Accelerating Statistical Static Timing Analysis Using Graphics Processing Units

Kanupriya Gulati and Sunil P. Khatri
Department of ECE, Texas A&M University, College Station, TX

Abstract— **In this paper, we explore the implementation of Monte Carlo based statistical static timing analysis (SSTA) on a Graphics Processing Unit (GPU). SSTA via Monte Carlo simulations is a computationally expensive, but important step required to achieve design timing closure. It provides an accurate estimate of delay variations and their impact on design yield. The large number of threads that can be computed in parallel on a GPU suggests a natural fit for the problem of Monte Carlo based SSTA to the GPU platform. Our implementation performs multiple delay simulations at a single gate in parallel. A parallel implementation of the Mersenne Twister pseudo-random number generator on the GPU, followed by Box-Muller transformations (also implemented on the GPU) is used for generating gate delay numbers from a normal distribution. The μ and σ of the pin-to-output delay distributions for all inputs and for every gate, are obtained using a memory lookup, which benefits from the large memory bandwidth of the GPU. Threads which execute in parallel have no data/control dependencies on each other. All threads compute identical instructions, but on different data, as required by the Single Instruction Multiple Data (SIMD) programming semantics of the GPU. Our approach is implemented on a NVIDIA GeForce GTX 8800 GPU card. Our results indicate that our approach can obtain an average speedup of about 260$\times$ as compared to a serial CPU implementation. With the recently announced quad 8800 GPU cards, we estimate that our approach would attain a speedup of over 785$\times$. The correctness of the Monte Carlo based SSTA implemented on a GPU has been verified by comparing its results with a CPU based implementation.**

I. Introduction

The impact of process variations is becoming increasingly significant with the rapidly diminishing minimum feature sizes of VLSI fabrication processes. In particular, the resulting increase of delay variations has strongly affected timing yields and maximum operating frequencies of designs. Processing variations can be random or systematic. Random variations are independent of the locations of transistors within a chip. An example is the variation of dopant impurity densities in the transistor diffusion regions. Systematic variations are dependent on locations, for example exposure pattern variations and silicon-surface flatness variations.

Static timing analysis (STA) is used in a conventional VLSI design flow to estimate circuit delay and the maximum operating frequency of the design. In order to deal with variations and to move beyond the limitations of the deterministic nature of traditional STA techniques, *statistical* STA (SSTA) was developed. The main idea of SSTA is to include the effect of variations in order to analyze circuit delay more accurately. Monte Carlo based SSTA is a simple and accurate method of performing SSTA. This method generates N samples of the gate delay random variables and executes static timing analysis runs for each sample. Finally, the results are aggregated to produce the full circuit delay distribution. Such a method is compatible with the process variation data from the fab line, which is essentially in the form of samples of the process random variables. Further, the most attractive property of Monte Carlo based SSTA is the level of accuracy obtained. However, its main drawback is the high runtime cost. By exploiting the parallelism in the Monte Carlo approach for SSTA, and exploring its implementation on a graphics processing unit, we show a *260$\times$ speed up* in the runtime, with no loss of accuracy. Our speedup numbers include the time incurred in transferring data to and from the GPU.

The application of GPUs for general purpose computations has been actively explored in recent times [1], [2], [3], [4]. The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data intensive algorithms has arguably had a contribution in encouraging GPU manufacturers to design GPUs that are easily programmable for general purpose applications. Additionally, the development of open-source programming tools and languages for interfacing with the GPU platforms, along with the continuous evolution of the computational power of GPUs has further fueled the growth of general purpose GPU (GPGPU) applications. The peak performance of GPUs 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 [5]. Memory bandwidths of the GPU have grown from 42 GB/s for the ATI Radeon X1800XT to 86.4 GB/s for the NVIDIA GeForce 8800 GTX GPU. The fully pipelined and highly parallel architecture of the GPU, along with its extremely high memory bandwidths, is responsible for its high peak computation power. However, the performance of a high-end microprocessor today, like the 3 GHz Pentium4 CPU, is $\sim$12 Gflops, and it has a memory bandwidth of 6 GB/s to main memory.

An application which has several instructions that can be issued in parallel, and independent of each other, is a natural match for the GPU's capabilities. Monte Carlo based SSTA fits this requirement well, since the generation of samples, and the corresponding static timing analysis for a single gate computation can be executed in parallel, with no data-dependency. We refer to this as *sample parallelism*. Further, gates at the same logic level can execute Monte Carlo based SSTA in parallel. We call this *data parallelism*, again with zero data-dependency. Employing sample-parallelism and data-parallelism simultaneously allows us to maximally exploit the high memory bandwidths of the GPU, as well as the presence of several processing elements on the GPU. In order to generate the random samples, the *Mersenne Twister* [6] pseudo-random number generator is employed. This pseudo-random number generator can be implemented in a SIMD fashion on the GPU, and thus proves to be extremely suitable for our Monte Carlo based SSTA engine. The μ and σ for pin-to-output falling and rising delay distribution for every input of every gate are stored in a lookup table (LUT) in the GPU device memory. The large memory bandwidth allows us to perform lookups extremely fast. The SIMD computing paradigm of the GPU is thus maximally exploited by our Monte Carlo based SSTA implementation.

In this paper we have only considered uncorrelated random variables while implementing SSTA. Our current approach can be easily extended to incorporate spatial correlations between the random variables, by using principal component analysis (PCA) to transform the original space into a space of uncorrelated principal components. PCA is heavily used in multivariate statistics. In this technique, the rotation of axes of a multidimensional space is performed such

978-1-4244-2749-9/09/$25.00 ©2009 IEEE

that the variations, projected on the new set of axes, behave in an uncorrelated fashion. The computational techniques for performing PCA are implementable in a parallel (SIMD) paradigm, as shown in [7], [8].

Although our current implementation does not incorporate the effect of input slew and output loading effects while computing the delay and slew at the output of a gate, these effects can be easily incorporated. Instead of storing just a pair of μ and σ values for each pin-to-output delay distribution for every input of every gate, we can store $K \cdot P$ pairs of μ and σ values for pin-to-output delay distributions for every input of every gate. Here K is the number of discretizations of the output load and P is the number of discretizations of the input slew values.

To the best of the authors' knowledge, this is the first paper which accelerates Monte Carlo based SSTA on a GPU platform. The key contributions of this paper are:

- We *exploit the natural match between Monte Carlo based SSTA and the capabilities of a GPU, a SIMD-based device*, and harness the tremendous computational power and memory bandwidth of GPUs to accelerate the same.
- The implementation satisfies all the key requirements which ensure maximal speedup on a GPU
  - Different threads which generate normally distributed samples and perform STA computations are implemented so that there are no data dependencies between threads.
  - All gate evaluation threads compute identical instructions but on different data, which exploits the SIMD architecture of the GPU.
  - The μ and σ for any pin-to-output delay of any gate, required for a single STA computation, are obtained using a memory lookup, which exploits the extremely large memory bandwidth of GPUs.
- Our Monte Carlo based SSTA engine is implemented in a manner which is aware of the specific constraints of the GPU platform, such as the use of texture memory for table lookup, memory coalescing, use of shared memory, use of a SIMD algorithm for generating random samples etc., thus maximizing the speedup obtained.
- Our implementation can obtain about 260× speedup compared to a CPU based implementation. This includes the time required to transfer data to and from the GPU.
- Further, even though our current implementation has been benchmarked on a single NVIDIA GeForce GTX 8800 graphics card, the NVIDIA SLI technology [9] supports up to four NVIDIA GeForce GTX 8800 graphic cards on the same motherboard. We show that our performance gains scale with the number of GPU cards, and hence Monte Carlo based SSTA can be performed about 785× faster on a quad GPU system than a conventional CPU based implementation.

Our Monte Carlo based timing analysis is implemented in the Compute Unified Device Architecture (CUDA) framework [10], [11], which is an open-source programming and interfacing tool provided by NVIDIA, for programming NVIDIA GPU devices. The GPU device used for our implementation and benchmarking is the NVIDIA GeForce 8800 GTX. The correctness of our GPU based timing analyzer has been verified by comparing its results against a CPU based implementation of Monte Carlo based SSTA.

The remainder of this paper is organized as follows. Some previous work in SSTA has been described in Section II. Section III details the architecture of the GPU device and the programming tool CUDA. Section IV details our approach for implementing Monte Carlo based SSTA on GPUs. In Section V we present results from experiments which were conducted in order to benchmark our approach. We conclude in Section VI.

II. Previous Work

The approach of [12], [13] are some of the early works in SSTA. In recent times, the interest in this field has grown rapidly. This is primarily due to the fact that process variations are growing larger and less systematic with shrinking feature sizes.

SSTA algorithms can be broadly categorized into *block-based* and *path-based*. In block-based algorithms, delay distributions are propagated by traversing the circuit under consideration in a levelized breadth-first manner. The fundamental operations in a block based SSTA tool are the *SUM* and the *MAX* operations of the μ and σ values of the distributions. Therefore, block based algorithms rely on efficient ways to implement these operations, rather than using discrete delay values. In path-based algorithms, a set of paths is selected for a detailed statistical analysis. While block-based algorithms [14], [15] tend to be fast, it is difficult to compute an accurate solution of the statistical MAX operation when dealing with correlated random variables or reconvergent fanouts. In such cases, only an approximation is computed, using the upper-bound or lower-bound of the probability distribution function (PDF) calculation, or by using the moment matching technique [16]. The advantage of path-based methods is that they accurately calculate the delay PDF of each path since they do not rely on statistical MAX operations, and can account for correlations between paths easily.

Similar to path-based SSTA approaches, our method does not need to perform statistical MAX and SUM operations. Our method is based on propagating the frontier of circuit delay values, obtained from the μ and σ values of the pin-to-output delay distributions for the gates in the design. Unlike path-based approaches, we do not need to select a set of paths to be analyzed.

The authors of [17] present a technique to propagate PDFs through a circuit in the same manner as arrival times of signals are propagated during STA. Principal component analysis enables them to handle spatial correlations of the process parameters. While the SUM of 2 Gaussian distributions yields another Gaussian distribution, the MAX of 2 or more Gaussian distributions is not a Gaussian distribution in general. As a simplification, and ease of calculation, the authors of [17], approximate the MAX of 2 or more Gaussian distributions to be Gaussian as well.

A canonical first-order delay model is proposed and an incremental block based timing analyzer is used to propagate arrival times and required times through a timing graph in the approach presented in [18]. In [19], [20], [21], the authors note that accurate SSTA can become exponential. Hence, they propose faster algorithms that compute only the bounds on the exact result.

In [22], a block based SSTA algorithm is discussed. By representing the arrival times as cumulative distribution functions and the gate delays as PDFs, the authors claim to have an efficient method to do the SUM and MAX operations. The accuracy of the algorithm can be adjusted by choosing more discretization levels. Reconvergent fanouts are handled through a statistical subtraction of the common mode. The authors of [23] propagate delay distributions through a circuit. The PDFs are discretized to help make the operation more efficient. The accuracy of the result in this case is again dependent on the discretization. The approach of [24] automates the process of false path removal implicitly (by using a sensitizable timing analysis methodology [25]). The approach first finds the primary input vector

transitions that result in the sensitizable longest delays for the circuit, and then performs a statistical analysis on these vector transitions alone.

In contrast to these approaches, our approach *accelerates* Monte-Carlo based SSTA technique by using off-the-shelf commercial graphic processing units (GPUs). The ubiquity and ease of programming of GPU devices, along with their extremely low costs makes GPUs an attractive choice for such an exercise. We aim at maximally harnessing the GPU's computational power in this paper.

In recent times, the implementation of general purpose computations on GPUs has been actively explored [1], [2], [3]. In [26], the authors accelerate fault simulation by using GPUs. However, to the best of our knowledge, the use of GPUs for Monte Carlo based SSTA has not been reported to date.

III. Architecture

We next discuss the architectural aspects of the NVIDIA GeForce 8800 GTX GPU device, which is the GPU used in our implementation. This brief discussion is provided to enable a better understanding of our implementation of the Monte Carlo based SSTA engine on the GPU. Additional details of the 8800 GTX can be found in [10], [11].

A. Hardware Model

The GeForce 8800 GTX architecture has a total of 128 cores, that are distributed such that there are 16 multiprocessors per chip and 8 processors per multiprocessor. Since the GPU operates in a SIMD fashion, all the processors (all 128 cores) execute the same instruction during any clock cycle, but may operate on different data. We next describe the memory organization of the 8800 device.

B. Memory Model

There are four types of on-chip memories on each multiprocessor [10], [11]:

- 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 amount of constant cache is 8 KB per multiprocessor.
- A read-only *texture cache* that is shared by all the processors in a multiprocessor. The size of texture cache is 8 KB per multiprocessor.

Global memory is read/write and is *not* cached. The global memory access latency for reading/writing a single floating point value can be 400 to 600 clock cycles. A considerable amount of this global memory access latency can be hidden if there are sufficient arithmetic instructions that can be issued while waiting for a global memory access to complete. Further, coalesced accesses (i.e. accesses which are aligned) 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. A texture cache fetch costs one memory read from device memory on a cache miss, otherwise it just costs a one cycle read. Device memory reads through *texture fetching* (provided in CUDA for accessing texture memory) present several benefits over reads from global or constant memory. We next discuss the GPU programming and interfacing tool.

C. Programming Model

The CUDA (Compute Unified Device Architecture), is a new hardware *and* software architecture which is used for interfacing with the GPU device. It allows the user to issue and manage computations on the GPU without the need of mapping them to traditional graphics APIs [10], [11].

In 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. The GPU device operates as a coprocessor to the main CPU, or host. Data-parallel, compute-intensive portions of an application running on the host can be off-loaded onto the GPU device. Such a portion of code is compiled into the instruction set of the GPU device and the resulting program is called a *kernel*. The kernel is executed on the GPU device.

A *thread block* (also referred to as a block) is a batch of threads that can cooperate efficiently. They do so by sharing data through fast shared memory and synchronizing their execution to coordinate memory accesses. *Synchronization points* can be specified in the kernel. During execution, threads in a block are suspended until they all reach the same synchronization point. Threads are grouped into *warps*, which are further grouped in *blocks*. Threads have one, two or three dimensional identity numbers or *threadIDs*. This helps in accessing the data for problems which have an underlying one, two or three dimensional geometry.

The GeForce 8800's synchronization paradigm is efficient, but it is *local* to a thread block. Threads belonging to different thread blocks cannot synchronize. We next discuss our implementation of Monte Carlo based SSTA on the 8800 GTX GPU.

IV. Our Approach

We accelerate Monte Carlo based SSTA by implementing it on a graphics processing unit (GPU). The following sections describe the details of our implementation. The first section discusses the details of implementing STA on a GPU, and the second section extends this discussion for implementing SSTA on a GPU.

A. Static Timing Analysis (STA) at a Gate

The computation involved in a single STA evaluation at any gate in a design is as follows. At each gate, the MAX of the SUM of the input arrival time at pin i plus the pin-to-output rising (or falling) delay from pin i to the output is computed. The details are explained with the example of a NAND2 gate.

Consider a NAND2 gate. Let AT_i^{fall} denote the arrival time of a falling signal at node i and AT_i^{rise} denote the arrival time of a rising signal at node i. Let the two inputs of the NAND2 gate be a and b, and the output be c.

The rising time (delay) at the output c of a NAND2 gate is calculated as shown below. A similar expression can be written to compute the falling delay at the output c.

$$AT_c^{rise} = MAX[(AT_a^{fall} + MAX(D_{11\rightarrow 00}, D_{11\rightarrow 01})), \\ (AT_b^{fall} + MAX(D_{11\rightarrow 00}, D_{11\rightarrow 10}))]$$

where, $MAX(D_{11\rightarrow 00}, D_{11\rightarrow 01})$ is the pin-to-output rising delay from the input a, while $MAX(D_{11\rightarrow 00}, D_{11\rightarrow 10})$ is the pin-to-output rising delay from the input b.

To implement the above computation on the GPU, a look-up table (LUT) based approach is employed. The pin-to-output rising and falling delay from every input, for every gate is stored in a LUT. The output arrival time of an n-input gate G is then computed by calling the two-input MAX operation n-1 times, after n computations of the SUM of the input arrival time plus the pin-to-output rising (or

falling) gate delay. The pin-to-output delay for pin i is looked up in the LUT at an address corresponding to the base address of gate G and the offset for the transition on pin i. Since the LUT is typically small, these lookups are usually cached. Further, this technique is highly amenable to parallelization as will be shown in the sequel.

In our implementation of the LUT based SSTA technique on a GPU, the LUTs (which contain the pin-to-output falling and rising delays) for all the gates are stored in the texture memory of the GPU device. This has the following advantages:

- Texture memory on a GPU device is cached unlike shared or global memory. Since the truth tables for all library gates easily fit into the available cache size, the cost of a lookup will typically be one cycle.
- Texture memory accesses do not have coalescing constraints as required for global memory accesses. This makes the gate lookup efficient.
- The latency of addressing calculations is better hidden, possibly improving performance for applications like STA that perform random accesses to the data.
- In case of multiple look-ups performed in parallel, shared memory accesses might lead to bank conflicts and thus impede the potential improvement due to parallel computations.
- In the CUDA programming environment, there are built-in texture fetching routines which are extremely efficient.

The allocation and loading of the texture memory requires non-zero time, but is done only once for a library. This runtime cost is easily amortized since several STA computations are done, especially in an SSTA setting.

The GPU allows several threads to be active in parallel. Each thread in our implementation performs STA at a single n-input gate G by performing n lookups from the texture memory, n SUM operations and $n-1$ MAX operations. The data, organized as a 'C' structure *type struct threadData*, is stored in the global memory of the device for all threads. The global memory, as discussed in Section III, is accessible by all processors of all multiprocessors. Each processor executes multiple threads simultaneously. This organization thus requires multiple accesses to the global memory. Therefore, it is important that the memory coalescing constraint for a global memory access is satisfied. In other words, memory accesses should be performed in sizes equal to 32-bit, 64-bit, or 128-bit values. The data structure required by a thread for STA at a gate with 4 input is:

```
typedef struct __align__(8){
int offset; // Gate type's offset
float a; float b; float c; float d; // input arrival times
} threadData;
```

The first line of the declaration defines the structure type and byte alignment (required for coalescing accesses). The elements of this structure are the offset in texture memory (type integer) of the gate for which this thread will perform STA, and the input arrival times (type float).

The pseudocode of the kernel (the code executed by each thread) for static timing analysis is given in Algorithm 1. The arguments to the routine *static_timing_kernel* are the pointers to the global memory for accessing the *threadData* (*MEM*) and the pointers to the global memory for storing the output delay value (*DEL*). The global memory is indexed at a location equal to the thread's unique *threadID* = t_x, and the *threadData* data is thus accessed. The pin-to-output rising (falling) delay for an input x of an *inverting* gate is accessed by indexing the LUT (in texture memory) at the sum of the gate's base address and even (odd) offset of the input x for falling (rising) transition. Similarly, the pin-to-output rising (falling) for an input x for a non-inverting gate, is accessed by indexing the LUT (in texture memory) at the sum of the gate's base address and the odd (even) offset of the input x for rising (falling) transition.

The CUDA inbuilt single-dimension texture fetching function $tex1D(LUT, index)$ is next invoked to fetch the corresponding pin-to-output delay values for every input. The fetched value is added to the input arrival time of the corresponding input. Then, using $n-1$ MAX operations, the output arrival time is computed.

In our implementation, the same kernel implements gates with n = 1, 2, 3 or 4 inputs. For gates with less than 4 inputs, the extra memory in the LUT stores zeroes. This enables us to invoke the same kernel for any instance of a 2, 3 or 4 input inverting (non-inverting) gate.

Algorithm 1 Pseudocode of the kernel for rising output STA for inverting gate

```
static_timing_kernel(threadData*MEM, float*DEL){
t_x = my_thread_id;
threadData Data = MEM[t_x];
p2pdelay_a = tex1D(LUT, MEM[t_x].offset + 2×0);
p2pdelay_b = tex1D(LUT, MEM[t_x].offset + 2×1);
p2pdelay_c = tex1D(LUT, MEM[t_x].offset + 2×2);
p2pdelay_d = tex1D(LUT, MEM[t_x].offset + 2×3);
LAT = fmaxf(MEM[t_x].a + p2pdelay_a, MEM[t_x].b + p2pdelay_b);
LAT = fmaxf(LAT, MEM[t_x].c + p2pdelay_c);
DEL[t_x] = fmaxf(LAT, MEM[t_x].d + p2pdelay_d);
}
```

B. Statistical Static Timing Analysis (SSTA) at a Gate

SSTA at a gate is performed by an implementation that is similar to the above discussed STA implementation. The additional information required is the μ and σ of the n Gaussian distributions of the pin-to-output delay values for the n inputs to the gate. The μ and σ used for each Gaussian distribution are stored in LUTs (as opposed to storing a simple nominal delay value as in the case of STA).

The pseudo-random number generator used for generating samples from the Gaussian distribution is the Mersenne Twister pseudo-random number generation algorithm [6]. It has many important properties like a long period, efficient use of memory, good distribution properties and high performance.

As discussed in [27], the Mersenne Twister algorithm maps well onto the CUDA programming model. Further, a special offline library called *dcmt* (developed in [28]) is used for the dynamic creation of the Mersenne Twisters parameters. Using *dcmt* prevents the creation of correlated sequences by threads that are issued in parallel.

Uniformly distributed random number sequences, produced by the Mersenne Twister algorithm, are then transformed into the normal distribution $N(0,1)$ using the Box-Muller transformation [29]. This transformation is implemented as a separate kernel.

The pseudocode of the kernel for SSTA is given in Algorithm 2. The arguments to the routine *statistical_static_timing_kernel* are the pointers to the global memory for accessing the *threadData* (*MEM*) and the pointers to the global memory for storing the output delay value (*DEL*). The global memory is indexed at a location equal to the thread's unique *threadID* = t_x, and the *threadData* data is thus accessed. The μ and σ of the pin-to-output rising (falling) delay for an input x of an *inverting* gate accessed by indexing LUT$^{\mu}$ and LUT$^{\sigma}$ respectively, at the sum of the gate's base address and the even (odd) offset of the input x for falling (rising) transition.

The CUDA inbuilt single-dimension texture fetching function $tex1D(LUT, index)$ is invoked to fetch the μ and σ corresponding

to the pin-to-output delay values for every input. Using the μ and σ, along with the Mersenne Twister pseudo-random number generator and the Box-Muller transformation, a normally distributed sample of the pin-to-output delay for every input is generated. This generated value is added to the input arrival time of the corresponding input. Then, by performing $n-1$ MAX operations, the output arrival time is computed.

Algorithm 2 Pseudocode of the kernel for rising output SSTA for inverting gate

```
statistical_static_timing_kernel(threadData*MEM, float*DEL){
t_x = my_thread_id;
threadData Data = MEM[t_x];
p2pdelay_a^μ = tex1D(LUT^μ, MEM[t_x].offset + 2×0);
p2pdelay_a^σ = tex1D(LUT^σ, MEM[t_x].offset + 2×0);
p2pdelay_b^μ = tex1D(LUT^μ, MEM[t_x].offset + 2×1);
p2pdelay_b^σ = tex1D(LUT^σ, MEM[t_x].offset + 2×1);
p2pdelay_c^μ = tex1D(LUT^μ, MEM[t_x].offset + 2×2);
p2pdelay_c^σ = tex1D(LUT^σ, MEM[t_x].offset + 2×2);
p2pdelay_d^μ = tex1D(LUT^μ, MEM[t_x].offset + 2×3);
p2pdelay_d^σ = tex1D(LUT^σ, MEM[t_x].offset + 2×3);
p2p_a = p2pdelay_a^μ + k_a × p2pdelay_a^σ; // k_a, k_b, k_c, k_d
p2p_b = p2pdelay_b^μ + k_b × p2pdelay_b^σ; // are obtained by Mersenne
p2p_c = p2pdelay_c^μ + k_c × p2pdelay_c^σ; // Twister followed by
p2p_d = p2pdelay_d^μ + k_d × p2pdelay_d^σ; // Box-Muller transformations.
LAT = fmaxf(MEM[t_x].a + p2p_a, MEM[t_x].b + p2p_b);
LAT = fmaxf(LAT, MEM[t_x].c + p2p_c);
DEL[t_x] = fmaxf(LAT, MEM[t_x].d + p2p_d);
}
```

In our implementation of Monte Carlo based SSTA for a circuit, we first levelize the circuit. In other words, each gate of the netlist is assigned a level which is one more than the maximum level of its fanins. The primary inputs are assigned a level '0'. We then perform SSTA at all gates with level *i*, starting with *i*=1. Note that we do not store (on the GPU) the output arrival times for all the gates at any given time. We reuse the GPU's global memory for storing the arrival times of the current level's gates and their immediate fanins. We reclaim the memory used by all gates which are not inputs to any of the gates at the current or a higher level. By doing this we incur no loss of data since the entire approach is carried out in a single pass and we don't revisit any gate. Although our current implementation simultaneously simulates all gates with level *i*, the number of computations at each gate is large enough to keep the GPU's processors busy. Hence, we could alternatively simulate one gate at a time on the GPU. Therefore, our implementation poses no restrictions on the size of the circuit.

GPUs allow extreme speedups if the different threads being evaluated have no data dependencies. The programming model of a GPU is the Single Instruction Multiple Data (SIMD) model, under which all threads must compute identical instructions, but on different data. Also, GPUs have an extremely large memory bandwidth, allowing multiple memory lookups to be performed in parallel.

Monte Carlo based SSTA requires multiple sample points for a single gate being analyzed. By exploiting sample-parallelism, several sample points can be analyzed in parallel. Similarly, SSTA at each gate at a specific topological level in the circuit can be performed independently of SSTA at other gates. By exploiting data parallelism, many gates can be analyzed in parallel. This maximally exploits the SIMD semantics of the GPU platform.

V. Experimental Results and Comparisons

In order to perform *S* gate evaluations for SSTA, we need to invoke *S* *statistical_static_timing_kernel*s in parallel. The total DRAM on an NVIDIA GeForce GTX 8800 is 768 MB. This off-chip memory can be used as global, local and texture memory. Also the same memory is used to store CUDA programs, context data used by the GPU device drivers, drivers for the desktop display and NVIDIA control panels. With our current implementation, we can issue 16M threads in parallel. The wall clock time taken for 16M executions of *statistical_static_timing_kernel*s (by issuing 16M threads in parallel) is 0.115 seconds. A similar routine using the conventional implementation on a 3.6 GHz CPU with 3 GB RAM, running Linux, took 37.158 seconds for 16M calls. Thus asymptotically, the speedup of our implementation is $\sim$320$\times$. The allocation and loading of the texture memory is a one time cost of about 0.65 ms, which is easily amortized in our implementation. Note that the Mersenne Twister implementation on the GTX 8800, when compared to an implementation on the CPU (3.6 GHz CPU with 3 GB RAM), is by itself about two orders of magnitude faster. On the GTX 8800, the Mersenne Twister kernel generates random numbers at the rate of 2.33×10^9 numbers/second. A CPU implementation of the Mersenne Twister algorithm, on the other hand, generates random numbers at the rate of 2.24×10^7 numbers/second. The results obtained from the GPU implementation were verified against the CPU results.

We ran 60 large IWLS, ITC and ISCAS benchmark designs, to compute the per-circuit speed of our Monte Carlo based SSTA engine implemented on a GPU. These designs were first mapped in SIS [30] for delay optimality. The Monte Carlo analysis was performed with 64K samples. The results for 30 representative benchmark designs for our GPU based fault simulation tool are shown in Table I. Column 1 lists the name of the circuit. Columns 2, 3 and 4 list the number of primary inputs, primary outputs and gates in the circuit. Columns 5 and 7 list the GPU and CPU runtime, respectively. The time taken to transfer data between the CPU and GPU was accounted for in the GPU runtimes listed. In particular, the data transfered from the CPU to the GPU is the arrival times at each primary input, and the μ and σ information for all pin-to-output delays of all gates. The data returned by the GPU are the 64K delay values at each output of the design. The runtimes also include the time required for the Mersenne Twister algorithm and applying Box-Muller transformations. Column 8 reports the speedup obtained by using a single GPU card.

By using the NVIDIA SLI technology with four GPU chips on a single motherboard, we can effectively increase the available global memory by 4$\times$. Hence we can invoke $\sim$64M calls of the *statistical_static_timing_kernel* in parallel. This allows for a 4$\times$ speedup in the processing time. The transfer times, however, do not scale. Column 6 lists the runtimes obtained when using a quad GPU system and the corresponding speedups against the CPU implementation is reported in Column 9.

VI. Conclusions

In this paper, we have presented the implementation of Monte Carlo based SSTA on a Graphics Processing Unit. Monte Carlo based SSTA is computationally expensive, but crucial in design timing closure since it enables an accurate analysis of the delay variations. Our implementation computes multiple timing analysis evaluations for a single gate in parallel. We used a SIMD implementation of the Mersenne Twister pseudo-random number generator, followed by Box-Muller transformations, implemented on the GPU, for generating delay numbers in a normal distribution. The μ and σ of the pin-to-output delay numbers, for all inputs and for every gate, are obtained using a memory lookup, which exploits the large memory bandwidth of the GPU. Threads which execute in parallel do not have data or control dependencies on each other. All threads execute identical instructions, but on different data. This is in accordance to the SIMD programming semantics of the GPU. Our results, implemented on a

Circuit	# Inputs	# Outputs	# Gates	GPU runtimes (s)		CPU runtime (s)	Speedup	
				Single GPU	SLI Quad		Single GPU	SLI Quad
b14	276	299	9496	4.734	1.404	1303.63	275.394 ×	928.408 ×
b15_1	483	518	13781	6.952	2.121	1891.884	272.116 ×	892.174 ×
b17	1450	1511	41174	20.736	6.3	5652.45	272.589 ×	897.283 ×
b18	3305	3293	6599	6.326	4.013	905.924	143.197 ×	225.769 ×
b21	521	512	20977	10.311	2.956	2879.765	279.298 ×	974.323 ×
b22_1	734	725	25253	12.519	3.665	3466.783	276.913 ×	945.897 ×
s832	23	24	587	0.298	0.092	80.585	270.376 ×	873.741 ×
s838.1	66	33	562	0.295	0.098	77.153	261.341 ×	785.937 ×
s1238	32	32	857	0.432	0.132	117.651	272.248 ×	893.594 ×
s1196	32	32	762	0.388	0.121	104.609	269.796 ×	867.713 ×
s1423	91	79	949	0.521	0.189	130.281	249.858 ×	690.5 ×
s1494	14	25	1033	0.508	0.145	141.812	279.414 ×	975.729 ×
s1488	14	25	1016	0.5	0.143	139.479	279.187 ×	972.973 ×
s5378	199	213	2033	1.16	0.447	279.094	240.58 ×	623.999 ×
s92341	247	250	3642	1.949	0.672	499.981	256.57 ×	744.309 ×
s13207	700	790	5849	3.512	1.461	802.963	228.633 ×	549.517 ×
s15850	611	684	6421	3.675	1.424	881.488	239.855 ×	619.14 ×
s35932	1763	2048	19898	11.318	4.341	2731.638	241.349 ×	629.197 ×
s38584	1464	1730	21051	11.544	4.163	2889.924	250.335 ×	694.158 ×
s38417	1664	1742	18451	10.341	3.871	2532.991	244.958 ×	654.326 ×
C1355	41	32	715	0.366	0.115	98.157	268.363 ×	853.062 ×
C1908	33	25	902	0.446	0.13	123.828	277.46 ×	952.312 ×
C2670	233	140	1411	0.797	0.303	193.705	242.906 ×	639.891 ×
C3540	50	22	1755	0.842	0.227	240.93	286.1 ×	1062.439 ×
C432	36	7	317	0.155	0.044	43.518	280.605 ×	990.414 ×
C499	41	32	675	0.347	0.11	92.665	267 ×	839.445 ×
C5315	178	123	2867	1.461	0.456	393.588	269.323 ×	862.843 ×
C6288	32	32	2494	1.197	0.323	342.381	285.927 ×	1060.055 ×
C7552	207	108	3835	1.899	0.555	526.477	277.214 ×	949.424 ×
C880	60	26	486	0.253	0.082	66.719	263.923 ×	809.761 ×
Avg							258.994 ×	788.014 ×

TABLE I
MONTE CARLO BASED SSTA RESULTS

NVIDIA GeForce GTX 8800 GPU card, indicate that our approach can provide about 260× speedup when compared to a conventional CPU implementation. With the recently announced quad 8800 GPU cards, our projected speedup is ∼785×.

REFERENCES

[1] 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.

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

[3] 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.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Philips, "GPU Computing," in *Proceedings of the IEEE*, vol. 96(5), May 2008.

[5] 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.

[6] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[7] D. Heras, J. Cabaleiro, V. Perez, P. Costas, and F. Rivera, "Principal component analysis on vector computers," in *Proceedings of VECPAR*, pp. 416 – 428, 1996.

[8] J. Cabaleiro, J. Carazo, and E. Zapata, "Parallel algorithm for principal component analysis based on hotelling procedure," in *Proceedings of EUROMICRO Workshop On Parallel and Distributed Processing*, pp. 144 – 149, 1993.

[9] "SLI Technology." `http://www.slizone.com/page/slizone.html`.

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

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

[12] J. Benkoski and A. J. Strojwas, "A New Approach to Hierarchical and Statistical Timing Simulations," vol. 6, pp. 1039–1052, Nov. 1987.

[13] H.-F. Jyu and S. Malik, "Statistical delay modeling in logic design and synthesis.," in *DAC*, pp. 126–130, 1994.

[14] L. Zhang, W. Chen, Y. Hu, and C. C.-P. Chen, "Statistical timing analysis with extended pseudo-canonical timing model," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, (Washington, DC, USA), pp. 952–957, IEEE Computer Society, 2005.

[15] J. Le, X. Li, and L. T. Pileggi, "STAC: statistical timing analysis with correlation," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, (New York, NY, USA), pp. 343–348, ACM, 2004.

[16] I. Nitta, T. Shibuya, and K. Homma, "Statistical static timing analysis technology," *FUJITSU Sci. Tech J.*, vol. 43, pp. 516–523, Oct 2007.

[17] H. Chang and S. S. Sapatnekar, "Statistical timing analysis under spatial correlations," vol. 24, pp. 1467–1482, Sept. 2005.

[18] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan, "First-order incremental block-based statistical timing analysis," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, (New York, NY, USA), pp. 331–336, ACM, 2004.

[19] A. Agarwal, V. Zolotov, and D. T. Blaauw, "Statistical timing analysis using bounds and selective enumeration," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, pp. 1243–1260, Sept 2003.

[20] A. Agarwal, D. Blaauw, and V. Zolotov, "Statistical timing analysis for intra-die process variations with spatial correlations," in *ICCAD*, pp. 900–907, 2003.

[21] A. Agarwal, D. Blaauw, V. Zolotov, and S. Vrudhula, "Statistical timing analysis using bounds," in *DATE*, pp. 62–67, 2003.

[22] A. Devgan and C. V. Kashyap, "Block-based static timing analysis with uncertainty.," in *ICCAD*, pp. 607–614, IEEE Computer Society / ACM, 2003.

[23] J.-J. Liou, K.-T. Cheng, S. Kundu, and A. Krstic, "Fast statistical timing analysis by probabilistic event propagation.," in *DAC*, pp. 661–666, ACM, 2001.

[24] R. Garg, N. Jayakumar, and S. P. Khatri, "On the improvement of statistical timing analysis," *International Conference on Computer Design*, pp. 37–42, Oct 2006.

[25] P. McGeer, A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli, *Logic Synthesis and Optimization*, ch. Delay Models and Exact Timing Analysis, pp. 167–189. Kluwer Academic Publishers, 1993.

[26] 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.

[27] "Parallel Mersenne Twister." `http://developer.download.nvidia.com/~MersenneTwister`.

[28] M. Matsumoto and T. Nishimura, *Monte Carlo and Quasi-Monte Carlo Methods*, ch. Dynamic Creation of Pseudorandom Number Generators, pp. 56–69. Springer, 1998.

[29] "Box-Muller Transformation." `http://mathworld.wolfram.com/Box-MullerTransformation`.

[30] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.