2010 Ninth International Symposium on Parallel and Distributed Computing

Parallel Cycle Based Logic Simulation using Graphics Processing Units

Alper Sen*, Baris Aksanli*, Murat Bozkurt*, and Melih Mert*
**Department of Computer Engineering,*
Bogazici University,
Istanbul, Turkey
Contact Email: alper.sen@boun.edu.tr

***Abstract*—Graphics Processing Units (GPUs) are gaining popularity for parallelization of general purpose applications. GPUs are massively parallel processors with huge performance in a small and readily available package. At the same time, the emergence of general purpose programming environments for GPUs such as CUDA shorten the learning curve of GPU programming. We present a GPU-based parallelization of logic simulation algorithm for electronic designs. Logic simulation is a crucial component of verification of electronic designs that allows one to check whether the design behaves according to the specifications. Verification of electronic designs consumes more than 60% of the overall design cycle. Any attempts to speedup the verification process (and logic simulation) results in great savings and shorter time-to-market. We develop a parallel cycle-based logic simulation algorithm that uses And Inverter Graphs (AIGs) as design representations and exploits the massively parallel GPU architecture. We demonstrate several orders of speedups on benchmarks using our system.**

***Keywords*-GPU, CUDA, design automation, verification**

I. Introduction

Recent emergence of general purpose programming models coupled with extremely high performance, huge memory bandwidth, and comparatively low cost of Graphics Processing Units (GPUs) are turning GPUs into a parallel processing hardware platform for several types of applications. Compute Unified Device Architecture (CUDA) by NVIDIA [1] is such a general purpose programming model that has accelarated the development of parallel applications beyond that of the original purpose of graphics processing. CUDA is an extension to C language and is based on a few abstractions for parallel programming. General Purpose GPU (GPGPU) computing with CUDA has spread in various application areas ranging from computational biology, to computational finance and electronic designs, where huge speedups have been achieved [2].

In this paper, we focus on parallelization efforts for verification of electronic designs. The complexity of electronic designs have been rapidly growing. The task of verifying such systems becomes an immense challenge and often products are delivered with bugs. Logic simulation allows us to check that the observed behavior of the design under test respects the specification of the design. However, logic simulation of designs with millions of components is time consuming and has become a bottleneck in the design process. Any means to speedup logic simulation results in productivity gains and faster time-to-market. We observe that electronic designs exhibit a lot of parallelism that can be exploited by parallel algorithms. In fact, there are parallel electronic design automation algorithms for almost all stages in the implementation of a design such as logic optimization, simulation, floor-planning, routing, and physical verification stages.

There are two main types of logic simulation; Cycle-Based Simulation (CBS) and Event-Based Simulation (EBS). In CBS, evaluation schedule of gates in the design for each step of simulation is determined once at compile time of the simulator. EBS has a more complicated scheduling policy where a gate is simulated only if at least one of its input values have changed. Both CBS and EBS are commonly used in the industry. In this paper, we work with CBS since it has a less complicated static scheduling policy that is amenable to better parallelization. However, we also note that logic simulation has been classified as the most challenging Computer Aided Design pattern to parallelize in [3].

Logic simulation proceeds in two phases; compilation and simulation. The compilation phase is part of any modern simulator and is required to convert the design into an appropriate form for the simulation phase. Our GPU based solution is also composed of these two phases. In the compilation phase, we perform several operations namely combinational logic extraction, levelization, and clustering operations. This phase has several GPU architecture dependent optimizations. Levelization helps determine the dependency between gates, where gates in the same level can be simulated in parallel. Clustering helps partition the design into a collection of smaller parts where each part can be simulated independent of other parts. These operations are necessary because in a gate level design, certain gates could have hundreds or thousands of fanouts while most will have a few fanouts resulting in irregular data access patterns. Clustering helps organize access patterns for effective simulation. The simulation phase is where thousands of threads are available to simulate the compiled design in parallel. We use several optimizations in simulation phase in order to reduce the communication overhead between the GPU and the CPU. In both phases, we exploit the GPU memory resources as efficiently as possible in order to have low latency.

Our design representation for logic simulation is different from the common design representations where we use And-Inverter Graphs (AIGs), in particular we use the AIGER format [4]. AIG is an efficient representation for manipulation of boolean functions. It is being growingly used in logic synthesis, technology mapping, verification, and boolean satisfiability checking. Since we use AIGs with only a single type of combinational gate (and-gate), our algorithms can efficiently use the limited low latency memory spaces provided by the GPU.

We validated the effectiveness of our parallel CBS algorithm with several benchmarks from OpenCores and IWLS. We compared our parallel CBS algorithm with that of a sequential CBS algorithm. Our experiments show that parallel CBS can speedup the simulation of designs several orders over the sequential algorithm. In particular, we obtained more than 5x speedup for a low density parity checker design.

This paper is organized as follows. In the next section, we give an overview of related work. We then describe background in CUDA, AIGER format and logic simulation in Section III. In section IV, we describe our parallel CBS algorithm. Our experiments are in section V. Finally, conclusions and future work is described.

II. Related Work

Design patterns for parallelization in Computer Aided Design (CAD) has been explained in [3]. The authors consider 17 different CAD algorithms and partition these algorithms in three categories; graph algorithms, branch and bound search, and linear algebra. They also state that graph algorithms in CAD (of which logic simulation is a member) are the hardest to parallelize among these categories. Similarly, CAD case studies of GPU acceleration can be found in [5]. Some of these case studies are SPICE simulations, fault simulation, static timing analysis, boolean satisfiability, fault dictionary computation, and power grid analysis. The authors describe optimization techniques for irregular EDA applications on GPUs in [6]. In particular, they make use of memory coalescing and shared memory utilizations that improve the speedup of sparse matrix vector product and breadth first search. There has been a lot of work on parallel logic simulation using architectures other than GPUs. There are several surveys on parallel logic simulation [7], [8]. In particular, cycle-based simulation approaches are used by IBM and others [9]. The simulation algorithms in these works are aimed at loosely coupled processor systems.

Different partitioning algorithms for electronic designs are described in [10], [11]. Some of these algorithms are based on performance, layout, clustering, network flow, and spectral methods. Our partitioning approach is similar to cone clustering described in [12], where a fanin cone of a circuit element embodies an area of combinational logic that has the potential to influence signal values provided by that element.

Several general purpose GPU applications can be found in [2], [1]. The application domain ranges from physics to finance and the medical field. The work in [13] gives a performance study of general purpose applications using CUDA and compares them with that of applications written using OpenMP. OpenCL [14] is a relatively new standard that is very similar to CUDA in that it is also an extension of C language. CUDA is specific to NVIDIA GPUs, whereas OpenCL can be run on different architectures and gives you portability at the expense of potentially sub-optimal performance for any specific platform. Also, CUDA is a more mature environment with high-performance libraries and accompanying mature tools like debuggers and profilers.

Our CBS algorithm is most similar to the work in [15]. However, there are several differences. We use AIGs as gate level representation whereas they support a generic library of gates. AIGs allow us to efficiently use the limited low latency memory spaces. We use a threshold value in clustering, whereas they start clusters from the primary outputs. In case the design has too few or too many outputs, their approach suffers from complication of the balancing operations. There is an earlier logic simulation algorithm using GPUs in [16]. However, this algorithm does not provide performance benefits since they do not optimize data transfer between GPU and CPU, use a different partitioning approach and do not use the general purpose programming language CUDA. There is also a recent work on event-based simulation algorithm, which is also a commonly used simulation technique in the industry, using CUDA [17].

III. Background

In this section, we are going to present background on CUDA programming, AIGER format, and sequential cycle based logic simulation of electronic designs.

A. NVIDIA CUDA Programming

Compute Unified Device Architecture (CUDA) is a small C library extension developed by NVIDIA to expose the computational horsepower of NVIDIA GPUs [1]. GPU is a compute *device* that serves as a co-processor for the *host* CPU. CUDA can be considered as an instance of widely used Single Program Multiple Data (SPMD) parallel programming models. A CUDA program supplies a single source code encompassing both host and device code. Execution of this code consists of one or more phases that are executed either on the host or device. The phases that exhibit little amount of parallelism are executed on the host and rich amount of parallelism are executed on the device. The device code is referred to as *kernel* code.

The smallest execution units in CUDA are *threads*. Thousands of threads can work concurrently at a time. GPU has its own device memory and provides different types of memory spaces available to threads during their execution. Each thread has a private local memory in addition to the registers allocated to it. A group of threads forms a CUDA *block*. A CUDA block can have at most 512 threads, where each thread has a unique id. A group of blocks forms a CUDA *grid*. Each thread block has a *shared memory* visible to all threads of the block within the lifetime of the block. Access to shared memory is fast like that of a register. Threads in the same block can synchronize using a barrier, whereas threads from different blocks can not synchronize. Each thread has access to the *global device memory* throughout the application. Acces to the global memory is slow around 400-600 cycles. There are also special memory spaces such as texture, constant, and page-locked (pinned) memories. All types of memory spaces are limited in size, and should therefor be handled carefully in the program. Figure 1 displays NVIDIA CUDA arcitecture with several multiprocessors that can execute one or more blocks in parallel.

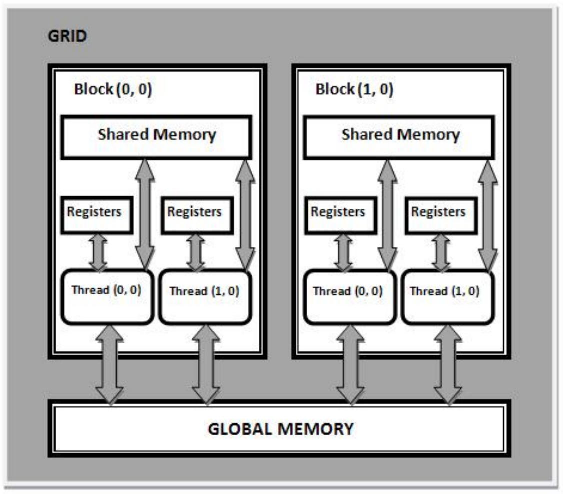

Figure 1. CUDA Architecture

B. AIGER Format

And-Inverter Graph (AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a design, circuit or network. AIGER format is an implementation of AIGs [4]. An AIG is composed of two-input and-nodes (combinational elements) representing logical conjunction, single input nodes representing memory elements (latches, sequential elements), nodes labeled with variable names representing inputs and outputs, and edges optionally containing markers indicating logical negation. We refer to and-nodes and latches as gates. AIG is an efficient representation for manipulation of boolean functions.

There has been a growing interest in AIGs as a functional representation for a variety of tasks in Electronic Design Automation (EDA) such as logic synthesis, technology mapping, verification, and equivalence checking. Especially, the recent emergence of efficient boolean satisfiability (SAT) solvers using AIGs instead using the Binary Decision Diagrams (BDD) has made AIGs popular in EDA.

Figure 2 displays an example circuit design. In this figure, and-gates are identified as *G#*, where # stands for the unique gate number; and latches are identified as *L#*, where # stands for the unique latch number. *i# and o#* stand for (primary) inputs and (primary) outputs, respectively.

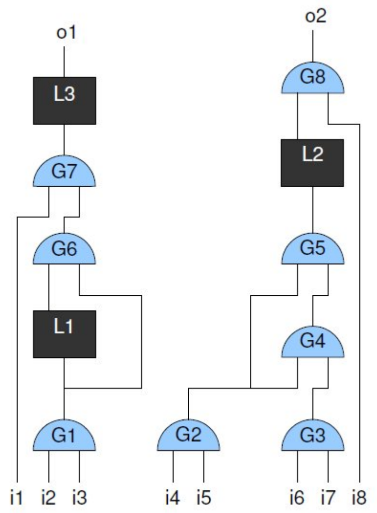

Figure 2. An example design

C. Sequential Cycle-Based Simulation

Cycle-based simulation (CBS) is a commonly used logic design simulation technique. Given a sequence of input vectors, the goal is to generate the output vectors of the design. CBS is a form of time driven or compiled mode simulation because the evaluation schedule of gates for each simulation step is determined once at the compile time of the simulator. The time in CBS is not a physical time rather an integer that denotes the current cycle. CBS represents a fast alternative to event-based simulation. In CBS, logic values of *all* gates are calculated at clock defined cycle boundaries, whereas event-based simulators calculate logic values of a gate if a change occurs at the inputs of that gate. This property of CBS may bring some redundancy since inputs of a combinational element may not change at every cycle. However, it eliminates costly management of scheduling of events as in event-based simulation resulting in efficient implementation rules and better performance.

Figure 3 displays a pseudocode for a sequential CBS algorithm. In our case, since we use AIGER format, the only combinational elements are and-gates. At each step, both combinational and sequential elements are evaluated. First, the combinational elements are evaluated generating the next latch values. The combinational elements can be simulated in any order as long as the inputs of an and-gate are ready for the current cycle. An optimization at this step would be to levelize the combinational elements and simulate them level by level, which we will explain in detail in the next sections. Second, sequential elements are simulated. The next latch values are stored in a second set of latch variables that will take the role of the present (current) latch variables in the next cycle. Finally, the output values are calculated at every cycle using the current values of sequential elements and combinational elements. For example, assuming that the latch values are initially all zero, the outputs of the design in Figure 2 with respect to a given input sequence is displayed in Table I.

```
obtain the number of simulation cycles;
for i=0 to number of cycles do
    read inputs;
    simulate all combinational elements;
    simulate all sequential elements;
    generate outputs;
end for
```

Figure 3. Sequential Cycle-Based Simulation Algortithm

Cycle	Inputs	Latches	Outputs
	$i_1i_2i_3i_4i_5i_6i_7i_8$	$L_1L_2L_3$	o_1o_2
0	11111111	000	00
1	01111111	111	11
2	11111111	110	01

Table I
CYCLE-BASED SIMULATION OF DESIGN IN FIGURE 2

IV. PARALLEL CYCLE-BASED SIMULATION USING GPUS

In this section, we describe our algorithm. Figure 4 displays a pseudocode for our parallel CBS algorithm with compilation and simulation phases. First, we extract combinational elements of a design since we can simulate combinational and sequential elements separately as described in Figure 3. Next, we levelize the extracted combinational design. Then, we divide the levelized design into clusters which are sets of gates that we define later. We then simulate combinational elements in parallel followed by sequential elements. Now we give details of our algorithm.

A. Levelization

In CBS, we can simulate combinational and sequential elements separately. Combinational elements can also be

```
{compilation phase}
extract combinational elements;
levelize combinational elements;
cluster combinational and sequential elements;
{parallel simulation phase}
obtain the number of cycles for simulation;
for i=0 to number of cycles do
    read inputs;
    simulate all clusters;
    generate outputs;
end for
```

Figure 4. Parallel Cycle-Based Simulation Algortithm.

simulated level by level, where the *level of a gate* is defined as the largest distance from the inputs and the memory elements of the design. For example, the level of gate $G7$ in Figure 2 is 3. The level of a gate also describes its evaluation order with respect to other gates, that is, a lower level gate is simulated before a higher level gate. Hence, a level encodes the dependency relation between gates in the design where the input of a gate can be the output of another gate from a lower level. This levelization enables parallelization of CBS since the simulation of gates in the same level are independent of each other and can be done in parallel.

Figure 5 displays the design in Figure 2 after levelization step. Note that there are no latches displayed in the figure since levelization is done on the combinational portion of the design. We also assume that all latch values and input values are given initially. Hence, latches and inputs can be considered to be in level 0. In the figure, we denote the present and next latch values by *PL#* and *NL#*, respectively.

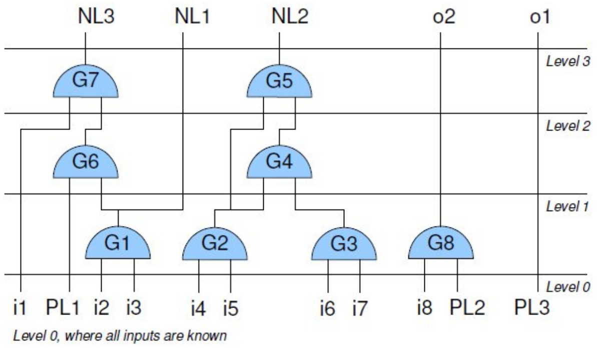

Figure 5. Levelization of the circuit in Figure 2. The upper levels are dependent on the lower levels, whereas the gates in the same level are independent of each other.

B. Clustering of Gates

Once the design is levelized, we can partition both the combinational and sequential elements into sets. We call each such set a *cluster* or *block*. Our goal is to generate clusters where each cluster can be simulated independently of other clusters. Then we can simulate each gate in a level of a cluster by a separate thread since simulation of gates in the same level are independent of each other.

We used several heuristics for clustering while exploiting the GPU architectural properties. In particular, we want to maximize the usage of available CUDA threads. Each CUDA block of threads can have up-to 512 threads and we can have up-to 192 blocks in CUDA. Furthermore, CUDA allows thread synchronization, that is, threads in a block can be synchronized; however, threads in different blocks cannot be synchronized. An attempt to distribute all gates such that each level is simulated by a separate block would require the synchronization of different blocks since each level has to be completely simulated before the next level. Hence, this approach is not possible.

Figure 6 displays the pseudocode of our clustering algorithm. Figure 7 displays application of this algorithm to the design in Figure 5. First, starting from the gates at the highest level, we search for the level where the number of gates in that level is greater than a given threshold value. We call this level the *threshold level*. This threshold value is crucial in obtaining a good partitioning of the gates. We add all the gates from the highest level down-to but not including the threshold level into a single cluster ($clusterH$). Once we reach the threshold level, we determine the cone of logic of every gate in that level, where the *cone of logic* of a gate is the set of elements encountered during a backtrace from a gate to inputs or memory elements. For example, the cone of logic of gate $G5$ in Figure 5 includes $G5, G4, G3, G2$. Each cone of logic is a cluster for our purposes, named $clusterGi$ for each gate Gi in threshold level. Note that since gates can be in the cone of logic of more than one cluster they may need to be duplicated for each cluster. This duplication allows us to simulate each cluster independent of other clusters. After these steps, there may still be some gates that are not part of any cluster. In particular, there may be combinational gates that are not in the cone of logic of any gate at the threshold level. For example, gate $G8$ is one such gate. Hence, we add such gates to $clusterR$. Also, we add all sequential elements to a separate $clusterS$. At the end of the clustering algorithm we have the following clusters, $clusterH$, $clusterGi$ for each gate Gi in threshold level, $clusterR$, and $clusterS$.

Although most clusters can be simulated independent of each other, there is a *dependency* among some clusters. In particular, $clusterH$ can be simulated only after all $clusterGi$ have been simulated. This is because all $clusteGi$ are in the cone of logic for gates in $clusterH$, hence $clusterH$ is dependent on all $clusterGi$. Furthermore, $clusterS$ for sequential elements can be simulated only after all combinational clusters have been simulated since the next latch values can only be calculated after all combinational clusters have been simulated.

```
$level$ = the highest level of gates;
**repeat**
  **if** number of gates in $level > threshold$ **then**
    add the gates from the highest level until the current level to $clusterH$;
    break; {found threshold level}
  **end if**
  level = level - 1;
**until** level = 0;
**if** $level = 0$ **then**
  update threshold value and rerun above steps;
**end if**
**for** i=1 to number of gates in $level$ **do**
  add the cone of logic for gate Gi to $clusterGi$;
**end for**
add remaining combinational gates to $clusterR$;
add sequential gates to $clusterS$;
```

Figure 6. Clustering Algortithm

In Figure 7, we apply clustering algorithm on the design in Figure 5. We assume that the threshold value is 2, in which case the top level (level 3) automatically becomes the *threshold level*. Clusters 1 and 2 are obtained by finding the cone of logic of gates in level 3. Whereas, cluster 3 is a cluster of remaining combinational elements. Similarly, there is a cluster 4 that includes all latches.

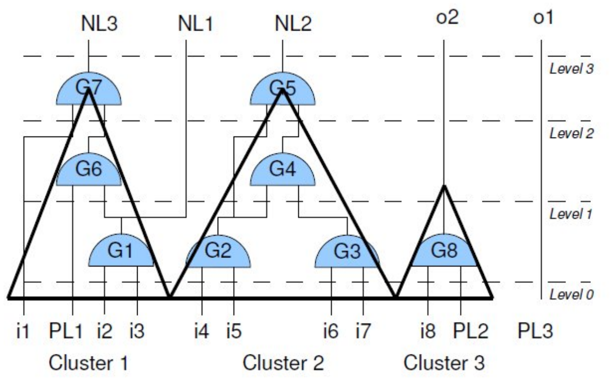

Figure 7. Clustering on the levelized design in Figure 5. Distinct clusters are independent of each other. However, there is a dependency between levels within a cluster.

In Figure 8, there is a conceptual view of clustering operation. Each triangle represents a block/cluster. The cluster above the threshold level is denoted by $clusterH$ in the algorithm. The intersections of clusters represent the design elements that are duplicated due to having common cone of logic elements. This duplication is necessary to ensure the simulation independence between the clusters.

In determining clusters, we first considered clusters starting from the primary outputs instead of starting from the

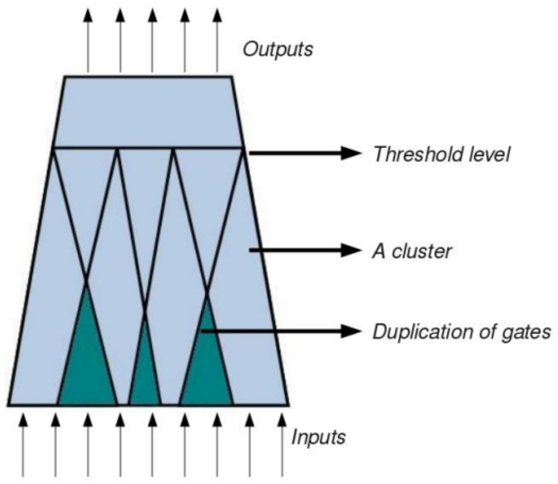

Figure 8. Visualizing Clustering. Threshold level is the level where the number of gates is at least the number specified by clustering threshold. The upper corner of each triangular represents a gate in the threshold level.

highest level of gates. However, since designs can have varying number of outputs ranging from a single output to several thousands, such clusters lead to inefficient distribution of gates and drastically increase the complexity of balancing step. Whereas, we can control the distribution of gates by changing the threshold value and obtain better performance.

Partitioning of gates should generate a balanced distribution of gates into blocks of similar size in order to increase efficiency. Moreover, the number of blocks should be neither too low nor too high. If it is too low, then our algorithm works similar to the sequential simulation. If it is too high, then more gates may need to be duplicated among clusters and computation overhead could increase.

C. Parallel Simulation Phase

```
obtain the number of cycles for simulation;
for i=0 to number of cycles do
    execute test-bench on the host and generate inputs
    for the design;
    transfer inputs from host to device;
    simulate all clusters on the device and generate
    outputs;
    transfer outputs to the host;
end for
```

Figure 9. Parallel Simulation Phase

Parallel simulation phase consists of several steps that can be seen in Figure 9. During each cycle, input values are transferred from host to device. The inputs are essentially generated by executing the test-bench on the host. In our experiments, we mainly used random test-benches, hence the input generation could be optimized as described in the next section. Once the inputs are ready, all clusters can be executed by executing a kernel function while respecting the cluster dependency explained above. An execution of a block with combinational elements proceeds level by level and after each level is simulated the threads of the block synchronize using a barrier. This process continues until all levels are completed. Blocks fetch relevant design data and input values from device memory before simulation of the levels and transfer output values and next latch values to the device memory after the simulation of the clusters. At the end of the cycle, output values are transferred from device to host. In particular, we stored the design structure, primary inputs and outputs and latches in the device memory and frequently accessed data structures such as intermediate and-gate values of a block in the shared memory.

D. CUDA Optimizations

We applied several optimization methods to increase performance of our algorithms. To develop an effective CUDA program, one needs to have knowledge of the GPU architecture such as shared memory bank conflicts and memory transfer overhead. Communication overhead is a major cost. We exploited CUDA's shared memory, register and pinned memories for our implementation.

We decreased the number of costly operations such as multiplication and division. In particular, if the same multiplication or division operation has to be done at every iteration of a loop, the outcome of this operation is done before the loop and recorded in a register to be used by other threads.

In CUDA, one can allocate memory space in host memory dynamically by using $CudaMallocHost$ function instead of the ordinary C $malloc$ function. This uses the pinned (paged-locked) memory. Allocating memory with $CudaMallocHost$ increases the performance of $CudaMemcpy$ operation. We used these functions to allocate data structures for our designs.

For designs with random input generation (random test-benches), we moved the test-bench code into the GPU and enabled parallel input generation. This removes the communication need between CPU and GPU. For designs without a random input generation, we used another optimization method where the inputs of the design and outputs of the design are not transferred between GPU and CPU during every cycle, rather these data transfers are done in bulks such as after every 10000 cycles. Although this optimization increases the data size to be copied between the host and the device, our experiments have proven that decreasing the number of $CudaMemcpy$ operations is more critical than increasing the data size.

One of the most important optimizations for CUDA programming is to make effective use of shared memory. Fetching a variable from device memory requires 400-600 cycles, whereas fetching it from the shared memory requires

Design	Inputs	Outputs	And-Gates	Latches	Levels	Clusters
ldpc-encoder	1723	2048	218961	0	19	288
des-perf	17850	9038	78299	8808	19	373
wb-conmax	1130	1416	47853	770	26	261
pci-bridge32	162	207	22784	3359	29	146
key-orig	382	82	24182	33	75	22
ethernet	98	115	69684	10544	31	364
vga-lcd	89	109	126711	17079	23	145

Table II
EXPERIMENTAL TEST CASES

only a few cycles. However, each shared memory for a CUDA block is limited by 16 KB. During the execution of a cluster, we copied the most frequently accessed data structures to the shared memory such as intermediate and-gate values in a cluster. Also, since we use AIGER format with only a single type of combinational gate (an and-gate), we do not need to store the truth table for other types of complex gates in the shared memory, wasting valuable space.

V. EXPERIMENTAL RESULTS

We validated the effectiveness of our GPU based parallel CBS algorithm on several test cases. We used test cases from IWLS [18], OpenCores [19] and AIGER [4]. These test cases ranged from a combinational ldpc encoder (low density parity check encoder), to complex sequential designs such as des-perf (triple DES optimized for performance), wb-conmax (wishbone conmax IP core), pci-bridge32, ethernet, and vga-lcd (wishbone compliant enhanced vga/lcd controller). These test cases either had their own test benches or we generated random test benches for them. Table II displays the list of test cases that we used in our experiments and their design characteristics. For some test cases, we generated AIGER format using ABC tool [20].

We performed our experiments on a dual quad core Intel Xeon (2.27GHz) processor with 32GB of memory and a CUDA–enabled NVIDIA Quadro FX3800 GPU with 1 GB device memory and 24 multiprocessors each with 8 cores.

In Table III, we demonstrate our experimental results. Column $Cycles$ denotes the number of cycles that the design has been simulated. Column SEQ denotes the results for sequential simulation using the default simulator that is available with AIGER format [4]. Column PAR denotes the results for our parallel simulation algorithm. Column $Speedup$ denotes the speedup of parallel algorithm over the sequential algorithm. The times do not include the compilation phase in both cases but include the time required to transfer the data between the GPU and the CPU.

Different speedups are expected for logic simulation due to the fact that logic simulation is heavily influenced by the circuit structure. We obtained speedups ranging from 1.7x to 5x. We obtained the best speedup for ldpc-encoder design, and when we investigated this design, we observed that the gates are almost equally distributed to the blocks. We also observed that when the size of the design gets bigger, the speedup ratio also gets bigger. We simulated the designs for different number of cycles ranging from 100K to 1M. The speedups were similar when the number of cycles were increased.

From Table II, we can see that the number of outputs and the number of clusters that gives the best results can be very different from each other. Hence, a clustering algorithm that considers the outputs as cluster heads could result in slower execution times. We also experimented with the threshold values. We observed that the threshold value is dependent on the circuit structure as well. For example, for key_orig testcase the execution time changes almost linearly with the threshold value. However, this was not the case for other test cases. We observed that the number of gates in the designs can be drastically higher using AIGER format than the other formats. Although our experiments showed that AIGs are effective in reducing the shared memory size, AIGs may result in higher number of levels that may lead to execution overheads.

Our compilation phase may spend more time than the sequential algorithm in order to find the threshold value. We note that, in practice, once the optimal threshold values are obtained, the simulations are run until the design product is taped-out. Hence, it is worth paying the initial cost.

VI. CONCLUSIONS

We introduced a novel parallel cycle-based simulation algorithm for digital designs using GPUs. Our algorithm results in a fast, efficient parallel logic simulator that can run on commodity graphics cards allowing verified designs while obtaining significant reduction in the overall design cycle. Our approach leverages the GPU architecture by optimizing on low latency memory spaces, and reducing host and device communications during compilation and simulation phases of a cycle based simulation algorithm. Our approach is unique in that we use the AIG representation for electronic designs that proves to be very efficient for boolean functions. We obtained speedups for various benchmarks. Since logic simulation is an activity that continues nonstop until the design is productized such speedups have a big impact on the design cycle. Our experiments confirm that the

Design	Cycles	SEQ(sec)	PAR(sec)	Speedup
ldpc-encoder	100K	382.21	74.79	**5.11**
	500K	1907.79	376.03	**5.07**
	1M	3796.85	749.47	**5.07**
des-perf	100K	180.62	65.68	**2.75**
	500K	891.84	330.10	**2.71**
	1M	1803.54	660.63	**2.73**
wb-conmax	100K	94.81	33.96	**2.79**
	500K	473.83	171.08	**2.76**
	1M	936.19	336.76	**2.78**
pci-bridge32	100K	50.12	21.70	**2.30**
	500K	250.56	105.47	**2.37**
	1M	499.82	209.46	**2.38**
key-orig	100K	46.47	24.62	**1.89**
	500K	220.14	123.46	**1.78**
	1M	438.53	246.96	**1.78**
ethernet	100K	155.51	52.89	**2.94**
	500K	778.87	261.23	**2.98**
	1M	1556.11	522.89	**2.97**
vga-lcd	100K	223.15	45.91	**4.86**
	500K	1118.76	231.57	**4.83**
	1M	2232.56	460.78	**4.84**

Table III
EXPERIMENTAL RESULTS

circuit structure has a dramatic influence on the performance of parallel CBS algorithm.

As a future work, we want to investigate other design formats and develop event-based simulation algorithms that are also commonly used in the industry. We also want to develop different clustering and balancing strategies and experiment with complex industrial test cases.

ACKNOWLEDGMENTS

We would like to thank Alan Mischenko for helping with AIGER benchmarks. This research was supported by a Marie Curie European Reintegration Grant within the 7th European Community Framework Programme and BU Research Fund 09HA101P.

REFERENCES

[1] "NVIDIA CUDA web site," http://www.nvidia.com/CUDA.

[2] H. Nguyen, *Gpu Gems 3*. Addison-Wesley Professional, 2007.

[3] B. Catanzaro, K. Keutzer, and B.-Y. Su, "Parallelizing CAD: a timely research agenda for EDA," in *Proceedings of the Design Automation Conference (DAC)*. ACM, 2008, pp. 12–17.

[4] "AIGER Format web site," http://fmv.jku.at/aiger/.

[5] J. F. Croix and S. P. Khatri, "Introduction to GPU programming for EDA," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. ACM, 2009, pp. 276–280.

[6] Y. S. Deng, B. D. Wang, and S. Mu, "Taming Irregular EDA Applications on GPUs," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. ACM, 2009, pp. 539–546.

[7] M. L. Bailey, J. V. Briner, Jr., and R. D. Chamberlain, "Parallel logic simulation of VLSI systems," *ACM Comput. Surv.*, vol. 26, no. 3, pp. 255–294, 1994.

[8] G. Meister, "A survey on parallel logic simulation," Dept. of Computer Engineering, University of Saarland, Technical Report, 1993.

[9] K. Hering, "A Parallel LCC Simulation System," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[10] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration, the VLSI Journal*, vol. 19, no. 1-2, pp. 1 – 81, 1995.

[11] F. M. Johannes, "Partitioning of VLSI circuits and systems," in *Proceedings of the Design Automation Conference (DAC)*. ACM, 1996, pp. 83–87.

[12] K. Hering, R. Reilein, and S. Trautmann, "Cone Clustering Principles for Parallel Logic Simulatio," in *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2002, pp. 93–100.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370 – 1380, 2008.

[14] "OpenCL web site," http://www.khronos.org/opencl/.

[15] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: High-performance gate-level simulation with GPGPUs," in *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, 2009, pp. 1332–1337.

[16] A. Perinkulam, "Logic Simulation using Graphics Processors," Master's thesis, University of Massachusetts Amherst, 2007.

[17] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Proceedings of the Design Automation Conference (DAC)*, 2009, pp. 557–562.

[18] "IWLS 2005 Benchmarks," http://www.iwls.org/iwls2005/benchmarks.html.

[19] "Opencores Benchmarks," http://www.opencores.org.

[20] "ABC web site," http://www.eecs.berkeley.edu/˜alanmi/abc/.