Gate Level Simulation using GP-GPUs with CUDA

by
Seçkin Savaşçı
seckin.savasci@boun.edu.tr

Boğaziçi University
Cmpe492 Senior Project Midterm Progress Report

November 23, 2012

Contents

0.1 Introduction

A typical digital design flow starts with creating an architectural model. Then designers convert this model into an HDL level design such as RTL model. VHDL or Verilog can be used for this step. After it, a synthesis tool synthesize structural gate level netlist consisting logic primitives. Finally this netlist is implemented on a silicon chip. If the prototype chips work as intended, then they will be released and used in end-user systems.

However more than half of the effort in the design phase goes for verification and validation of the design, which is known as pre-silicon verification. In RTL level, validation is easier to do with extensive tool support. On the other hand netlist validation(functional validation) is mainly done by simulation. Performance for gate level netlist simulation is extremely low; typically it takes days to validate a particular design.On the other hand, netlist simulation performance is quite significant since short times-to-market limit the coverage that can be achieved in verification. Thus, faster verification methods are needed to improve coverage. Any improvement in any phase of the validation can improve the overall performance of the validation flow. In this context, GP-GPUs can deliver faster gate level simulation by exploiting the parallelism.

0.1.1 Project Description & Prerequisites

As my senior project, it is given to implement event-driven gate level simulation on GP-GPUs using CUDA. Having little experience on event-driven simulation, digital design implementation and CUDA makes the project a self-learning task as well as an implementation task.I subdivided the project into the learning and implementation tasks. Even if the learning and implementation tasks can be interleaved to some extent, I've chosen to minimize interleaving of them to avoid major design and problems and wrong and costly decisions since I'm new to this field. To complete the learning task and to start to the implementation, I must:

- learn and practice CUDA,
- revise core digital design topics including logic components,
- revise core simulation topics with special focus on event-driven simulation,

- learn how event-driven simulation is applied in logic level circuitry.

Learning and Practicing CUDA

CUDA stands for *Compute Unified Device Architecture*. It is a parallel programming platform and a programming model created by NVIDIA. It enables programmers to develop programs using massively parallel architecture of GPUs.

Not surprisingly, GPU programming didn't start with CUDA. Computer graphics developers were(and still they are) programming for GPUs using so-called shader languages such as GLSL. The main problem in this approach is that shading languages reside natively in the computer graphics domain.So anyone interested in GPU programming must have learnt at least some computer graphics terminology before developing programs that run on GPU. CUDA solved this problem by presenting a domain agnostic way to program GPUs. CUDA offers a subset of C language and some additional tags like *__device__* to solve any programming task that requires parallelism. It doesn't require any more than mediocre development experience in any C-like language. Currently all GPUs of NVIDIA on the market supports CUDA. Since these GPUs have quite different specifications and they have released in different years, CUDA feature capabilities are rated by an index called *CUDA Compute Capability*. For example, devices with compute capability 2.0 supports atomic floating point operations. CUDA is still under development, currently version 5.0 is released which presents a subset of C++ class capabilities for GPU computing.

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

- **Scattered reads**, code can read from arbitrary addresses in memory
- **Shared memory**, CUDA exposes a fast shared memory region (up to 48KB per Multi-Processor) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups

But it also has disadvantages to consider:

- Texture rendering is not supported (CUDA 3.2 and up addresses this by introducing "surface writes" to CUDA arrays, the underlying opaque data structure).
- Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency (this can be partly alleviated with asynchronous memory transfers, handled by the GPU's DMA engine)
- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task.
- Unlike OpenCL(an alternative to CUDA), CUDA-enabled GPUs are only available from Nvidia
- Valid C/C++ may sometimes be flagged and prevent compilation due to optimization techniques the compiler is required to employ to use limited resources.
- Double precision (CUDA compute capability 1.3 and above) deviate from the IEEE 754 standard: round-to-nearest-even is the only supported rounding mode for reciprocal, division, and square root. In single precision, denormals and signalling NaNs are not supported; only two IEEE rounding modes are supported (chop and round-to-nearest even), and those are specified on a per-instruction basis rather than in a control word; and the precision of division/square root is slightly lower than single precision.

Since I'm quite experienced with C language, I started learning CUDA as soon as the project overview is presented. For learning CUDA, I started reading and completed the book *CUDA by Example* in two weeks. The book is structured in a way that each chapter covers a fundamental feature of CUDA in addition to many core concepts of parallel computing. I implemented the examples presented in the book rather than merely reading the

text to get even more familiar and fast with CUDA programming.In addition to this book, I watched the webinars in CUDA Learning Zone in NVIDIA website. These webinars offer a jump-start point for developers with past experience on sequential computing.

At the moment that this report is written, I completed any learning task associated with CUDA. I've enabled myself to develop applications using CUDA with techniques and algorithms exploiting parallelism which hopefully results in improved performance.

Revision of Digital Design Topics

Computer Engineering Undergraduate Program in Bogazici University contains a compulsory digital design course in sophomore year, namely CMPE240 Digital Systems. The textbook of this course *Frank Vahid, Digital Design, Wiley, 2011.* was a solid source of information for me at the time that I'd taken the course.

To revise digital design topics, I've recently read very first chapters of the book and solved some of the exercises at the end of each chapter. Yet I might further consult the book in future if I need more information about digital design and logic components. To indicate specifically, a future student or researcher with the same topic of this project must cover *hazards* and related topics essentially to excel in gate level simulation.

0.1.2 Revision of Simulation Topics

Computer engineering curriculum contains a system simulation course from industrial engineering department, namely IE306 Systems Simulation. The course aims to make students familiar with simulation systems with different approaches, indicating pros and cons of each of the techniques. Mainly discrete event simulation is taught and practiced in projects of IE306, so it suits my intentions.

To revise simulation topics, I consulted presentations available in the course website. They cover fundamentals of simulation topics in a paced manner. Yet it makes no more than an introduction to simulation systems. Extensive knowledge from past courses could be very beneficial for this project.

0.1.3 Learning Event-Driven Simulation in Gate Level

For this learning task, the professor guided me to Peter.M.Maurer's unpublished chapters on Design Automation: Logic Simulation. Chapters of the book covers simulation concepts and target common issues:

- A Review of logic design
- Levelized simulation
- Event-driven simulation
- Multi-delay simulation
- The PC-Set method
- The Parallel Technique
- The Inversion Algorithm

Starting from levelized simulation, it presents event driven simulation in gate level with various techniques such as shadow algorithm. It touches implementation of delay in logic simulation, also gives extensible information about simulation in parallel.

0.2 Gate Level Simulation

Simulating individual AND, OR, and NOT gates is simple. First of all, allocate an integer for each net. Then keep the value of the net in the low-order bit of the integer. Finally, use bit-level AND, OR and NOT operators to perform the simulations. This method can be extended to networks of gates, but some care is necessary for handling memory components. Yet, The strategy here is to simulate every gate for every time-step. This is wasteful because if the inputs of a gate don't change, then the output doesn't change, either.Avoiding simulating those gates whose inputs do not change can result in improvements in terms of speed. However, a very straightforward approach of continuously checking if the inputs are changed or not for each of the gates, will not help, since testing the inputs presents additional workload to the simulation of a gate.

Event-Driven Simulation is purposed to eliminate unnecessary gate simulations without introducing an unacceptable amount of additional testing.

It is based on the idea of an event, which is a change in the value of a net. In a typcial event driven simulation, each events are represented as a data structure, and these data structures acts as a trigger the simulation of gates and the creation of other events.

During simulation, events trigger gate simulations and gate simulations produce events. If there is no new events then no gates will be simulated. The initial set of events is created by comparing each bit of an input vector with the corresponding bit in the previous input vector. An event is created for each pair of bits that is different. Thereafter, nets are tested for changes only after a gate simulation. When an event is processed, any gates that use the net as an input will be scheduled for simulation. The order in which gate simulations are performed cannot be predicted ahead of time, so dynamic queues are used to schedule both event processing and gate simulation. When an event is detected, an event structure will be created and stored in the Event Queue for future processing. Event processing continues until all events have been processed and removed from the event Queue. At this point, there will usually be several gates in the gate queue. Once all events have been processed, it is necessary to simulate each gate in the gate queue, test outputs for changes, and schedule events. An alternative approach is to eliminate the gate queue or the event queue and use a single scheduling queue. Such an approach is called Single-List or One-List scheduling to contrast it with Two-List scheduling. Event-driven simulation presents by default a unit delay timing strategy yet multi-delay strategies can be adopted by using techniques like time wheel.

0.3 Related Work on Gate Level Simulation

Research on logic simulators gain importance and interest when the concepts of circuit netlist compilation, oblivious and event-driven simulation were first discovered. Particularly,[2] provides an analysis of early attempts to parallelize event-driven simulation by dividing the processing of individual events across multiple machines with fine granularity. But fine granularity comes with a high communication overhead and, depending on the solution, the issue of deadlock avoidance needs sophisticated event handling. There are also parallel algorithms for event-driven simulation for distributed systems [16, 15] and multiprocessors [12]. In these solutions, threads run on separate netlist clusters and communication is done with an event-driven fashion.

Today, several commercial simulators building on these concepts are available: they execute on a single CPU and adopt aggressive compiled-code optimization techniques to boost their performance. In addition, specialized hardware solutions (emulation systems) have also been implemented to boost simulation performance. Modern emulators can deliver 3-4 orders of magnitude speedup and they can handle very large designs. However, their cost is high and the process of successfully mapping a netlist to an emulator can take up to few months. Most recently, a few research solutions have been presented to run simulations on GPUs: an early attempt by Perinkulam [20] fails because of not providing performance benefits due to lack of general purpose programming primitives (CUDA like language, data transfer overhead etc.) for their platform and the high communication overhead. An oblivious simulator solution can be seen in [8].Yet expectedly, the size of the circuits that can be simulated with the solution in [8] is severely limited by the size of the shared memory in the GPU platform. [chatterjee] introduces macro-gate concept and proposes a solution targets fast simulation of complex designs which circuit partitioning and optimizations techniques in order to enhance the parallelism of the target platform. [sen] provides a similar Cycle-based simulation solution to chatterjee yet using And-Inverter Graphs.

0.4 Decisions for implementation phase

At the time that this report is written, I pretty much covered enough of simulation topics and started the core parts of the implementation. This early stage of implementation is omitted for this document but it will be covered extensively in the final report. With my supervisor's guiding, we decided on implementing a multi-delay event-driven simulation engine which must simulate networks containing at least hundred thousands of gates.

Data structures and programming techniques will be revealed in time. Yet, we can probably say that exploiting spatial locality is the first priority to come up with a successful implementation. Deciding on single-queue or double-queue implementation is a bit trivial, since both techniques are easy to implement and they trade performance between memory and computation. I have started developing a double-queue model. After it will become mature enough, I will implement the single queue model and consider my options. Input output formats are nearly out of discussion and out of topic because developing a working simulator with good performance is the main goal.

However, for simulation and finalization of the project, necessary input and output mechanisms will be implemented, tested, and used.

0.5 Conclusion

Gate level simulation and CUDA programming is out of curriculum for undergraduates in Computer Engineering department. So any project on these topics need extensive additional work and study hours to catch up deadlines. In the first phase of the project,

- I've learnt CUDA and gained enough experience for developing the required assets of the project.
- I've reviewed my digital design knowledge and become inclined on the area.
- I've learnt gate level simulation fundamentals which is required for successfully completing the project.
- With my supervisor, we decided on the specifications of the final deliverable. Decisions are well-justified for now, yet they are subject to change in case of a misjudgment in a particular aspect of the project, especially for the next stage of implementation.