Efficient Smart Monte Carlo based SSTA on Graphics Processing Units with Improved Resource Utilization

Vineeth Veetil, Yung-Hsu Chang, Dennis Sylvester, David Blaauw
University of Michigan, EECS Department, Ann Arbor, MI 48109
{tvvin,yunghsuc,dennis,david}@umich.edu

Abstract

To exploit the benefits of throughput-optimized processors such as GPUs, applications need to be redesigned to achieve performance and efficiency. In this work, we present techniques to speed up statistical timing analysis on throughput processors. We draw upon advancements in improving the efficiency of Monte Carlo based statistical static timing analysis (MC SSTA) using techniques to reduce the sample size or smart sampling techniques. An efficient smart sampling technique, Stratification + Hybrid Quasi Monte Carlo (SH-QMC), is implemented on a GPU based on NVIDIA CUDA architecture. We show that although this application is based on MC analysis with straightforward parallelism available, achieving performance and efficiency on the GPU requires *exposing more parallelism* and *finding locality in computations*. This is in contrast with random sampling based algorithms which are inefficient in terms of sample size but can keep resources utilized on a GPU. *We show that SH-QMC implemented on a Multi GPU is twice as fast as a single STA on a CPU for benchmark circuits considered*. In terms of an efficiency metric, which measures the ability to convert a reduction in sample size to a corresponding reduction in runtime w.r.t a random sampling approach, we achieve 73.9% efficiency with the proposed approaches compared to 4.3% for an implementation involving performing computations on smart samples in parallel. Another contribution of the paper is a critical graph analysis technique to improve the efficiency of Monte Carlo based SSTA, leading to 2-9X further speedup.

Categories and Subject Descriptors

J.6 **[Computer Applications]** Computer-Aided Design - *computer-aided design (CAD).*

General Terms

Algorithms, Verification

Keywords

Monte Carlo, Statistical timing, Graphics Processing Units

1. Introduction

Recent years have seen the rapid scaling of throughput-optimized processors, such as Graphics Processing Units (GPUs). Modern GPUs deliver over 1 TeraFlops of computational power with more than 100 GB/second of memory bandwidth while conventional processors face difficulties with frequency scaling and are increasingly incorporating multiple cores on a chip to keep up with Moore's law. Throughput processors recognize two crucial aspects of machine organization which are *parallel execution* and *hierarchical memory organization*. To increase performance in throughput processors, applications will need to expose parallelism while finding locality in their computations to overcome restrictions arising from communication bandwidth bottlenecks. In this work we show the importance of these two aspects for improving performance and efficiency in the context of statistical timing analysis by drawing inferences from the implementation on a specific GPU architecture.

Process variations have taken on increasing importance in nanometer-scale CMOS. Rather than using simple corner models that capture worst-case behavior at the device level (and lead to large guard bands), modern CAD tools have moved towards a more probabilistic view of circuit timing behavior. Two primary approaches have been proposed to replace corner models by incorporating process parameter uncertainty in timing analysis. The first is statistical static timing analysis (SSTA), which models gate delay as a function of process parameters and propagates these distribution functions to compute the circuit delay distribution [1-5]. We refer to these approaches as traditional SSTA. The second approach is Monte Carlo based SSTA (MC SSTA), which involves analyzing samples of the process variation space to obtain statistical distributions of circuit timing behavior. MC techniques are "embarrassingly parallel" and have inherent advantages over traditional SSTA in exposing parallelism for performance improvements in throughput processors. Hence, in this work we focus on MC based algorithms. However, the difficulty is that the standard MC approach of random selection of samples in the process variation space requires too many samples for sufficient accuracy, resulting in high runtimes.

An effective solution to address the high runtime cost of MC SSTA is to use techniques to reduce the sample size. Sample size reduction is achieved using a combination of standard techniques in statistics called variance reduction techniques [6] and the use of circuit specific information. These techniques have been studied in recent years for parametric yield analysis, statistical timing, crosstalk and leakage analysis [7-13]. In [7], the authors propose mixture importance sampling for statistical SRAM design and analysis. However, while several approaches are reviewed, no results are presented. In [8], the authors propose to use Quasi Monte Carlo Analysis for yield estimation. This approach cannot be directly extended to systems with large number of dimensions (variables) which is often the case with process variation. In [9], the authors attempt to address this issue by reduction of problem dimension using a Karhunen-Loeve expansion (KLE) model of spatial correlation. The proposed problem formulation considers a grid-less spatial correlation model with assumptions of continuity, positive definiteness and bounded variance. The results report significant speed-ups in terms of sample size reduction. One drawback of the work is that it is not clear if existing design flows that consider a grid-based spatial correlation model can use the properties of stochastic processes with a covariance kernel [10], while also achieving a significantly reduced set of variables which can be handled by QMC. In [11] the authors propose to combine QMC and LHS to address the issue of QMC's inability to handle high dimensionality. [11] also proposes to use stratified sampling for additional performance improvement. In [12] the authors present a robust theoretical framework for the ability of QMC and LHS-based methods to speed up statistical timing analysis. Further, they propose techniques to generate QMC samples tuned for optimal performance in SSTA. An efficient method for statistical analysis of full cheap leakage using an application of QMC is proposed in [13].

In this work we draw upon these advancements in MC SSTA. However, with a reduced sample size the objective of exposing parallelism for performance on throughput processors poses new challenges. In the case of GPUs, the level of parallelism required in the application is much higher than the units of parallelism to hide bottlenecks including memory access time. In a random sampling based approach the sample size is in the range of tens of thousands, about two orders of magnitude higher than the units of parallelism available in the GPU hardware. Therefore, this enables high utilization of resources on the GPU simply by performing computations on the samples in parallel. An implementation of random sampling MC SSTA on GPUs was explored in [14], where it is illustrated that such an implementation is sufficient for adequate resource utilization. However smart sampling algorithms can achieve accurate results with a sample size that is typically in the range of 100-200 [11], which is the same order of magnitude as the hardware parallelism available on a

DAC'10, June 13-18, 2010, Anaheim, California, USA

GPU. This reduction in sample size cannot be translated to a corresponding reduction in runtime for a GPU with such a straightforward implementation. In addition to enabling fast statistical timing analysis of chips with millions of gates, this additional improvement opens up possibilities for using SSTA in a design optimization loop.

The main contribution of this work is to illustrate performance and efficiency improvements in the context of smart sampling based MC SSTA by recognizing the aspects of *parallel execution* and *hierarchical memory organization* in throughput processors. This translates to the following key ideas leading to the implementation.

- *Expose more parallelism.* In the context of smart sampling based MC SSTA, gates in a circuit that do not depend on each other for input data given the computations already performed can be analyzed in parallel, leading to data parallelism or *gate parallelism*. We propose a smart scheduling algorithm for allocation of gates to parallel threads to make use of this parallelism. We show that exposing gate parallelism is crucial to achieving *parallel execution* on GPUs in the context of smart sampling based MC SSTA.
- *Find locality in computations.* Finding locality in computations is critical to avoid restrictions arising from communication bandwidth bottlenecks. We lump together computations that are manageable within the fast local memory to avoid bottlenecks from accessing slow global memory.

We attempt to illustrate these general principles for throughput processors through an implementation of the smart sampling based MC SSTA technique called SH-QMC (Stratified Hybrid + Quasi Monte Carlo), which was proposed in [11], on Nvidia's CUDA-based GPU platform. Though the implementation itself is specific to the platform, this serves to illustrate the effectiveness of these concepts. The algorithm in [11] achieves a significant reduction in the number of samples needed to achieve accurate timing results while also considering a detailed process variation model incorporating within die variation. We compare the proposed implementation of SH-QMC with a straightforward sample level parallelism approach. Average speedups over random sampling MC SSTA improve from 11.2X to 192.5X for the two implementations of SH-QMC on benchmark circuits ranging from 15,000 to 60,000 gates. When the GPU system is compared with a CPU an average speedup of 153X is achieved. The average runtimes normalized to a single STA on a CPU is 0.46, pointing to the result that *smart sampling based MC SSTA on a GPU is faster than a single STA on a CPU*.

A second contribution of this work is a critical graph analysis technique to speed up MC SSTA. Nominal STA is used to identify gates with very low probabilities of falling on critical paths under process variation, and are pruned from further consideration, without impacting the accuracy of statistical timing analysis. This enables fast evaluation of circuit samples leading to a 6.8X runtime reduction for the benchmark circuits considered.

The paper is organized as follows. Section 2 describes the important relevant hardware and software features in GPUs. Section 3 briefly discusses the SH-QMC algorithm for smart sampling based MC SSTA. Section 4 describes the implementation of MC SSTA on GPUs and proposes techniques to achieve resource utilization when mapping SH-QMC onto GPUs. Section 5 discusses the critical graph analysis technique. Section 6 presents results and the paper concludes in Section 7.

2. CUDA Platform

NVIDIA CUDA (Compute Unified Device Architecture) is a general purpose parallel computing architecture that is easily programmable and exhibits good performance in scientific applications [15]. The CUDA architecture is built around multiprocessors, each consisting of several scalar processor (SP) cores. From a software perspective, threads are the basic unit for parallel computation and the code they execute is called the kernel. A thread block, also referred to as a block, contains a batch of threads. Threads in the same block can efficiently share information through shared memory and run on the same multiprocessor. Within a block, 32 consecutive threads are grouped into a *warp*. All threads in a warp follow the exact same sequence of instructions. CUDA threads have accesses to multiple memory spaces during their execution. Global memory has the largest size but also exhibits long access times compared to other on-chip memory. The CUDA warp consists of two half-warps of 16 threads each. If all 16 threads of a half-warp access consecutive words from global memory, the overhead is significantly lower than when non consecutive words are accessed [16]. Other types of fast on-chip memory in the targeted CUDA architecture include *register memory* and *shared memory*. Shared memory can be shared within a block and is significantly faster than global memory.

3. Smart Sampling based MC SSTA: SH-QMC

We propose to implement a smart sampling based MC SSTA approach called SH-QMC (Stratification + Hybrid Quasi Monte Carlo) [11] on GPUs. This algorithm significantly speeds MC based SSTA using sample size reduction. In this technique, circuit timing criticality information is used for intelligent selection of samples. It is shown that 100-200 samples are sufficient for accurate statistical timing analysis. The process variation model is based on [2], which considers intra-die spatially correlated variation by partitioning the die into $n * n$ grids and assuming identical parameter variations within a grid [11]. SH-QMC uses a combination of standard techniques and and circuit timing criticality information to reduce sample size for MC based analysis (variance reduction techniques). The variance reduction techniques employed are Quasi Monte Carlo (QMC), stratified sampling, and Latin Hypercube Sampling (LHS) [6]. These techniques are employed on variables based on their convergence properties and the ability to handle multiple variables. A detailed analysis of the algorithm is presented by the authors in [11].

4. Monte Carlo based Statistical Static Timing Analysis on GPUs

This section describes techniques for efficient implementation of MC SSTA on GPUs. In a random sampling based MC approach, samples of the chip are generated using process variation information. These samples have no data dependence on each other and therefore are directly amenable to parallelism. This is referred to as sample parallelism. Each thread is dedicated to the computation of one sample (representing one virtually fabricated die) of the circuit. Gates are visited in the topological order in the circuit by a thread and delay computations are performed. For the computation of process variation samples we use a Mersenne Twister based random number generator [14]. A detailed discussion is omitted for brevity.

4.1 Enhanced resource utilization for implementation of SH-QMC on GPU

Sample parallelism is sufficient to keep resources utilized on GPUs when employing random sampling[14]. However the sample size in SH-QMC is typically only 100-200, which is comparable to the number of streaming processors available in GPUs. A straightforward implementation in the spirit of the approach in [14] leads to under-utilization of resources. In this section, we describe techniques which adhere to the two key ideas for performance and efficiency in throughtput processors introduced in Section 1. With this improved resource utilization, performing SSTA repeatedly in a design optimization loop with hundreds of thousands of iterations becomes a possibility for moderately sized circuits.

a. *Parallelism - exposing gate parallelism*

For gates with no data dependence gate delay calculations can be performed in parallel. To leverage this parallelism we propose static scheduling of gates in the circuit using a scheduling algorithm prior to performing statistical timing analysis. A schedule table assigns gates to levels such that gates at a given level are assigned to parallel threads only after computations on previous levels have been completed (Figure 1). All threads working on gates in the same sample are grouped together within a CUDA block. A block consists of threads that can efficiently communicate with each other using shared memory. Threads working on

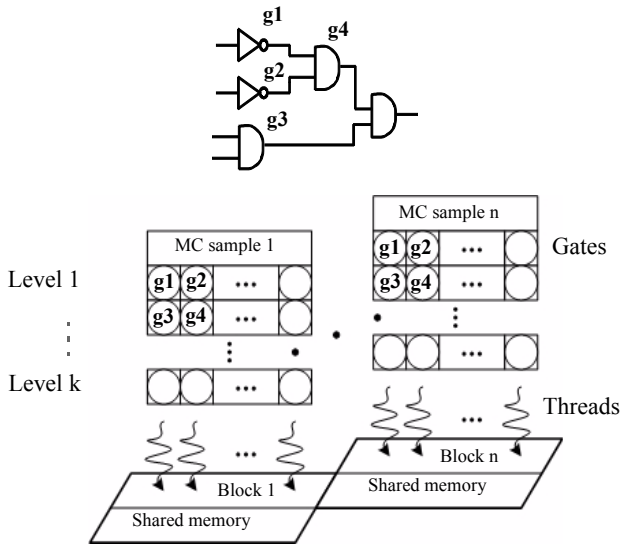

Figure 1. Gate scheduling. Gates in a sample with no dependence are computed in parallel. In graph shown, gates g1,g2,g3 have no dependence and can be assigned to the same level. However, g3 is a 2-input gate which if assigned to the same level as g1 and g2, increases the computational steps in the level. Therefore, g3 is assigned to the next level alongwith gate g4.

different samples of the circuit are grouped into different blocks, allowing flexible use of memory and resources. For assigning gates to the schedule table, we propose two algorithms - *Algorithm 1* and *Algorithm 2*. The pseudocode for both algorithms is presented in Figure 2. In Algorithm 1, the gates are sorted in topological order. A gate is ready to be scheduled when all its fanin gates have been scheduled in previous levels of the schedule table. All ready gates are assigned to levels such that the number of gates per level does not exceed the key parameter *MaxPerLevel*. Algorithm 2 performs smart scheduling of gates to reduce the total number of computation steps. In this case gates that are ready to be scheduled are preferentially grouped together in a level based on three criteria:

1. *Fanout count*: Gates with large fanouts are assigned a higher preference for scheduling. This allows more freedom for gate choices in subsequent levels where more gates are likely to have their fanin gates already scheduled.
2. *Global memory access*: Gates with common fanin gates are grouped together to avoid redundant fetching of delay data from slow global memory.
3. *Pin count*: Gates with the same or similar number of inputs are assigned to the same level. Gates with higher input pin counts involve more delay computation steps. Since all threads within a CUDA warp are forced to perform the same number of computations, all threads in a warp complete at the same time as the thread for the gate with the largest pincount. Therefore, grouping together gates with lower input counts in the same level leads to speed up.

Given the list of ready gates, a gate *g* is selected for scheduling to the next level such that a linear sum of costs based on the above three criteria is maximized. *ExtraFanin* is the number of fanin gates of *g* that are not already fanins for other gates in the current level. The algorithm tries to select gates with low values of *ExtraFanin* to minimize the total memory accesses from global memory required to perform computations on gates in the current level. *FaninDiff* is positive if the new gate selected has more pins than the gates in the current level. A maximum of *MaxPerlLevel* gates are selected per level and the list of ready gates is updated before gates are allocated to the next level in the schedule table.

b. *Localizing computations in shared memory*

Since global memory has much higher latency and lower bandwidth than on-chip memory, global memory accesses should be minimized. Shared memory is a fast on-chip memory resource and therefore ideal for

Scheduling Algorithm 1

```
Topologically sort gates in the circuit
queue ReadyGate
Level = 0
while all gates are scheduled
{
   for all gates g not scheduled, if g is ready
         Add g to queue ReadyGate
   while ( size of Schedule[Level] < MaxPerLevel .AND. ReadyGate is not empty)
           Add to list Schedule[Level] ( ReadyGate.pop() )
    Level ++
}
return schedule
```

Scheduling Algorithm 2

```
Topologically sort gates in the circuit
queue ReadyGate
Level = 0
while all gates are scheduled {
   for all gates g not scheduled, if g is ready
         Add g to queue ReadyGate
   faninlist = {}
   faninM = 1
   while size of Schedule[Level] < MaxPerLevel .AND. ReadyGate is not empty {
         Find g in ReadyGate to maximize Weight(g,faninlist,faninM)
         Add g to Schedule[Level]
         Remove g from ReadyGate, Add fanin gates of g to faninlist
         faninM = max( fanincount(g), faninM )
    }
    Level ++
}
return Schedule
Weight(g,faninlist,faninM)
Fanout = fanoutcount(g)
ExtraFanin = number of fanin gates of g not in faninlist
FaninDiff   = max( fanincount(g) - faninM, 0 )
Weight =  Fanout - CoeffMem*ExtraFanin - CoeffDelayStep*FaninDiff
```

Figure 2. *Algorithm 1* and *Algorithm 2* for gate scheduling.

storing all intermediate information. However, the shared memory size for each multiprocessor is small (16KB in typical CUDA architectures), which is small compared to global memory. The maximum size of shared memory allocated to a block is no larger than 16KB, which is not sufficient to store all intermediate information in practical sized circuits. Hence global memory is used to store the delay information. To minimize access of this data from global memory, we propose a technique to localize computations such that they are manageable within the limits of shared memory. As mentioned in Section 4.2a, the circuit is scheduled into multiple levels in a schedule table to expose gate parallelism. We group *N* levels into one entity or subcircuit, where the parameter *N* is a function of the shared memory size. Before gates in the first level of the subcircuit are scheduled, input data for this subcircuit is loaded into the shared memory. When gates in subsequent levels require input data already accessed or computed by gates in the previous levels within the same subcircuit, this is accessed from the shared memory. This minimizes access of data from global memory. In addition, while computations on gates in the current subcircuit are being performed, the algorithm fetches data required for the next subcircuit (defined by the next *N* levels) if not an output of the current computation. This allows overlapping of memory access steps and arithmetic computations so that global memory latency is effectively hidden.

Figure 3 summarizes these techniques. Computations for one circuit sample are illustrated in the figure. The schedule table resides in global memory. Gates in the current level of the schedule table are assigned to different threads. Process variation samples for the gates are computed in parallel and gate delay computations are performed. The input delay

information required for the computations is accessed from *current_subckt* in shared memory. Here *N* levels in the schedule table are grouped together. The *current_subckt* in shared memory consists of all input information required by gates in *N* levels including the current level. This data was previously loaded into the shared memory from global memory. While arithmetic computations are performed one level at a time on the set of *N* levels, delay information for the next set of levels *i+N* to *i+2*N-1* is loaded from global memory to shared memory as illustrated. The table *next_subckt* stores this data in shared memory. This allows better hiding of latency for the global memory access within each thread. Also, access of data for *N* levels at a time avoids repeated accesses from global memory. The output information from the current delay computations are stored in the delay table in global memory and also updated appropriately in the table *next_subckt*.

5. Critical Graph Analysis for MC SSTA

In this section we propose a technique to improve the performance of SSTA through critical graph analysis. The basic idea is to identify critical paths in the graph by performing heuristics. In other words, gates which are expected to have a negligible effect in determining the worst case arrival time of the circuit can be pruned or avoided from consideration in subsequent analyses, leading to speedups in the overall statistical analysis. In the context of variability, criticality is statistical. The challenge here is to assign probability values to gates/paths in the circuit based on a measure of criticality. In [17], the authors propose an algorithm to compute criticality probability of gates in the circuit. This algorithm computes criticality accurately, however it can potentially add a significant runtime overhead to the SSTA. It may be noted that the proposed critical graph analysis technique only requires that all sufficiently critical gates be selected for accuracy in subsequent SSTA. The exact values for criticality probability are not required in the further analysis. Therefore, we propose a simpler technique for critical graph identification. We propose that slack information obtained from STA performed at the nominal process corner be used to identify the critical graph. The timing overhead for this technique is significantly lower.

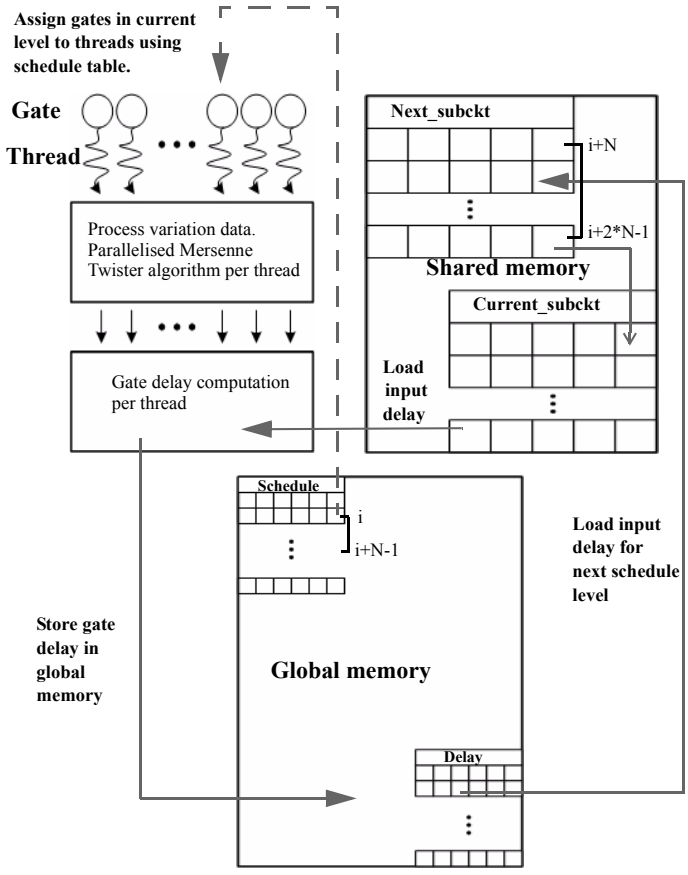

Figure 3. Summary of proposed approaches to improve resource utilization. Concurrent computation on gates in the same level and use of shared memory are illustrated.

- ***Nominal STA based Critical Graph Identification***

This technique uses information obtained from timing analysis of the circuit at the nominal process corner. The example in Figure 4 illustrates the technique. Nominal STA is performed and slack information is obtained at all gates in the circuit. Gates with significant slack, in this case higher than a threshold value of 0.3 are excluded from consideration when applying MC based SSTA. The reduced graph size will allow the runtime-dominant MC STA runs to be reduced roughly linearly with circuit size. The threshold slack is defined as *s%* of the worst arrival time at the nominal sample, where *s* is the pruning parameter. For instance, s is 30% in the above example if circuit delay is 1 unit.

6. Results

We implement the proposed approach on an Nvidia Tesla S1070 GPU with a 3.16 GHz Intel Xeon-based Linux machine serving as the host. The GPU system has 4 GPU cards totalling 960 streaming processor cores [18]. Results in this section are based on a 65nm commercial technology library. In our implementation we only consider channel length variation as a source of process variation for simplicity, however other sources can be readily implemented. The inter-die, spatially correlated intra-die and uncorrelated random components of channel length variation are considered. The overall standard deviation is 10% of nominal channel length. Simulations are performed on four large circuits, Viterbi Decoder 1 (VD1), Viterbi Decoder 2 (VD2), USB 2.0 Core (USB), and an Ethernet MAC Core (ETHER), with gate counts varying from approximately 15,000 to 60,000. We perform synthesis and APR on all the circuits using commercial tools.

Table 1 compares the implementation of a random sampling based MC SSTA approach with the SH-QMC approach, both on the Tesla S1070 GPU system. The results indicate that a straightforward implementation of SH-QMC exploiting sample level parallelism does not lead to speedups corresponding to the reduced sample size w.r.t a random sampling based approach, whereas much higher speedups are obtained using the proposed techniques to improve resource utilization for smart sampling techniques. The sample size used in the random sampling based approach is 50,000. The number of samples used in the SH-QMC approach is 192 (the exact number is related to the granularity of sample size in the SH-QMC approach based on [11]). The SH-QMC approach is implemented in three variants:

1) SP: Only sample level parallelism is considered. This is based on the implementation in [14]. An active thread is dedicated to computations on a single sample;

2) SGP: Multiple threads perform computations on a sample by exploiting gate parallelism;

3) SGP+S.Mem. In this case shared memory is utilized and prefetching of data is performed.

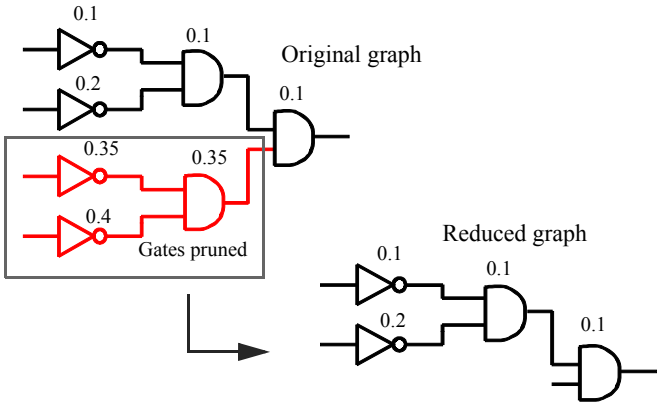

Figure 4. Illustration of graph reduction. Slacks for nodes are indicated next to corresponding gates. Gates with slacks higher than a threshold of 0.3 at output node are removed to obtain the reduced graph in the example.

Table 1. Comparison of runtime for SH-QMC (192 samples) vs random sampling based MC SSTA on GPU. SH-QMC is implemented with (a) sample level parallelism or SP (b) sample + gate parallelism or SGP. (c) SGP + efficient shared memory usage or SGP+S.Mem.

Circuit	# of Gates	RS 50k (ms)		SH-QMC on 1 GPU (ms)/Speedup w.r.t RS on 1 GPU			SH-QMC on 4 GPU (ms)/Speedup w.r.t RS on 4 GPU		
		1 GPU	4 GPU	SP	SGP	SGP+S.Mem	SP	SGP	SGP+S.Mem
VD1	**14503**	**4630**	**1620**	155/30X	32/147X	**28/164X**	148/11X	15/109X	**13/123X**
VD2	**34082**	**10870**	**3810**	366/30X	70/155X	**64/170X**	349/11X	33/116X	**30/128X**
USB	**32898**	**11360**	**4050**	364/31X	71/160X	**65/175X**	344/12X	17/232X	**16/256X**
Ether	**57327**	**19370**	**6810**	634/31X	119/163X	**106/182X**	600/11X	29/233X	**26/263X**

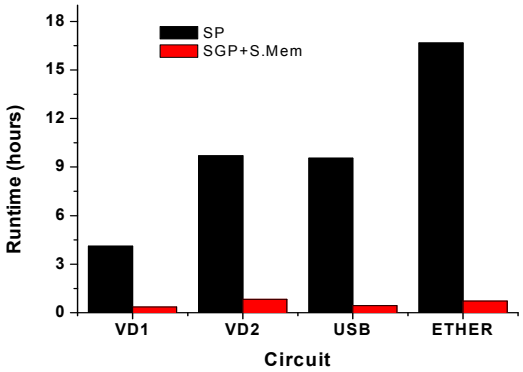

Figure 5. Comparison of runtime for SH-QMC performed with successive gate sizing (100,000 sizing steps) on 4 GPU cards for Ether circuit. Runtime for an implementation utilizing sample level parallelism (SP) is compared with the proposed approach (SGP+S.Mem).

We define the *efficiency* of the SH-QMC implementation as the runtime per sample of the random sampling approach divided by the runtime per sample for the SH-QMC approach. If the reduction in sample size for SH-QMC w.r.t random sampling could be translated into a corresponding reduction in runtime by the same factor then the efficiency is 100% according to the definition. In the GPU implementation we show results using both a single GPU card and four GPU cards in the S1070 GPU system. For the single GPU implementation, the speedup of the implementation compared to a random sampling approach increases from 30.5X for sample parallelism (SP) to 172.7X for SGP+S.Mem. This demonstrates that exposing gate parallelism and exploiting shared memory greatly increases resource utilization in the GPU for smart sampling based MC SSTA. The efficiency metric for the implementation increases from 11.7% for SP to 66.3% for SGP+S.Mem. When all cards in the multi-GPU system are used, we achieve 192.5X speedup on average for SGP+S.Mem compared to 11.2X for SP. In this case the runtime improvement is more pronounced compared to the case of a single GPU, since more resources are available per sample, leading to even lower resource utilization without the proposed techniques. The efficiency metric increases from 4.3% for SP to 73.9% for SGP+S.Mem for SH-QMC. Figure 6 illustrates the trend in performance improvement versus sample size for SP and SGP+S.Mem. As the sample size decreases the performance improvement with resource utilization increases significantly, underlining the synergy of these techniques with smart sampling based MC techniques.

Table 2 compares the efficiency of SH-QMC with 192 samples on a CPU versus a GPU system. The results are also shown for a single STA run on the CPU. On average a 48X speedup is achieved for a single GPU card over a quad core CPU. The average speedup is 153X when the multi-GPU system is compared with the quad-core CPU. The runtimes normalized to that of a single STA on a CPU are 1.26X and 0.46X, respectively, for a single GPU and multi-GPU. Thus MC SSTA runtime on a GPU is comparable to that of a single STA run on a CPU.

As shown in Table 1, the runtime for SH-QMC on the Ether circuit (57K gates) is only 600ms on a GPU even with a simple implementation using sample parallelism. Extrapolating from this data, this means that we can perform statistical analysis on large designs with millions of gates with low runtime. In addition, for circuits of similar sizes, the additional

Table 2. Comparison of runtime for SH-QMC (192 samples) on GPU vs CPU. and single STA on CPU. The CPU is a 3.16GHz Intel Xeon processor.

Circuit	Single STA on CPU	SH-QMC on quad core CPU	SH-QMC on GPU 1 card/Speed up w.r.t. quad core	SH-QMC on GPU 4 cards/ Speed up w.r.t quad core CPU	Runtime of SH-QMC on Tesla GPU norm. to CPU STA	
					1 card	4 cards
VD1	**20ms**	**1.1s**	**28ms/39X**	**13ms/85X**	**1.4**	**0.65**
VD2	**50ms**	**2.9s**	**64ms/45X**	**30ms/97X**	**1.28**	**0.6**
USB	**50ms**	**3.2s**	**65ms/49X**	**16ms/200X**	**1.3**	**0.32**
Ether	**100ms**	**6.0s**	**106ms/57X**	**26ms/231X**	**1.06**	**0.26**

20X improvement with the proposed approaches opens up possibilities for the use of SSTA in a design optimization setting where the anlaysis can be performed repeatedly in a loop for better quality of results. To illustrate the basic idea we demonstrate a simple experiment where SH-QMC is performed in tandem with gate sizing. Here, we select gates randomly for sizing, and perform SH-QMC after every sizing step. This is repeated for 100,000 sizing steps and the runtime is reported in Figure 5. The proposed approach for implementation of SH-QMC (SGP+S.Mem) is compared with the sample level parallelism based implementation (SP) for the benchmark circuits studied. For the largest circuit with 57K gates, the runtime is reduced from 16.7 hours to 42 minutes with the proposed approach. In a similar spirit, we compare the runtime of SH-QMC on both CPU and GPU with STA on a CPU, in Figure 7. The analyses are performed in a loop involving 100,000 sizing iterations for the Ether circuit (57K gates). The runtime for the cases of SH-QMC on a CPU and STA on a CPU are 166 hours and 2.7 hours respectively, compared to 42 minutes for the proposed SH-QMC implementation on GPU.

Figure 8 illustrates the runtime improvement versus the degree of gate parallelism when using scheduling algorithm 2 from Figure 2. The improvement in runtime saturates around a gate parallelism of about 200. The discontinuity in the graph at a gate parallelism of 64 is because an additional warp is required in the block to accommodate a new thread in this case (64 is a multiple of 32, the warp size in CUDA). Beyond this

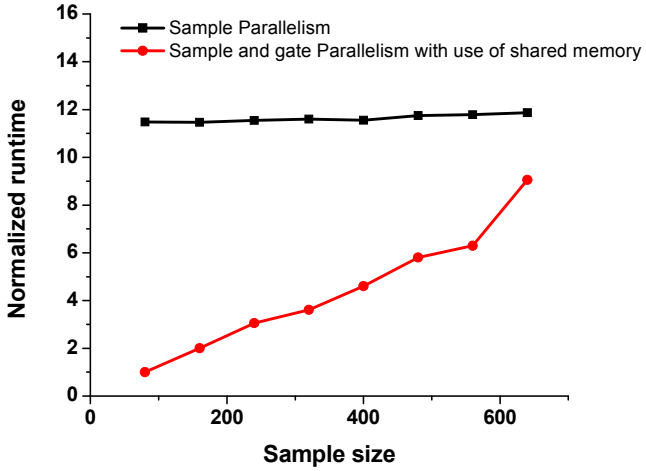

Figure 6. Improvement in runtime due to the techniques proposed for improved resource utilization.

Table 3. Quality of results for the critical graph analysis technique.

Circuit	% gates pruned	% of samples with zero error	Error in 99th percentile AT
VD1	65.4	99.74	0.018
VD2	43.2	99.60	0.015
USB	81.7	98.60	0.080
Ether	89.3	98.82	0.045

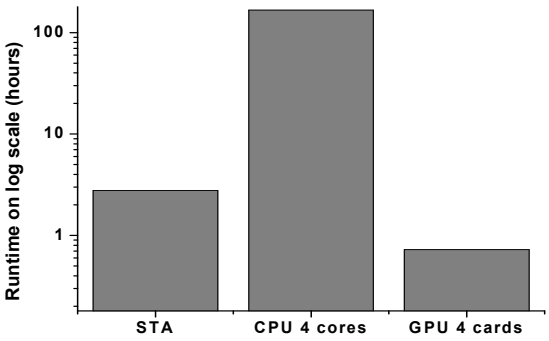

Figure 7. SH-QMC with 192 samples on GPU is compared with SH-QMC on CPU and STA on CPU, on a logarithmic scale, for Ethernet circuit with 57327 gates. SH-QMC on multi-GPU is faster than STA on CPU.

point, each block requires more registers (for the new warp) and the number of registers required per multiprocessor exceeds the capacity. Therefore, the number of blocks that can be active per multiprocessor is reduced. This leads to the runtime overhead.

We evaluate the accuracy of the critical graph identification approach in Table 3. The second column shows the percentage of gates pruned from consideration after critical graph analysis. We perform experiments on 80,000 samples where the possible error arising from critical graph analysis is computed at each sample. The third column shows the percentage of samples which incur absolutely no error. The error in computation of the 99th percentile of the circuit delay distribution (using 80,000 samples) is shown in the fourth column. We see that across all the benchmarks studied, more than 98.6% of samples incur absolutely no error. Also, the error in computation of the 99th percentile of worst arrival time is negligible. Table 4 illustrates the results from graph reduction for the benchmarks studied. The point on the sizing curve such that the hardware intensity (defined as the magnitude of the ratio of percentage change in power to percentage change in timing constraint) is 1 is selected for analysis in the table. On average a 6.8X speedup is achieved through the new pruning approach on the GPU implementation of SH-QMC, which is orthogonal to the speedups described above.

7. Conclusions

We present an implementation of smart sampling based MC SSTA on a GPU system. We show that a straightforward implementation of smart sampling that exposes only sample parallelism under-utilizes resources in the GPU, in contrast to random sampling based MC SSTA approaches where this type of parallelism is sufficient. We propose several techniques to achieve high resource utilization for the case of smart sampling based MC SSTA, particularly gate parallelism and enhanced use of shared memory. While sample parallelism leads to only 11.2X speedups using 192 samples compared to random sampling with 50,000 samples, our techniques lead to 192.5X speedups for the same comparison. In terms of an efficiency metric, the proposed techniques achieve an efficiency of 73.9% for smart sampling MC SSTA compared to a modest 4.3% for sample parallelism. Most significantly, MC SSTA runtime on a multi-GPU is shown to be over twice as fast as a single STA run on a CPU. This work also proposes a critical graph analysis technique to further speedup MC SSTA, achieving a 2-9X speedup on several benchmarks.

Table 4. Runtime improvement from graph reduction combined with the proposed technique.

Circuit	# of Gates	% gates pruned	S.G.P+ S.Mem (ms)	S.G.P.+ S.Mem+ G.Red (ms)	Speedup due to graph reduction
VD1	14503	65.4	13.1	4.5	2.9
VD2	34082	43.2	29.7	16.9	1.8
USB	32898	81.7	15.8	2.9	5.5
Ether	57327	89.3	25.9	2.7	9.4

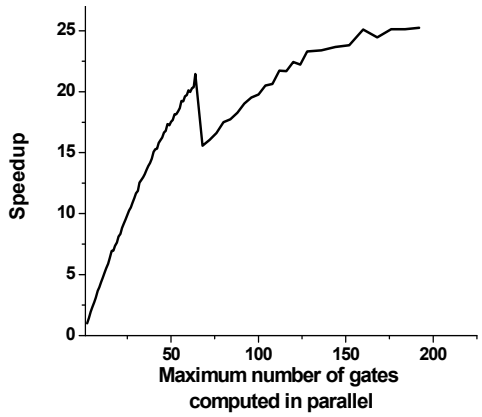

Figure 8. Speedup of SH-QMC algorithm implemented on multi-GPU for a USB circuit (14,503 gates) with smart scheduling algorithm Algorithm 2. X axis indicates the maximum number of gates computed in parallel. A discontinuity in resource requirements above a parallelism of 64 leads to the discontinuity in the graph.

Acknowledgements

The authors gratefully acknowledge Nvidia for enabling this work through their donation of the Nvidia Tesla S1070 GPU system.

References

[1] C. Visweswariah, et al., "First-Order Incremental Block-Based Statistical Timing Analysis," *Proc. Design Automation Conference*, pp 331-336, 2004.

[2] H. Chang, and S.S.Sapatnekar, "Statistical Timing Analysis Considering Spatial Correlations using a Single Pert-Like Traversal," *Proc. International Conference on Computer-Aided Design*, pp. 621-625, 2003.

[3] K. Chopra, et al., "Parametric yield maximization using gate sizing based on efficient statistical power and delay gradient computation", *Proc. International Conference on Computer Aided Design*, pp 1023-1028, 2005.

[4] F. N. Najm,N. Menezes, "Statistical timing analysis based on a timing yield model," *Proc. Design Automation Conference*, pp. 460-465, 2004.

[5] R. Gandikota, D. Blaauw, D. Sylvester, "Modeling crosstalk in statistical static timing analysis", *Proc. Design Automation Conference,* pp. 974-979, 2008.

[6] R.Y. Rubinstein, Simulation and the Monte Carlo Method, John Wiley & Sons, Inc., 1981.

[7] R. Kanj, R. Joshi, and S. Nassif, "Mixture Importance Sampling and Its Application to the Analysis of SRAM Designs in the Presence of Rare Failure Events," *Proc. Design Automation Conference*, pp. 69-72, 2006.

[8] A. Singhee and R.A. Rutenbar, "From Finance to Flip Flops: A Study of Fast Quasi-Monte Carlo Methods from Computational Finance Applied to Statistical Circuit Analysis", *Proc. ISQED*, pp. 685-692, 2007.

[9] A.Singhee, S.Singhal and R.A.Rutenbar, "Practical, fast Monte Carlo statistical static timing analysis: why and how," *Proc. International Conference on Computer-Aided Design*, pp. 190-195, 2008.

[10] A.Singhee, S.Singhal and R.A.Rutenbar, "Exploiting Correlation Kernels for Efficient Handling of Intra-Die Spatial Correlation, with Application to Statistical Timing," *Proc. Design, Automation and Test in Europe*, pp. 856-861, 2008.

[11] V. Veetil, D. Sylvester and D.Blaauw, "Efficient Monte Carlo based Incremental Statistical Timing Analysis," *Proc. Design Automation Conference*, pp. 676-681, 2008.

[12] J.Jaffari, and M.Anis, "On efficient Monte Carlo-based statistical static timing analysis of digital circuits," *Proc. International Conference on Computer-Aided Design*, pp. 196-203, 2008.

[13] V.Veetil, D.Sylvester, D.Blaauw, S.Shah, and S.Rochel, "Efficient Smart Sampling-Based Full-Chip Leakage Analysis for Intra-Die Variation Considering State Dependence," *Design Automation Conference*, 2009.

[14] K. Gulati, S.P. Khatri, "Accelerating Statistical Timing Analysis Using Graphics Processing Units", *Proc. ASP-DAC*, pp. 260-265, 2009.

[15] J.D.Owens, et al., "GPU Computing" *Proceedings of the IEEE*, vol. 96(5), pp. 879-899, 2008.

[16] http://www.nvidia.com/object/cuda_home.html#

[17] J.Xiong et al, " Criticality computation in parameterized statistical timing", *Proc. Design Automation Conference*, pp. 63-68, 2006.

[18] http://www.nvidia.com/object/product_tesla_s1070_us.html