GCS: High-Performance 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

In recent years, the verification of digital designs has become one of the most challenging, time consuming and critical tasks in the entire hardware development process. Within this area, the vast majority of the verification effort in industry relies on logic simulation tools. However, logic simulators deliver limited performance when faced with vastly complex modern systems, especially synthesized netlists. The consequences are poor design coverage, delayed product releases and bugs that escape into silicon. Thus, we developed a novel GPU-accelerated logic simulator, called GCS, optimized for large structural netlists. By leveraging the vast parallelism offered by GP-GPUs and a novel netlist balancing algorithm tuned for the target architecture, we can attain an order-of-magnitude performance improvement on average over commercial logic simulators, and simulate large industrial-size designs, such as the OpenSPARC processor core design.

1. INTRODUCTION

Logic simulation of structural netlists is a notoriously time-consuming process, yet essential to determine that a synthesized design matches its specifications and behavioral description. Simulation farms consisting of thousands of machines are used by design houses to accomplish the task of validation. The task of verifying the correctness of a design is the most resource-consuming one in the entire development effort [4, 7]. Despite this large scale effort, many execution scenarios of a design often go unverified, particularly corner case situations that occur deep in a design's state space. This research addresses precisely this issue by proposing a novel solution that leverages inexpensive, off-the-shelf electronics, namely, graphic processing units, to boost the performance of logic simulation for synthesized netlists by more than an order of magnitude. Such a performance boost has great potential to increase the correctness of future released digital systems, while decreasing their time to market. We developed our solution using commercial NVIDIA graphic devices, which can be easily deployed on today's simulation farms.

Logic simulation pervades the verification effort in the digital design industry: it is relied upon to thoroughly validate the behavioral description of a design (register-transfer level model) and to check that the behavior corresponds to the original specification. After synthesis, simulation is once again employed to check that a design is synthesized correctly, that is, no timing errors have been introduced, and no functional errors arose due to logic optimizations. Finally, simulation is also the underlying engine for many power estimation and timing analysis tools. In the past decade, several solutions have been proposed to aid engineers in using logic simulation effectively for design verification, including verification languages, coverage tools, constrained random generators, and regression suite management, *etc.* [25]. While these tools have greatly contributed to improvements in verification productivity, the raw performance of simulation engines has been closely following the performance trends of the underlying hardware.

The logic simulation of a circuit's netlist typically begins by *levelizing* the circuit, determining a sequencing for gate simulation compatible with the dependencies imposed by the netlist structure. Gates in a same level do not have any input/output dependency on each other and can be simulated concurrently, an unrealized potential in sequential simulators.

Simulators can be classified based on how they process individual gates: *oblivious simulators* evaluate each gate during each simulation cycle, while *event-driven simulators* only evaluate a gate if a change occurs at its input nets. The former can leverage very simple scheduling, static data structures and better data locality; while the latter require a dynamic analysis of which gates must be scheduled for re-evaluation. The static scheduling of gate evaluations in oblivious simulation makes this approach an ideal candidate for aggressive parallelism available in GPUs.

Evolving from the intense computation requirements imposed on graphic processors, general purpose graphic processing units (GP-GPUs) have recently become available. NVIDIA, a market leader in graphic processors, has developed an architecture and programming interface (called CUDA) [19] enabling end users to control GPU activity directly and develop parallel applications. CUDA provides an architecture amenable to logic simulation, providing the opportunity for concurrent simulation.

1.1 Contributions

In this work we explore the inherent parallelism of netlist simulation in developing a novel cycle-based, oblivious, compiled logic simulator that executes on an NVIDIA CUDA GP-GPU. The simulator, called GCS – Gate-level Concurrent Simulator – enables a specialized design compilation process, which partitions a netlist, optimizes it, and maps gates to the CUDA architecture. Our solution includes novel clustering and gate balancing algorithms, optimized to strike a balance between the demands of large circuits and the architectural resources available in the CUDA hardware. In addition, we developed a clever organization of the data structures to exploit the memory hierarchy characteristics of these devices.

Our experimental results show that GCS can simulate designs of industrial complexity while delivering an order of magnitude performance speedup on average, when compared to state-of-the-art commercial logic simulators. The performance boost delivered by this simulation platform can be easily leveraged by digital design houses with no change to their current verification methodology.

2. RELATED WORK

For several decades the majority of the verification effort in industry has revolved around logic simulators. Initial work from the 1980s addressed several key algorithmic aspects that are still utilized by modern solutions, including netlist compilation, management of event-driven simulators, propagation delays, *etc.* [5, 3, 14]. The exploration of parallel algorithms for simulation started at approximately the same time [2, 17, 23], targeting both shared memory multiprocessors [11] and distributed memory systems [16, 15]. In addition, much effort has been dedicated to parallelize simulation exploiting specialized architectures, also called *emulation systems*, whose computational units are optimized for the simula-

tion of a single logic gate or a small block of gates [1, 12, 13]. One of the very first emulation systems, the Yorktown Simulation Engine [6], was developed at IBM and consisted of an array of special purpose processors. Only recently has the effort of parallelizing simulation algorithms targeted data-streaming architectures (single-instruction multiple-data), as the solution proposed by Perinkulam, *et al.* [21]; however, the communication overhead of this system had a high impact on the overall performance. Another recent work parallelized a fault simulation solution for the CUDA architecture [9]. This solution operates by partioning the problem over the faults to be simulated and different test vectors on the same circuit thus having each block of threads simulate the circuit monolithically for a different fault. In contrast, GCS strives to deliver fast simulation of complex designs, thus we developed a series of circuit partioning and optimization techniques.

The *algorithms for parallel simulation* can be grouped into two families: *synchronous* ones, as the solution proposed in this work, where for each simulation clock cycle several parallel threads are forked off and then joined at a barrier at the end of the step. In contrast, *discrete event* algorithms partition a circuit into non-overlapping sub-circuits and assign each portion to a distinct thread.

A key aspect of all parallel simulation solutions lies in the choice of netlist *partitioning algorithm*, because of its heavy impact on communication overhead. Previous solutions include random [8], activity-based partitioning [16], balanced workload [10], and *cone partitioning* [22], where logic clusters are created by grouping the cones of influence of circuit outputs with the goal of minimizing the number of gates overlapping among multiple clusters. Our solution relies on a variant of cone partitioning tailored to the constraints of our target architecture.

3. NVIDIA CUDA ARCHITECTURE

General purpose computing on graphics processing units brings new opportunities for parallel processing previously unavailable in commodity hardware. NVIDIA's Compute Unified Device Architecture (CUDA) is a new hardware architecture and software framework for using the GPU as a data parallel computing device. In CUDA, the GPU can be viewed as a co-processor capable of executing many threads in parallel. Code is compiled by the host CPU to a proprietary instruction set and the resulting program, called a *kernel*, is executed on the GPU device, with each parallel thread running the same kernel code on distinct data blocks.

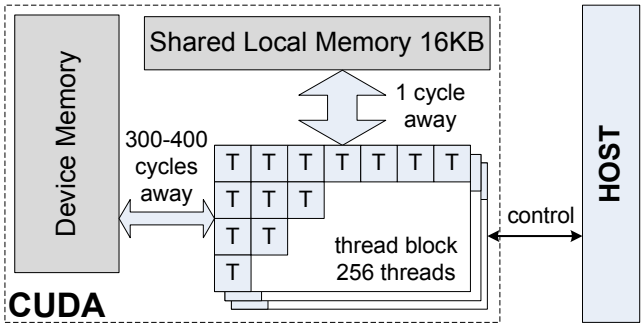

Figure 1: The NVIDIA CUDA architecture is organized as an array of multiprocessors. Each multiprocessor can execute a block of concurrent threads. A 16KB local shared memory block is available at each multiprocessor, which also has access to a larger, shared device memory.

The CUDA architecture [19] (Figure 1) is built around *multiprocessors*, whose number typically doubles with each new device generation. Each multiprocessor can execute a *thread block* comprising up to 512 concurrent threads, all running identical kernel code. Available in each multiprocessor, a fast *local memory* block is accessible within 1 clock cycle and shared among all the threads running on the same multiprocessor. An additional *device memory* block, sized between 256MB to 1GB, is accessible by all multiprocessors with a latency of 300 to 400 clock cycles. Efficient memory usage is key to good CUDA programs. For example, it is possible to mask the latency of device memory by time-interleaving thread execution within a multiprocessor: while a group of threads is executing on local data, others are suspended, waiting for their data to be transferred from the device memory. All execution takes place on GPU hardware, the host CPU serves only to invoke the beginning execution of a thread batch and waits for completion of one cycle of netlist simulation.

4. GCS ARCHITECTURE

GCS operates as a compiled-code simulator, first performing a *compilation*, where it considers a gate-level netlist as input, compiles it and maps it into CUDA. A *simulation* proper follows, where GCS considers a CUDA-mapped design, simulating over a number of several cycles, possibly reusing the same mapped design while running with many distinct testbenches. The process of compilation and simulation progresses in 5 steps (Figure 2). First, a behavioral netlist is synthesized to a gate-level netlist and mapped to GCS's internal representation. From here, the combinational elements are extracted, since the design will be simulated in a cycle-based fashion. Next, GCS partitions the netlist into *clusters*, that is, logic blocks of appropriate size to fit within the constraints of the CUDA architecture. In this phase, the compiler prepares rough clusters, based on size estimates quickly computed on the fly. The following step, *balancing*, is an optimization phase, where each cluster is carefully restructured to maximize compute efficiency during simulation. Finally, all the required data structures are compiled into the CUDA kernel and transferred to the GP-GPU device.

4.1 Synthesis

The GCS compiler requires a gate-level netlist as input. This can either be a synthesized version of a design under verification, or a behavioral description to which we can apply a relaxed synthesis step. In our experimental evaluation, we consider a broad range of designs, including a pool of behavioral descriptions that we synthesized using Synopsys Design Compiler targeting the GTECH library. We selected GTECH because of its broad availability and simplicity of use in an experimental environment. Note, however, that the GCS compiler could easily target any other technology library and, depending on the pool of cells available in the library of choice, this could also bring forward additional performance benefits compared to what we report in this work. Within the GTECH library we excluded non-clocked latches (but not flip-flops), since a cycle-based simulator cannot properly handle the sub-cycle delays involved in the simulation of a non-clocked latch. Multiple clock designs can still be handled by using a logical clock that generates all other clock signals.

When the netlist is read into GCS, an internal representation based on GTECH is created. In GCS we represent each gate's functionality by a 4-valued (0,1,X,Z) truth table. Since each execution thread must execute the same CUDA code, all gates must be specified in the same format. Note that this information is accessed every time a gate is simulated, thus quick accessibility is important.

4.2 Combinational Netlist Extraction

During the compilation phase, GCS extracts the combinational

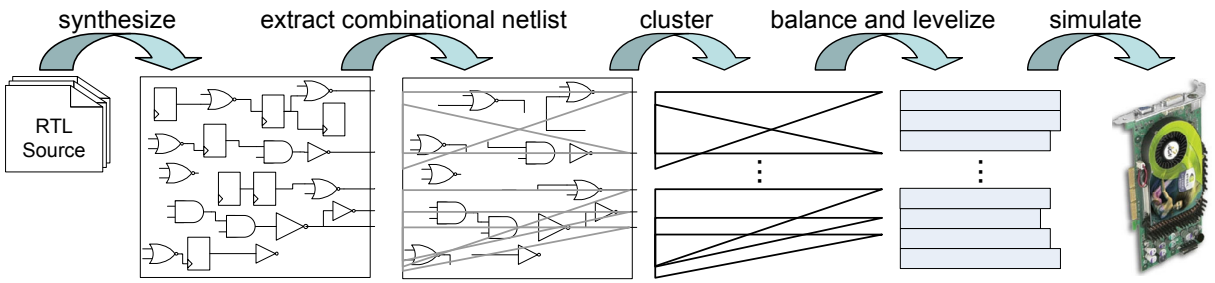

Figure 2: The GCS compiler considers a gate-level netlist or synthesizes a behavioral netlist. It then extracts the combinational logic block and partitions it into clusters, that is portions of the circuit that approximately fit within the resources of a single CUDA multiprocessor. The balancing step then optimizes each cluster to satisfy CUDA resource constraints. Finally, balanced clusters are transferred to the GP-GPU device and the simulation commences.

portion of the gate-level netlist and maps it to CUDA, creating data structures to represent the gates, as well as their input and outputs. The netlist can be viewed as a directed graph, where vertices correspond to logic gates and edges correspond to interconnect wires. Moreover, the graph has multiple outputs (a forest), which may connect to one of the storage elements or primary netlist outputs. Multiple inputs are also present: either primary inputs or coming from storage elements. Finally, during simulation, dedicated data structures store the simulated values for the storage elements (the *input* and *output buffer vectors*) and specialized testbench code feeds primary input values and extracts primary output values at each simulation cycle.

Because of the memory hierarchy of CUDA, an optimal memory layout can lead to significant improvements in the performance of a GP-GPU simulator. GCS places the most frequently accessed data structures in local shared memory (Figure 3). Here, we store intermediate net values (called the *value matrix*), which are computed for each internal netlist node during simulation. Each net requires 2 bits of storage in a 4-valued simulator. Also in local shared memory the *gate-type truth tables* are stored, which are consulted for the evaluation of each gate.

All other data structures reside in the higher-latency device memory: the *input* and *output buffers* and the *netlist topology* information. Note that the netlist topology information is required just as often as the data that we store in the local memory. However, the latter is data that is shared among several threads (gates) and thus its locality can benefit multiple threads.

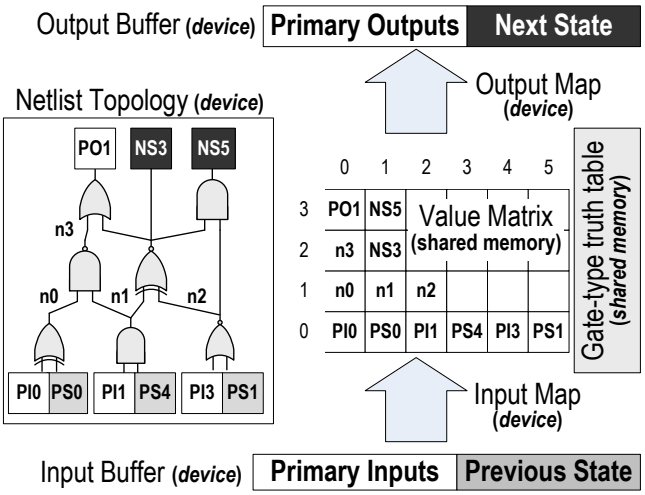

Figure 3: GCS's compiled-netlist data structures. The picture shows the data structures required for the simulation of a small netlist. Thread blocks store and retrieve intermediate net values from the value matrix in the local shared memory. Note that there is a one-to-one correspondence between a row of intermediate values and a netlist's logic level.

4.3 Clustering

GCS's clustering algorithm (Figure 4) divides a netlist into clusters, each to be executed as a distinct thread block on the CUDA hardware. Since CUDA does not allow information transfer among thread blocks within a simulation cycle, all thread blocks must be independent. The central goals of the clustering algorithm are (i) minimizing redundant computation, (ii) data structure organization and (iii) maximizing data locality.

The requirement of creating netlist clusters that are self-contained and do not communicate to other clusters within a simulation cycle led us to choose a *cone partitioning* approach. In cone partitioning, a netlist is viewed as a set of logic cones, one for each of the netlist's outputs; each cone includes all the gates that contribute to the evaluation of that output. Due to the lack of inter-cluster communication capability, each cluster must include one or more cones of logic, and each cone must be fully contained within a cluster. As a result, once a cluster has been completely simulated, one or more output values have been computed and can be stored directly into the output buffer vector. Cone overlap necessarily requires that some gates are duplicated, because they belong to multiple cones. However, as we show in the experimental section, the incidence of this extra computation is small in practice.

During the simulation of a cluster, several data blocks must be readily available. Because each thread block has fast access only to the small local shared memory, the size of this structure becomes the constraining parameter in our clustering algorithm.

With the goal of minimizing cluster overlap, the clustering algorithm proceeds by assigning one cone of logic – we start from the one with the most gates – to a cluster. Additional cones are subsequently added to this cluster until memory resources have been exhausted. The criteria for adding a cones is the *maximal number of overlapping gates*; for example, the second logic cone is the cone that overlaps the most with the first one already included in the cluster. Upon completion of the clustering algorithm, GCS has mapped all gates to a set of clusters, minimizing logic overlap while satisfying the constraints of shared memory resources.

4.4 Cluster Balancing

The *cluster balancing* algorithm minimizes the critical execution path of thread blocks (clusters) on the CUDA hardware. It considers each cluster individually and optimizes the scheduling of each

```
clustering (netlist){
  sort(output_cones)
  for each (output_cone) {
    new cluster = output_cone;
    while (size(cluster) < MAX_SIZE) do {
       cluster += max_overlap(
                    output_cones, cluster);
    } append (cluster, clusters);
 }
 return clusters;
}
```

Figure 4: Pseudo-code for the clustering algorithm. Combinational logic cones are grouped into clusters, netlist blocks that are estimated to fulfill CUDA's resource constraints, with minimal logic overlap.

gate simulation so that the number of logic levels (the limiting factor for execution speed) is minimized. The simulation latency of a single cycle is limited by the cluster with the most logic levels, since each additional level requires another access to device memory 300-400 cycles away. Considering the number of logics levels (cluster *height*) and the number of concurrent threads simulating distinct gates (cluster *width*), the algorithm balances these within the constraints of the CUDA architecture: a maximum of 256 concurrent threads. Since this is a functional simulaton, intra-cycle timing can be safely ignored and thus the transformation is guaranteed to generate equivalent simulation results.

From a visual standpoint, cluster balancing attempts to reshape the natural triangular clusters to a rectangle with a 256-wide base. In this analogy, each gate occupies one entry in a bi-dimensional matrix. Clusters tend to be triangular shapes because they are a collection of cones of logic, which are usually triangular: a wide set of inputs computes one output through several stages of gates. In Figure 6, we show the cluster generated during the GCS compilation of a JPEG decompressor design. The original cluster has a base width of 3,160 gates and a height of 67 levels of logic, where most of the deeper levels require significantly less than 3,000 threads. After balancing, we reshaped the cluster to a rectangular shape with a base width of 256 gates, fitting perfectly on a single thread block, and a height of 81 levels. As a result, it will take 81 subsequent gate simulations per thread to completely simulate this cluster.

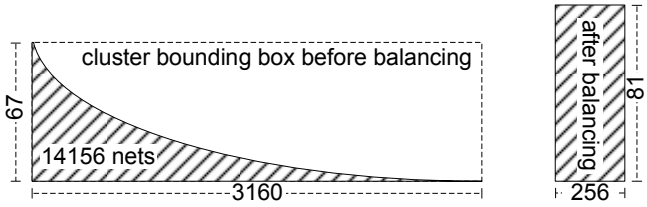

Figure 5: Cluster balancing. Example of balancing of one cluster in the JPEG decompressor design. Height represents the number of logic levels and width the number of gates. The cluster balancing algorithm takes clusters and reshapes them to a maximum 256 gates wide (limited by CUDA resources), while minimizing the height, which is the limiting factor in simulation latency.

Figure 6 shows the pseudocode for the balancing algorithm. It is adapted from a variant of list scheduling algorithm [18] to CUDA data structure constraints, using the slack available at each gate in attempting to minimize the balanced clusters' height.

4.5 Simulation

After the balancing step, the GCS compiler has generated a finite number of clusters, optimized them and generated all the support data structures necessary for the kernel code to simulate all gates

```
balance_cluster() {
  for each level in height
    for each column in width
      balanced_cluster[level][column] =
        select_gate()
    }
  }
  return balanced_cluster
}
select_gate() {
  sort gates in cluster by height
  for each gate in cluster {
    if not assigned_to_balanced_cluster(gate)
      return gate
  }
}
```

Figure 6: Pseudo-code for the balancing algorithm. Clusters are considered one at a time and reshaped to fit into a thread block with a maximum of 256 threads, while minimizing the number logic levels. The algorithm proceeds in a bottom-up fashion, filling the cluster with gates, minimizing the level of each gate and maintaining a maximum width restriction.

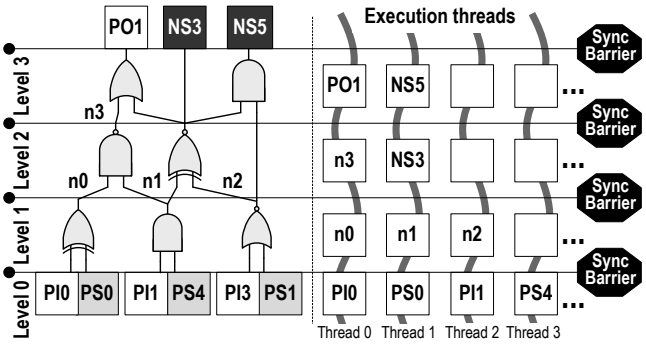

Figure 7: GCS simulation on CUDA. Simulation of the small netlist of Figure 3. Each thread is responsible for computing the output of one gate at a time, vertical waved lines connect the set of value matrix slots for which a single thread is responsible at subsequent time intervals. Note also how each level is followed by a synchronization.

in a netlist with a high level of parallelism while respecting data dependencies. At this point, cluster data and kernel code can be transferred to the GP-GPU device and simulated cycle by cycle.

Simulation begins with the host CPU, which transfers the kernel code and data structures to the GPU and starts the simulation of each cycle. The GPU hardware now takes over without the assistance of the CPU, scheduling clusters for parallel execution. When all clusters have executed, one simulation step is complete and control returns to the host CPU. The host reads primary outputs, evaluates the testbench, sets primary inputs and invokes the next cycle.

Cluster execution on the GPU proceeds in three phases: *scattering*, *logic evaluation* and *gathering*. During scattering, the cluster's primary input data is retrieved from the device memory and copied to the value matrix (Figure 3). Next, logic evaluation progresses when each thread begins execution. The threads, each simulating one gate, retrieve the relevant portion of the netlist from device memory, as well as gate truth tables and net matrices from local shared memory. With this information, the threads evaluate their gates by consulting the truth table. During the gather step, computed results are copied from the value matrix to the output buffer vectors in device memory. Finally, the threads synchronize after simulating their respective gates and the process is repeated for all the subsequent logic levels in the cluster. Figure 7 shows an example of cluster execution for the sample netlist of Figure 3.

5. OPTIMIZATIONS AND TESTBENCHES

GCS employs a number of optimizations to speed simulation, including running testbenches as kernel threads, as well as using the GPU's texture memory to speed memory accesses.

Testbenches. In GCS we implemented the testbench in a separate kernel so that, during each simulation cycle, the netlist-kernel and lightweight testbench-kernel alternate execution on the GP-GPU device. *Microprocessor test kernels* are very simple kernels that can be used when simulating a processor design. These designs are usually simulated by executing an assembly program, thus we upload the program to device memory and include a specialized kernel whose job is simply to serve memory requests from the processor. When possible, *synthesizable test kernels* can deliver high performance, since they can themselves be mapped to a netlist, and thus the simulation can be viewed as a co-simulation between two digital circuits. Another approach we explored uses a *trace playback test kernel*, where the input stimuli are captured while running the test generator on a separate system and later uploaded to device memory. The kernel then transfers these inputs to the simulated design. Hybrid or altogether different solutions are also viable, for instance, in the case of our JPEG decompressor design, the testbench was an image to be decompressed and resided in device memory. Complex testbenches involving constructs that can not be represented as a kernel in CUDA still be executed on the host CPU, but an additonal communication penalty will be incurred with every cycle. Debugging support can also be implemented at the cost of storing internal values in device memory and incurring the related latency penalty.

Optimizations. Several optimizations in GCS take advantage of the raw performance of the GP-GPU. First, we leveraged the *texture memory* to reduce the access latency when loading the netlist topology from device memory. The texture memory block acts similarly to a direct memory access device, launching monolithic requests for large, consecutive data reads. Thus, with proper data layout, once the first thread receives the required information, a number of other threads will receive their data as well. The result is an overall reduction in cluster simulation latency.

Another optimization focuses on minimizing cluster latency. We observed that if we mapped two clusters to the same multiprocessor, then CUDA could interleave their execution and mask most of the time spent while waiting for data from device memory. However, in order to fit two clusters in a multiprocessor, we had to accommodate the value matrices for both clusters in the shared local memory. We found that this option delivered valuable performance improvements, hence the GCS compiler uses an 8KB memory bound (instead of 16KB) when estimating the number of logic cones that can fit within one cluster. Additionally, the physical GPU's limit on the number of threads for each multiprocessor is 512, but we used 256 so that we could fit two clusters executing at the same time. We found that reducing clusters to smaller sizes did not bring a further performance advantage.

6. EXPERIMENTAL EVALUATION

We evaluated the performance of GCS on a set of Verilog designs ranging from a combinational LDPC (Low-Density Parity Check) encoder to an industrial microprocessor core from the OpenSPARC T1. Designs were obtained from OpenCores [20] and from the Sun OpenSPARC project [24]; the 5 stage processors and NoC designs were designed by student teams.

Table 1 shows the key characteristics of the designs. We report the number of gates and flip-flops in the corresponding synthesized netlist for each design and indicate the testbench used for simulation. The first two designs, 5 stage in-order and pipelined processors, implement a subset of the Alpha instruction set and simulated a recursive Fibonacci program. The LDPC encoder is a combinational design simulated with random stimulus. The JPEG decompressor reads a 1920x1080 pixel image and decompresses it. The NoC designs are torus networks of 5-channel routers connected to a random stimulus generator, which sends legal packets over the network. Finally, the SPARC core is a single processor from the OpenSPARC T1 multi-core chip (excluding caches) and runs test programs provided with Sun's open-source distribution. In our evaluation, we did not run into the theoretical maximum design size limit imposed by the GP-GPU hardware, indeed it is possible to run much larger designs. The design size limit is governed by the amount of shared memory available as well at the number of multiprocessors and is given by $\frac{shared\_memory\_bits}{2} * num\_multiprocessors$, about 1 million gates with our GPU device. Note that simulations of over 6 million gated designs would be possible with the largest available NVIDIA system today.

Design	Testbench	# Gates	# Flops
5 stage in-order	recursive Fibonacci program	17546	2795
5 stage 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	167973	23875
SPARC core	v9allinst.s lsu_mbar.s lsu_stbar.s	262201	62001

Table 1: **Designs used for evaluating GCS.**

design	cycles	Seq Sim(s)	GCS time(s)	Speed up
5 stage in-order	12,889,495	40,427	9,942	4.07x
5 stage pipeline	13,423,608	67,560	10,688	6.32x
LDPC encoder	100,000	12,014	193	62.25x
	1,000,000	120,257	1,993	60.34x
	10,000,000	>48h	19,859	
JPEG decompressor	2,983,674	14,740	929	15.87x
3x3 NoC routers	111,823	386	50	7.72x
	1,225,245	2,819	324	8.7x
	1,967,155	4,258	504	8.45x
4x4 NoC routers	120,791	561	82	6.84x
	1,298,438	3,263	424	7.7x
	2,018,450	5,061	659	7.68x
	10,000,001	34,503	4,656	7.41x
SPARC - v9allinst.s	119,017	3,221	756	4.26x
- lsu_mbar.s	137,497	3,726	880	4.23x
- lsu_stbar.s	101,720	2,762	640	4.32x

Table 2: GCS performance. Comparison of GCS simulation performance against a state-of-the-art event-driven simulator. GCS outperforms the sequential simulator by 14.4x on average.

6.1 Performance

We evaluated GCS's performance by comparing it to a state-of-the-art, event-driven, compiled code simulator, considered among the fastest available today. GCS simulations were run on a CUDA-enabled 8800GT GPU with 14 multiprocessors and 512MB of device memory, running the cores at 600 MHz and the memory at 900 MHz. The sequential simulations were run on a 3.4GHz Pentium 4 workstation with 2GB of memory. Both were configured to simulate without monitoring values of internal nets. All our testbench circuits were synchronous designs with latches driven by a single clock, hence the commercial simulator and GCS both worked on a single clock boundary. Table 2 reports the number of cycles, runtimes for both GCS and the commercial simulator, and the relative

speedup. Times are in seconds and reflect total simulation runtime, excluding compilation for both solutions. It can be noted that GCS always outperforms the commercial simulator by a factor of 4 to 60, averaging 14.4 times speedup. The speedup was most significant for the LDPC design because high switching activity in this experiment affected the performance of the event-driven commercial simulator, but not that of GCS.

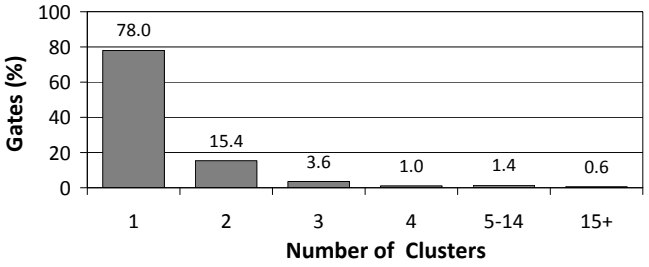

Figure 8: Gate duplication due to cluster overlap. Percentage of gates present in multiple clusters is shown; the first bar indicates no duplication.

6.2 Clustering Efficiency

In the next experiment, we evaluated the impact of cluster overlap, as discussed in Section 4.4. Figure 8 reports the percentage of gates present in multiple clusters, averaged over all designs. We found that nearly 80% of the gates were not duplicated, 15% were present in two clusters and less than 7% were present in more than two clusters. We conclude that the cone-based clustering algorithm is both effective in removing inter-cluster dependencies and does not introduce a relevant increase in the required computation.

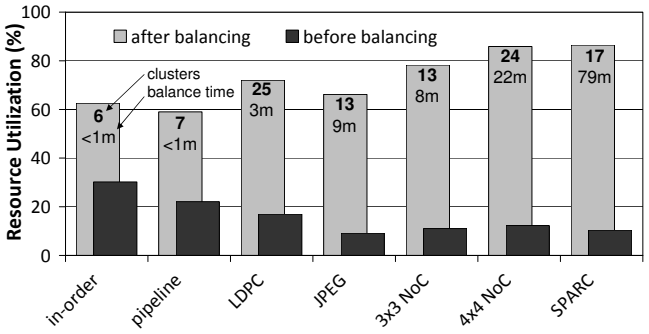

Figure 9: Cluster balancing. Resource utilization averaged over all clusters in each design, before and after balancing.

6.3 Cluster Balancing Efficiency

Finally, we evaluated GCS's ability to effectively optimize individual clusters with its cluster balancing algorithm. An example of balancing for a cluster in the JPEG decompressor design is shown in Figure 5. Figure 9 reports the average utilization ratio for all clusters in each design, before and after the balancing step. The utilization ratio is computed as the ratio of the area covered by the scheduled netlist in a cluster over the cluster bounding box. For example, with reference to Figure 5, the utilization of the cluster before balancing is $\frac{14156}{67 \cdot 3160}$, while the utilization after balancing is $\frac{14156}{256 \cdot 81}$. Figure 9 also reports the total number of clusters in each design and the total compilation time. We observed that the total number of clusters in each design was relatively low despite large latch counts, due to the aggregation of a large number of cones, all balanced to run simultaneously on the same multiprocessor. It can be easily noted that balancing brings a sharp boost in the clusters' utilization factors across all designs.

7. CONCLUSIONS

We have presented GCS, a high-performance, concurrent simulator for gate-level netlists on parallel GP-GPU architectures. GCS maps complex netlists to NVIDIA GPUs by employing a novel clustering and balancing algorithm. The algorithm cleverly orchestrates the use of GPU resources to convert their high computing power into simulation performance. In our experimental results, we show that GCS is capable of order-of-magnitude speed-ups over state-of-the-art commercial simulators. GCS opens new horizons in the performance of logic simulators, which are the workhorse of verification in the industry. In the near future, we plan to explore possible solutions for an event-driven, timing-aware simulation solution for CUDA able to handle arbitrarily large designs.

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] J. Bergeron. *Writing testbenches: functional verification of HDL models*. Kluwer Academic Publishers, 2000.

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

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

[7] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *IEEE Computer*, 2004.

[8] E. Frank. Exploiting parallelism in a switch-level simulation machine. *Proc. DAC*, 1986.

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

[10] S. Karthik and J. A. Abraham. Distributed VLSI simulation on a network of workstations. In *Proc. ICCD*, 1992.

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

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

[13] H. Kohler, J. Kayser, H. Pape, and H. Ruffner. Code verification by hardware acceleration. *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, pages 65–69, 2001.

[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] G. Meister. A survey on parallel logic simulation. Technical report, University of Saarland, Dept. of Computer Science, Misra J, 1993.

[18] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[19] NVIDIA. *CUDA Complete Unified Device Architecture*, 2007.

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

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

[22] S. Smith, W. Underwood, and M. R. Mercer. An analysis of several approaches to circuit partitioning for parallel logic simulation. In *Proc. ICCD*, 1987.

[23] L. Soulé and T. Blank. Parallel logic simulation on general purpose machines. In *Proc. DAC*, 1988.

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

[25] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification*. Morgan Kaufmann Publishers Inc., 2005.