# Gate-Level Simulation on a GPU

Alvin Tung, Ben Fu, Rohan Nagar

## ABSTRACT

The purpose of this project is to simulate a gate-level netlist on a GPU in order to introduce parallelism during simulation. Gate-level logic networks are currently simulated with software such as ModelSim and is usually based on event driven evaluations on a CPU. Our approach allows for multiple gate layers to be evaluated simultaneously to improve the overall speed of simulation. This project involves parsing synthesized EDF files, constructing a graph of the represented network, and then performing the simulation on a GPU. Our results show that this approach is more performant than software simulators provided by the Digital System Design course of the University of Texas at Austin, and our implementation can be used by students to speed up their development and testing processes.

## INTRODUCTION

Simulations of gate-level logic networks has long been a time-consuming and difficult task. In this work, we attempt to improve the performance of simulation and design verification by simulating gate-level networks on GPUs. Using CUDA, we can perform concurrent simulation and boost the performance compared to existing simulation software, such as ModelSim. Such a performance improvement has the potential to speed up the verification process for gate-level networks. Our attempt, based on our time constraints and resources, focuses on creating a streamlined process that takes Verilog designs from synthesized RTL gate-level netlist files to a CUDA representation and simulation.

## MOTIVATION

After noticing the parallels between the GPU execution style and the flow of gate level logic in hardware, we performed research to determine if simulating gate level netlist in parallel has been attempted. From

our search, we found papers by D. Chatterjee et al. that described a GPU-accelerated logic simulator written in CUDA to verify digital designs [1]. These papers, along with the fact that we experienced slow compute times in our own gate-level simulations with ModelSim, provided us the motivation to tackle this problem. We wanted to combine our experience in digital logic design with GPU programming and provide a modularized network simulator that is easy to use and would decrease development time.

**Problem Background**

One of the biggest problems in the digital design industry today is the time-consuming nature of verification using simulation software [2]. Simulation is heavily relied upon to verify that the behavioral description of a design is functioning correctly and corresponds to the original design specification. It also plays a big role in estimating the power of networks and the timing of each gate. For these reasons, it would be beneficial to the industry if the simulation process was sped up. In order to achieve this, some parallelization, specifically on a GPU, must be introduced. It is best to approach this problem by designing an oblivious simulator, one that evaluates every gate during every simulation, since this provides for easy static scheduling. To interact directly with the GPU on a machine, NVIDIA has developed a programming framework called CUDA, which will provide us with an opportunity to perform parallel execution [3].

**Design Goals and Functionality**

Our design goal is to implement a modularized gate-level network simulator on a GPU that is comparable to ModelSim. First, the simulator will be able read a synthesized EDF file that represents a gate-level network as input and build a graph from the described network. Then, the system will perform simulation concurrently using CUDA. The simulator will also be easy to use by potential users. Since it reads EDF files, a standard file type produced by the synthesis process of software such as Vivado, users will not

have to do any additional work to run their verification on our system. Due to time constraints, our simulator will not be able to simulate sequential logic designs, but will be able to take a Vivado synthesized design and convert it to be run on TACC supercomputers with CUDA.

## IMPLEMENTATION

We took a three-part approach to implement a simulator that could run in parallel, consisting of a script to parse files that contain a hardware design as a gate netlist, a module to build a graph representation of the netlist, and a simulation engine written using CUDA. This modularization gives us the option to swap out parts in the future if we wanted to extend to various file types or use a different simulation algorithm. Each part of our implementation is described in further detail in the following subsections. Additionally, we discuss alternatives to our design that we previously considered.

### Parsing EDF Files

The first stage of our solution is a Python script that converts an EDF file to a text file with only the information necessary to build a network graph. EDF files are generated after synthesis of a gate-level network in software such as Vivado and represents a gate-level network [4]. Individual network components are represented as *cells*, and cells are composed of inputs, outputs, instances of other components, and network connections. The connections are represented as *nets* in the file. This first script rewrites the information from an EDF file to a text file in order to condense the information and to provide a representation that we can easily build a graph from.

The script works by reading an EDF file one line at a time and constructing each cell in memory in the order that they appear before writing them to the output file. Each cell (or component) can be composed of many other cells, and each of those cells have inputs and outputs. Network connections for each cell

include connections of inputs and outputs from the cells that are encompassed by the cell currently being parsed, which can make the parsing process more complex. Thus, it is necessary to keep track of each type of cell and the names of their inputs/outputs so that we can accurately determine which labels in the connections represent inputs and which represent outputs. Once a cell is completely parsed, it is written to the output file and execution continues to the next cell until the entire file is read.

**Building the Graph**

After converting the original file, we move to our second stage of constructing the graph using C++. In this graph, each gate is a node and the inputs and outputs are connected. After building the graph in-memory, we perform a topological sort on the graph in order to force all of the dependencies of a gate to a lower level. That is, each input of a gate will always be at a level below it so it will always be evaluated and ready to be used. This construction provides a one-to-one correspondence between a row of values and a logic level [1].

Once the design is created as a graph, we save that in a format that can be loaded and run on a NVIDIA GPU using CUDA. To do this, we take each gate's information and store it as an entry into a memory matrix. The gate's location in the matrix corresponds to its netlist location, where the row is the gate's level in the netlist and the column is its assigned gate position horizontally. Each entry of the matrix allocates space for the output value and holds the gate type and its the two input locations. The gate entry allocates space to store its output logic level. The entry also holds the gate type so that the GPU can perform the corresponding logic operation. The 2 input locations are stored in the entry based on their row and column in the matrix. The gate matrix, along with the max design height, width, input and output port's name and location are saved as a header file to be used for simulation with CUDA.

**Simulating with CUDA on TACC**

The final stage of our design is the simulation on NVIDIA GPUs using CUDA. We first load the design from the header file and the inputs from the input file. From the header file, we recreate the gate matrix, and from the input file, we obtain the number of passes to simulate and their corresponding inputs. We then initialize a block on the GPU and copy the matrix and inputs from the host to the device. We exploit the power of the GPU by having each thread evaluate a gate at one level. Each thread first finds its inputs in the memory matrix, evaluates them based on the gate logic, and stores it as the output in its gate entry. When all threads complete this task, they are synchronized and move onto the next level where the process is repeated, using the previously calculated outputs as the inputs. We finish at the last gate level and the outputs are read and stored. If we are doing more than one pass, we then reload the next inputs at the bottom gate level and repeat. When the simulations are complete, the output is copied into the CPU host memory and is printed in order for the user to view.

**Design Extensions and Alternatives**

While we believe that our design is robust and flexible, there are a number of extensions, optimizations, and alternatives from our existing design that we have considered. One major extension we considered was to support sequential logic. According to D. Chatterjee et al., registers can be simply represented as memory locations. At the start of the cycle, the memory locations are read as the register outputs for the current cycle, and after the cycle, the registers are read as the register inputs to the next cycle [2]. Additionally, there are some optimizations that would improve our simulator's efficiency and speed. These include balancing the execution list by reorganizing the gates so that each available thread is always actively processing a gate and not idling, as well as minimizing overlapping redundant gates that are duplicated when clustering is done over many GPU blocks [1]. Finally, in terms of alternative design approaches, we could have designed a parallel simulator without a GPU and used frameworks like

OpenMP or Java Concurrency. This would allow access for students and professionals who do not have a GPU on their personal computers. However, the advantage that a GPU gives in terms of concurrency and optimization was the key factor in deciding to use CUDA.

## TEST RESULTS

In order to test our simulator, we built and synthesized three combinational circuits using Verilog and timed how long simulation took with our design. The three designs are available in the repository and include a simple gate, a four bit subtractor, and an array multiplier. The key focus of this project is speed, so we focused our testing on how quickly we can perform simulations. For comparison, we simulated the same circuit design on ModelSim that uses a event driven style simulator. We ran our code on TACC and produced the timing results of the simulation in an output file. The results of two of these tests are shown in Appendix A. Our four bit subtractor design simulated in 27.82 ms on average, which is a 34x speed up compared to ModelSim. Our array multiplier test simulated in 30.70 ms on average, which is a 15x speed up compared to the same simulation in ModelSim.

## CONCLUSION

As an attempt to solve the problem of long computational times for simulating gate-level netlists, we built a system to perform simulation on a GPU using CUDA. Basing our work off that of D. Chatterjee et al., we built a system with 3 modules that interact with each other in an easy-to-use way so that students and professionals can speed up the netlist verification process. Our focus was on combinational synthesized designs, and we show in our test results that we were successful in improving the performance of simulation. With the amount of speed up that we achieved in our system, we have provided a viable alternative to that of simulators that run sequentially. As this system continues to develop, we can focus on adding more features and supporting more types of netlists.

**REFERENCES**


[1] D. Chatterjee et al., “GCS: High-Performance Gate-Level Simulation with GP-GPUs,” University of Michigan, July 2009. [Online] Available: http://andrewdeorio.com/research/assets/DATE09GCS.pdf.

[2] D. Chatterjee et al., “Gate-Level Simulation with GPU Computing,” University of Michigan, June 2011. [Online] Available: http://web.eecs.umich.edu/~valeria/research/publications/TODAES0611.pdf.

[3] NVIDIA, “CUDA Complete Unified Device Architecture,” 2007.

[4] B. Kemp et al., “A simple format for exchange of digitized polygraphic recordings,” Electroencephalogr Clin Neurophysiol, May 1992. [Online] Available: https://www.ncbi.nlm.nih.gov/pubmed/1374708.

[5] "ModelSim 6.0 SE Performance Guidelines", *ece.uwaterloo.ca*, 2016. [Online]. Available: https://ece.uwaterloo.ca/~ece327/old/2006t3/docs/welcome/ModelSim6.0PerfGuidelines.pdf. [Accessed: 16- Nov- 2016].

# APPENDIX A - TEST RESULTS

### A.1 Four Bit Subtractor

The test shown in Table 1 iterates through all possible inputs (512). ModelSim results were calculated using a file to force the inputs and run for all iterations. Simstats was used to get the actual simulation time. For our system, we used the built-in CUDA API to get time results using timing events.

**Table 1. Four Bit Subtractor Results**

| Test | ModelSim (ms) | PLD Sim (ms) |
|---|---|---|
| 1 | 910.00 | 27.62 |
| 2 | 1120.00 | 27.81 |
| 3 | 780.00 | 27.99 |
| 4 | 970.00 | 27.84 |
| 5 | 1100.00 | 27.82 |
| **Average** | **964.00** | **27.82** |

### A.2 Array Multiplier

The test shown in Table 2 iterates through all possible inputs (256). Results were found in the same way as those in Table 1.

**Table 2. Array Multiplier Results**

| Test | ModelSim (ms) | PLD Sim (ms) |
|---|---|---|
| 1 | 500.00 | 24.34 |
| 2 | 503.00 | 24.92 |
| 3 | 360.00 | 24.52 |
| 4 | 580.00 | 54.88 |
| 5 | 420.00 | 24.83 |
| **Average** | **964.00** | **30.70** |