Event-Driven Gate-Level Simulation with GP-GPUs

Debapriya Chatterjee, Andrew DeOrio and Valeria Bertacco
Department of Computer Science and Engineering, University of Michigan
{dchatt, awdeorio, valeria}@umich.edu

ABSTRACT

Logic simulation is a critical component of the design tool flow in modern hardware development efforts. It is used widely – from high-level descriptions down to gate-level ones – to validate several aspects of the design, particularly functional correctness. Despite development houses investing vast resources in the simulation task, particularly at the gate-level, it is still far from achieving the performance demands required to validate complex modern designs.

In this work, we propose the first event-driven logic simulator accelerated by a parallel, general purpose graphics processor (GP-GPU). Our simulator leverages a gate-level event-driven design to exploit the benefits of the low switching activity that is typical of large hardware designs. We developed novel algorithms for circuit netlist partitioning and optimized for a highly-parallel GP-GPU host. Moreover, our flow is structured to extract the best simulation performance from the target hardware platform. We found that our experimental prototype could handle large, industrial scale designs comprised of millions of gates and deliver a 13x speedup on average over current commercial event-driven simulators.

Categories and Subject Descriptors. B.6.3 [**Logic Design**]: Design Aids—*Simulation*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel Processors*

General Terms. Verification, Performance

Keywords. Gate-level simulation, High-performance simulation, General Purpose Graphics Processing Unit(GP-GPU)

1. INTRODUCTION

Logic simulation is the validation workhorse of modern digital designs. It is used to verify designs at the behavioral level, as well as the structural level, ensuring that a synthesized circuit's netlist matches the functionality and timing of the behavioral model. Structural netlists are particularly cumbersome for simulation because of their low-level specification and the fine granularity of the structural definition, which consists of gate primitives in the target technology library. It is typical for design houses to invest the computational power of large simulation "farms" to complete as many simulation cycles as possible before final design tapeout. However, even with such investment in today's development efforts, large portions of a design go unverified. The result is unforeseen bugs that are released into silicon, which may have drastic impacts, ranging from silicon respins to market recalls.

The root cause of this situation lies in the vast complexity of modern designs (several million gates) and the fact that the performance of commercial logic simulators is inversely proportional to their size. In addition, the technology in commercial logic simulators today is fairly mature, thus their performance improvement between subsequent releases largely relies on the performance trends of the underlying simulating hardware host.

DAC 2009, July 26 - 31, 2009, San Francisco, California, USA.

Modern gate-level simulators proceed in two phases: during the first phase, the circuit netlist to be simulated is restructured and optimized (compilation phase); in the second phase, the netlist is simulated ("executed") using the input stimuli specified in the testbench. The performance of the simulator is driven by this second phase, since the compilation step is only required once per netlist. In this work, we propose a novel simulator design that leverages the high-performance of general purpose graphics processing units (GP-GPUs) for the execution phase of the simulation, leading to a major improvement in simulation performance. During the compilation phase, a netlist is "levelized", that is, gates are organized into levels so that all the gates in one level depend only on simulation values generated in previous levels. Thus, gates in a same level are not directly connected and can be simulated in parallel, an advantage that can be leveraged when many parallel processing units are available, as in GPUs. During the execution phase, gates are simulated by level; however, in an *event-driven* simulation, a gate is simulated only if at least one of its input values had changed, while in an *oblivious* simulator all gates are evaluated with each cycle. While oblivious simulation has the advantage of simple, efficient static gate scheduling, event-driven simulation has been noted to perform better in practice. This is because it is typical for large designs to only simulate a small fraction of the gates (1 to 10%) during any given cycle. Thus, even in face of a more complex dynamic scheduling architecture, most commercial simulators rely on an event-driven approach for performance reasons.

The recent availability of general purpose computing programming models for high-performance and highly parallel GPUs led us to explore a new simulation architecture targeting these hardware platforms, with the hope of delivering a conspicuous performance advantage at a small hardware cost (that of a GPU peripheral). Specifically, the NVIDIA's CUDA architecture provides a programming interface that enables users to develop software applications for their vastly parallel co-processor GPU. However, CUDA exposes its parallel architecture directly to the programmer, with the result that applications must be designed specifically for this structure in order to derive benefit from it.

1.1 Contributions

In this work, we present the first event-driven GPU-based logic simulator, which leverages GPUs' massive parallelism to achieve large performance speedups over commercial logic simulators. Our solution leverages a novel *macro-gate* segmentation algorithm, designed specifically to benefit from the CUDA architecture. A macro-gate comprises several gates of the original netlist connected to each other. The macro-gates generated cover the entire circuit's netlist; they are compiled into a suitable data structure and transferred to the GPU's memory. During simulation, those macro-gates that require simulation because their input values have changed, are tagged for execution and handed over to the CUDA's low-level scheduler. We developed a prototype of our simulator and applied it to a range of designs, including a SPARC multiprocessor of more

than a million gates. We developed several testbench infrastructures, from random generators running on the GPU, to assembly programs for processor designs; and simulated the designs for millions of cycles. We found that our GPU-based simulator delivers performance speedups from 4 to 44 times over the performance of a top-end commercial simulator, with 13 times being the average.

2. RELATED WORK

Research on logic simulators bloomed in the 1980s, when the concepts of circuit netlist compilation, oblivious and event-driven simulation were first explored [6, 3, 14, 2]. In particular, [2] provides a comparative analysis of early attempts to parallelize event-driven simulation by dividing the processing of individual events across multiple machines with fine granularity. This fine granularity would generate a high communication overhead and, depending on the solution, the issue of deadlock avoidance could require specialized event handling. Parallel logic simulation algorithms were also proposed for distributed systems [16, 15] and multiprocessors [12]. In these solutions, individual execution threads would operate on distinct netlist clusters and communicate in an event-driven fashion, with a thread being activated if switching activity was observed at the inputs of its netlist cluster. Both conservative [7, 17, 10] and speculative techniques, such as time warp [5, 4], were proposed to handle synchronization in these discrete event algorithms. 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 explored to boost simulation performance. These systems typically consist of several identical hardware units connected together, with units optimized for the simulation of small logic blocks. To emulate a circuit netlist, a "compiler" partitions the netlist into blocks and then loads each block into separate units [9, 1, 13]. Modern emulators can deliver 3-4 orders of magnitude speedup and they can handle very large designs. However, their cost is prohibitively large 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 proposed to run simulations on GPUs: a first attempt by Perinkulam [20] did not provide performance benefits due to lack of general purpose programming primitives for their platform and the high communication overhead generated by their solution. An oblivious simulator solution was proposed in [8]: their software design is simpler, and can be optimized statically, but simulating all gates in each cycle limits the performance of this approach. Moreover, 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. Another recent solution in this space [11] introduces parallel fault simulation on a CUDA GPU target. It derives its parallelism by simulating distinct fault patterns on distinct processing units, with no partitioning within individual simulations or the design. In contrast, we target fast simulation of complex designs, thus we must explore circuit partitioning and optimizations techniques in order to leverage the parallelism of the target platform. Moreover, we optimize the performance of individual simulation runs, in contrast with [11], which optimizes over all faults simulations.

3. INTRODUCTION TO CUDA

The architecture of modern graphic processing units (GPUs) comprises a large number of data streaming processing units. They are commonly fairly simple, programmable and together can execute an astonishing amount of floating point (or integer) instructions in parallel. Typically, GPUs can be programmed via a graphics library interface, however NVIDIA has made available a general purpose programming interface for their CUDA platform (Compute Unified Device Architecture [18]), enabling the development of a broader range of applications. A CUDA GPU consists of a set of multiprocessors (Figure 1, each comprising several functional units (FUs), which can execute multiple program threads concurrently (up to 512). Threads are organized into blocks; with one or more blocks in concurrent execution on individual multiprocessors. All threads running in a multiprocessor have fast access (1 cycle) to a small shared local memory (16KB), and also to a much larger device memory (up to 1GB - 300-400 cycles latency for access). The CUDA architecture can be programmed using C language extension in a SIMD (single-instruction-multiple-data) fashion: all FUs across the entire GPU must be executing the same code, operating on different data. Finally, data placement to shared or device memory must be handled explicitly by the programmer. When executing a program on CUDA, also called a *kernel*, the host computer uploads the data and compiled program (in our case the netlist and simulation code) to the GPU's device memory, and then relinquishes control to the GPU scheduler, which executes all required threads autonomously until simulation completes.

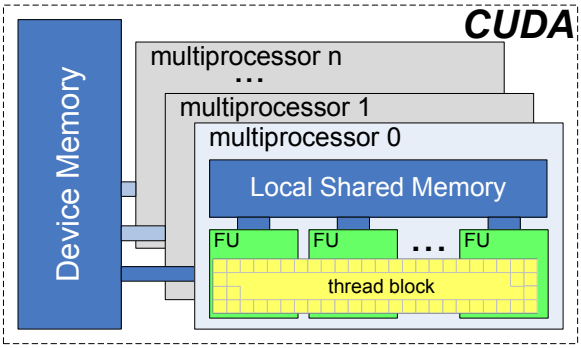

Figure 1: NVIDIA CUDA architecture. A GPU includes a number of multiprocessors, each comprising 8 execution units. Several threads (up to 512) may execute concurrently within a multiprocessor and communicate through a small shared memory bank (16KB). The larger device memory has much higher access latency.

4. OVERVIEW

Our event-driven CUDA-accelerated simulator first applies a *compilation phase*, during which it transforms the netlist to leverage the raw performance of the target architecture. This is followed by a *simulation phase* where the compiled netlist is uploaded to the GPU co-processor and one or more simulations may be executed with different input testbenches.

The compilation phase is responsible for segmenting a large monolithic netlist into blocks amenable to simulation by individual execution units within the GPU. This requires segmenting the netlist into *macro-gates*: a set of several connected gates within the netlist of ideal size, optimizing the logic within each macro-gate, and finally producing the data structures and the CUDA programs necessary to carry out the simulation. During simulation, both program and data reside on the GPU. Testbenches can be implemented using many different solutions; if they are encoded in a CUDA program (possibly with associated stimuli data), then the simulation can be completely offloaded from the host with direct performance benefits. If the testbench resides on the host, control alternates between host and GPU to simulate and generate stimuli.

4.1 Netlist generation

The first step of compilation considers a digital design and synthesizes it to a flattened netlist using a target technology library (we

used the GTECH library by Synopsys for our experiments). If the design is a gate-level description (as in the case of synthesis validation), the synthesis step may be unnecessary. Finally, the combinational portion of the netlist is extracted for further processing, while the storage elements will be mapped to memory during simulation. Note that in our implementation, we excluded tri-state buffer and latches from the synthesis library to obtain a simple synchronous netlist. Latches could be easily included by adapting our simulator to operate at a finer granularity, that is, time units instead of clock cycles. Tri-state elements can be included by using 4-valued logic instead of binary. Both of these are straightforward extensions to the simulator. The combinational netlist is finally *levelized*, that is, logic gates are organized into levels, so that the fanin of all gates in one level is computed in previous levels. With this organization, it is possible to simulate the entire netlist one level at a time, from inputs to outputs, with no backward dependency. In our prototype implementation, we used an ALAP (as-late-as-possible) levelization, though other solutions are also possible.

4.2 Segmentation into macro-gates

To exploit the parallelism available in the GPU, we must segment the gate-level netlist into several logic blocks (called *macro-gates*), and assign the simulation of each macro-gate to a distinct CUDA multiprocessor. During simulation, we maintain a *sensitivity list* of nets at the inputs of each macro-gate: if any net in a sensitivity list changes value, then the corresponding macro-gate will be affected by the change and must be simulated (*i.e.activated*). Otherwise, the macro-gate can be skipped during the current cycle.

In determining how to partition the netlist into macro-gates, we took into consideration several factors: (i) the time required to simulate a macro-gate should be greater than overhead of determining which macro-gates to simulate; (ii) CUDA's multiprocessors can only communicate through device memory, thus macro-gates should not share data. To this end, we occasionally duplicate small portions of logic, so that each macro-gate can compute the value of its outputs independent of other concurrent macro-gates. Finally, (iii) we want to avoid cyclic dependencies between macro-gates, so to simulate each macro-gate at most once per cycle.

To address the constraint list, we segment the netlist by partitioning the netlist into *layers*: each layer encompasses a fixed number of the netlist's levels. Macro-gates are then defined by selecting a set of nets at the top boundary of a layer, and including its cone of influence back to the input nets of the layer. The number of levels within each layer is called the *gap* and corresponds to the height of the macro-gate. By using this procedure, it is possible that a given logic gate is assigned to two or more macro-gates. In this case, we duplicate it, so that each macro-gate can compute the value of its output nets without sharing any data with other macro-gates (second requirement). Finally the number of output nets used to generate each macro-gate is a variable parameter (called *lid*), whose value is selected so that the number of logic gates in all macro-gates is approximately the same. Figure 2 shows a schematic of the segmentation technique, while figure 3 presents the pseudo-code of the algorithm. The set of nets that cross the boundary between each pair of layers is monitored during simulation to determine which macro-gates should be activated.

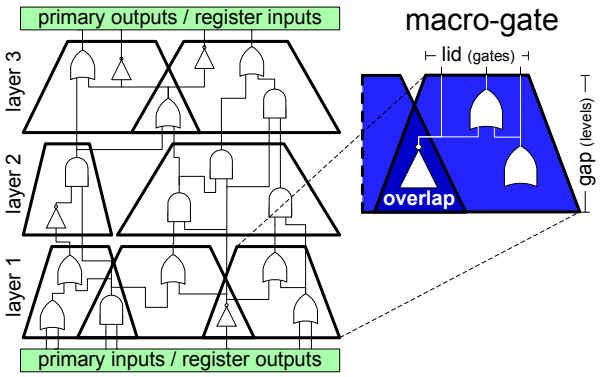

Figure 2: Segmentation topology. The levelized netlist is partitioned into layers, each encompassing a fixed number of levels (gap). Macro-gates are then carved out by extracting the transitive fanin from a set of nets (lid) at the output of a layer, back to the layer's input. If an overlap occurs, the gates involved are duplicated to all associated macro-gates.

Section 5 discusses the process that we used to select optimal values for gap and lid, so as to achieve a high-level of parallelism during simulation with little macro-gate overlap and low activation rates. Note that, in our prototype implementation, we fixed gap and lid across the entire netlist: additional performance could be achieved if each layer had its own associated gap and each macro-gate had an associated lid.

```
segmentation (netlist, gap, lid) {
  levelized_netlist = ALAP_schedule(netlist)
  layers = gap_partition(levelized_netlist)
  for (layer in layers) {
      macro-gates = lid_partition(layer)
      macro-gates_pool = append(macro-gates);
      compute_monitored_nets (layer);
  }
  return macro-gates_pool }
```

Figure 3: Macro-gate segmentation algorithm. The levelized netlist is partitioned into layers: several macro-gates are carved from each layer and appended to the macro-gates pool to be simulated. The nets to be monitored are also tagged at this stage.

4.3 Macro-gate balancing

Each macro-gate is designed to be simulated in a single CUDA multiprocessor. Because our lowest-level primitives are basic logic gates, we designed our CUDA simulation program so that the execution threads simulate all the gates in the same level, then move on to the next level, and so on, until an entire macro-gate has been simulated. Thus the gap is directly proportional to layer simulation performance. However, the segmentation procedure tends to generate macro-gates with a large base (many gates) and a narrow tip. Correspondingly, we have many active threads in the lower levels, and just a few in the top levels.

To maximize concurrency throughout the simulation, we optimize each macro-gate individually with a *balancing* step, as outlined in the schematic of Figure 4. This is the last step of the compilation phase: it exploits the slack available in the levelization within each macro-gate and restructures macro-gates to have approximately the same number of logic gates in each level. As a result, a smaller number of threads will be required to simulate the base of the macro-gate. Note that it is always possible to "shrink" the size of the base, at the price of an increased gap.

4.4 Simulation phase

As mentioned earlier in this section, simulation is carried out directly on the GPU co-processor. Each multiprocessor is responsible for the simulation of one or more macro-gates. Each macro-gate corresponds to one thread block. In determining the number of macro-gates that should be simulated concurrently on a multiprocessor, the number of concurrent thread blocks allowed in a multiprocessor (3), was the limiting factor. A single allocation would enable larger macro-gates, however, mapping several smaller ones

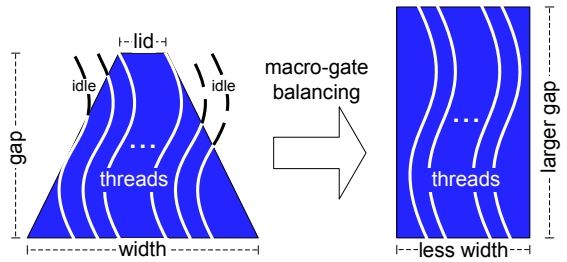

Figure 4: Macro-gate balancing. The balancing algorithm exploits the levelization slack within a macro-gate to restructure it so that fewer execution threads are required to simulate the lower levels, and idle threads are minimized at the top levels.

concurrently allows us to hide the memory latency in retrieving structural netlist data from device memory. We found experimentally that the latter solution provides better performance.

The overall simulation alternates executing all active macro-gates in a layer, with analyzing the corresponding monitored nets to determine which macro-gates should be activated for the next layer. The CUDA scheduler is responsible for assigning activated macro-gates to individual multiprocessors. Figure 5 illustrates the layered structure of macro-gates and monitored nets. It also shows how activated macro-gates are transferred from the pool to a multiprocessor for execution. Within a macro-gate simulation, multiple concurrent threads simulate all the gates in same level, then synchronize, and finally advance to the next level, until completion.

Data placement is organized as follows: primary inputs, outputs, register values and monitored nets are mapped to device memory, since they must be shared among several macro-gates (multiprocessors). Truth tables for the gates in the technology library are mapped to shared memory because of their frequent access. In addition, intermediate net values generated within a macro-gate are also placed in shared memory. Finally, the netlist structure is stored in device memory and accessed during each macro-gate simulation.

5. OPTIMIZATIONS

5.1 Macro-gate sizing and activation

In segmenting a netlist into macro-gates, the selection of *gap* and *lid* values have critical impact on the simulation performance (see also Section 4.2). During the compilation phase, we select these values by evaluating a range of solution points; for each candidate value we collect several metrics: number of macro-gates, number of monitored nets, size of macro-gates and activation rate. The activation rate is obtained by a mock-up of the simulation on a micro testbench. We then select the locally optimal values and perform

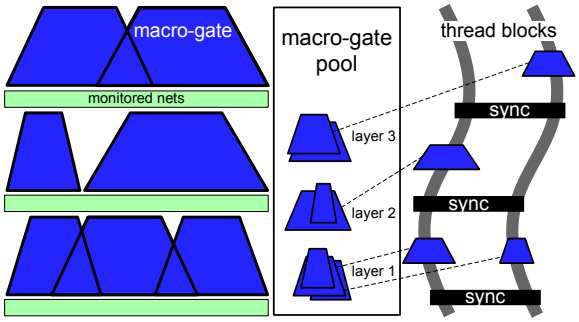

Figure 5: The event-driven simulation operates by layer. Within each layer, it simulates activated macro-gates and then analyzes the monitored nets to tag additional macro-gates for activation. Activated macro-gates are transferred by the CUDA scheduler to an available multiprocessor for simulation.

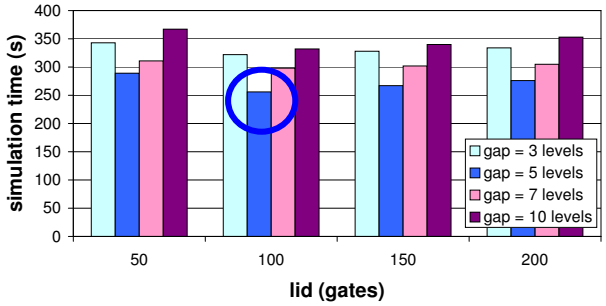

Figure 6: Estimation of optimal gap and lid for the LDPC testbench design. We run a mock simulation with a micro testbench using a range of gap and lid values and found that optimal performance is achieved for gap=5 and lid=100.

detailed segmentation. Figure 6 shows an example of this selection. The chart reports the simulation times for the LDPC benchmark design when running the micro-testbench: each bar corresponds to a unique <gap,lid> value pair. In this example the best estimates are 5 for gap and 100 for lid. The boundaries for the range of gap values considered are derived from the number of monitored nets generated: we only consider gap values for which no more than 50% of the total nets are monitored. In practice, small gap values tend to generate many monitored nets, while large gap values trigger high activation rates. For lid determination, we bound the analysis by estimating how many macro-gates will be created at each layer: We strive to run all the macro-gates concurrently. The GPU used for our evaluation included 14 multiprocessors and the CUDA scheduler allows at most three thread blocks in concurrent execution on a same multiprocessor. Thus we only consider lid values that generate no more than $14 \cdot 3 = 42$ macro-gates per layer. Note that this analysis is performed only once per compilation.

After the simulation of all active macro-gates in a layer is completed, the GPU executes a *scheduling kernel* that evaluates the array of monitored nets to determine which macro-gates should be activated in the next layer. This array is organized as a bit vector, with each monitored net being implicitly mapped to a unique location in the array. If a macro-gate simulation modifies the value of any of these nets, its corresponding location is tagged. Each macro-gate has a corresponding *sensitivity list* where all the input nets triggering its activation are tagged. With this structure, a simple bit-wise AND operation between the monitored nets array and a macro-gate's sensitivity list determines if any input change has occurred and the macro-gate should be activated. The alternative of maintaining the sensitivity lists as linked lists within the monitored nets array would require variable size data structures, which are extremely cumbersome to manage in a GP-GPU architecture.

5.2 CUDA-specific optimizations

We also explored a few optimizations that are specific of the CUDA architecture. For instance, CUDA has an additional memory block, called *texture memory* that can be used as an intermediary to access device memory. The texture memory controller operates by conglomerating adjacent memory accesses and sending block requests to device memory. We leveraged this memory when retrieving the netlist structure of a macro-gate during simulation: since gates in a same level are placed in contiguous locations in device memory, the access through texture memory could bypass most of the latency for these data.

6. EXPERIMENTAL RESULTS

We evaluated the performance of our simulator on a broad set of designs ranging from purely combinational circuits such as an

Design	Testbench	# Gates	# Flops
Alpha no pipeline	recursive Fibonacci program	17546	2795
Alpha pipeline	recursive Fibonacci program	18222	2804
LDPC encoder	random stimulus	62515	0
JPEG decompressor	1920x1080 image	93278	20741
3x3 NoC routers	random legal traffic	64432	13698
4x4 NoC routers	random legal traffic	144098	23875
OpenSPARC core	OpenSPARC regression suite	262201	62001
OpenSPARC-2 cores	OpenSPARC regression suite	610670	124002
OpenSPARC-4 cores	OpenSPARC regression suite	1221340	248004

Table 1: Testbench designs for evaluation of the simulator.

LDPC encoder, to a multicore SPARC design containing over 1 million logic gates. Designs were obtained from OpenCores [19] and from the Sun OpenSPARC project [21]; the Alpha processors and NoC designs were developed in advanced digital design courses by student teams at the University of Michigan.

We report in Table 1 the key aspects of these designs: number of gates, flip-flops and type of stimulus that was used during simulation. The first two designs are processors implementing the Alpha instruction set, the first can execute one instruction at a time, while the second has a 5-stage pipelined architecture. Both were simulated executing a binary program that computed Fibonacci series recursively. The LDPC encoder outputs an encoded version of its input; for this design we developed a random stimulus generator that run directly on the GPU platform. The JPEG decompressor would decode an input image. The NoC designs consist of a network of 5-channel routers connected in a torus network and simulated with a random stimulus generator sending legal packets through the network. Finally, the OpenSPARC designs use processors from the OpenSPARC T1 multi-core chip (excluding caches) and run a conglomeration of assembly regressions provided with Sun's open source distribution. We built several versions of this processor: single-core, two cores, and four cores and we simulated local cache activity by using playback of pre-recorded signal traces from processor-crossbar and processor-cache interactions.

6.1 Macro-gates

We studied several aspects of the compilation phase of our simulators and report here our findings. Figure 7 shows the total number of macro-gates generated for each design when using the gap and lid values determined in Section 5.1. On average each macro-gate includes 400 logic gates. In addition, we indicate the number of layers used in the segmentation of each design. Note that the largest design include many more macro-gates in each layer that could be simulated concurrently(42 as per section 5.1).

As mentioned in Section 4.2, gate duplication is a necessary consequence of the high communication latency between multiprocessors. However, we strive to keep duplication low, so not to inflate the number of simulated gates during each cycle. Figure 8 plots the fraction of gates that were duplicated, averaged over all our experimental designs: more than 80% incurred no duplication, less than

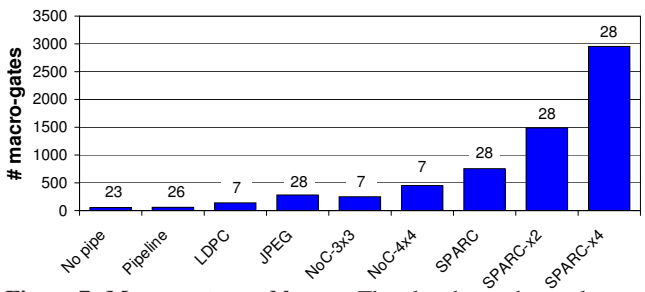

Figure 7: Macro-gates and layers. The plot shows the total number of macro-gates for each design. The value above each bar indicates the number of layers in the segmentation.

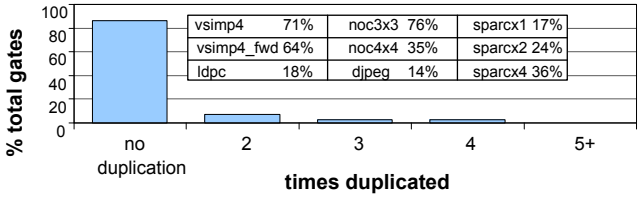

Figure 8: Gate duplication due to macro-gate overlap. The graph reports the number of times that gates were duplicated. The overlapping table indicates the gate inflation that each design incurred as a result of duplication.

10% were duplicated once, very few incurred more than one duplication. The table reports the overall rate of "gate inflation" in each design, resulting in an overall average of 39%.

6.2 Monitored nets

The number of monitored nets has a high impact on the simulator performance thus segmentation strives to keep the fraction of nets that are monitored low. As an example, in Figure 9 we plot the structure of the LDPC encoder design after segmentation: for each layer, we plot the corresponding number of macro-gates and monitored nets. Note how middle layers have more macro gates and how lower layers tend to generate the most monitored nets. Finally, we analyzed the fraction of total nets in the design that require monitoring because they cross layer boundaries. The compilation phase should strive to keep this fraction low, since it is directly related to the size of the sensitivity list that must be checked when evaluating a macro-gate for possible activation. Figure 10 reports our findings for experimental testbench designs after the segmentation phase.

6.3 Macro-gate activity

The activation rate of macro-gates is an important metric for event-driven simulation (an oblivious simulator has an activation rate of 100% on any design). The goal of an event-driven simulator is to keep this rate as low as possible, thus leveraging the fact that not all gates in a netlist switch on every cycle. Figure 11 reports the macro-gate activation rates for a number of our designs. Plots show the cumulative distribution of activation rates among all the macro-gates for distinct designs. Note how, for most designs, the majority of the macro-gates have an activation rate between 10 and 30% only. However, for LDPC, most macro-gates experience a high activation rate ($> 80\%$): this is due to the inherent nature of this design. The designs that are not reported had a cumulative distribution similar to that of the OpenSPARC and NoC designs. Note that activation rate in our solution does not directly relate to performance gain over oblivious simulation. As an example, the JPEG decoder has an average activation rate of 40%. This does not mean that, on average, the JPEG decoder is simulated 2.5 times faster when compared to an oblivious simulation. Indeed, even if

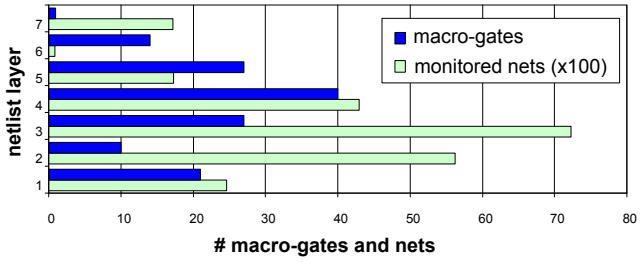

Figure 9: Segmented structure for LDPC encoder design. The plot shows the geometry of the LDPC encoder after segmentation. For each layer we report the number of macro-gates and of monitored nets in hundreds.

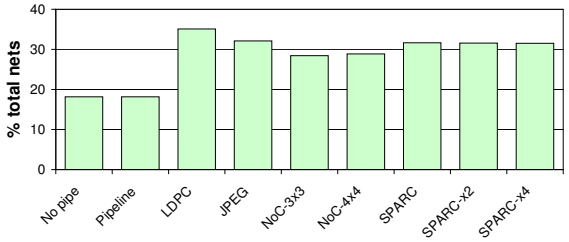

Figure 10: Fraction of monitored nets. Percentage of all nets that are monitored for each testbench design

very few macro-gates are activated, one for each layer, the performance would be just the same as if several macro-gates were simulated in each layer. This is because the parallel processing units can hide much of the additional computation required when activating many macro-gates, while the synchronization barriers force macro-gates to be simulated in layer-order. The overall performance of the JPEG design in event-driven simulation is 1.55 times faster than in oblivious simulation.

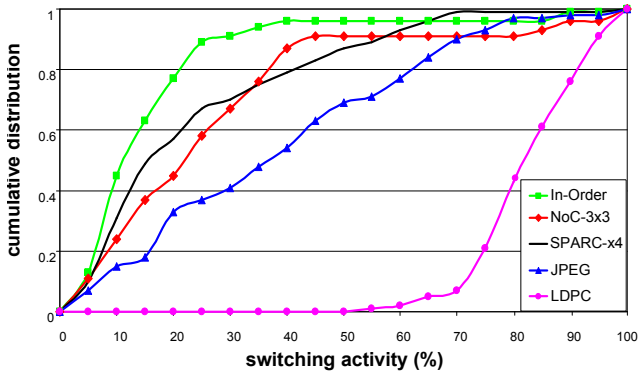

Figure 11: Cumulative distribution of macro-gates w.r.t. to activation rate. The plots show which fraction of macro-gates have an activation rate below a threshold indicated on the x axis. Most designs have very low activation rate (<30%).

6.4 Performance Evaluation

Finally, we evaluated the performance of our prototype event-driven simulator against that of a commercial, event-driven sequential simulator. Our graphics coprocessor was a CUDA-enabled 8800GT GPU with 14 multiprocessors and 512MB of device memory, operating at 600 MHz for the cores and 900MHz for the memory. The current implementation has 83% occupancy and achieves a bandwidth of 20.4 GB/s. The commercial simulator was run on a 2.4 GHz Intel Core 2 Quad running RH-EL5, enabling 4 parallel simulation threads. For each design, Table 2 reports the number of cycles simulated, the runtimes in seconds for both the GPU-based simulator and the commercial simulator (compilation times are excluded), and the relative speedup. Note that our prototype simulator outperforms the commercial simulator by 4 to 44 times. Despite the LDPC encoder having a very high activation rate, we report the best speedup for this design. As mentioned before, most gates in this design are switching in each cycle: this affects our activation rates, but hampers the sequential simulator performance. Thus, the speedup obtained is due to sheer parallelism of our architecture.

7. CONCLUSIONS

In this work, we have presented a novel event-driven simulator architecture that leverages the high-level of parallelism of general purpose GPUs. By extracting parallelism in the simulation of gate-level netlists, we are able to realize a 13 times speedup over traditional sequential simulators, on average. Our simulator carves out

design	sim cycles	seq sim(s)	GPU sim(s)	speed up
Alpha no pipeline	12,889,495	31,678	2,567	**12.15x**
Alpha pipeline	13,423,608	54,789	7,781	**7.04x**
LDPC encoder	1,000,000	115,671	2,578	**44.87x**
	10,000,000	>48h	25,973	**43.49x**
JPEG decompressor	2,983,674	12,146	599	**20.28x**
3x3 NoC routers	1,967,155	3,532	397	**8.90x**
4x4 NoC routers	10,000,001	28,867	3,935	**7.34x**
sparc core x1	1,074,702	27,894	6,077	**4.59x**
sparc core x2	1,074,702	40,378	8,229	**4.91x**
sparc core x4	1,074,702	61,678	10,983	**5.62x**

Table 2: GP-GPU simulator performance. Performance comparison between our CUDA-based event-driven simulator and a commercial event-driven simulator. Our prototype simulator outperforms the commercial simulator by 13 times on average.

macro-gates from the structural netlist of a design and schedules them for simulation on the multiprocessors of the NVIDIA CUDA architecture, only if they are activated by switching events at their inputs. We show in our experimental results that the simulator is capable of delivering a remarkable performance speedup on large, industrial-scale designs of over a million gates, thus bringing about new validation frontiers for the digital design industry. In the future, we plan to explore further optimizations in the segmentation algorithm to deliver even higher simulation performance.

8. REFERENCES

[1] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Trans. on CAD*, 1997.

[2] W. Baker, A. Mahmood, and B. Carlson. Parallel event-driven logic simulation algorithms: Tutorial and comparative evaluation. *IEEE Journal on Circuits, Devices and Systems*, 1996.

[3] Z. Barzilai, J. Carter, B. Rosen, and J. Rutledge. HSS–a high-speed simulator. *IEEE Trans. on CAD*, 1987.

[4] H. Bauer and C. Sporrer. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. *Proc. ANSS*, 1993.

[5] O. Berry and G. Lomow. Speeding up distributed simulation using the time warp mechanism. In *Proc. of workshop on Making distributed systems work*, 1986.

[6] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: a compiled simulator for MOS circuits. In *Proc. DAC*, 1987.

[7] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. ACM*, 1981.

[8] D. Chatterjee, A. DeOrio, and V. Bertacco. High-performance gate-level simulation with GP-GPUs. In *Proc. DATE*, 2009.

[9] M. Denneau. The Yorktown simulation engine. *Proc. DAC*, 1982.

[10] R. Fujimoto. Parallel discrete event simulation. *Comm. ACM*, 1990.

[11] K. Gulati and S. Khatri. Towards acceleration of fault simulation using graphics processing units. *Proc. DAC*, 2008.

[12] H. Kim and S. Chung. Parallel logic simulation using time warp on shared-memory multiprocessors. *Proc. IPPS*, 1994.

[13] Y.-I. Kim, W. Yang, Y.-S. Kwon, and C.-M. Kyung. Communication-efficient hardware acceleration for fast functional simulation. *Proc. DAC*, 2004.

[14] D. Lewis. A hierarchical compiled code event-driven logic simulator. *IEEE Trans. on CAD*, 1991.

[15] N. Manjikian and W. Loucks. High performance parallel logic simulations on a network of workstations. *Proc. of workshop on Parallel and distributed simulation*, 1993.

[16] Y. Matsumoto and K. Taki. Parallel logic simulation on a distributed memory machine. *Proc. EDAC*, 1992.

[17] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 1986.

[18] NVIDIA. *CUDA Compute Unified Device Architecture*, 2007.

[19] Opencores. http://www.opencores.org/.

[20] A. Perinkulam and S. Kundu. Logic simulation using graphics processors. In *Proc. ITSW*, 2007.

[21] Sun microsystems OpenSPARC. http://opensparc.net/.