High Performance Gate-level Simulation with GP-GPU Computing

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

***Abstract*— Functional verification of modern digital designs is a mission critical and time-consuming task. Verification is essential since it ensures the correctness of the final product; however, due to the complexity of modern designs, verification has become the primary bottleneck of time to market. Logic simulation forms the core of most current verification efforts, as it is almost ubiquitously used to verify the functional correctness of a design over a broad range of abstraction levels. In particular at the gate-level granularity, a design consists of millions of logic primitives, thus making simulation excruciatingly slow. In mainstream industry setups, complex digital systems are validated by distributing logic simulation tasks among vast server farms for weeks at a time. Yet, the performance of simulation keeps falling behind the demand, leading to incomplete verification and escaped functional bugs. It is no surprise that the EDA and semiconductor industries are always seeking for faster simulation solutions.**

Recently, advances in graphic processing unit (GPU) technology has made GPUs emerge as a cost effective parallel processing solution. Modern GPUs offer vast execution parallelism, which is a natural fit for the inherent parallel structure of gate-level netlists. Based on these observations, we propose novel algorithms for the efficient mapping of large netlists to the concurrent architecture of GPU hardware. Our GCS simulation architecture maximizes the utilization of concurrent hardware resources, while minimizing expensive communication overhead using a novel hybrid simulation method. Experimental results indicate that our GPU-based gate-level simulator delivers an order-of-magnitude performance improvement, on average, over commercial simulators on industrial-size designs.

I. Introduction

Logic simulation is of paramount importance in the development of modern VLSI circuits. A variety of design aspects, including the functional correctness of the system, are primarily validated by logic simulation. A major fraction of time and resources in big design houses are spent precisely on this task [1], to ensure that the design meets the original functional specification. Large server farms, comprising thousands of machines, execute billions of cycles of simulation for months at a time. The simulation of gate-level netlists, in particular is extremely computation-intensive, as it involves large netlists presented in a low-level description, comprising millions of logic primitives (gates). The main purpose of logic simulation is to evaluate the output sequence of a design over time in response to a set of input stimuli, usually provided by a testbench program on per-cycle basis. Responses are validated with a range of techniques, including assertions, functional checkers and/or by comparison with a golden model. Based on their internal architecture, simulators can be grouped into two families: *oblivious* simulators compute the output of all gates in the system during each simulation cycle, and thus benefit from simple software structure. In contrast, *event-driven* simulators use a dynamic runtime scheduler to selectively compute only those gates whose inputs have not changed since the previous cycle, and thus are more efficient by construction. The performance of oblivious simulators suffer from the redundant computation of all gates whose input values have not changed in the latest simulation cycle; while the dynamic scheduler of event-driven simulators entail more software complexity and scheduling overhead. Today, the event-driven approach is more common in commercial simulators, as only 1 to 10% of all gates switch in any given cycle, and thus the scheduling overhead is more than compensated for.

A. Bringing GPU acceleration to simulation

Graphics processing units consist of a large number of simple and identical compute units capable of executing same instruction sequence on different data sets, and are typically deployed in the fast execution of graphics primitives in a parallel fashion. General-purpose GPU computing has emerged as a recent extension of traditional graphics processing, providing a general purpose programming interface for the GPU. This recent trend has made the vast parallel computation resources available to general purpose applications that are amenable to a single-instruction multiple-thread (SIMT) structure. Platforms for GPU computing include AMD's FireStream [2] and NVIDIA's CUDA [3]. In addition, vendor-independent parallel computing standards, such as OpenCL [4], have also been developed. Many EDA applications such as fault simulation [5], power grid analysis [6], etc., present an algorithmic structure suitable for GP-GPU. In these applications, a same execution sequence must be applied to different sets of data, hence the available parallelism is explicit.

In investigating the potential for large performance improvements in logic simulation, we found that large gate-level netlists enjoy a high degree of structural parallelism. In a levelized netlist, many gates can be independently simulated in a concurrent fashion, making it suitable for the type of parallelism of GPUs. However, the constraints of the GPU hardware dictate distributing the computation over several multi-processors, each with limited shared memory space and a high penalty of global synchronization. Hence the netlist must be partitioned so that the implicit parallelism available in the application is explicitly mapped to the hardware to attain efficient execution. Moreover the event-driven approach to simulation, which provides great performance benefits in sequential computation environments, presents an additional challenge in the massively parallel computation environment of the GPU: event scheduling is an intrinsically sequential process, and hence can cause a large overhead in a parallel setting. As a result, we developed our GPU-based simulator as a unique hybrid simulator where a design is partitioned into clusters of gates (called *macro-gates*), with clusters being simulated in an oblivious fashion in each multi-processor, while the scheduling of individual clusters is organized in an event-driven fashion. GCS strives to deliver the best gate-level simulation performance under the constraints of the underlying GPU architecture: consequently, the segmentation algorithm, the simulation data structures and the execution flow are tuned to the nuances of the GPU execution model.

II. Related Work

For several decades the majority of verification effort in the industry has revolved around logic simulators. Initial work from the 1980s addressed several key algorithmic aspects that are still utilized today in modern solutions, including netlist compilation, design of event-driven schedulers, propagation delays, *etc.* [7]. The exploration of parallel algorithms for simulation started approximately at the same time [8], targeting both shared memory multiprocessors [9] and distributed memory systems [10]. In these solutions, individual execution threads operate on distinct netlist clusters and communicate in an event-driven fashion, with a thread being activated only if switching activity is observed at the inputs of its netlist cluster. In particular, [8] provides a comparative analysis of early attempts to parallelize event-driven simulation by partitioning the processing of individual events across multiple machines using a fine granularity.

Specialized hardware solutions (*emulation systems*) have also been explored to boost simulation performance. These systems typically consist of several identical hardware units connected together, with individual 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 a distinct unit [11], [12]. Modern emulators can deliver 3-4 orders of magnitude speedup over software simulators and can handle very large designs. However, their cost is prohibitively large and the process of successfully mapping a netlist to an emulator can take up to a few months.

The effort of parallelizing simulation algorithms has only recently targeted data-streaming architectures (single-instruction multiple-thread), as in the solution proposed by [13]; however, the communication overhead of that system had an overwhelming impact on its overall performance. Since the emergence of efficient software interfaces for general purpose programming of GPUs, a collection of research works targeting the acceleration of a range of EDA applications has appeared in the literature. These solutions can be classified in the following three broad categories:

a) Pattern parallelism: These problems demand application of a same execution pattern on a same data structure using different set of values. A classic example of this type of solution is [5], which proposes to parallelize fault simulation on a CUDA GPU target. It achieves its concurrency by simulating distinct fault patterns on distinct processing units, with no partitioning within individual simulations or within the design. Other examples include power grid analysis [6] and statistical timing analysis [14].

b) Concurrency from problem structure: In contrast to pattern parallelism, applications in this category extract parallelism by leveraging alternative algorithms suitable for efficient execution on a GPU platform. GCS simulator is an example of such approach, as we target fast simulation of complex designs, leveraging specialized algorithms to partition the design and target parallel processing elements to maximize memory locality in both oblivious [15] and event-driven [16] simulation architectures. The specific choice of partitioning algorithm is key to the performance of this approach, *e.g.*, our solution relies on a variant of cone partitioning [17] tailored to the constraints of our target architecture. Moreover the underlying irregular data structures (*e.g.* large graphs) must be reorganized and made more regular to be suitable for GPU execution. Other examples of this category include fast circuit optimization [18].

c) Parallelization of core algorithms: Utilizing GPU computing to accelerate core algorithms that are shared across many EDA applications has also been proposed. Occasionally, algorithms which are not optimal in a sequential environment are found to be better candidates in a GPU setting, such as in [19]. Algorithms that provide concurrency without sacrificing asymptotic complexity are also proposed, such as in [20].

III. Introduction to CUDA

General purpose computing on graphics processing unit enables parallel processing on commodity hardware. NVIDIA's Compute Unified Device Architecture (CUDA) is a hardware architecture and complementary software interface to design data parallel programs executing on a GPU. According to the CUDA model, a GPU is a co-processor capable of executing a parallel task off-loaded from primary CPU. In CUDA, this data parallel computation process, is called a "kernel". The model of execution is known as single-instruction multiple-thread (SIMT), where thousands of threads execute the same code, while operating on different data values. Each thread can identify itself by thread ID and thread-block ID, and can thus access the data assigned to it.

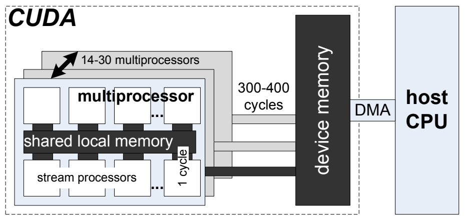


Fig. 1. **The NVIDIA CUDA architecture** The GPU contains a set of multiprocessors, all multiprocessors have access to a global device memory and to a dedicated shared memory block.

The CUDA architecture [3] (Figure 1) consists of several multiprocessors (14-30 in current generations) packaged in a single GPU chip. Each multiprocessor comprises 8 or more stream processors and can execute up to 1,024 concurrent threads, all running exactly the same code. The block of threads contained in one multiprocessor has exclusive access to up to 48KB of shared memory, at an access latency of 1 clock cycle. All multiprocessors also have access to a larger global memory block, ranging in size from 256 MB to 1 GB, but with higher access latency (300-400 cycles). It is possible to amortize the cost of accessing global memory by coalescing accesses from several threads. It is also possible to transfer data from the host's main memory to device memory, preferably in large blocks, since the communication is through DMA interface.

IV. Simulator overview

GCS is designed as a hybrid event-driven simulator, combining the advantages of dynamic gate scheduling with the requirements of the underlying GPU architecture, entailing uniform control flows. It draws inspiration from both methods of logic simulation: oblivious and event-driven, with design decisions being driven by the constraints of the GPU platform. A design to be simulated with GCS is first partitioned into several clusters of gates, called macro-gates. Simulation of each macro-gate is carried out in oblivious fashion by individual thread-blocks taking advantage of the dedicated shared memory and local synchronization; 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

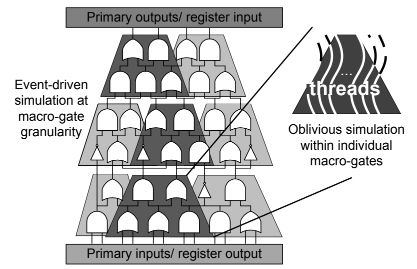

Primary outputs/ register input
Event-driven simulation at macro-gate granularity
threads
Oblivious simulation within individual macro-gates
Primary inputs/ register output

Fig. 2. **Hybrid event-driven simulator.** The GCS architecture is event-driven at the granularity of macro-gates, while individual 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.

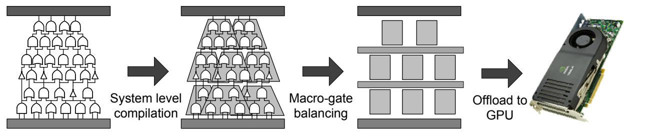


Fig. 3. **Simulator organization.** A compilation step produces macro-gates, which are optimized and offloaded to the GPU for simulation.

Figure 2, where the netlist's logic-gates are grouped into macro-gates, Figure 2 also shows an example of macro-gate activation by the event-driven mechanism.

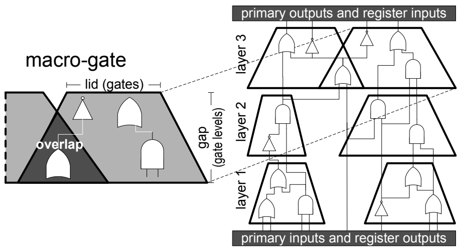
The GCS simulator is organized as a compiled code simulator, first performing a compilation process to convert the netlist into internal data structures to be mapped to the CUDA memory hierarchy, and then transferring the compiled data structures to the GPU. During the simulation phase, the CUDA-mapped design is simulated based on the input stimuli provided by the validation testbench. 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. The second phase is *macro-gate balancing*: during this phase each macro-gate is reshaped for optimal latency of execution on the GPU platform. After the data structures are prepared, they are offloaded to the GPU for the simulation proper.

V. Compilation

The GCS compiler processes a gate-level netlist as input and generates kernels that can be executed on a GPU. The combinational portion of the flattened netlist is extracted and levelized in an as-late-as-possible (ALAP) fashion, to apply macro-gate segmentation.

A. 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.


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.

Three important factors govern the macro-gate formation process: (i) The objective of forming macro-gates is to perform event-driven simulation at a coarse granularity: hence the simulation time of a macro-gate should be substantially larger than the overhead to decide which macro-gate to activate. (ii) GPU multiprocessors should not communicate among themselves, 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, to avoid cyclic dependencies between macro-gates, the netlist must be levelized at the granularity of macro-gates as well.

To address the above list of constraints, we segment the netlist by partitioning it into *layers*: each layer encompasses a fixed number of 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). 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. The baseline policy attempts to cluster together cones of influence of those nets with the most number of gates in common. Figure 4 shows a schematic of the segmentation technique. 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.

B. 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 has changed, in which case simulation of the entire macro-gate is carried out in an oblivious manner 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, level-by-level. However, the segmentation procedure tends to generate macro-gates with a large base (many gates) and a narrow tip. Correspondingly, we have many active threads in the lower levels, and just a few in the top levels. To maximize concurrency throughout the simulation, we optimize each macro-gate individually with a *balancing* step, as outlined in the schematic of Figure 5. This is the last step of the compilation phase: it exploits the slack available in the levelization within each macro-gate and restructures macro-gates to have approximately the same number of logic gates in each of their levels. The balancing step introduces a trade-off between latency of execution of an individual macro-gate and the utilization of threads in a thread block.

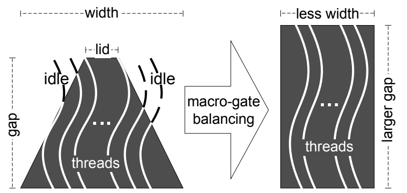


Fig. 5. **Macro-gate balancing.** The balancing algorithm restructures each macro-gate to minimize the number of execution threads required. The result is a more efficient utilization of thread block resources.

VI. 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 multiprocessors; and second, logic gates at the same level within a macro-gate are simulated in parallel by different threads.

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. Two kernels (GPU programs) alternate execution on the GPU, driving the simulation. First, the simulation kernel simulates all active macro-gates in a layer. This is followed by the execution of a *scheduling kernel*, that evaluates the array of monitored nets to determine which macro-gates should be activated in the next layer.

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 thread block synchronize after each level, so that all output values of gates at a given level are written before the next level is simulated. Figure 6 illustrates this process graphically.

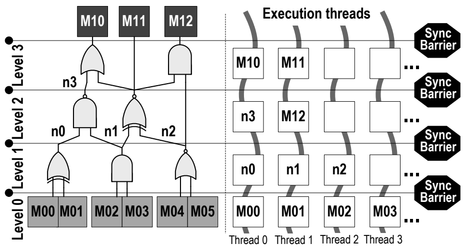


Fig. 6. **Gate simulation within a macro-gate.** The logic gates within a macro-gate are simulated in an oblivious fashion. Each thread is responsible for computing the output of one gate at a time: vertical waved lines connect the set of logic-gates for which a single thread is responsible at subsequent time intervals. Note that each level is followed by a synchronization step.

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 layout, thus creating the scope of very regular execution suited for GPUs. Each thread fetches the topology information corresponding to a gate, that is the 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.

VII. Experimental Results

Finally, we evaluated the performance of our prototype event-driven simulator against that of a commercial, event-driven sequential simulator on a broad set of designs, ranging from purely combinational circuits such as an LDPC encoder, to a multicore SPARC design containing over 1 million logic gates. Our graphics coprocessor was a CUDA-enabled 8800GT GPU with 14 multiprocessors and 512MB of device memory, operating at 600 MHz for the cores and 900MHz for the memory. The commercial simulator was run on a 2.4 GHz Intel Core 2 Quad running RH-EL5, enabling 4 parallel simulation threads. For each design, Table I reports the number of cycles simulated, the runtimes in seconds for both the GPU-based simulator and the commercial simulator (compilation times are excluded), and the relative speedup. Note that our prototype simulator outperforms the commercial simulator by 4 to 44 times. Despite the LDPC encoder having a very high activation rate, we report the best speedup for this design. As mentioned before, most gates in this design are switching in each cycle: this affects our activation rate but hampers the sequential simulator performance. Thus, the speedup obtained is due to sheer parallelism of our architecture.

design	sim cycles	seq sim(s)	GPU sim(s)	speed up
Alpha no pipeline	12,889,495	31,678	2,567	**12.34x**
Alpha pipeline	13,423,608	54,789	7,781	**7.04x**
LDPC encoder	1,000,000	115,671	2,578	**44.87x**
	10,000,000	>48h	25,973	**43.49x**
JPEG decompressor	2,983,674	12,146	599	**20.28x**
3x3 NoC routers	1,967,155	3,532	397	**8.90x**
4x4 NoC routers	10,000,001	28,867	3,935	**7.34x**
sparc core x1	1,074,702	27,894	6,077	**4.59x**
sparc core x2	1,074,702	40,378	8,229	**4.91x**
sparc core x4	1,074,702	61,678	10,983	**5.62x**

TABLE I

GP-GPU simulator performance. Performance comparison between our CUDA-based event-driven simulator and a commercial event-driven simulator.

References

[1] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian, “2003 technology roadmap for semiconductors,” *IEEE Computer*, vol. 37, no. 1, pp. 47–56, 2004.

[2] *ATI Stream Technology*, AMD, 2008.

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

[4] Khronos Group. [Online]. Available: http://www.khronos.org/opencl/

[5] K. Gulati and S. Khatri, “Towards acceleration of fault simulation using graphics processing units,” in *Proc. DAC*, June 2008, pp. 822–827.

[6] J. Shi, Y. Cai, W. Hou, L. Ma, S. X.-D. Tan, P.-H. Ho, and X. Wang, “GPU friendly fast poisson solver for structured power grid network analysis,” in *Proc. DAC*, 2009, pp. 178–183.

[7] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, “COSMOS: a compiled simulator for MOS circuits,” in *Proc. DAC*, 1987, pp. 9–16.

[8] W. Baker, A. Mahmood, and B. Carlson, “Parallel event-driven logic simulation algorithms: tutorial and comparative evaluation,” *IEEE Journal on Circuits, Devices and Systems*, vol. 143, no. 4, pp. 177–185, 1996.

[9] H. K. Kim and S. M. Chung, “Parallel logic simulation using time warp on shared-memory multiprocessors,” in *Proc. International Symposium on Parallel Processing*, Apr 1994, pp. 942–948.

[10] Y. Matsumoto and K. Taki, “Parallel logic simulation on a distributed memory machine,” in *Proc. European Design Automation Conference*, Mar 1992, pp. 76–80.

[11] M. Denneau, “The Yorktown simulation engine,” in *Proc. DAC*, June 1982, pp. 55–59.

[12] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, “Logic emulation with virtual wires,” *IEEE Trans. on CAD*, vol. 16, no. 6, pp. 609–626, 1997.

[13] A. Perinkulam and S. Kundu, “Logic simulation using graphics processors,” in *Proc. International Test Synthesis Workshop*, March 2007.

[14] K. Gulati and S. P. Khatri, “Accelerating statistical static timing analysis using graphics processing units,” in *Proc. ASPDAC*, 2009, pp. 260–265.

[15] D. Chatterjee, A. DeOrio, and V. Bertacco, “GCS: High-performance gate-level simulation with GP-GPUs,” in *Proc. DATE*, 2009, pp. 1332–1337.

[16] D. Chatterjee, A. DeOrio, and V. Bertacco, “Event-driven gate-level simulation with GP-GPUs,” in *Proc. DAC*, 2009, pp. 557–562.

[17] S. Smith, W. Underwood, and M. R. Mercer, “An analysis of several approaches to circuit partitioning for parallel logic simulation,” in *Proc. ICCD*, 1987, pp. 664–667.

[18] Y. Liu and J. Hu, “GPU-based parallelization for fast circuit optimization,” in *Proc. DAC*, 2009, pp. 943–946.

[19] Y. S. Deng, B. D. Wang, and S. Mu, “Taming irregular EDA applications on GPUs,” in *Proc. ICCAD*, 2009, pp. 539–546.

[20] L. Luo, M. Wong, and W. M. Hwu, “An effective GPU implementation of breadth-first search,” in *Proc. DAC*, 2010, pp. 52–55.