Gate-Level Simulation with GPU Computing

DEBAPRIYA CHATTERJEE
University of Michigan
ANDREW DEORIO
University of Michigan
and
VALERIA BERTACCO
University of Michigan

Functional verification of modern digital designs is a crucial, time-consuming task impacting not only the correctness of the final product, but also its time to market. At the heart of most of today's verification efforts is logic simulation, used heavily to verify the functional correctness of a design for a broad range of abstraction levels. In mainstream industry verification methodologies, typical setups coordinate the validation effort of a complex digital system by distributing logic simulation tasks among vast server farms for months at a time. Yet, the performance of logic simulation is not sufficient to satisfy the demand, leading to incomplete validation processes, escaped functional bugs, and continuous pressure on the EDA industry to develop faster simulation solutions.

In this work we propose GCS, a solution to boost the performance of logic simulation, gate-level simulation in particular, by more than a factor of 10 using recent hardware advances in graphic processing unit (GPU) technology. Noting the vast available parallelism in the hardware of modern GPUs, and the inherently parallel structures of gate-level netlists, we propose novel algorithms for the efficient mapping of complex designs to parallel hardware.

Our novel simulation architecture maximizes the utilization of concurrent hardware resources while minimizing expensive communication overhead. The experimental results show that our GPU-based simulator is capable of handling the validation of industrial-size designs while delivering more than an order-of-magnitude performance improvements on average, over the fastest multi-threaded simulators commercially available.

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

Additional Key Words and Phrases: Gate-level simulation, High-performance simulation, General Purpose Graphics Processing Unit (GP-GPU), GPU Computing, Parallel CAD

1. INTRODUCTION

Logic simulation is a central aspect of the modern integrated circuit development process. It is the primary tool used to validate a wide range of design aspects, foremost among these being the correctness of the system's functionality, both in its behavioral description, as well as in its structural (gate-level) one. Most industry design flows invest the largest fraction of their time and resources precisely on this task [Edenfeld et al. 2004], in an attempt to provide the best possible guarantee that the system satisfies its original functional specification. Large server farms, comprising thousands of machines, execute billions of cycles of simulation for months at a time. Within this effort, the simulation of gate-level netlists is an especially onerous task, as it involves large netlists at a fairly low-level description, comprising many components to be simulated. However, despite the vast effort of time and resources, functional validation remains an incomplete task, with large portions of the design going unverified. Indeed, while common case scenarios are often checked in this process, buggy and rare corner cases frequently slip through validation, and are consequently latent in the final product, potentially causing malfunctions in the field. For these reasons, exacerbated by the constantly increasing complexity of digital systems, there is a strong need for increased performance in logic simulation, especially in time-consuming gate-level simulation, to improve the productivity and cost of digital developments.

Logic simulation entails evaluating the response of a design over time when subjected to a set of input stimuli, typically selected by the designer to be representative of practical use situations. For most synchronous designs the response is computed once for each cycle of simulated execution. Modern logic simulators read in a design description, then "compile" it to produce machine code emulating the same functionality as the design's primitives, and finally optimize it to minimize the amount of computation required to provide the responses that the user wishes to observe. The input stimuli are commonly provided in the form of a testbench, that is, a program describing implicitly or explicitly the set of input values for each clock cycle of simulation. The testbench may be direct, where input values are selected by a verification engineer, or pseudo-random, that is, inputs are set by a generator abiding pre-set constraints and statistical distributions.

Simulators can be grouped into two families based on their internal architecture: *oblivious* simulators compute all gates in the system during every simulation cycle and entail a simpler software design. Oblivious simulators have the advantage of low control overhead, but can spend significant computation time unnecessarily evaluating gates over and over whose output values do not change from cycle to cycle. *Event-driven* simulators limit the amount of computation by selectively simulating in each cycle only those gates who inputs have changed since the previous cycle, and whose output may thus change in response to the switching stimulus. While the sequencing of gate evaluation in oblivious simulation can be statically determined at compile-time, event-driven simulators require a dynamic runtime scheduler, hence entail a more complex software structure. However, this latter approach is vastly more common in commercial tools because the scheduler performance overhead is largely offset by the fact that for many designs only 1 to 10% of the gates switch at each cycle, thus requiring significantly less computation.

In investigating the potential for large performance improvements in logic simulators, we noted that logic netlists present a high degree of structural parallelism that could be exploited by simulating individual gates concurrently. An ideal platform leveraging such concurrency is the modern graphics processing unit (GPU), as it includes many simple and identical computational units capable of operating concurrently by executing same instruction sequence on different data. GPU computing is a recent extension of traditional graphics processing, providing a general purpose programming interface for GPU devices, and making the vast parallel computational resources available for applications beyond the processing of graphic primitives. Platforms for GPU computing include AMD's FireStream [AMD 2008] and NVIDIA's CUDA [NVIDIA 2007]. In addition, vendor-independent parallel computing standards such as OpenCL [Khronos Group] have also been developed.

In this work, we provide a solution to gate-level simulation which strives to leverage the concurrency of GPUs to vastly boost the performance of gate-level simulation, and we deliver an efficient, architecture-aware algorithm for mapping large designs to GPU hardware.

1.1 Contributions

Our novel logic simulation solution executes on a GPU platform and it is called GCS, GPU-based Concurrent Simulator (GCS). We leverage GPUs' massive parallelism to achieve large performance improvements compared to the fastest modern commercial simulators. One of our design goals is to grasp the advantages of event-driven simulators so that only a small fraction of the netlist's gates are simulated at each cycle. However, it is critical that our solution incurs only minimal overhead for run-time event scheduling: event scheduling is an intrinsically sequential process, while we want to maintain a massively parallel computation environment for the majority of the time. As a result, GCS is a unique hybrid simulator where the design is partitioned into clusters of gates (called *macro-gates*): clusters are then simulated in an oblivious fashion, while the scheduling of individual clusters is organized in an event-driven fashion. In addition, all algorithms involved in the simulation are optimized for an underlying GPU architecture, characterized in particular by limited shared memory space and by the inclusion of additional components designed to optimize the execution of graphic primitives (for instance, texture memory). Specifically, our contributions to deliver this novel solution include:

(1) a hybrid event-driven/oblivious scheduling algorithm optimized for GPU computing;
(2) a netlist partitioning and macro-gate sizing solution targeting maximal concurrency within the constraints of the GPU's hardware resources;
(3) a balancing algorithm to optimize resource utilization during the simulation of individual macro-gates;
(4) testbench solutions minimizing the performance cost of testbench evaluation on the GPU platform.

We provide a wide range of experimental evaluations highlighting different aspects and design trade-offs of our solution. In addition, our experiments show that our GPU-based simulator achieves an order-of-magnitude performance gain over state-of-the-art commercial logic simulators while tackling industrial-size designs, such as the OpenSPARC T1 multiprocessor [OpenSPARC].

2. RELATED WORK

For several decades the majority of industry verification effort 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.*[Barzilai et al. 1987; Bryant et al. 1987; Lewis 1991]. The exploration of parallel algorithms for simulation started at approximately the same time [Baker et al. 1996; Meister 1993; Soulé and Blank 1988], targeting both shared memory multiprocessors [Kim and Chung 1994] and distributed memory systems [Manjikian and Loucks 1993; Matsumoto and Taki 1992]. In these solutions, individual execution threads operate on distinct netlist clusters and communicate in an event-driven fashion, with a thread being activated if switching activity is observed at the inputs of its netlist cluster. In particular, Baker et al. [1996] provides a comparative analysis of early attempts to parallelize event-driven simulation by partitioning 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. Both conservative [Chandy and Misra

1981; Fujimoto 1990; Misra 1986] and speculative techniques, such as time warp [Bauer and Sporrer 1993; Berry and Lomow 1986], 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.

Emulation systems and specialized hardware solutions for high performance simulation, 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 [Babb et al. 1997; Denneau 1982; Kim et al. 2004]. Modern emulators can deliver 3–4 orders of magnitude speedup over simulation software and can handle very large designs. However, their cost is prohibitive and the process of successfully mapping a netlist to an emulator can take up to a few months.

The effort of parallelizing simulation algorithms has only recently targeted data-streaming architectures (single instruction multiple thread), as the solution proposed by Perinkulam and Kundu [2007]; however, the communication overhead of this system had a high impact on its overall performance. Another recent solution in this space, by Gulati and Khatri [2008], introduces parallel fault simulation on a CUDA GPU target. It extracts parallelism by simulating distinct fault patterns on distinct processing units, with no partitioning within individual simulations or within the design. In contrast, we target fast simulation of complex designs, requiring specialized algorithms to partition the design and target parallel processing elements while leveraging memory locality. We have performed preliminary studies in this direction using oblivious [Chatterjee et al. 2009b] and event-driven [Chatterjee et al. 2009a] simulation architectures. Moreover, in our work we focus on optimizing the performance of individual simulation runs, in contrast with Gulati and Khatri [2008], which optimizes over all fault simulations. A key aspect of all parallel simulation solutions lies in the choice of a netlist partitioning algorithm, because of its heavy impact on communication overhead. Previous solutions include random [Frank 1986], activity-based partitioning [Matsumoto and Taki 1992], balanced workload [Karthik and Abraham 1992], and *cone partitioning* [Smith et al. 1987], 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.

Finally, in recent years, acceleration of various computationally intensive processes using general purpose graphics processing units (GPUs) has been suggested in several other domains of EDA, such as power grid analysis [Shi et al. 2009], fast circuit optimization [Liu and Hu 2009], statistical timing analysis [Gulati and Khatri 2009], circuit-level simulation [Gulati et al. 2009]. Utilizing GPU computing to accelerate a few core algorithms that are shared across many EDA applications has also been proposed [Deng et al. 2009].

3. CUDA OVERVIEW

General purpose computing on graphics processing units enables parallel processing on commodity hardware. NVIDIA's Compute Unified Device Architecture (CUDA) is a hardware architecture and complementary software interface empowering GPUs to do general purpose computing. In the CUDA execution model, the GPU is a co-processor capable

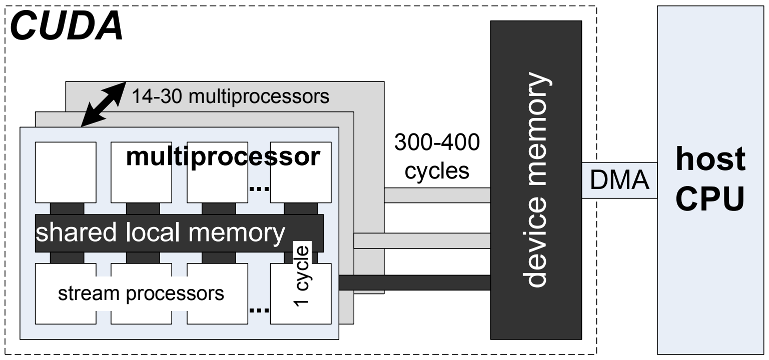


Fig. 1. **NVIDIA CUDA architecture.** The GPU contains an array of multi-processors, each containing individual stream processors. Within a multiprocessor, the stream processors have access to a small, fast shared local memory. Multiprocessors can also access higher latency device memory. Finally, the GPU device communicates with the host CPU via DMA.

of executing many threads in parallel. A data parallel computation process, known as a kernel, can be offloaded to the GPU for execution. This model of execution is known as single instruction multiple thread (SIMT), where thousands of threads execute the same code, each operating on different portions of data. Threads identify their spatial location within the data by thread ID and thread block ID.

The CUDA architecture [NVIDIA 2007] (Figure 1) consists of a number of multiprocessors (14-30 in the G80 generation) contained in a single GPU chip. Each multiprocessor is comprised of multiple stream processors (8 in G80 generation, 32 in current GF100 generation) and can execute a large number of concurrent threads (up to 512 in G80 generation, 1024 in current generation) all running the same code. The block of threads contained in one multiprocessor has access to a small amount of shared memory (16 KB in G80 generation, up to 48 KB in current generation) at an access latency of 1 clock cycle. All multiprocessors also have access to a global memory called device memory, which can be 256 MB to 1 GB in current CUDA enabled GPU's and has higher access latency (300-400 cycles). While the access latency to global memory is high, it is possible to amortize the cost by coalescing accesses from multiple threads. Finally, communication with the host CPU's main memory is achieved by means of direct memory access (DMA) transfers, which are most efficiently executed in large blocks.

Threads belonging to a single thread block can synchronize among themselves using fast barrier synchronization, and also co-operatively access shared memory. Synchronization among different thread blocks, however, is cumbersome and has a high overhead. Due to this architecture, providing memory locality within individual thread blocks is critical for performance.

4. SIMULATOR OVERVIEW

Noting the extensive concurrency available in data-parallel GPUs, our primary design goal is to match these resources with the parallelism present in gate-level netlists. In a levelized gate-level netlist, where each level depends only on those computed at previous levels, all gates in a same level can be simulated in parallel. Moreover, in the best scenario, only those gates whose inputs have changed since the previous simulation cycle would be computed. Scheduling only the correct gates for computation requires a central event queue to manage

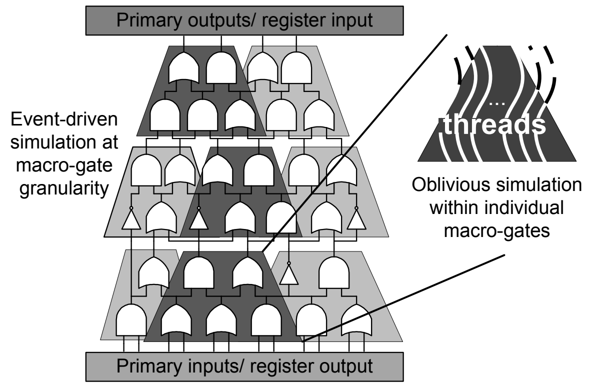

Primary outputs/ register input
Event-driven simulation at macro-gate granularity
threads
Oblivious simulation within individual macro-gates
Primary inputs/ register output

Fig. 2. **Hybrid event-driven simulator.** The GCS architecture is event-driven at the granularity of macro-gates, while the macro-gates themselves are simulated in oblivious fashion. As an example, during a simulation cycle, only the darker macro-gates could be activated, each of them simulated one after the other by a single thread block in an oblivious fashion.

the list of ready gates after the completion of each netlist level. However, note that central event queues incur a significant performance overhead in a parallel processing model, due to the overhead of synchronization. In the CUDA GPU model, local synchronizations (within a thread block) are efficient, while synchronizing among distinct thread blocks is costly. However the total number of logic gates that is activated in each clock cycle in a typical design may greatly exceed the maximum number of allowed threads in a thread block (512 in the GPU hardware we used). Thus local synchronization is not sufficient to handle all events in the netlist. Moreover, best thread block performance is achieved when the memory accesses from the thread block have a regular structure. This situation can not be achieved if individual threads were to update a global event-queue, as it would be the case if simulation within the thread blocks were event-driven. Thus, oblivious simulation is best suited for within thread block execution. These observations suggest an optimal solution based on a hybrid execution model, where execution within a same thread block is uniform (thus oblivious) but it can be heterogeneous across thread blocks, making an event-driven flow effective.

Matching data locality in the netlist with that of the GPU is also a critical design goal. In a netlist, gates outputs are often inputs to several other gates in subsequent netlist levels. With this data flow, memory locality can be leveraged by storing locally the most accessed values, that is, the intermediate output values generated during simulation. Based on these observations we derive a hybrid simulator design for GCS that uses event-driven simulation at a coarse granularity and oblivious simulation within each coarse grain group, with data locality being exploited during the oblivious simulation of each group.

4.1 Hybrid event-driven simulator

GCS is a hybrid event-driven simulator, balancing the advantages of dynamic gate scheduling with the GPU architecture requirements, which necessitates identical control flows. Our design is inspired by two common approaches to gate-level simulation of digital designs. The first approach, oblivious simulation, simulates every logic gate in the design at every simulation cycle. While this has the advantage of uniform control flow, it can result in the superfluous computation of gates whose inputs did not change. By contrast, event-

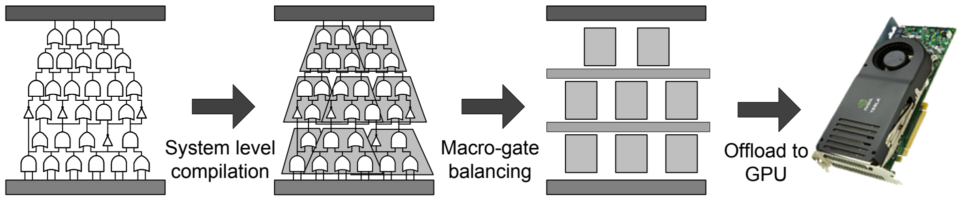


Fig. 3. **Simulator organization.** A compilation step produces macro-gates, which are optimized and offloaded to the GPU for simulation.

driven simulation computes a subset of the gates at each cycle, computing only those gates whose input values have changed. This requires dynamic control flow, incurring a scheduling overhead. Usually this overhead is worthwhile in sequential solutions, as only a small fraction of gates are active in any given cycle, hence event-driven simulators achieve better performance for most designs, and they are the basis for most commercial simulators.

With a goal of optimum performance for gate-level simulation, our design decisions were driven by the constraints of the GPU platform. Thus, the design is partitioned into several clusters of logic gates, and each cluster is called a macro-gate. Simulation of each macro-gate is carried out in oblivious fashion, while macro-gates are scheduled for simulation only if some of their inputs have changed since the previous simulation cycle. Thus the simulation is event-driven at the granularity of macro-gates: this process is shown in Figure 2, where we show a possible grouping of a netlist's logic gates into macro-gates, and we highlight a possible simulation sequence activating only three macro-gates because no inputs have changed for all others.

4.2 Simulator organization

For performance reasons, our GCS simulator is organized as a compiled code simulator, first performing a compilation process to convert the netlist into internal data structures to be mapped to the CUDA memory hierarchy, and then transfering the compiled data structures to the GPU platform for the simulation proper. During the simulation phase, the CUDA-mapped design is simulated based on the input stimuli provided by the validation testbench. Note that it is possible and recommended to reuse the same compiled design several times to simulate with many different testbenches. The GCS compiler proceeds in two phases (see Figure 3): the first phase is *system level compilation*, where a gate-level netlist is considered as input. Segmentation is applied to the netlist to partition it into a set of levelized *macro-gates*: each macro-gate includes several gates within the netlist connected by input/output relations. Companion data structures are also created for each macro-gate to facilitate event-driven simulation. The second phase is *macro-gate balancing*: during this phase each macro-gate is reshaped for optimal latency of execution on the GPU platform. There are several possible variations in the process of segmentation with regard to sizing the macro-gates, as will be discussed in Section 5.3.

During simulation proper, a data parallel GPU program, known as a "kernel" (see Section 3) operates on the data structures generated by the GCS compiler, which have been transferred to the GPU. In addition, this phase requires a set of input stimuli for the design under validation. Typically, the stimuli are generated by a testbench program that reads the netlist outputs generated at the end of each simulated clock cycle and, based on those values, generates inputs for the next clock cycle. We investigated several solutions for testbench design that are viable for a GPU computing framework and we discuss them in Section 6.3. We found that one of the most effective techniques consists of writing the testbench as a GPU kernel (program) executing at the completion of each simulation cycle by evaluating the netlist outputs from the previous cycle and producing the input stimuli for the following cycle directly in device memory. Note that one important goal is to have all the simulation data structures, testbench data and GPU kernels reside in GPU memory, so as to avoid time consuming data transfers to and from the host processor.

5. COMPILATION

GCS operates as a compiled code simulator, processing a gate-level netlist as input and generating machine code that can be executed on a GPU. First, system level compilation segments the netlist into macro-gates. The second phase, macro-gate balancing, reshapes the macro-gates to more regular structures for achieving best utilization of GPU thread resources.

5.1 System level compilation

The goal of system level compilation is to segment the netlist into macro-gates, groups of gates that will be simulated in an oblivious fashion by a single thread block. Different macro-gates are scheduled in an event-driven fashion in our hybrid simulator. Three pre-processing steps are required before macro-gates can be extracted: *synthesis*, *combinational logic extraction* and *levelization*. Following these steps, macro-gate segmentation can be performed, producing a set of macro-gates.

5.1.1 *Synthesis.* Synthesis, the first step of system level compilation, produces a gate-level netlist as output, that is, a synthesized version of the design under verification. To generate a gate-level netlist, a digital design is synthesized to a flattened netlist using a target technology library. In our experimental evaluation, a range of behavioral circuit descriptions were synthesized using Synopsys Design Compiler targeting a subset of the GTECH library. The choice of GTECH was due to its generality and simplicity of use in an experimental environment; however, other libraries may be used as well, since gates are replaced by functional primitives by the simulator. Note that, if a synthesized version of the design is already available, this step is unnecessary.

The subset of the GTECH library used in synthesis excludes non-clocked latches (but includes flip-flops), making cycle-by-cycle simulation possible. Sub-cycle delays involved in the simulation of non-clocked latches would require further detailed modeling. Note that multiple clock designs can still be handled by using a logical clock that generates all other clock signals, and operates at a period of the greatest common divisor of all original clock signals. When the netlist is read into the compiler, an internal functional primitive of each gate based on GTECH is created. This functional primitive is represented by a 4-valued (0, 1, X, Z) truth table. Since each execution thread must execute the same instruction sequence for best performance, all gates must use an uniform format to indicate their input nets and functionality. Moreover, this representation is most efficiently packed at the bit level, so as to not waste precious memory bandwidth, since this information is accessed every time a gate is simulated.

5.1.2 *Combinational logic extraction.* Once the design is synthesized and read into internal functional primitives, the combinational portion is extracted. Since the design is simulated cycle by cycle, the contents of registers that retain state across clock cycles can

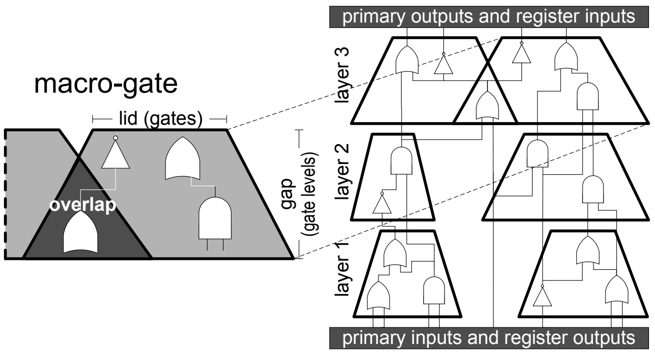

macro-gate
lid (gates)
overlap
gap
(gate levels)
primary outputs and register inputs
layer 3
layer 2
layer 1
primary inputs and register outputs

Fig. 4. **Macro-gate segmentation.** The levelized netlist is partitioned into layers, each encompassing a fixed number of gate 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 replicated to all associated macro-gates.

be modeled as memory locations. These storage elements are written with the values at register inputs, and read in the next simulation cycle as register outputs. Thus, the entire circuit can be viewed as a combinational network from the simulator's point of view. In the absence of combinational loops, this network is a directed acyclic graph (DAG), whose vertices correspond to logic gates and edges correspond to connecting wires. This DAG has multiple outputs (a collection of trees, hence a forest), which may be outputs of registers or netlist primary outputs. Also, multiple inputs are present: either primary inputs or from register outputs.

5.1.3 *Levelization.* The combinational network graph is now levelized: a topological sorting is performed on the DAG, so that the fan-in of all gates in each 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) policy during levelization, though other solutions are also possible.

5.1.4 *Macro-gate segmentation.* Macro-gate segmentation partitions the levelized combinational network graph into blocks of logic with multiple inputs and outputs, referred to as macro-gates. In addition to the segmentation itself, sensitivity lists are generated, noting the relation of macro-gates to each other to inform event-driven simulation.

Three important factors govern the macro-gate formation process: (i) since the objective of forming macro-gates is to perform event-driven simulation at a coarse granularity (compared to individual gates), the time required to simulate a certain macro-gate should be substantially larger than the overhead to decide which macro-gate to activate. (ii) GPU multiprocessors can only communicate through slower device memory: thus for best performance, there should not be any communication among the thread blocks simulating different macro-gates, a goal that can be achieved if the tasks that execute on distinct multiprocessors are independent of each other. This can be assured only if concurrently simulated macro-gates are independent, as can be attained by replicating small portions of shared logic. (iii) Finally, we want to avoid cyclic dependencies between macro-gates so that no macro-gate is repeatedly simulated during one clock cycle, implying that the netlist must be levelized at the granularity of macro-gates as well.

```
segmentation (netlist, gap, lid) {
  levelized_netlist = ALAP_levelize(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; }
```

Fig. 5. **Macro-gate segmentation algorithm.** First, the netlist is levelized, and the resulting levels are grouped into layers. Each layer is then divided into macro-gates and added to the pool of gates to be simulated. The nets to be monitored for activity are also tagged at this stage.

To address the above list of constraints, we segment the netlist by partitioning it 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 (in gates) of the macro-gate. In this procedure, it is possible that a logic gate may be assigned to two or more macro-gates, and then replicated to avoid data sharing (second requirement).

There are several possible policies for selecting the nets whose cones of influence should be clustered in a single macro-gate. To minimize replication, our baseline policy attempts to cluster together nets with the greater number of gates in common. Additionally, 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 4 shows a schematic of the segmentation technique, while Figure 5 presents the pseudo-code of the algorithm. The set of nets that crosses the boundary between each pair of layers is monitored during simulation to determine which macro-gates should be activated.

Section 5.3 discusses how to select a suitable value for gap and lid so as to achieve a high-level of parallelism during simulation as well as maintaining the event-driven structure of simulation. In Section 5.3.1 we use a profiling technique to determine suitable values of gap and lid and also an alternative clustering policy, targeting better simulation performance. We further extend these ideas to enable flexible values of gap and lid across the segmentation process and analyze the resulting trade-offs.

5.2 Macro-gate balancing

After macro-gate segmentation has been performed, each macro-gate can be treated independently as a block of logic gates having a set of inputs and a set of outputs. In the simulation phase, a macro-gate is simulated only if the value at one of its inputs change, in which case simulation of all gates within the macro-gate is carried out in an oblivious manner by the parallel threads in a single thread block. The step of macro-gate balancing reshapes each macro-gate to enable the best use of execution resources.

Within a thread block, a number of threads concurrently simulate all the gates in a level, then move on to the next level, and so on until an entire macro-gate has been simulated. The number of gate levels in a macro-gate, *i.e.*, the gap, is inversely proportional to the macro-gate simulation performance. Since all threads must execute through each level, if

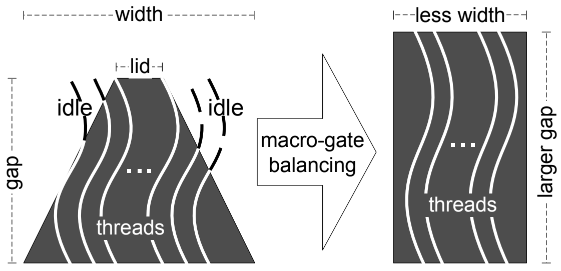


Fig. 6. **Macro-gate balancing.** The balancing algorithm restructures each macro-gate to minimize the number of execution threads required. The result is a more efficient utilization of thread block resources.

there is a level where gates are fewer than threads, then some threads will be idle. This situation occurs naturally because often macro-gates present a large base (many gates) and a narrower tip in a trapezoidal shape. This shape results from our segmentation algorithm which includes the fan-in cone of a few wires from a layer boundary. As a result a large number of active threads are required at the lower levels, and just a few at the top levels.

Thus, to maximize concurrency, we strive to reduce the number of threads required at the lower levels with a *balancing* step, outlined in the schematic of Figure 6. This is the last step of the compilation phase: it exploits the slack available in the levelization within each macro-gate, reshaping macro-gates to have approximately the same number of logic gates in each level by using the algorithm of Figure 7, thus leading to a more rectangular shape. The height of the rectangle corresponds to the number of levels in the longest executing thread. We introduced the balancing step specifically to address the ‘long tail’ effect within the thread blocks resulting from the shape of macro-gates: it promotes some gates from lower levels to upper ones to fill idle thread slots such that utilization is increased. Width of the thread block may be set to be less than the maximum number of gates in a macro-gate level. Note how the balancing step introduces an inherent trade-off between latency of execution of an individual macro-gate and the utilization of threads in a thread block.

In our experimental evaluation Each thread block is 128 threads wide, a design parameter that was the result of a number of considerations: (i) the width of a thread block has to be a multiple of the warp-size (32 as per CUDA specification) (ii) obeying all other best practice utilization criteria for thread occupancy, maximal width allowed was 256, but was leading to low utilization at higher macro-gate levels and (iii) values lower than 128 resulted in very high gaps after balancing, and consequently high macro-gate latencies.

5.3 Macro-gate segmentation heuristics

The quality of macro-gate partitioning is key to simulation performance. Ideally, event-driven simulation is most efficient when only a small fraction of macro-gates are active in each simulation cycle; thus specific gates included in each macro-gate are relevant to this aspect. Macro-gates are governed by two parameters: gap and lid, which control the granularity at which event-driven mechanism operates. In this section, we will first describe a method to select ideal values for gap and lid, the two key parameters controlling the granularity of event-driven simulation, and then we will explore an alternative method to perform clustering during macro-gate formation. Finally, we will consider the scenario where gap and lid are not fixed across the entire segmentation process, but instead may vary from one macro-gate to another.

```
balance_macro-gate() {
  for each (level in height)
    for each (column in width)
      balanced_macro-gate[level][column] = select_gate();
    }
  }
  return balanced_macro-gate;
}

select_gate() {
  sort gates in macro-gate by increasing level;
  for each (gate in macro-gate) {
    if(not assigned_to_balanced_macro-gate(gate))
      return gate;
  }
}
```

Fig. 7. **Macro-gate balancing algorithm.** Macro-gates are considered one at a time and reshaped to fit into a thread block with a maximum of 128 threads, while striving to minimize the number of logic levels. The algorithm proceeds in a bottom-up fashion, filling each macro-gate level with gates while satisfying the maximum width restriction.

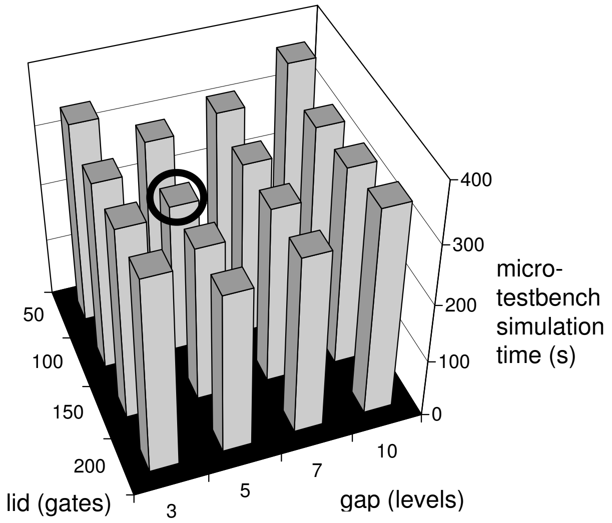


Fig. 8. **Ideal gap and lid estimation** for the LDPC testbench design. The figure shows the runtime for a micro-testbench simulation using a range of gap and lid values. The ideal gap and lid values, achieving minimum simulation time within the range considered, is highlighted by the dark circle.

The goal of gap and lid selection is to create macro-gates with low activation rates. Gap and lid values are selected during the compilation phase by evaluating a range of candidate $\langle gap, lid \rangle$ value pairs; for each candidate pair, we collect several metrics: number of macro-gates that would be generated, number of monitored nets, size of macro-gates (due to limited amount of shared memory per thread block) and activation rates. Activation rates are obtained by a simulation mock-up on a micro testbench. After this analysis, we select the locally optimal values and perform detailed segmentation. Figure 8 shows an example

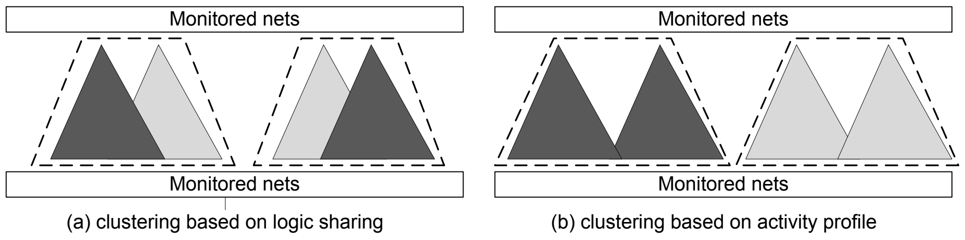

Monitored nets
Monitored nets
(a) clustering based on logic sharing
Monitored nets
Monitored nets
(b) clustering based on activity profile

Fig. 9. **Profile-based clustering** compared to baseline algorithm. The baseline clustering algorithm (a) groups cones of logic by degree of logic sharing, while profiling (b) is based on the activation frequency of logic cones. In the picture a cone's shading is proportional to its activation frequency. Clustering based on activity profile results in the consolidation of frequently activated logic cones.

of this study, reporting simulation times for the LDPC benchmark design. In the example, best performance is achieved for $\langle gap, lid\rangle$ = $\langle 5, 100\rangle$.

The range of gap values considered is 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 values, we bound the analysis by estimating how many macro-gates will be created at each layer, with a goal of running all those macro-gates concurrently in the worst case. The GPU used for our evaluation included 14 multiprocessors, while the CUDA scheduler allows at most 3 thread blocks in concurrent execution on a same multiprocessor. Thus, we only considered lid values that generate no more than $14 \times 3 = 42$ macro-gates per layer unless any other constraints are violated. Finally, note that the analysis described above must only be performed once per compilation.

5.3.1 *Macro-gate clustering based on profiling.* The policy selecting which logic cones to include in each macro-gate has a great impact on the activation rates and, consequently, on the simulator performance. Indeed, any macro-gate containing a frequently activated logic gate will result in the entire macro-gate being simulated, degrading overall performance. Thus, we strive to consolidate those logic cones with frequently activated gates into a same macro-gate by means of profiling. In this approach, representative simulations are performed to determine which gates are most frequently activated. Figure 9 shows an example of clustering based on activity profile, where cones are shaded proportional to their activation frequencies (darker shades correspond to more frequent activation). If cones were clustered by degree of logic sharing, two higher activity cones are clustered with lower activity cones, resulting in both macro-gates having high activation frequency. However, if they are clustered based on activation frequency, then both high activity cones will be placed in the same macro-gate. The result is one frequently activated macro-gate and one rarely activated, leading to better performance. A higher degree of gate replication may result from that policy, as qualitatively shown in the figure. This additional replication is amortized by the significant reduction in the total number of macro-gate simulations that must be performed.

To estimate activity profiles we use profiling, first by simulating a micro-testbench (10,000 cycles long) using the default clustering policy. During this simulation process, we aggregate additional data corresponding to activation rates of each logic cone in each layer. The activation frequency of a cone corresponds to the activation frequency of the set of its input wires, since that cone will need to be simulated if any of its input nets undergoes a value change. Moreover, note that the input nets that form the base of the cone are part of the monitored nets. Hence, we can compute the activation frequency of all cones by recording the activation frequency of all monitored nets. Once these are computed, the segmentation process is performed again, but this time cones are included in macro-gates based on activation frequency, rather than by logic sharing. We evaluate the impact of this alternate clustering method in Section 7.4. Finally, we note that this analysis needs to be performed only once during compilation.

5.3.2 *Flexible gap and lid.* Gap and lid values need not be fixed throughout the entire segmentation process. Indeed, it is typical that within each cone the lowest levels have the most switching activity, while fewer and fewer gates switch on the higher levels. Based on this observation, we define the annihilation ratio of a cone of logic as $1 - \frac{activation\ frequency(output)}{activation\ frequency(inputs)}$. That is, the ratio of cycles between, when the output of the cone does not switch while its inputs are activated. Consequently, we could devise a segmentation process which first sets an annihilation ratio, and then groups together cones that produce that ratio in some number of levels. With this segmentation each macro-gate could have a distinct gap value. We estimate adequate annihilation ratios by first performing a segmentation with a fixed gap value of one, leading to all intermediate wires to become monitored nets. A micro-testbench (10,000 cycles long) is then simulated on this segmented version and the activation frequencies of all monitored nets are recorded. The annihilation ratio of any cone of arbitrary gap can then be computed from these values. From this information, we can then apply the segmentation scheme just described and generate macro-gates with distinct gap values.

One additional challenge brought forward by this approach is a potentially irregular pattern of monitored nets in the final segmented circuit. Specifically, almost every level may have a few monitored nets, leading to a situation where switching activity of monitored nets has to be checked every few macro-gate simulations, thus impacting performance. We overcome this issue by forcing a constant gap throughout each layer, and allowing different gap on different layers. In this scenario, the gap value is set to be the number of levels that create an acceptable value of annihilation ratio or higher, for all cones in that layer. In our experiments, we found 0.25 to be an acceptable value for the annihilation ratio. The performance benefits from this technique are described in Section 7.4.

6. SIMULATION PHASE

Once the compilation process is completed, simulation can be carried out directly on the GPU co-processor. Macro-gates are simulated in an event driven fashion, alternating between execution of all active macro-gates in a layer and observation of the value changes in the monitored nets of the next layer. The gates within a macro-gate are simulated by single thread blocks in an oblivious fashion. There are two kinds of parallelism exploited in this approach: first, independent macro-gates are simulated by distinct thread blocks possibly executing concurrently on different multi-processors; and second logic gates at the same level within a macro-gate are simulated in parallel by different threads.

6.1 Event-driven simulation of macro-gates

Each macro-gate corresponds to one thread block, and each multiprocessor executes multiple thread blocks. We found experimentally that allocating 3 thread blocks for each mul-

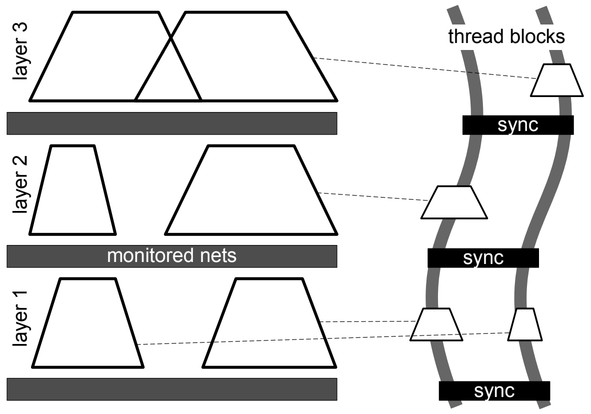


Fig. 10. **Event-driven simulation** proceeds layer by layer at the macro-gate granularity. Within each layer, activated macro-gates are simulated and monitored nets are analyzed to determine which subset of the next layer's macro-gates should be activated. Activated macro-gates are transferred by the CUDA scheduler to an available multiprocessor for simulation.

```
scheduler (layer, monitored_nets) {
  switching_monitored_nets = monitored_nets.previous
                             XOR monitored_nets.current;
  for each (macro_gate in layer){
    macro-gate.to_schedule = macro-gate.sensitivity_list
                             AND switching_monitored_nets;
    if(macro-gate.to_schedule!=0){
      active_list.append(macro_gate);
    }
  }
  return active_list;
}
```

Fig. 11. **Event-driven simulation scheduler.** The scheduling algorithm considers all macro-gates in the next layer of simulation, intersecting their sensitivity list with the monitored nets that have switched during the current simulation cycle. Macro-gates with a non-empty intersection are scheduled for simulation in the next layer.

tiprocessor provides best performance in hiding memory access latency. Indeed, while warps from one thread block is suspended waiting for data from device memory, one of the others can execute.

The CUDA scheduler is responsible for determining which multiprocessor will execute which thread blocks hence the scheduling of macro-gates; after they have been marked for simulation, is implicit. Figure 10 illustrates the layered structure of macro-gates and monitored nets, and shows how only activated macro-gates are scheduled for execution at each layer. In addition, the pseudo-code for event-driven scheduling is presented in Figure 11.

Two kernels (part of a GPU program) alternately execute on the GPU, driving the simulation. First, the simulation kernel simulates all active macro-gates in a layer. This is followed by execution of a *scheduling kernel* that evaluates the array of monitored nets to determine which macro-gates should be activated in the next layer. The monitored nets array is organized as a bit vector, with each net mapped to a memory location that is tagged

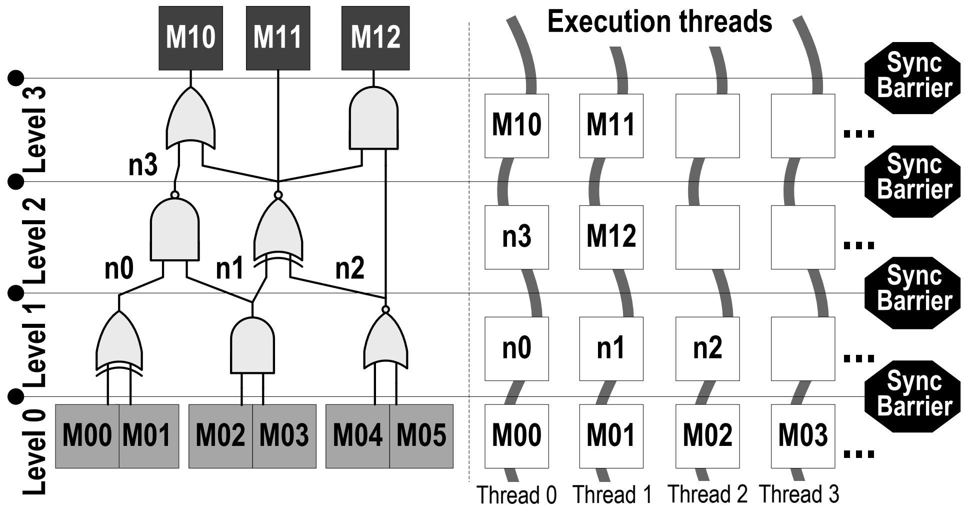


Fig. 12. **Gate simulation within a macro-gate.** The logic gates within a macro-gate are simulated in an oblivious fashion. Each thread is responsible for computing the output of one gate at a time: vertical waved lines connect the set of logic-gates for which a single thread is responsible at subsequent time intervals. Note that each level is followed by a synchronization step.

if a macro-gate simulation modifies the net's value. Correspondingly, each macro-gate has a *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 whether the macro-gate should be activated.

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). The netlist structure is also stored in device memory and accessed during each macro-gate simulation. However, all internal net values of an activated macro-gate are stored in shared memory, since they are often shared among all threads participating in the oblivious simulation of that macro-gate.

6.2 Oblivious simulation within macro-gates

Each macro-gate is simulated by a thread block and each thread within the block simulates one logic gate, one level at a time. The threads in each thread block synchronize after each level so that all output values of gates at a given level are written before the next level gates are simulated. Figure 12 illustrates this process graphically and the pseudo-code is outlined in Figure 13.

The value of each gate is computed by accessing its corresponding truth table, stored in shared memory because of its frequent access. Also, intermediate net values (outputs of internal gates) are stored in shared memory, since they are often accessed by several gates and are the most frequently accessed values. Macro-gate topology is instead stored in device memory and each single thread fetches the information for the gate it must compute. Logic gates information is stored in a matrix: the location corresponds to the position of the output net in the balanced macro-gate. Net values in shared memory follow a matching

```
simulate_macro-gate(){
  macro-gate_index=active_list[block_ID];
  for each (level) {
    launch_gate_simulation(assigned_gate[level][thread_ID]);
    sync_gate_simulation();
  }
}
```

Fig. 13. **Oblivious simulation routine for individual threads.** The routine is executed by each thread within a thread block. Each thread operates on a different set of gates, and the simulation of an entire macro-gate is complete after the last synchronization step.

layout, thus creating the scope of very regular execution suited for GPUs.

Each thread fetches the topology information corresponding to a gate, that is locations of input nets and logic function that this gate should compute. Moreover, since the balanced macro-gate has a regular structure, all such fetch operations are contiguous and thus can be coalesced to a minimum number of device memory loads. Input net values are then read from shared memory, and access to the corresponding truth table determines the output value to be written to shared memory. At the completion of a macro-gate simulation, the outputs generated are transferred to device memory for value change detection, used to determine other macro-gates activations.

6.3 Testbenches

Testbenches are a critical aspect of simulation, since simulation is only useful when the design is validated with a correct and relevant sequence of stimuli. Hence, for our GCS simulator to be useful, we need to provide methods for incorporating testbenches. Since the simulator is cycle-based, the function of the testbench is to read outputs after each simulation cycle and provide suitable inputs for the next cycle. Testbenches are implemented as a separate GPU kernel, and their execution alternates with that of the simulation proper kernels (macro-gate simulation and scheduling). At the end of each cycle, the outputs produced by the circuit are read from device memory, and suitable inputs are written for the simulation kernels to read during the next simulation cycle. There are several ways of implementing a testbench kernel, each can be ideal for different types of designs. Below we discuss two types: synthesizable testbench kernels and software kernels emulating a behavioral hardware description.

6.3.1 *Synthesizable testbenches.* When the testbench can be designed to conform to synthesizable hardware, high performance testbench kernels are possible. Since they can themselves be mapped to a netlist, the simulation can be viewed as a co-simulation between two digital circuits. If the testbench entails additional complex structures such as assertions, those can often be synthesized to a netlist, too, whose output is an output of the testbench circuit. In this latter case, assertion output signals must be monitored every cycle by the host to terminate simulation or record useful information when appropriate.

6.3.2 *Software testbenches.* In some cases, behavioral testbenches can be expressed as GPU programs operating on a memory block in device memory to create input values at every clock cycle. The best example of such testbenches are microprocessor test kernels, often implementing simple kernels that can be used when simulating a processor design. These designs are usually simulated by executing a binary program, which is uploaded

Design	Testbench	# Gates	# Flops
Alpha no pipeline	recursive Fibonacci program	17,546	2,795
Alpha pipeline	recursive Fibonacci program	18,222	2,804
LDPC encoder	random stimulus	62,515	0
Wide LDPC encoder	random stimulus	125,003	0
JPEG decompressor	1920x1080 image	93,278	20,741
3x3 NoC routers	random legal traffic	64,432	13,698
4x4 NoC routers	random legal traffic	144,098	23,875
5x5 NoC routers	random legal traffic	252,238	37,334
OpenSPARC core	OpenSPARC regression suite	262,201	62,001
OpenSPARC 2 cores	OpenSPARC regression suite	610,670	124,002
OpenSPARC 4 cores	OpenSPARC regression suite	1,221,340	248,004

Table I. **Testbench designs** used to evaluate the simulator.

to device memory. The corresponding testbench kernel simply serves memory requests from the processor. For more complex processor designs, the memory controller can be mimicked by a kernel which processes the processor's transactions. A number of other testbenches can make use of this solution, for example in the case of our experimental JPEG decompressor design, the testbench was the image to be decompressed. The image resided in device memory and the testbench kernel supplied bytes from the image to the simulated circuit. Complex testbenches involving constructs that are arduous to represent as a kernel in CUDA can still be executed on the host CPU, but an additonal communication penalty is incurred with every cycle. Debugging support can also be implemented at the cost of storing internal values in device memory and incurring the related memory latency penalty.

7. EXPERIMENTAL RESULTS

We evaluated the performance of our simulator on a broad range of designs, from combinational circuits, such as LDPC encoders, to a SPARC multiprocessor of over 1 million logic gates. The experimental designs were collected from OpenCores [OpenCores] and the Sun OpenSPARC project [OpenSPARC]. Moreover, the Alpha processors and NoC designs have been developed by student teams at the University of Michigan in advanced digital design courses.

Table I reports the key characteristics of these designs: number of gates, flip-flops and the type of testbench stimuli that was used during simulation. The first two designs are processors implementing a subset of the Alpha instruction set, the first one is non-pipelined, 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, and was fed with random stimuli. The JPEG decompressor decodes an input image. The NoC designs consist of 5-channel routers connected in a torus topology and simulated with a random stimulus generator sending legal packets through the network. Finally, the OpenSPARC designs use processor cores from the OpenSPARC T1 multi-core chip(excluding caches) and run a set of assembly regressions provided with Sun's open source distribution. Several versions of the processors were used: single-core, two cores, and four cores. The local cache activity was simulated by using playback of pre-recorded signal traces from the processor-crossbar and processor-cache interactions.

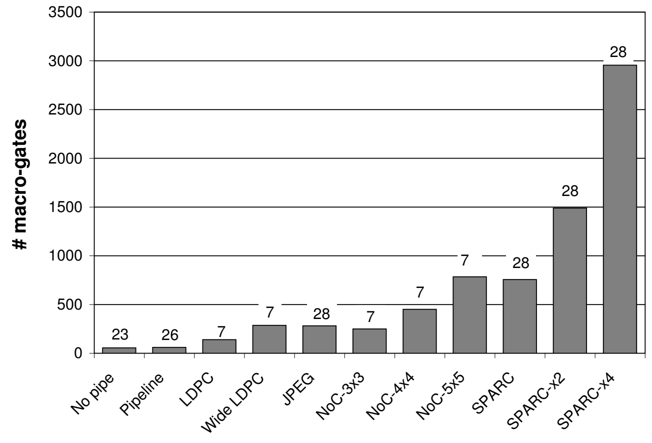


Fig. 14. **Number of macro-gates and layers.** The figure plots the total number of macro-gates for each design using the baseline segmentation algorithm. The value above each bar indicates the number of layers in the segmentation. Larger designs result in more macro-gates.

Design	gap	lid	# layers	# macro-gates	activation rate
Alpha no pipeline	7	100	23	56	18.5
Alpha pipeline	7	100	26	60	37.8
LDPC encoder	5	100	7	140	84.7
wide LDPC encoder	5	100	7	281	83.9
JPEG decompressor	5	150	28	282	40.5
3x3 NoC routers	5	100	7	250	29.1
4x4 NoC routers	5	100	7	451	29.5
5x5 NoC routers	5	100	7	709	28.9
OpenSPARC core	5	150	28	756	23.3
OpenSPARC-2 cores	5	150	28	1,489	24.9
OpenSPARC-4 cores	5	150	28	2,955	26.0

Table II. **Macro-gate segmentation statistics.** The table reports the parameters used in macro-gate segmentation for each design, the resulting number of macro-gates. It also reports the average activation rate of all macro-gates in the design over the complete GCS simulation.

7.1 Macro-gates

We studied several aspects of the compilation phase, evaluating the effect of our segmentation algorithm on the generated macro-gates. Figure 14 shows the total number of macro-gates generated for each design when using the gap and lid values determined by the baseline method presented in Section 5.1.4. The same results are also used when following the alternative policy of cone clustering based on activation profiling. On average, each macro-gate included 400 logic gates. In addition, the number of layers generated for each design are also reported above the bars in the figure.

The characteristics of the macro-gates that result from segmentation of each design, are presented in Table II. We report the gap and lid values used to segment the designs, as determined by the baseline method for choosing gap and lid, as described in Section 5.3. Additionally, the fourth column indicates the number of layers obtained by segmenting with the indicated gap. Note that the largest design, OpenSPARC-4 cores, includes many more macro-gates in each layer that could be simulated concurrently (42 as computed

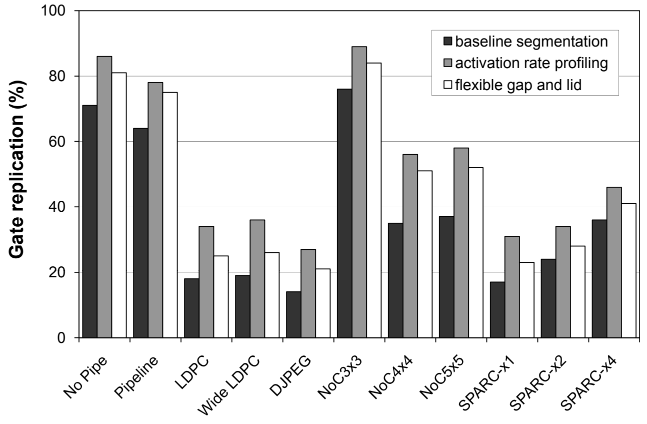


Fig. 15. **Gate replication during macro-gate segmentation.** The graph plots the increase in the number of gates in the design due to gate replication over all three segmentation algorithms discussed. The least replication is provided by the baseline algorithm.

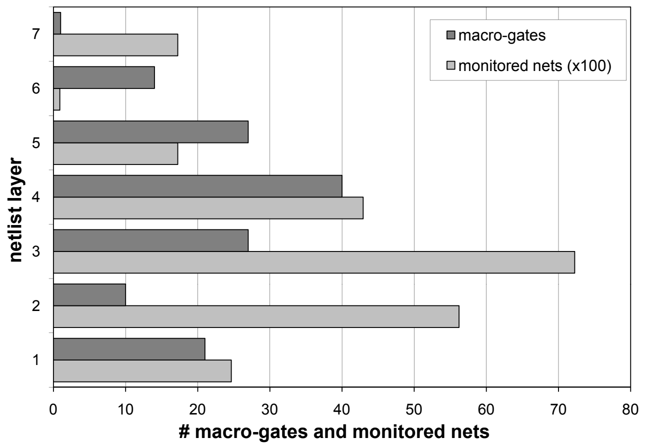


Fig. 16. **Geometry after baseline segmentation** for the LDPC design. For each layer we report the number of macro-gates as well as monitored nets in hundreds generated when segmenting the design according to baseline algorithm.

in Section 5.3), since it has 2,955 macro-gates distributed over only 28 layers. The last column reports the average activation rate of all macro-gates over the entire simulation of the design.

As discussed in Section 5.1.4, gate replication is necessary to limit communication among multiprocessors. As a result the total number of gates after macro-gate segmentation process increases. Figure 15 reports this increase as a percentage over the baseline number of gates for all three segmentation algorithms presented. The main goal of the baseline segmentation algorithm is that of minimizing gate replication: correspondingly, Figure 15 reports the lowest gate increase for this algorithm. The other two policies focus on minimizing activation rates and present a high rate of gate replication.

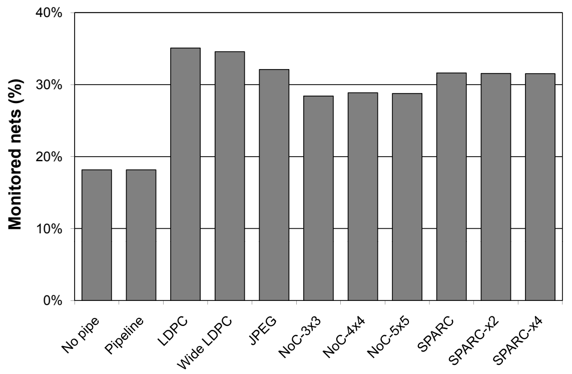


Fig. 17. **Percentage of monitored nets.** Percentage of all nets that are monitored for each testbench design after applying baseline segmentation.

7.2 Monitored nets

The number of monitored nets has a high impact on simulator performance because they must be checked after each layer of simulation, thus, segmentation strives to keep the fraction of nets to be monitored low. As an example, in Figure 16 we plot the structure of the LDPC encoder design after segmentation: the number of macro-gates and monitored nets are shown for each layer. Note that, middle layers have more macro gates. In contrast lower layers tend to generate the most monitored nets as a side effect of ALAP levelization.

We also analyzed monitored nets as a fraction of total nets in the design. Figure 17 reports our findings over all experimental designs after the segmentation phase. We note that the simpler designs, the pipelines, presents a markedly smaller percentage of monitored nets: this is due to the small number of macro-gates in these designs.

7.3 Macro-gate activation

The activation rate of macro-gates is an important metric for event-driven simulation. The goal of an event-driven simulator is to keep the activation rates at a minimum, so as to minimize the number of macro-gates simulated at every clock cycle. In order to achieve this, the segmentation algorithm should generate macro-gates so that the fraction that is frequently active is small.

In Figure 18, we plot the cumulative distribution of macro-gates with respect to their activation rates, when applying the baseline segmentation policy. For example, the non-pipelined processor design (No pipe) shows that 80% of its macro-gates are active for only 20% or less of the total simulation cycles. Most of the other designs follow a similar trend, with the majority of the macro-gates having an activation rate between 10 and 30%. Thus, the baseline segmentation algorithm partitions the design effectively for event-driven simulation. An exception to the common trend is the LDPC design, where most macro-gates experience a high activation rate ($>$ 80%). In this design, most logic gates are active during each simulation cycle, thus leading to exceptionally high macro-gate activation rates. The designs not reported in the Figure present cumulative distributions similar to those of the OpenSPARC and NoC designs.

Finally, in Figure 19 we evaluate the impact of activation rates on the performance of our GCS simulator. For each of our experimental designs the scatter plot reports their average activation rate and speed up over the sequential simulator. Note that overall there

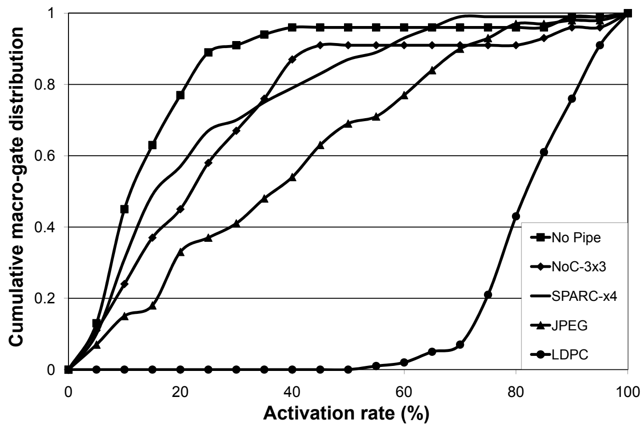


Fig. 18. **Cumulative distribution of macro-gates** with respect to activation rate for the baseline segmentation algorithm. The plots show the fraction of macro-gates with an activation rate below the threshold indicated on the X axis. Note that for most designs, the majority of macro-gates have activation rates below 30%.

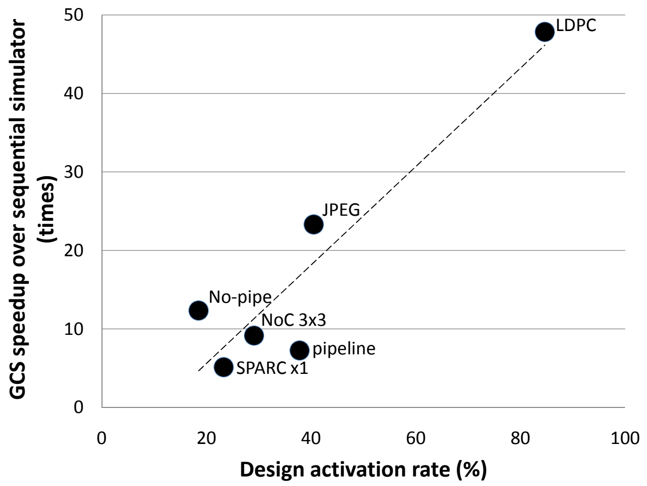
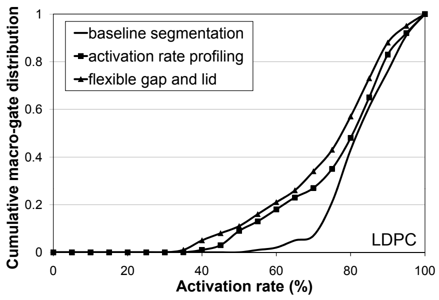


Fig. 19. **GCS relative speedup versus activation rate.** The scatter plot shows the relation between these two parameters for a representative subset of our experimental designs indicating an approximately linear relation.

is approximately a linear relation between these two parameters. At first this trend may seem counter-intuitive, because we would expect lower performance from designs with high activation rates. However note that high activation rates impair the performance of the sequential simulator because may logic gates are concurrently available for execution, but only one can be simulated at a time. In contrast GCS can schedule many macro-gates concurrently, gaining further performance speedup over a sequential simulator.

7.4 Impact of macro-gate segmentation heuristics

We now examine the impact of different segmentation algorithms on activation rates. Using two representative designs, LDPC and JPEG, in Figure 20 we plot the cumulative distribution of macro-gates with respect to activation rates. The plots show all three algorithms presented: baseline segmentation (Section 5.1.4), activity profiling (Section 5.3.1)



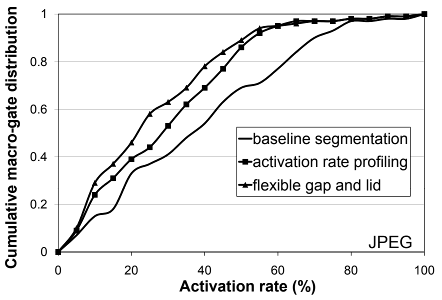


Fig. 20. **Cumulative distribution of macro-gates** with respect to activation rates for LDPC and JPEG designs. The plot shows the trends for all three segmentation heuristics presented: baseline segmentation, activation rate profiling and flexible gap and lid. The flexible gap and lid heuristic performs the best with the greatest improvement on macro-gates with low activation rates.

	GPU-based simulator			Commercial simulator
Design	system level(s)	balancing(s)	total(s)	(s)
Alpha no pipeline	13	8	21	7
Alpha pipeline	21	15	36	13
LDPC encoder	78	47	125	63
JPEG decompressor	245	92	337	156
3x3 NoC routers	212	164	376	189
4x4 NoC routers	302	130	432	237
OpenSPARC core	456	187	643	275
OpenSPARC-2 cores	873	259	1,132	504
OpenSPARC-4 cores	1,670	575	2,245	1,278

Table III. **Compilation performance.** The table compares the time in seconds required for compilation, for both our GCS simulator (with baseline policy) as well as a commercial sequential simulator.

and flexible gap and lid (Section 5.3.2). Note that the baseline segmentation line is the same as in Figure 18.

Note that in both plots the activation rate profiling algorithm brings a noticeable advantage leading to a larger fraction of macro-gates with lower activation rates. This is the result of gathering cones with high activation rates together in a small number of macro-gates. Lower activation rates lead to fewer macro-gate simulations and fewer device memory accesses, with a resulting 1–12% performance improvement over the use of baseline segmentation (as shown in Table IV).

The flexible gap and lid algorithm present an even more marked improvement, with an overall performance benefit of up to 15% over the baseline approach (see Table IV). However this significantly more complex algorithm also brings an increase in compilation time.

7.5 Design compilation

We now consider the time spent compiling the design for simulation on the GPU-based target. Table III reports the times for system-level compilation and for the balancing phase. The table also provides the total compilation time and compares it against that of a commercial logic simulator executing on a general purpose machine. The commercial simulator that we used for the comparison is considered among the fastest available in the market today. Note that the compilation time is not a critical aspect in the performance of a sim-

Design	simulation cycles	sequential time(sec)	GCS simulator: baseline segmentation time (sec)	baseline segmentation speedup (times)	activation rate time (sec)	activation rate % extra speedup	flexible gap and lid time (sec)	flexible gap and lid % extra speedup
Alpha no pipe	12,889,495	31,678	2,567	**12.34x**	2,435	**5.14%**	2,401	**6.47%**
Alpha pipeline	13,423,608	54,789	7,781	**7.04x**	7,561	**2.83%**	7,523	**3.32%**
LDPC encoder	1,000,000	115,671	2,578	**44.87x**	2,421	**6.09%**	2,345	**9.04%**
	10,000,000	>48 hrs	25,973	**43.49x**	24,321	**6.36%**	23,451	**9.71%**
wide LDPC enc.	1,000,000	257,891	5,832	**44.22x**	5,761	**1.22%**	5,667	**2.83%**
	10,000,000	>72 hrs	25,973	**43.29x**	2,567	**1.44%**	2,567	**2.85%**
JPEG decomp.	2,983,674	12,146	599	**20.28x**	521	**13.02%**	509	**15.03%**
3x3 NoC routers	1,967,155	3,532	397	**8.90x**	386	**2.77%**	374	**5.79%**
4x4 NoC routers	10,000,001	28,867	3,935	**7.34x**	3,842	**2.36%**	3,774	**4.09%**
5x5 NoC routers	10,000,001	48,113	6,789	**7.09x**	6,654	**1.99%**	6,451	**4.98%**
SPARC core x1	1,074,702	27,894	6,077	**4.59x**	5,451	**10.30%**	5,321	**12.44%**
SPARC core x2	1,074,702	40,378	8,229	**4.91x**	7,456	**9.39%**	7,342	**10.78%**
SPARC core x4	1,074,702	61,678	10,983	**5.62x**	10,005	**8.90%**	9,876	**10.08%**

Table IV. **GCS simulator performance.** Performance comparison between our GCS event-driven simulator and a commercial sequential simulator. Our prototype simulator outperforms the commercial simulator by 13 times on average when using the baseline segmentation algorithm. Further performance improvements are obtained when using activation rate profiling and flexible gap and lid segmentation. The speedup reported for these techniques is relative to using baseline segmentation.

ulator, since this time can be amortized over many simulations and over very long (hours or days) simulation runs. However, we felt it was relevant to convey information on the approximate time scale of compilation performance.

7.6 Performance evaluation

Finally, we evaluated the overall performance of our GCS event-driven simulator against a multi-threaded, commercial, event-driven, sequential simulator. Our graphics coprocessor was a CUDA-enabled NVIDIA 8800GT GPU with 14 multiprocessors equipped with 512MB of device memory. The GPU operated at 600MHz for the processors and at 900MHz for the memory. The current implementation has 83% occupancy and achieves a bandwidth of 20.4 GB/s when transferring data to device memory. The commercial simulator executed on a 2.4 GHz Intel Core 2 Quad running Redhat Enterprise Linux 5, leveraging four parallel simulation threads. The GPU simulator performed compilation on the same host machine used by the commercial simulator, and was connected to it as well. For each design, Table IV reports the number of cycles simulated, the runtime in seconds for both the GPU-based simulator and the commercial simulator (compilation times are excluded), and the relative speedup for all three types of segmentation heuristics. Note that our prototype simulator outperforms the commercial simulator by 4 to 44 times when using baseline segmentation. Performance can be improved by up to 15% in certain cases by using our more advanced segmentation techniques. Despite the LDPC encoder having a very high activation rate, we report the best speedup for this design. As mentioned before, most logic gates in this design are switching in each cycle: this affects our activation rates, but also hampers the sequential simulator performance. Thus, the speedup obtained is due

Design	GPU bound (sec)	CPU bound (sec)	Relative CPU time (%)
Alpha pipeline	7,564	217	2.87
JPEG decompressor	576	23	3.99
3x3 NoC routers	385	12	3.11
OpenSPARC core	5,665	419	7.40

Table V. **Time spent on CPU and GPU** execution during simulation for a few representative designs. The time spent in CPU is due to data transfers and kernel switching.

to the sheer parallelism of our simulator architecture and underlying hardware.

For all designs used in our evaluation the testbenches are implemented as GPU kernels that alternate execution with the simulation kernels. The data sets needed by the testbenches are also stored in the GPU device memory. Hence, the time spent in CPU computation during the simulation phase is minimal: it includes the sum of (i) the time spent in transferring all data sets for simulation and testbench to the device memory, (ii) copying it back after simulation ends and (iii) the penalty for invoking different kernels when switching between simulation proper and testbench execution. Quantitative values corresponding to these times are presented in Table V for a few representative designs. Overall, the time spent on the CPU accounted for less than 8% of the execution for all testbenches. For the SPARC testbenches, which are closest to the 8% cap, large memory transfers were necessary for the long replay traces.

8. CONCLUSIONS

In this work, we have presented GCS, a novel GPU-based logic simulator that leverages the high degree of parallelism available in general purpose GPUs. By exploring the concurrency in the simulation of gate-level netlists and providing a number of optimizations tuned for the underlying GPU architecture, we could realize a 13 times speedup over commercial-strength sequential implementations of logic simulation, on average.

The simulator is event-driven at system-level and oblivious within each block of logic, thus capturing both the concurrency benefits provided by SIMT execution on a GPU, as well those of event-driven simulation in simulating only a small fraction of a netlist's gates. Our simulator carves out macro-gates(that is, blocks of logic) from the structural netlist of a design and schedules them for simulation on the multiprocessors of the NVIDIA CUDA architecture, only if and when they are activated by switching events at their inputs.

We show in our experimental results that GCS is capable of delivering a remarkable performance speedup on large, industrial-scale designs of over a million gates, thus pushing further away the limits of validation for the digital design industry.

REFERENCES

AMD 2008. *ATI Stream Technology*. AMD.

Babb, J., Tessier, R., Dahl, M., Hanono, S., Hoki, D., and Agarwal, A. 1997. Logic emulation with virtual wires. *IEEE Trans. on CAD 16,* 6, 609–626.

Baker, W., Mahmood, A., and Carlson, B. 1996. Parallel event-driven logic simulation algorithms: tutorial and comparative evaluation. *IEEE Journal on Circuits, Devices and Systems 143,* 4, 177–185.

Barzilai, Z., Carter, J., Rosen, B., and Rutledge, J. 1987. HSS–a high-speed simulator. *IEEE Trans. on CAD 6,* 4, 601–617.

Bauer, H. and Sporrer, C. 1993. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. In *Proc. Annual Simulation Symposium*. 12–20.

BERRY, O. AND LOMOW, G. 1986. Speeding up distributed simulation using the time warp mechanism. In *Proc. of Workshop on Making Distributed Systems Work*. 1–14.

BRYANT, R., BEATTY, D., BRACE, K., CHO, K., AND SHEFFLER, T. 1987. COSMOS: a compiled simulator for MOS circuits. In *Proc. Design Automation Conference*. 9–16.

CHANDY, K. AND MISRA, J. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. ACM 24,* 4, 198–206.

CHATTERJEE, D., DEORIO, A., AND BERTACCO, V. 2009a. Event-driven gate-level simulation with GP-GPUs. In *Proc. Design Automation Conference*. 557–562.

CHATTERJEE, D., DEORIO, A., AND BERTACCO, V. 2009b. GCS: High-performance gate-level simulation with GP-GPUs. In *Proc. Design, Automation and Test in Europe*. 1332–1337.

DENG, Y. S., WANG, B. D., AND MU, S. 2009. Taming irregular EDA applications on GPUs. In *Proc. International Conference on Computer-Aided Design*. 539–546.

DENNEAU, M. 1982. The Yorktown simulation engine. In *Proc. Design Automation Conference*. 55–59.

EDENFELD, D., KAHNG, A. B., RODGERS, M., AND ZORIAN, Y. 2004. 2003 technology roadmap for semiconductors. *IEEE Computer 37,* 1, 47–56.

FRANK, E. 1986. Exploiting parallelism in a switch-level simulation machine. In *Proc. Design Automation Conference*. 20–26.

FUJIMOTO, R. 1990. Parallel discrete event simulation. *Comm. ACM 33,* 10, 30–53.

GULATI, K., CROIX, J. F., KHATRI, S. P., AND SHASTRY, R. 2009. Fast circuit simulation on graphics processing units. In *Proc. Asia and South Pacific Design Automation Conference*. 403–408.

GULATI, K. AND KHATRI, S. 2008. Towards acceleration of fault simulation using graphics processing units. In *Proc. Design Automation Conference*. 822–827.

GULATI, K. AND KHATRI, S. P. 2009. Accelerating statistical static timing analysis using graphics processing units. In *Proc. Asia and South Pacific Design Automation Conference*. 260–265.

KARTHIK, S. AND ABRAHAM, J. A. 1992. Distributed VLSI simulation on a network of workstations. In *Proc. International Conference on Computer Design*. 508–511.

Khronos Group. Khronos Group.

KIM, H. K. AND CHUNG, S. M. 1994. Parallel logic simulation using time warp on shared-memory multiprocessors. In *Proc. International Symposium on Parallel Processing*. 942–948.

KIM, Y.-I., YANG, W., KWON, Y.-S., AND KYUNG, C.-M. 2004. Communication-efficient hardware acceleration for fast functional simulation. In *Proc. Design Automation Conference*. 293–298.

LEWIS, D. 1991. A hierarchical compiled code event-driven logic simulator. *IEEE Trans. on CAD 10,* 6, 726–737.

LIU, Y. AND HU, J. 2009. GPU-based parallelization for fast circuit optimization. In *Proc. Design Automation Conference*. 943–946.

MANJIKIAN, N. AND LOUCKS, W. 1993. High performance parallel logic simulations on a network of workstations. In *Proc. of Workshop on Parallel and Distributed Simulation*. 76–84.

MATSUMOTO, Y. AND TAKI, K. 1992. Parallel logic simulation on a distributed memory machine. In *Proc. European Design Automation Conference*. 76–80.

MEISTER, G. 1993. A survey on parallel logic simulation. Tech. rep., University of Saarland, Dept. of Computer Science.

MISRA, J. 1986. Distributed discrete-event simulation. *ACM Computing Surveys 18*, 39–65.

NVIDIA 2007. *CUDA Compute Unified Device Architecture*. NVIDIA.

OpenCores. http://www.opencores.org/.

OpenSPARC. Sun Microsystems OpenSPARC. http://opensparc.net/.

PERINKULAM, A. AND KUNDU, S. 2007. Logic simulation using graphics processors. In *Proc. International Test Synthesis Workshop*.

SHI, J., CAI, Y., HOU, W., MA, L., TAN, S. X.-D., HO, P.-H., AND WANG, X. 2009. GPU friendly fast poisson solver for structured power grid network analysis. In *Proc. Design Automation Conference*. 178–183.

SMITH, S., UNDERWOOD, W., AND MERCER, M. R. 1987. An analysis of several approaches to circuit partitioning for parallel logic simulation. In *Proc. International Conference on Computer Design*. 664–667.

SOULÉ, L. AND BLANK, T. 1988. Parallel logic simulation on general purpose machines. In *Proc. Design Automation Conference*. 166–171.