (on average 75%) of the SPICE runtime is spent in evaluating transistor model equations. The paper reports our early efforts to accelerate transistor model evaluations using a GPU. We have integrated this accelerator with a commercial fast SPICE tool, and have shown significant speedups (2.36× on average). The asymptotic speedup that can be obtained is about 4×. With the recently announced quad GPU systems, this speedup could be enhanced further, especially for larger designs.

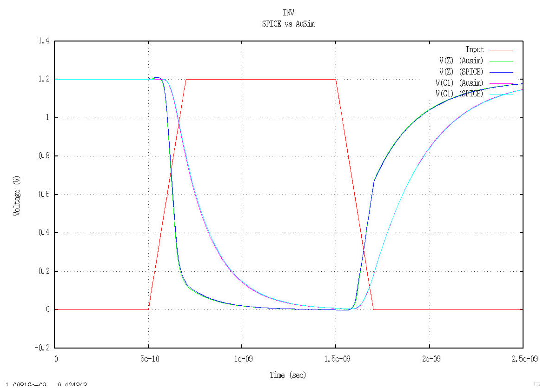

Fig. 3. Industrial_2 waveforms

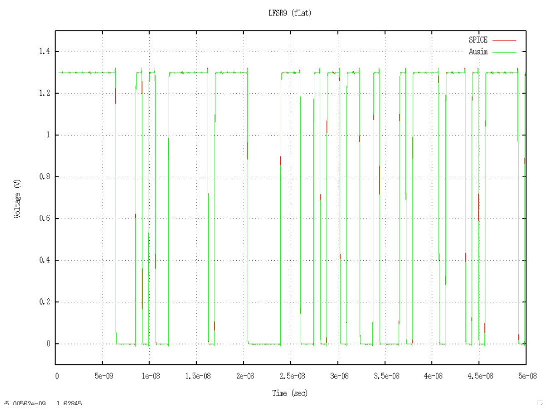

Fig. 4. Industrial_3 waveforms

## REFERENCES

[1] L. Nagel, "SPICE: A computer program to simulate computer circuits," in *University of California, Berkeley UCB/ERL Memo M520*, May 1995.

[2] "BSIM3 Homepage." `http://www-device.eecs.berkeley.edu/~bsim3`.

[3] L. T. Pillage, R. A. Rohrer, and C. Visweswariah, *Electronic Circuit & System Simulation Methods*. Mcgraw-Hill, Dec 1994. ISBN-13: 978-0070501690 (ISBN-10: 0070501696).

[4] G. Hachtel, R. Brayton, and F. Gustavson, "The sparse tableau approach to network analysis and designlation," *Circuits Theory, IEEE Transactions on*, vol. 18, no. 1, pp. 101–113, 1971.

[5] "BSIM4 Homepage." `http://www-device.eecs.berkeley.edu/~bsim4`.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), p. 47, IEEE Computer Society, 2004.

[7] J. Owens, "GPU architecture overview," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 2, ACM, 2007.

[8] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: general-purpose computation on graphics hardware," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 208, ACM, 2006.

[9] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, and T. Chiueh, "Graphic-card cluster for astrophysics (GraCCA) – performance tests," in *Submitted to NewAstronomy*, July 2007.

[10] "OmegaSim Mixed-Signal Fast-SPICE Simulator." `http://www.nascentric.com/product.html`.

[11] "Virtuoso UltraSim Full-chip Simulator." `http://www.cadence.com/products/custom_ic/ultrasim/index.aspx`.

[12] "FineSIM SPICE." `http://www.magmada.com/c/SVX0QdBvGgqX/Pages/FineSimSPICE.html`.

[13] "Capsim Hierarchical Spice Simulation." `http://www.xcad.com/xcad/spicesimulation.html`.

[14] F. Dartu and L. T. Pileggi, "TETA: transistor-level engine for timing analysis," in *DAC '98: Proceedings of the 35th annual conference on Design automation*, (New York, NY, USA), pp. 595–598, ACM, 1998.

[15] P. Agrawal, S. Goil, S. Liu, and J. Trotter, "Parallel model evaluation for circuit simulation on the PACE multiprocessor," in *Proceedings of the seventh international conference on VLSI Design*, pp. 45–48, 1994.

[16] P. Agrawal, S. Goil, S. Liu, and J. A. Trotter, "PACE: A multiprocessor system for VLSI circuit simulation," in *Proceedings of SIAM Conference on Parallel Processing*, pp. 573–581, 1993.

[17] P. Sadayappan and V. Visvanathan, "Circuit simulation on shared-memory multiprocessors," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1634–1642, 1988.

[18] K. Gulati and S. P. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proceedings of the 45th annual conference on Design automation*, pp. 822–827, 2008.

[19] "NVIDIA CUDA Introduction." `http://www.beyond3d.com/content/articles/12/1`.

[20] "NVIDIA CUDA Homepage." `http://developer.nvidia.com/object/cuda.html`.

[21] L. Nagel and R. Rohrer, "Computer analysis of nonlinear circuits, excluding radiation," *IEEE Journal of Solid States Circuits.*, vol. SC-6, pp. 162–182, Aug 1971.

[22] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," *Proceedings of AFIPS*, vol. 30, pp. 483–485, 1967.