2010 11th International Workshop on Microprocessor Test and Verification

Using Graphics Processing Units for Logic Simulation of Electronic Designs

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

***Abstract*—Logic simulation is the major verification technique used for electronic system designs. Speeding up logic simulation results in great savings and shorter time-to-market. We parallelize logic simulation using Graphics Processing Units (GPUs). We present a parallel cycle-based logic simulation algorithm that uses And Inverter Graphs (AIGs) as design representations. We partition the gates in the design into independent blocks and simulate these blocks using the GPU. Our algorithm exploits the massively parallel GPU architecture featuring thousands of concurrent threads, fast memory, and memory coalescing for optimizations. We demonstrate up-to 21x speedup on several benchmarks using our simulation system.**

***Keywords*-verification, logic simulation, Graphics Processing Units (GPU)**

I. Introduction

Logic simulation of electronic 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 this paper, we focus on parallelization of logic simulation of electronic designs using Graphics Processing Units (GPUs).

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 accelerated 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. CUDA has accelerated the development of parallel applications beyond that of the original purpose of graphics processing.

CAD case studies of GPU acceleration can be found in [2]. Some of these case studies are spice 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 [3]. There has been a lot of work on parallel logic simulation using architectures other than GPUs. There are several surveys on parallel logic simulation [4], [5]. In particular, cycle-based simulation approaches are used by IBM and others [6]. The simulation algorithms in these works are aimed at loosely coupled processor systems. Earlier, we have developed non-parallel verification techniques [7].

We use And-Inverter Graphs (AIGs) for design representation. AIG is an efficient representation for manipulation of boolean functions. It is increasingly 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 present two clustering algorithms The first one [8] clusters gates using a given threshold, and the second one improves clustering with that of merging and balancing steps and incorporates memory coalescing and efficient utilization of shared memory. We validated the effectiveness of our parallel CBS algorithm for both types of clustering with several benchmarks. 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 over the sequential algorithm. In particular, we obtained up-to 5x speedup with the first clustering algorithm and up-to 21x with the second clustering algorithm.

II. Background

A. 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 cannot synchronize. Each thread has access to the *global device memory* throughout the application. Access to the global memory is slow and 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 therefore be handled carefully in the program. For an NVIDIA FX3800 GPU, each multiprocessor can execute 768 threads in parallel and shared memory size is limited where each multiprocessor can have 16 KB of shared memory.

B. And-Inverter Graph (AIG)

And-Inverter Graph (AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a design or circuit. AIGER format is an implementation of AIGs [9]. 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.

The combinational logic of an arbitrary Boolean network can be factored and transformed into an AIG using DeMorgans rule. The following properties of AIGs facilitate development of robust applications in synthesis, mapping, and formal verification. Structural hashing ensures that AIGs do not contain structurally identical nodes. Inverters are represented as edge attributes. As a result, single-input nodes representing inverters and buffers do not have to be created. This saves memory and allows for applying DeMorgans rule on-the-fly, which increases logic sharing. The AIG representation is uniform and fine-grain, resulting in a small, fixed amount of memory per node. The nodes are stored in one memory array in a topological order, resulting in fast, CPU-cache-friendly traversals.

III. PARALLEL CYCLE-BASED SIMULATION (CBS) USING GPUS

Algorithm 1 displays the pseudocode for our parallel CBS algorithm. Our algorithm has compilation and simulation phases. In CBS, we can simulate combinational and sequential elements separately. A speed-up technique for the simulation of combinational elements is to simulate them level by level. Once the design is levelized, we can partition both the combinational and sequential elements into sets. We call each such set a *cluster*. 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 then balance these clusters in order to maximize CUDA thread usage and map these balanced clusters to CUDA blocks. We developed two different clustering algorithms. Figure 1 is a conceptual view of the result of our first clustering algorithm. Each triangle represents a cluster. We use a threshold value in order to control the number of CUDA blocks. Starting from the gates at the highest level, we search for the level where the number of gates in that level is greater than or equal to a given threshold value. We call this level the *threshold level*. The intersections of clusters represent the design elements that are duplicated. This duplication is necessary to ensure the simulation independence between the clusters.

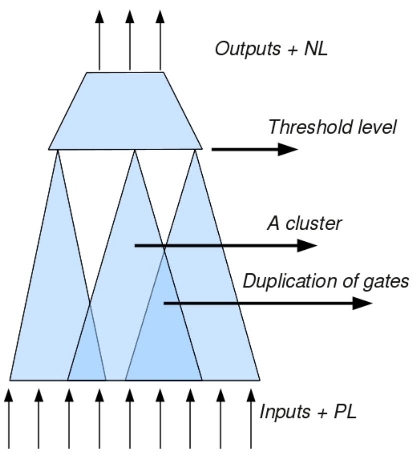

Figure 1. Visualizing Clustering Algorithm 1. The threshold level is the level where the number of gates is at least the number specified by clustering threshold. The top corner of each triangle represents a gate in the threshold level.

It is clear that we can control the number of CUDA blocks using the threshold value with this clustering algorithm. However, this becomes a manual effort and some designs may not have as many gates as specified in the threshold value or may have many more gates than the threshold value. Furthermore, clustering of remaining gates needs extra effort of searching and finding out the gates that are not part of any cluster. In such cases, the first clustering approach may not be useful. Hence, we developed an optimized approach that allows us to better control the number of CUDA blocks, the shared memory and ultimately obtain better speedups.

In the second clustering algorithm, we build clusters starting from the primary outputs and latches instead of starting from the user given threshold level as in the first clustering algorithm. This allows us not to have any remaining gates as was the case above. Also, since designs can have varying number of outputs and latches, the number of clusters and the sizes of clusters can vary. This necessitates further steps to optimize thread usage. For this purpose, we develop new steps for merging and balancing of clusters. This avoids assigning every cluster to a CUDA block as in the previous algorithm, rather we reshape clusters into an optimized format then assign those clusters to CUDA blocks. Figure 2 is a conceptual view of the result of our first clustering algorithm.

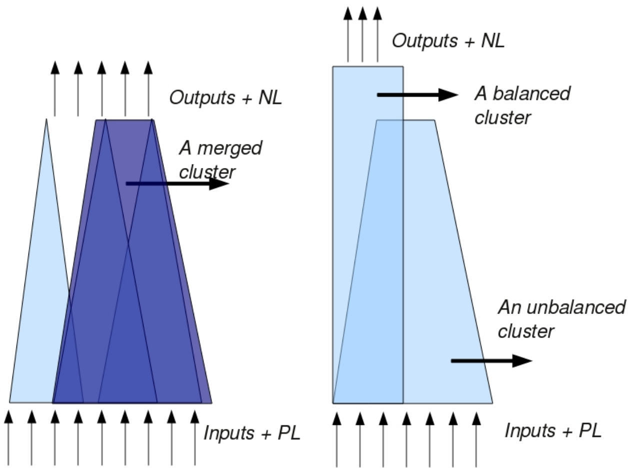

Figure 2. Merging and balancing clusters results in using all available threads efficiently at all levels.

After obtaining CUDA blocks, we are ready for parallel simulation. In parallel simulation phase, we execute the test-bench in order to generate primary input values at each cycle, next we transfer these values to the GPU. Then, we simulate in parallel CUDA blocks for combinational and sequential elements.

Parallel simulation phase consists of several steps. 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 used random test-benches, hence the input generation could be optimized. Once the inputs are ready, all CUDA blocks can be executed by executing a kernel function. 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. We used coalesced memory access for reading and writing gate variables between the global memory and the CUDA processors. We increased the number of gates that can be efficiently simulated in a block by encoding the intermediate gate outputs and storing these in the shared memory. Also, the intermediate gate output values are stored in shared memory in an encoded fashion. At the beginning of a cycle, blocks fetch relevant design data (gate variables) and input values from device memory in a coalesced manner. At the end of the cycle, output and latch values are transferred from device to host. In particular, we stored the design structure, primary inputs, primary 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.

Algorithm 1 Parallel Cycle-Based Simulation Algorithm

```
Input: circuit, number of simulation cycles $num$, test-bench
Output: output values
    // compilation phase
 1: extract combinational elements;
 2: levelize combinational elements;
 3: cluster combinational and sequential elements;
 4: balance clusters;
    // parallel simulation phase
 5: for i=0 to $num$ do
 6:     execute test-bench on the host and generate primary input values;
 7:     transfer inputs from the host to the device;
 8:     simulate all clusters on the device and generate primary output values;
 9:     transfer outputs from the device to the host;
10: end for
```

IV. Experimental Results

We validated the effectiveness of our two GPU based parallel CBS algorithms on several test cases. We used test cases from IWLS [10], OpenCores [11] and AIGER [9]. These test cases include ldpc-encoder (low density parity check encoder), des-perf (triple DES optimized for performance), wb-conmax (wishbone conmax IP core), aes-core (AES Encryption/Decryption IP Core) pci-bridge32, ethernet, vga-lcd (wishbone compliant enhanced vga/lcd controller), and several other designs. These test cases either had their own test benches or we generated random test benches for them. For some test cases, we generated AIGER format of designs using ABC tool [12].

We performed our experiments on an Intel Xeon CPU with two 2.27GHz multiprocessors, 32GB of memory and a CUDA–enabled NVIDIA Quadro FX3800 GPU with 1 GB device memory and 24 streaming multiprocessors each with 8 streaming processors.

Figure 3 graphically displays our speedups. $PAR1$ and $PAR2$ denote the results for our parallel simulation algo-

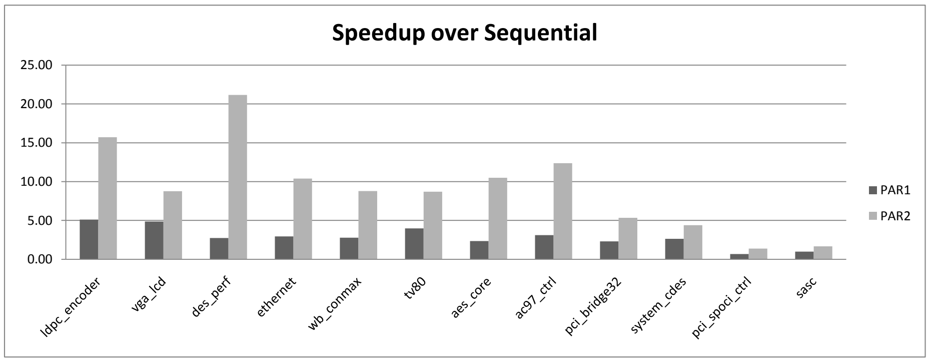

Figure 3. Parallel Simulation Speedups over Sequential Simulation for 100,000 cycles

rithms using the first and the second clustering algorithms, respectively. 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. We simulated the designs for different number of cycles ranging from 100K, 500K to 1M. The speedups are similar for different number of cycles. We present results for 100K cycles in the rest of the experiments.

Different speedups are expected for logic simulation due to the fact that logic simulation is heavily influenced by the circuit structure. We obtained speedups up-to 5x for PAR1 and up-to 21x for PAR2. Both PAR1 and PAR2 outperform Sequential simulation (SEQ) except for two cases where PAR1 performs worse than SEQ. In all test cases, PAR2 significantly outperforms both SEQ and PAR1.

V. Conclusions

Our GPU based parallel logic simulation algorithms result in faster, more efficient simulators that can run on commodity graphics cards allowing verified designs while obtaining significant reduction in the overall design cycle. Our approach is unique in that we use the AIG representation for electronic designs that proves to be very efficient for boolean functions. Our algorithm exploits the massively parallel GPU architecture featuring thousands of concurrent threads, fast memory, and memory coalescing for optimizations. We obtained speedups ranging from 5x to 21x with two different clustering algorithms, respectively, 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 verification and the overall design cycle. Our results show that we obtain better speedups with larger circuits. Hence, further studies with industrial designs hold promise.

Acknowledgments

We would like to thank Alan Mischenko from the UC Berkeley for suggesting and providing AIG benchmarks. This research was supported by a Marie Curie European Reintegration Grant within the 7th European Community Framework Programme and BU Research Fund 5483.

References

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

[2] 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.

[3] 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.

[4] 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.

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

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

[7] A. Sen and V. K. Garg, “Formal Verification of Simulation Traces Using Computation Slicing,” *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 511–527, Apr. 2007.

[8] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, “Parallel Cycle Based Logic Simulation using Graphics Processing Units,” in *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC)*, 2010.

[9] “AIGER Format web site,” http://fmv.jku.at/aiger/.

[10] “IWLS 2005 Benchmarks,” http://www.iwls.org/iwls2005/benchmarks.html.

[11] “Opencores Benchmarks,” http://www.opencores.org.

[12] “ABC web site,” http://www.eecs.berkeley.edu/˜alanmi/abc/.