Massively Parallel Logic Simulation with GPUs

Article *in* ACM Transactions on Design Automation of Electronic Systems · June 2011

DOI: 10.1145/1970353.1970362 · Source: DBLP

Yuhao Zhu
University of Texas at Austin
17 PUBLICATIONS **526** CITATIONS
SEE PROFILE

Yangdong Deng
Tsinghua University
138 PUBLICATIONS **2,983** CITATIONS
SEE PROFILE

Massively Parallel Logic Simulation with GPUs

YUHAO ZHU, Beihang University
BO WANG and YANGDONG DENG, Tsinghua University

In this article, we developed a massively parallel gate-level logical simulator to address the ever-increasing computing demand for VLSI verification. To the best of the authors' knowledge, this work is the first one to leverage the power of modern GPUs to successfully unleash the massive parallelism of a conservative discrete event-driven algorithm, CMB algorithm. A novel data-parallel strategy is proposed to manipulate the fine-grain message passing mechanism required by the CMB protocol. To support robust and complete simulation for real VLSI designs, we establish both a memory paging mechanism and an adaptive issuing strategy to efficiently utilize the GPU memory with a limited capacity. A set of GPU architecture-specific optimizations are performed to further enhance the overall simulation performance. On average, our simulator outperforms a CPU baseline event-driven simulator by a factor of 47.4X. This work proves that the CMB algorithm can be efficiently and effectively deployed on modern GPUs without the performance overhead that had hindered its successful applications on previous parallel architectures.

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 logic simulation, discrete event-driven, CMB algorithm, GPU

ACM Reference Format:
Zhu, Y., Wang, B., and Deng, Y. 2011. Massively parallel logic simulation with GPUs. ACM Trans. Des. Autom. of Electron. Syst. 16, 3, Article 29 (June 2011), 20 pages.
DOI = 10.1145/1970353.1970362 http://doi.acm.org/10.1145/1970353.1970362

1. INTRODUCTION

Logic simulation has been the fundamental and indispensible means to verify the correctness of IC designs. For instance, IC designers use logic simulators to evaluate different design trade-offs at the RTL design stage and validate if the design implementation finally delivers the expected behavior before tape-out. The rapid increasing complexity of mod ern VLSI circuits, however, is continuously posing challenges to the simulation software. As a result, the gate-level logic simulation has become a time-consuming process. One example is the recently released NVidia graphics processing units (GPUs), Fermi, which consist of 3 billion transistors; it takes months to

Y. Zhu was with Department of Computer Science, Beihang University, Beijing, China, when he conducted the work reported in this article. B. Wang was with Institute of Microelectronics, Tsinghua University, Beijing, China, when he conducted the work reported in this article.
Authors' addresses: Y. Zhu (corresponding author), Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX; email: yuhao.zhu@mail.utexas.edu; B. Wang, Department of Electrical and Engineering, Stanford University, Palo Alto, CA; Y. Deng, Institute of Microelectronics, Tsinghua University, Beijing, China.

DOI 10.1145/1970353.1970362 http://doi.acm.org/10.1145/1970353.1970362

run a gate-level simulation even on clusters designed for parallel simulation [NVIDIA 2009]. In addition, the trend of integrating the whole system with multiprocessors into a single chip (MPSoC) makes overall system verification even more difficult. In fact, verification tasks could take over 70% of the total design time in a typical SoC design and 80% of the NRE cost [Rashinkar et al. 2000]. A large body of research has been devoted to accelerate the logic simulation process [Fujimoto 2000]. Besides improving the efficiency of algorithms, parallel computing has long been widely considered as the essential solution to provide scalable simulation productivity. Unfortunately, logic simulation has been one of the most difficult problems for parallelization due to the irregularity of problems and the hard constraints of maintaining causal relations [Bailey et al. 1994]. Today commercial logic simulation tools depend on multicore CPUs and clusters by mainly exploiting the task-level parallelism. In fact, large simulation farms could consist of hundreds of workstations, which would be expensive and power hungry. Meanwhile, the communication overhead might finally outweigh the performance improvement through integration of more machines.

Recently Graphics Processing Units (GPUs) are emerging as a powerful but economical high-performance computing platform. With hundreds of small cores installed on the same chip, GPUs could sustain both a computing throughput and a memory bandwidth that are one order of magnitude higher than those of CPUs. On applications that can be properly mapped to the parallel computing resources, modern GPUs can outperform CPUs by a factor of up to a few hundreds [Blythe 2008].

On the other hand, in the timed simulation of VLSI circuits, the parallelism provided by the simultaneously happened event might not be sufficient [Bailey et al. 1994]. The asynchronous, distributed time logic simulation mechanism exemplified by the CMB algorithm [Bryant 1977; Chandy and Misra 1979] would extract more parallelism by allowing different computing elements to evaluate events according to their local schedules as long as the correct causal relations can be maintained. CMB-like approaches had not been successful for VLSI simulation in the past due to the communication overhead on previous parallel platforms [Soule and Gupta 1991]. With a large number of computing cores integrated on-chip and a flexible memory system, current GPUs support much more efficient communication. As a result, it is essential to investigate the potential of modern GPUs for CMB-styled parallel logic simulation.

In this article, we propose a GPU-based logic simulator based on an asynchronous event-driven algorithm, the CMB algorithm. We present a fine-grain mapping scheme to effectively assign simulation activities to the computing elements of GPUs for a high level of available data-level parallelism. The original CMB algorithm deploys a priority queue to store events for every gate and such a data structure incurs a significant performance penalty due to the divergent branches. We develop a distributed data structure so that the events can be separately stored into FIFOs attached on pins. A dynamic GPU memory management is also proposed for efficient utilization of the relatively limited GPU memory. We also build a CPU/GPU cooperation mechanism to effectively sustain the computation power of GPUs. By combining the proceeding techniques, we are able to achieve a speedup of 47X on GPUs on average. The contributions of this article are summarized as follows.

— To the best of the authors knowledge, this work is the first GPU-based implementation of the CMB algorithm.
— A dynamic memory paging mechanism is developed for GPUs. It allows dynamic allocation and recycling of GPU memory to guarantee efficient memory usage in spite of the irregular memory demand inherent in logic simulations. To the best of our knowledge, this is the first dynamic memory manager for GPU-based applications.

In addition, the technique is generic enough to be also deployed in applications arisen from other domains.

— We redesigned the underlying data structures for CMB to make it suitable for efficient GPU executions. The new distributed data structure allows a higher level of parallelism to be unleashed.
— We presented a gate reordering mechanism by simultaneously considering input dependency, logic depth, and gate type. The reordering procedure could significantly reduce the chance of divergent program execution paths by different threads.
— We proposed a heterogeneous simulation platform by closely coupling CPU and GPU. Such a platform could effectively take advantage of high-speed memory transfer mechanisms such as asynchronous copy and zero copy. Through carefully organizing the execution flow, the CPU-GPU data transfer can be completely overlapped with GPU execution during a typical simulation process.

The remainder of this article is organized as follows. Section 2 introduces the background and motivation for this research. Section 3 presents the basic processing flow of our GPU-based logic simulator. Various optimization techniques that are essential for simulation throughput are outlined in Section 4. Section 5 presents the experimental results and detailed analysis. Related works are reviewed in Section 6. In Section 7, we conclude the work and propose future research directions.

2. BACKGROUND AND MOTIVATIONS

In this section, we first briefly present the algorithmic framework of parallel, asynchronous event-driven simulation with a focus on the CMB algorithm. A more complete picture of (circuit) simulation will be further discussed in Section 6. In this work, the parallel simulator is implemented on NVidia GPUs. So we will also introduce the corresponding GPU architecture and programming model [GTX 280 2011; Lindholm et al. 2008].

2.1 EVENT-DRIVEN SIMULATION ALGORITHM

Today the event-driven simulation algorithm is the workhorse of virtually all real-world simulation engines [Fujimoto 2000]. The event-driven simulation algorithm is built on the basis of the event, which is composed of both a value and a timestamp. An event indicates that the simulation state would have a transition on value at the time of timestamp. Another important notion is the physical time. Different from the simulation time, the physical time is the time passed in the physical system that is being simulated. The logic simulation of VLSI circuits can be naturally performed by an event-driven simulator. Generally, a logic simulator would use a global event queue to store all pending logic transitions. The event queue can be implemented as a priority queue so that the events are automatically chronologically ordered as they are inserted into the queue. At every simulation cycle, a logic simulator fetches the first event in the queue and performs evaluation according to the type of gate at which the event happens. The evaluation may create new events, which are again inserted into the event queue. The process is repeated until no events are available any more.

The event-driven logic simulation is a very efficient sequential algorithm. However, the parallelization of the event-driven logic simulation algorithm turns out to be challenging [Bailey et al. 1994]. In fact, it could not extract sufficient parallelism by simply evaluating events that happen at the same timestamp [Bailey et al. 1994]. The concept of distributed time, event-driven simulation, was developed by Chandy Misra, and [Bryant 1977] and designated as the CMB algorithm. Such an approach is based on the concept of *Logic Processes* (LPs). In a parallel simulation process, different modules of a simulated system are abstracted as *logic processes*. A LP maintains its

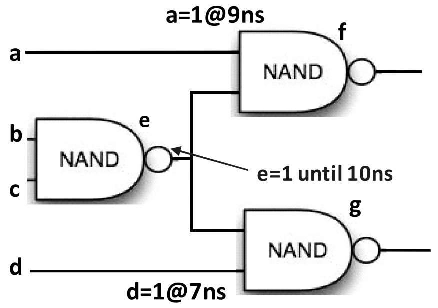


Fig. 1. The basic concept of the CMB simulation algorithm.

local simulation time and could have several inputs and an output. At each simulation step, an LP may receive several new events from its inputs and generate an event at its output. The key idea of the CMB algorithm is that different LPs could independently push forward their evaluation as long as the causal relations are not violated. Figure 1 is an illustration of the CMB algorithm. In the 3-gate circuit, the gate e has a pending event at 10ns and this fact is known to gates f and g. Meanwhile, input pin a has an awaiting event at 9ns, while input pin d has one at 7ns. Obviously, since e would not change until 10ns, the states of f and g would be completely determined by a and d before 10ns. In addition, gates f and g can be evaluated because a and d are independent. In other words, the safe evaluation time, T_{min}, of gates f and g is 10ns. After evaluating the events of a and d, the local time of gates f and g would be 9ns and 7ns, respectively.

The CMB simulation could run into a deadlock due to the distributed time mechanism [Chandy et al. 1979]. The most commonly used approach to prevent deadlock is through the usage of null events [Bryant 1977]. When an LP, A, will not generate new events (i.e., logic value remains the same) after an evaluation, it will instead send a null event with timestamp T_{null}. When another LP, B, receives the null event, its simulation time can then proceed to T_{null}. It can be formally proved that the CMB algorithm can correctly proceed until completion with the help null messages [Chandy and Misra 1979].

2.2 PARALLEL PROCESSING ON GPU

Although originally designed for graphic-specific algorithms, Graphic Processing Units (GPUs) are now increasingly used for general-purpose computing applications as a massively parallel accelerator [Blythe 2008]. Both NVidia and AMD, the largest two independent GPU manufacturers, have released programming models and software tools for general-purpose GPU program development. As a result, now programmers could use a general-purpose API to map their domain-specific applications onto the many cores installed on GPU chips. The complete tool chain greatly facilitates the rapid increase of usage of GPUs as a mainstream computing solution. In this article, our work is based on NVidias GPU and its programming environment CUDA [CUDA 2.3 2011].

Figure 2 illustrates the microarchitecture of GTX280, which is one of the off-the-shelf Tesla architecture processors. GTX280 consists of 30 Streaming Multiprocessors (SMs). Each SM contains 8 Scalar Processors (SPs). A SP is a simple scalar multiply-add unit, and the 8 SPs in one SM share a common multithreading instruction scheduling unit which would fetch and issue instructions to be executed across the SM. From such a point of view, the GTX280 can be regarded as a Single Program Multiple Data (SPMD), or Single Instruction Multiple Thread (SIMT) processor.

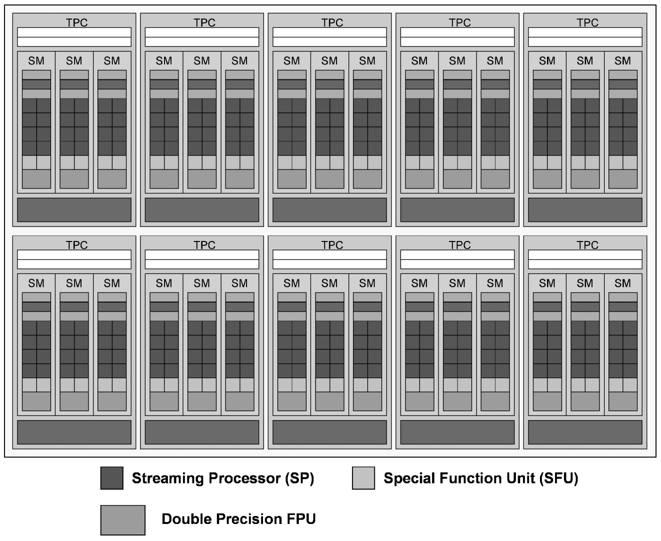


Fig. 2. NVidia GPU architecture.

During GPU execution, a large number of threads are organized as 32thread groups, or warps. Each SM in GTX280 could concurrently manipulate up to 24 active warps (i.e., 768 threads). Within every instruction cycle, one instruction is fetched for a warp within which all threads have had their data ready. So ideally, threads in a warp should follow the same flow of instructions. Such an ideal situation, of course, is hard to maintain for real-world applications. Therefore, CUDA allows that threads in one warp take different paths of program execution. If this does happen, the execution of divergent branches in that warp has to be serialized. In other words, each branch is taken one after another by corresponding threads, and finally threads reconverge at the end of the divergence. Such a performance overhead indicates that complex branch structures should be maximally avoided.

Modern GPUs are always equipped with a flexible memory hierarchy consisting of various types of memory circuits with different accessing latencies and read/write modes. Each SM has a 64K-Byte on-chip shared memory shared by its internal SPs. From the perspective of software, blocks of threads assigned to one SM would exchange data through the shared memory. When a group of accessing requests contains no bank conflict, the shared memory provides peak performance as fast as the registers inside an SP. Besides, through an interconnection network, SMs can access the off-chip global DRAM, with a 1G-Byte capacity on GTX280. However, it takes up to 400∼600 cycles of accessing latency and there is no cache support. Thus it is strongly recommended that the global memory accessing should be coalesced. A coalesced access means that the memory requests distributed in a given range can be combined into a single memory operation and processed in one memory latency. Meanwhile, there are two off-chip memory spaces, namely constant memory and texture memory, which are backed up by on-chip caches for improved irregular accessing. Both memory spaces are read-only, with the former intended for data declared at compiled time and the latter designed for data determined at runtime.

3. GPU BASED LOGICAL SIMULATOR

This section explains the details of our GPU-based massively parallel simulator. First the basic simulation flow is presented. Next we show how to sustain a high-level parallelism through carefully manipulating the communication of messages. In the last subsection, we propose novel techniques for dynamic GPU memory management and adaptive issuing of input patterns according to the memory page's utilization ratio.

3.1 THE GENERAL FRAMEWORK

During the process of parallel simulation, the input circuit is converted into an internal representation in which every gate is mapped to a Logic Process (LP). During simulation, an LP receives events from their inputs, performs logic evaluations, and then creates and sends events to outputs. In such a manner, the simulation process proceeds like waves travel forward. In order to handle both internal and I/O signals of a circuit in a uniform manner, every PI (Primary Input) is treated as a virtual gate. The simulation flow is organized as three consecutively executed primitives, *extract, fetch,* and *evaluate*. First, virtual gates *extract* pending input patterns and insert them into their corresponding event queues. Logically these stimuli are equivalent to the outputs of virtual gates. Next, all gates (including virtual gates) send the output events to the queues of the pins at the succeeding level. This phase can also be regarded as a *fetch* primitive from the viewpoint of an input pin. In real designs, we use *fetch* rather than *send* to extract more parallelism, because the number of pins is much larger than that of gates. Finally, in compliance with timing orderings, the real gates (i.e., all gates excluding virtual gates) *evaluate* the inputs to generate new events and add them into the event queues of the corresponding output pins.

Although the previous three primitives exhibit sufficient parallelism that is suitable for GPU implementation, two problems remain to be resolved. First, the workload has to be efficiently distributed to the hundred of cores installed on a graphic processor such that the fine-grained data-level parallelism can be effectively exposed. Second, an efficient memory management mechanism is essential, because GPUs are only backed up by a limited capacity of memory storage. To address the first problem, every pin (gate) is assigned to a single thread in the *fetch (evaluate)* kernel. In the *evaluate* kernel, each thread actually serves as an LP. For the second problem, a dynamic memory management mechanism is indispensable to best utilize the memory resource. Because current CUDA releases only offer preliminary support for flexible memory management, we designed an efficient paging mechanism for memory allocation and recycling. To further cushion the pressure on the memory usage, the issuing rate of input stimuli can be determined in an adaptive manner by considering the memory usage. The overall GPU-based simulation flow is outlined in Figure 3. The "**for each**" structure indicates that the following operation would be executed in parallel.

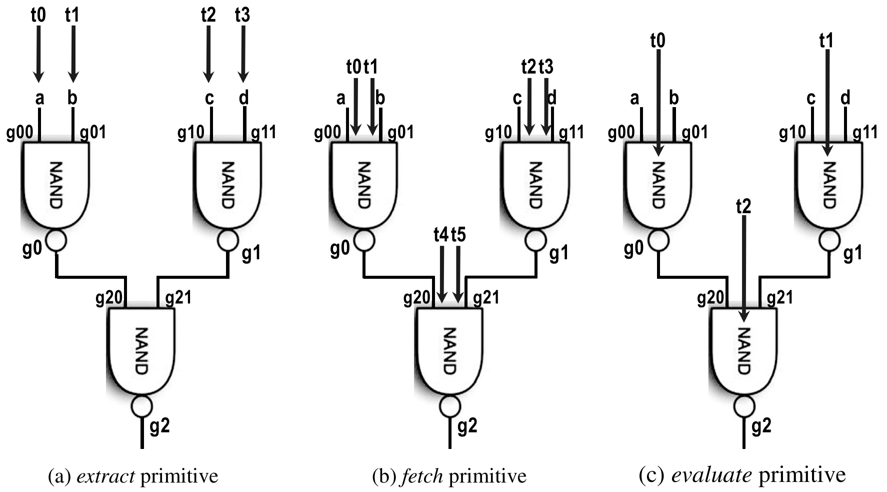
In Figure 4, the fine-grained mapping mechanism is illustrated with a simple example circuit consisting of three 2-input NAND gates, *g0, g1,* and *g2*. The four primary inputs are *a, b, c,* and *d*. The input pins for a given gate *gi* are labeled as *gi0* and *gi1*, where *i = 0, 1,* or *2*. In the extract primitive, the primary inputs are concurrently handled by multiple threads labeled as *t0* to *t4*. In the *fetch* primitive, each thread handles a different input pin. And finally in the *evaluate* primitive, one thread is assigned to every gate.

3.2 DATA STRUCTURES FOR FINE-GRAINED MESSAGE PASSING

The CMB algorithm is an event-driven algorithm and thus intrinsically suitable for a message passing model, such as MPI [MPI 2011]. However, modern GPUs are based on a shared memory architecture where several multiprocessors uniformly access the same global memory. Therefore, we emulate a message passing mechanism in our simulator through manipulating three arrays, namely *output_pin, event_queue,* and *gate_status*. The *output_pin* array reserves an entry for the output pin of every gate

```
while completion requirements not meet do

for each PI do
    if  memory to allocate is enough
        extract the stimuli to the PI output pins
    else
        issue null message the PI output pins
end for each

for each input pin do
    fetch messages from output pins
end for each

allocate memory if needed

for each gate do
    insert new events from its input pins to the event FIFOs
    evaluate the earliest message in its event FIFOs
    write evaluation result to the output pin
end for each

release memory if possible

end while
```

Fig. 3. GPU-based logic simulation flow.

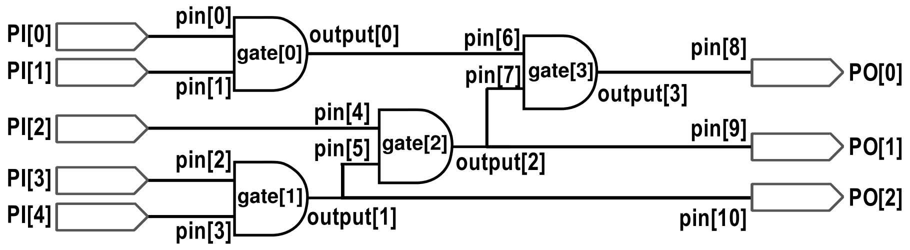


(a) *extract* primitive (b) *fetch* primitive (c) *evaluate* primitive

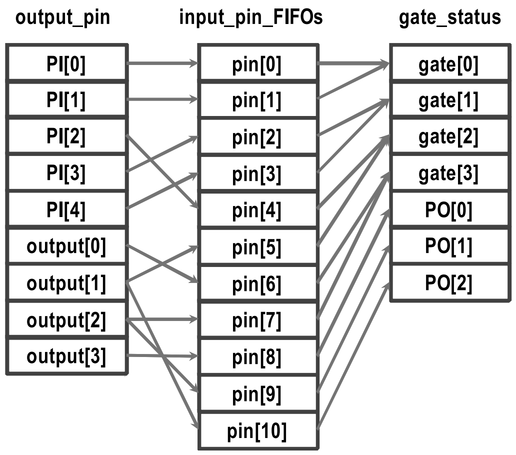
Fig. 4. Fine-grain mapping of workload.

and stores events generated during gate evaluation. Note that the events for virtual gates are extracted from the input stimuli in the extract primitive rather than from evaluation. The *event_queue* array stores the events to be evaluated by a gate. The *gate_status* array records related information for each gate. Such information includes the input logic values and the safe evaluation time T_{min}, which is defined in Section 2.1.

Figure 5(a) is a sample circuit whose corresponding data structures are illustrated in Figure 5(b). At the beginning of simulation, the virtual gates, that is primary inputs, *extract* stimuli into the corresponding portion of *output_pin*. Then all events stored



(a) sample circuit



(b) corresponding message passing data structures

Fig. 5. Sample circuit and its data structures.

in *output_pin* are written into *event_queue* to emulate one round of message passing. Finally, the event with the earliest timestamp T_{pin} is chosen from its *event_queue*. As formulated in the original CMB algorithm, if T_{pin} is smaller than T_{min}, that event could be safely fetched for evaluation.

To maintain the ordering of the events passed to a gate, the *event_queue* mentioned before has to be a priority queue. However, it is extremely challenging to find an efficient GPU implementation for managing a heap-based priority queue due to the large number of branches incurred by heap operations. To avoid this bottleneck, we decompose the *event_queue* of a gate into several FIFOs, with one FIFO for an input pin. Such FIFOs are designated as *input_pin_FIFOs*. The advantages of such an approach are trifold. First, by decomposing the gate-wise priority queue into a group of lightweight distributed FIFOs, the insertion operation for a priority queue is much simpler. In fact, those events arriving at a given input pin are naturally chronological and thus newly arrived events can be safely appended at the end of the FIFO. Otherwise, the insertion operation of a priority queue would incur serious performance overhead because different queues would have different structures, which would inevitably lead to diverse program execution paths varying. Meanwhile, the distributed FIFO mechanism enhances the potential parallelism because the insertion operations for different FIFOs are independent. Otherwise, a centralized, gate-wise storage would result in resource contention and restrict the maximum parallelism to be the number of gates. Additionally, if a newly arrived event has the same event as the latest one in the FIFO, it will be discarded and only the last-come

```
typedef struct{
      unsigned int  size;
      unsigned int  *page_queue;
      unsigned int  head_page;
      unsigned int  head_offset;
      unsigned int  tail_page;
      unsigned int  tail_offset;
}FIFO_T;
```

Fig. 6. FIFO data structure.

event timestamp is updated. This mechanism greatly reduces the number of active events in the simulation and is proved essential in the experiments.

3.3 MEMORY PAGING MECHANISM

The limited GPU memory would pose an obstacle for the simulation of large-scale circuits. First of all, industry-strength, large-scale circuits would require considerable space to store necessary information. In addition, a significant number of events would be created and sent in the simulation process. Such events also need to be buffered in memory before they are processed. Current GPUs do not support dynamic memory management features such as malloc()/free() in C language and new()/delete() in C++. Accordingly, a straightforward implementation would allocate a fixed size for the FIFO, that is, *input_pin_FIFO*, for each input. However, there is no easy way to optimally determine the FIFO size before runtime. A smaller FIFO size would lead to frequent FIFO overflow, while a larger size suffers low utilization of memory storage. To overcome this problem, we introduce a memory paging mechanism to efficiently manage the GPU memory.

The feasibility and advantage of such a dynamic memory management mechanism is justified by two basic observations in the CMB-based simulation experiments. First, we found that the numbers of events received at different pins would fluctuate dramatically. A relatively small number of pins are very hot in the sense that they receive many more events than those cold ones by orders of magnitude. This fact certainly suggests that a similar nonuniform pattern should be applied to the allocation of FIFO size on each input pin to avoid both starvations and overflow. The second key observation is that, in a different period, the occupation of a FIFO varies noticeably. Thus, the management should be able to recycle the memory storages.

Based on these observations, we designed a paging mechanism for dynamic management. A large bulk of memory *MEM_SPACE* with *MEM_SIZE* bytes is allocated on GPU before the simulation. It is uniformly divided into small memory pages with a constant size of *PAGE_SIZE*, each of which can hold up to *EVENT_NUM* events. A page is the minimum unit of memory management and each page is uniquely indexed in a continuous manner. This memory space is reserved for the *input_pin_FIFO* arrays of all the inputs pins. Initially, a specific number of pages, *INIT_PAGE_NUM*, are allocated to every FIFO of input pins. Indexes of all remaining pages are inserted into a global FIFO called *available_pages*, which is used to record the index of available pages. As the simulation process moves on, preallocated pages are gradually consumed. As a result, free pages are needed to store newly created events, while pages storing processed events can be recycled. In our simulator, we deal with the requests for allocating new pages and releasing useless pages after every execution of fetch and evaluate primitives, respectively.

The FIFO structure for a pin is defined as Figure 6. The *page_queue* stores the indexes of pages that have been allocated to that pin. The events are extracted and stored in *MEM_SPACE* according to the FIFO pointers. As illustrated in Figure 7, the

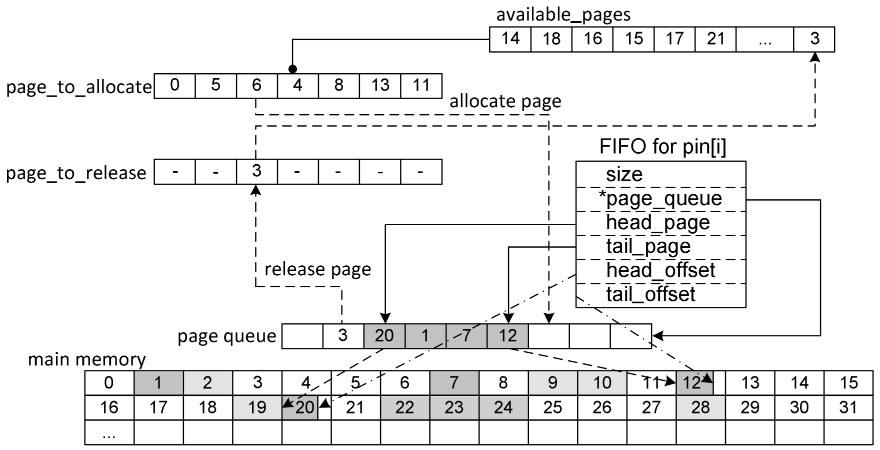


Fig. 7. Illustration of a sample memory layout.

locations of first and last events in the FIFO are pointed by *page_queue[head_page] * PAGE_SIZE + head_offset and page_queue[tail_page] * PAGE_SIZE + tail_offset*, respectively.

A few additional data structures are needed to guarantee the correct working of the paging mechanism. Two index arrays, *page_to_allocate* and *page_to_release*, are employed to support page release and allocation. Every pin has an entry in both arrays. Elements in *page_to_allocate* denote the page indexes that are to be allocated to corresponding pins, while elements in *page_to_release* hold the page indexes of those pins that can be released. When a pin i runs out of pages, the GPU kernel fetches the *i*-th value in *page_to_allocate* and inserts it into pin *i*s *page_queue*. Similarly, when events on a page of pin i are all processed, the GPU kernel writes the index of that page to the i-th entry of *page_to_release*, and pops that page index from *page_queue*. Because all event information is stored in *MEM_SPACE*, no explicit copy has to be performed for release and allocation. The only overhead is introduced when the control flow returns to the CPU side. At that time, the host thread would recycle the pages in *page_to_release* by inserting them into *available_pages*, and designate pages to be instantly allocable by writing indexes of those pages into related entries in *page_to_allocate*. The update of these two arrays to the *available_pages* is done sequentially on the host CPU. Further improvement of this step will be discussed in Section 5.

Finally, in the experiments, we found that under proper conditions, events can be created at a much higher rate than that of evaluation. Under such a circumstance, all pages in *MEM_SPACE* may be occupied by events that have not been evaluated. This would prevent the new input stimuli to be issued due to the lack of memory space. To attack this problem, we propose an adaptive strategy to control the issuing speed of input stimuli.

Before issuing stimuli, one host-side thread is responsible for checking the number of allocable pages residing in *available_page*. If the number is under a predefined threshold, the issue of stimuli is paused. Instead, *null* events are issued with the same timestamp as the last valid event. These *null* events will not be inserted into pin FIFOs but can push forward the simulation process. Those events ready to be evaluated will be processed simultaneously and some pages may be released. When the number of obtainable pages in *available_page* is once again above the threshold, normal issue would continue. In this work, we choose a relatively conservative threshold value no smaller than the number of pins, since it is possible that all pins require a page in the next simulation iteration.

4. GPU-ORIENTED PERFORMANCE OPTIMIZATION

The techniques presented in Section 4 provide a basic flow for a massively parallel logic simulator. However, performance tuning and optimization are essential to successfully unleash the computing power of GPUs. For GPUs, major hurdles for computing performance include divergent program execution paths, communication overhead between CPU and GPU, as well as uncoalesced memory access. In this section, we discuss a set of optimizations to improve the simulation throughput.

4.1 GATE REORDERING

The efficient execution of a GPU program requires the data to be organized with good locality and predictability. When the threads are organized into warps, ideally these threads should take the same actions. However, gates in an input circuit are generally arbitrarily ordered and the ordering is used to assign gates to threads. As a result, generally threads in a warp would follow different instruction paths when performing the table lookup for gate delay and logic output value as well as the calculation of T_{pin}.

To reduce the divergent paths, we proposed a gate reordering heuristic according to circuit information. The ordering process can be performed prior to simulation. Gates are sorted and then ordered with regard to gate types and number of fan-ins. When the gates are sequentially assigned to threads, threads in a warp are very likely to follow the same execution path in the *evaluate* primitive. Our experiments show that this heuristic could enable an additional speedup up to 1.5X on some designs.

4.2 MINIMIZING DIVERGENT EXECUTION PATHS

Although hardware techniques like dynamic warp formation [Wilson et al. 2007] have been proposed to minimize the overhead of intrawarp divergent branches, current off-the-shelf GPUs are still sensitive to flow control instructions such as if, switch and while. These instructions would potentially lead to a serialization of parallel threads. As stated in Section 3.2, the design of the fundamental data structure already mitigates the divergent path problem by replacing the gate-wise priority queue with multiple distributed pin-wise FIFOs so as to avoid the insertion of priority queues.

4.3 HIDING HOST-DEVICE INTERACTION OVERHEAD

The only interaction between host and device threads during simulation is introduced by the paging mechanism. To dynamically allocate and recycle GPU memory, it requires explicit copy of data (*page_to_allocate and page_to_release*) and the corresponding sequential processing during every iteration. The resultant overhead is twofold. First, data transfer between GPU and CPU is through a PCI Express [PCIe 2011] interface, which usually cannot meet the consumption speed of the GPU hardware. The second overhead is due to the sequential processing on the CPU. In fact, Amdahls law [Amdahl 1967] suggests that the sequential part of a program could severely drag down the performance of a parallel program. In fact, the CPU processing might overweigh the performance gain offered by the massively parallel execution on the GPU. Figure 10 illustrates how seriously the host-device interaction can affect the overall performance by dividing the whole simulation time into three parts, GPU processing, CPU processing, and data transfer. It can be seen that the latter two consume a far larger percentage of total execution time.

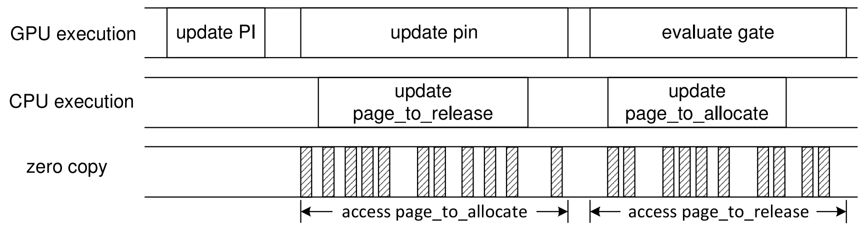


Fig. 8. Illustration of ideal overlap.

Certainly the ideal situation would be to overlap the three parts as illustrated in Figure 8. Such an overlapping is indeed possible through a careful organization of the primitives. A key observation is that the *fetch* primitive that inserts events into event FIFOs only needs to execute the memory allocation operations when the allocated pages for that FIFO are full. In contrast, the *evaluate* primitive that evaluates events from FIFOs just requires the memory release operations, because some allocated pages may have be completely processed. Therefore, the CPU processing of *page_to_allocate* and the *evaluate* kernel can be overlapped. The same goes with the processing of *page_to_release* and the receive kernel. This can be straightforwardly realized with the *asynchronous execution* feature provided in CUDA. We create two streams and assign *fetch* / *release* within one stream, and *evaluate* / *allocate* in another. In this way, the receive primitive is executed on GPUs while the *release* operation runs on the CPU host simultaneously. The evaluate primitive and *allocation* operation work in the same way. Because the amount of data transferred is relatively small, we take advantage of a new feature called *zero copy* [CUDA 2.3 2011] released in CUDA 2.2 and later version. With zero copy technique, data transfer is invoked within GPU kernels, rather than on the host size. Zero copy has the advantage of avoiding superfluous copies since only the request data will be transferred. Another advantage is that it also schedules those kernel-originated data transfers implicitly along with the kernel execution, without the need for explicit programmer interventions.

To further overlap CPU computation time with GPU processing time, we propose a *group flag* strategy. Every 32 (i.e., the warp size in CUDA) pins are aggregated into a group. Every group has two flags, one for release and one for allocation. During GPU processing, if any pin in the group requires a new page or releases a useless page, corresponding flags are marked. Therefore, the CPU thread only needs to check those groups of threads whose flags are marked and skip those unmarked ones. This strategy functions like a skip-list and proves to be important for a high level of efficiency.

4.4 ENHANCING MEMORY ACCESS EFFICIENCY

As suggested in CUDA Best Practice Guide [CUDA 2.3 2011], one of the top concerns for CUDA programming is how to guarantee the efficiency of memory access. Accordingly, we perform extensive optimizations on the memory usage. For instance, a FIFO has a set of descriptive data organized as a structure with multiple members such as *head* pointer, *tail* pointer, *size* value, and *offset* value. To avoid the structure accesses that are hard to coalesce, we actually use a Structure Of Array (SOA) to store the data. In addition, we minimize the access to long-latency global memory by storing read-only data in constant and texture memories, which are both supported by on-chip cache. The circuit-topology-related data which are determined before simulation are loaded into texture memory. Static circuit data structures like the truth table are located in constant memory by declaring them as __constant__. With these storage patterns, the penalty of irregular memory accessing is significantly reduced.

Table I. Characteristics of Simulation Benchmark

Design	#Gates	#Pins	Description
AES	14511	35184	AES encryption core
DES	61203	138419	DES3 ENCRYPTION CORE
M1	17700	42139	3-stage pipelined ARM core
SHA1	6212	13913	Secure Hashing algorithm core
R2000	10451	27927	MIPS 2000 CPU core
JPEG	117701	299663	JPEG image encoder
B18	78051	158127	2 Viper processors and 6 80386 processors
NOC	71333	181793	Network-on-Chip simulator
LDPC	75035	148022	LDPC ENCODER
K68	11683	34341	RISC PROCESSOR

Table II. Distribution of Number of Events Received by Pins

#events	AES	DES	M1	SHA1	R2000	NOC	K68	B18
$0 \sim 99$	34444	138317	39708	13591	26106	157202	32669	158086
$100 \sim 9999$	737	102	2431	321	1764	24591	1672	40
$>= 10000$	3	0	0	1	3	0	0	1

5. RESULTS AND ANALYSIS

5.1 EXPERIMENTAL FRAMEWORK

We choose ten open-source IC designs from OpenCores.org [OpenCores 2011] and ITC99 Benchmarks (2nd release) [ITC99 2011] to test our GPU-based simulator. Critical parameters of these circuits are listed in Table I. Among these designs, b18 was released as a gate-level netlist. The other nine designs were downloaded as RTL-level Verilog code and then synthesized with Synopsys Design Compiler using a 0.13 μm TSMC standard cell library. Similar to the common industry practices, we use two sets of simulation stimuli, deterministic test patterns released with the design test-bench and randomly generated patterns. Usually the deterministic patterns can be used to validate if a circuit delivers the expected behavior, while the random patterns are used to cover possible design corners. For randomly generated stimuli, we set the maximum simulation time as 250,000 cycles. Every five cycles the random generator would create a new true or false value so that valid stimuli actually occupy 50,000 cycles.

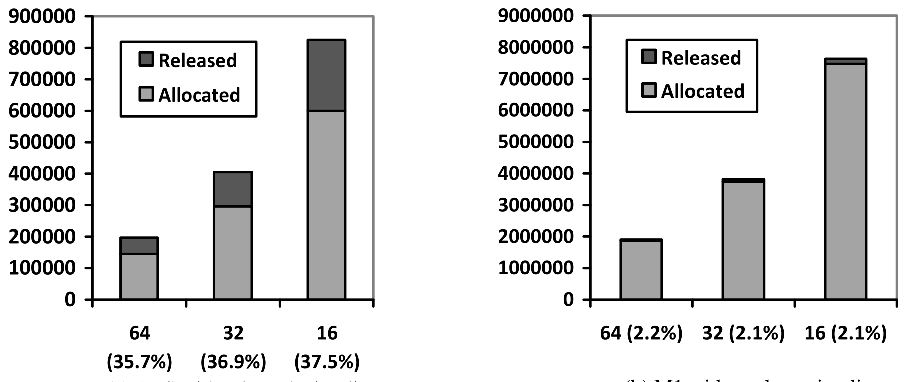
To evaluate the performance of our GPU simulator, we also hand-coded a baseline CPU simulator as a reference. The baseline simulator uses a classical event-driven algorithm instead of the CMB algorithm, because the CMB algorithm is intrinsically slower on sequential machines due to the extra work of message passing and local time maintaining [Soule and Gupta 1991]. Our baseline simulator is up to 2X faster than the Synopsys VCS simulator, because such commercial simulators have to handle complex verification computations that incur performance overhead.

5.2 JUSTIFICATION OF THE OPTIMIZATION TECHNIQUES

We mentioned in Section 4.3 that there exist significant variations in the events received by each pin. This fluctuation can be several orders of magnitude across one specific design. Detailed statistics for a group of benchmark circuits are listed in Table II. In Table II, pins are grouped into three categories with regard to the peak number of events that they could receive during a complete run of simulation. The data are collected after 50,000 simulation cycles using randomly generated input stimuli.

Table III. Memory Page Recycling

Design	Allocated pages	Released pages	Recycled ratio
AES	145085	51793	35.69%
DES	429312	93312	21.74%
R2000	27336	1938	7.09%
JPEG	1045482	46292	4.43%
SHA1 (R)	245596	14355	5.84%
M1 (R)	1868242	40075	2.15%
B18 (R)	45742	21935	47.95%
NOC (R)	2072609	167061	8.06%
K68 (R)	1430792	128857	9.00%



(a) AES with released stimuli (b) M1 with random stimuli

Fig. 9. Page recycling with different *PAGE_SIZE*.

It can be seen from the table that the activity intensities of different pins are under dramatic deviation, which implies that the dynamic paging mechanism is imperative.

We also conducted experiments to quantitatively analyze the efficiency of GPU memory paging. The page recycling algorithm is detailed in Section 4.3. Here we choose four circuits to apply deterministic input pattern and five circuits (marked with a letter "R") to exert randomly generated stimuli. Table III quantifies the numbers of pages released and allocated when simulating each circuit. The 4th column is the ratio of released pages to allocated pages.

The data in Table III are collected by configuring that *PAGE_SIZE* as 32 bytes. For a give design, the circuit topology and stimuli pattern would largely determine the progress of the simulation and the consumption of memory. Accordingly, we observe an obvious variation in the page-recycle frequency. On average, about 14.2% pages could be recycled. While the *PAGE_SIZE* decreases, the number of recycled pages would increase as illustrated in Figure 9, where the x-axis is page size (i.e., *PAGE_SIZE*) and y-axis represents the number of pages. The number beside the page size is the ratio of the number of recycled pages to the number of total allocated pages. There is no noteworthy difference in the relative ratio with varying page size, because both released and reallocated pages vary in a similar trend. However, a small page size means that more pages are needed. A larger number of pages in turn results in more frequent page recycling and would potentially drag down the overall performance. On the other hand, a large page size enlarges the granularity of the page recycling process and thus lowers the flexibility of memory management. The extreme situation would

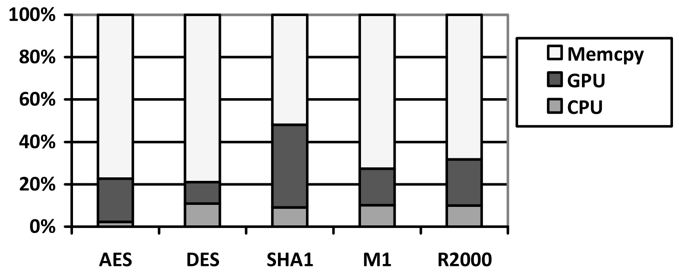


Fig. 10. Detailed timing analysis.

be so large a *PAGE_SIZE* that every pin can only have one page. Considering the contradicting impacts, a *PAGE_SIZE* of 64 offers a balanced solution empirically.

Finally, we justify the overlap of CPU processing, GPU processing, and memory copy. The distributions of CPU time consumed by these three components are illustrated in Figure 10. The most significant observation is that memory copy accounts for most of the whole simulation time. This necessitates the use of *zero copy* to reduce redundant data copy and overlap computation with communication. Besides, CPU processing plays a nontrivial role in all designs. In DES, CPU time even outweighs the GPU part. Therefore, we need to not only overlap CPU processing with GPU computation, but also employ the *group flag* mechanism to further reduce the overhead on CPU in achieving an optimal performance.

5.3 PERFORMANCE EVALUATION

In this section, we compare the performance of our GPU simulator against the CPU simulator described in Section 6.1. The CPU baseline simulator is compiled by gcc 4.2.4 with O3 optimization. The GPU-based simulator reported in this article is realized in CUDA 2.2 version. The experiments were conducted on a 2.66 GHz Intel Core2 Duo server with an NVidia GTX 280 graphics card. We present results on both deterministic and random stimulus. Table IV reports results of all ten designs under randomly generated input patterns. Some designs (K68 and b18) have no testbench released, and the other two designs (LDPC and NOC) are inherently tested by random patterns. Table V only reports six designs simulated under the released deterministic stimuli. Thanks to the highly parallel GPU architecture described in Section 2.2, the GPU simulator attains an average speed of 47.4X on average for random patterns, and up to 59X speedup for four designs with deterministic patterns. Meanwhile, there do exist two designs on which the GPU simulator is slower under deterministic patterns. The reasons that the speedup varies dramatically among different circuits and input patterns will be explained in the next section.

5.4 SPEEDUP DISCUSSION

it is worth noting that the GPU-based simulator is slightly slower than its CPU counterpart for m1 and it is worth noting that the GPU-based simulator is slightly slower than its CPU counterpart for M1 and SHA1 when applying the deterministic stimuli released to the designs. However, when applying random patterns, a significant speedup can be observed. In fact, for all six designs listed in Table V, the speedup on random stimuli is much more significant. The reasons are twofold. First, we find that the deterministic stimuli are applied in a rather sparse manner. For example, with the deterministic stimuli of M1, input values only receive new values in 176,500 cycles (total 99,998,019 cycles simulated). In other words, the primary inputs receive

Table IV. Simulation Results With Randomly Generated Stimuli

Design	Baseline simulator (S)	GPU based simulator (S)	Speedup (Column 2/Column 3)
AES	676.96	9.45	71.64
DES	881.68	16.38	53.83
M1	88.85	8.57	10.37
SHA1	51.48	5.57	9.24
R2000	223.93	8.33	26.89
JPEG	10030.64	37.11	270.29
LDPC	51.91	10.41	4.99
K68	54.46	6.74	8.08
NOC	43.32	9.71	4.46
B18	299.92	21.70	13.82

Table V. Simulation Results with Officially Released Stimuli

Design	Baseline simulator (S)	GPU based simulator (S)	Simulated cycles	Speedup (Column 2/Column 3)
AES	90.50	5.01	42,935,000	18.06
DES	17.38	8.06	307,300,000	2.16
M1	13.56	22.11	99,998,019	0.61
SHA1	0.33	0.42	2,275,000	0.79
R2000	4.594	0.937	5,570,000	4.90
JPEG	2121.71	35.71	2,613,200	59.42

new patterns in about every 500 cycles. On the other hand, new assignments are applied very frequently in the random patterns (every 5 cycles for the results reported in Table IV). The simulator is thus kept much busier by the random stimuli. Second, when the deterministic input patterns are applied, one pin would receive the same value consecutively. Again taking circuit M1 as an example, a great portion of pins repeatedly receive the same true or false value throughout the simulation. As described before, events with the same value as the previous one will not be added into FIFO. So the total number of events that are to be evaluated is actually even more sparse. With randomly generated stimuli, however, pins would randomly receive a new assignment. As a result, the number of events would be significantly higher. Considering the preceding the two factors as well as the parallelization overhead, it is not strange that the GPU simulator would be sometimes slower on deterministic patterns, but still could achieve a dramatic speedup on random patterns. In other words, the lack of activities in the deterministic input stimulus for M1 and SHA1 is the major reason for the insufficient acceleration. In fact, when using a GPU simulator, IC designers could use high-activity testbenches to extract sufficient parallelism for an even higher throughput, as exemplified in Figure 11. On the benchmark circuits AES, SHA1, and M1, we created three sets of random test patterns by gradually reducing the gap between two patterns from 50 cycles to 5 cycles. We normalize the relative speedup to the 50-cycle one and show them in Figure 11. Clearly, when the input patterns have a higher level of activity, the GPU will perform better with a larger acceleration ratio.

6. RELATED WORKS

The IC design process involves different simulation tools. For digital circuits, discrete algorithms are used because the output voltage can be abstracted to always take a few

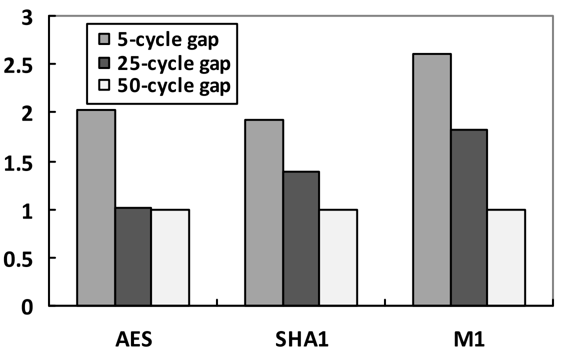


Fig. 11. Speedup of GPU simulator over CPU baseline with varying input activity intensities.

fixed values defined in a discrete space. On the other hand, the simulation of analog circuits depends on numerical methods developed based on continuous models (the continuous nature would preclude the concept of event). In this work, we only focus on digital circuit simulation.

There are two major approaches to perform simulations: *oblivious* and *event-driven* [Fujimoto 2000]. The oblivious simulator evaluates all gates at every clock cycle, while the event-driven one only evaluates those whose inputs have changed. Though oblivious simulation is simple, it is not adopted in practice. Due to the fact that only a small fraction of gates are active, that is, the inputs of them change, every clock cycle in a large design, event-driven approaches, and these are the most appropriate and practical methods. In fact, most current commercial simulators are based on event-driven approaches.

Considering the enormous time required for the simulation of large VLSI circuits, much emphasis has been paid to accelerate logic simulation by leveraging parallel and distributed computer architectures. Modern parallel logic simulators are typically based on event-driven simulation algorithms which can be divided into two basic categories: synchronous algorithms and asynchronous algorithms. In a synchronous simulator, multiple LPs progress in a step-locked manner as coordinated by a global clock. The events are stored in a global data structure for shared access. Such synchronous algorithms can be implemented in SIMD architectures (e.g., Bataineh et al. [1992]) in a straightforward manner. To extract a higher level of parallelism, an asynchronous simulator like the one implemented in this work assigns every LP with a local clock. The basic flow of asynchronous simulation protocol was first proposed by Chandy and Misra [1979] and Bryant [1977], as detailed in Section 2.1. The overall asynchronous flow is amenable to parallel implementations because the synchronization overhead can be avoided. However, it poses challenges for an SIMD implementation since LPs tend to follow different execution paths. In addition, the communication cost of passing messages could be too expensive for CPUs located on different chips. To the best knowledge of the authors, this work is the first one to successfully resolve the previous problems on a shared memory chip multiprocessor (CMP). It should be noted that there are many variants (e.g., Lubachevsky [1988] and Nicol [1991]) of CMB, but the fundamental ideas are the same. Our technology is generic and can be applied to all variants in this category.

Asynchronous simulation algorithms can be classified into two categories, *conservative* and *optimistic*. The conservative approach exemplified by the CMB algorithm guarantees that no event with a timestamp smaller than the evaluated ones would be received in succeeding simulation steps. This protocol enforces the causal relation of the simulation. On the other hand, the optimistic approach [Jefferson and Sowizal 1985; Jefferson 1985; Reiher et al. 1990] allows processing events whose timestamps may be larger than those of later events. When a prior evaluation turns out to be incorrect after related events have been processed, a rollback mechanism or a reverse computation will be initialized to restore the results before the incorrect evaluation. In this work, we adopt the conservative approach to avoid the complex flow control inherent in the optimistic approach because such a divergent control flow would drag down the efficiency of GPU execution.

Also as mentioned in Section 2.2, the very mechanism of the CMB algorithm could lead to deadlock [Chandy et al. 1979]. Besides the technique of null message [Bryant 1977] that we employ in this work, various other solutions such as probe message [Holmes 1978; Peacock et al. 1979] and virtual time [Jefferson 1985] have also been introduced to prevent or recover from deadlock. We use the null message approach as originally proposed by the CMB algorithm because it is most amenable to the SIMD processing style of GPUs.

The emergence of GPU makes it possible to perform massively parallel logic simulation because it provides a large number of scalar processors with fast inter-processor communication. The work proposed in Gulati and Khatri [2008] uses GPU to accelerate fault simulation by simulating multiple input patterns in parallel. A GPU-based simulator, GCS, was presented in Chatterjee et al. [2009a]to carry out high-performance gate-level logical simulation. It is based on an oblivious algorithm in which every gate is evaluated at every clock cycle. The authors of Chatterjee et al. [2009b] proposed a GPU-based logic simulator by synchronously evaluating multiple events happening simultaneously. Different from the preceding works, our GPU-based CMB simulator is asynchronous and does not require maintaining a global clock across all gates in a circuit.

Outside the logic simulation domain, there are several related works also using GPUs for a specific application (e.g., Xu and Bagrodia [2007], Park and Fishwick [2008], Rybacki et al. [2009]), or a general framework (e.g., Perumalla [2006a, 2006b], Park and Fishwick [2009]). One early work reported in Xu and Bagrodia [2007] focuses on a high-fidelity network modeling and adopts a synchronous event-driven algorithm. In Park and Fishwick [2008], a discrete-event simulation framework is proposed. Also based on a synchronous event-driven algorithm, that paper developed a distributed FEL (Future Event List) structure to avoid the time-consuming global sorting. We build a different approach to solve the similar problem in this article. Basically, the priority queue of a gate is distributed to the gate inputs such that the sorting is no longer necessary. The mechanism is enhanced with a dynamic memory management mechanism to overcome the obstacle of a limited GPU memory capacity.

7. CONCLUSION AND FUTURE WORKS

This article proposes the first CMB-based logic simulation algorithm on modern GPUs. The CMB algorithm is efficiently and effectively mapped to GPUs in a fine-grain manner. To guarantee a high level of parallelism in the simulation, we redesigned the fundamental data structures of the CMB algorithm to overcome the previous obstacle that hinders an efficient implementation on previous shared memory architectures. A gate reordering heuristic is introduced to improve the data locality and reduce the execution divergence on GPUs. A dynamic GPU memory paging mechanism and an adaptive issuing mechanism are introduced to support the robust simulation of large-scale circuits. We also develop a complete set of optimization techniques to maximize the effective memory bandwidth. Our GPU simulator is tested on a group of real VLSI designs with both deterministic and random stimuli. The experimental results show that a speedup of 47.4X could be achieved against a CPU baseline simulator using a classical event-driven algorithm.

In the future, we plan to extend this work in several directions. First, the simulator will be enhanced to support RTL. Complex processes would need to be mapped to computing resources under such a context. A process decomposition mechanism would be necessary. Second, we will generalize our simulator to perform simulation in other application domains such network and transportation simulations. Another interesting topic is to explore the possibility of parallelizing SystemC's simulation kernel with our techniques. The current SystemC simulation model defined by IEEE [IEEE System 2011] follows a sequential event-driven approach. A previous work on GPU-based SystemC simulation [Nanjundappa et al. 2010] takes a straightforward parallelization strategy rather similar to Chatterjee et al. [2009b]. In our future work, we will be investigating the potential of using the CMB protocol to rewrite the SystemC simulation engine.

REFERENCES

AMDAHL, G. M. 1967. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS'67)*. ACM, New York, 483–485.

BAILEY, M. L., BRINER JR., J. V., AND CHAMBERLAIN, R. D. 1994. Parallel logic simulation of vlsi systems. *ACM Comput. Surv. 26*, 3, 255–294.

BATAINEH, A., ÖZGÜNER, F., AND SZAUTER, I. 1992. Parallel logic and fault simu- lation algorithms for shared memory vector machines. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'92)*. IEEE, Los Alamitos, CA, 369–372.

BLYTHE, D. 2008. Rise of the graphics processor. *Proc. IEEE 96*, 5, 761–778.

BRYANT, R. E. 1977. Simulation of packet communications architecture computer system. Tech. rep. MIT-LCS-TR-188, MIT.

CHANDY, K. M. AND MISRA, J. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Engin. SE-5*, 5, 440–452.

CHANDY, K. M., MISRA, J., AND HOLMES, V. 1979. Distributed simulation of networks. *Comput. Netw. 3*, 105–113.

CHATTERJEE, D., DEORIO, A., AND BERTACCO, V. 2009a. High-Performance gate- level simulation with GP-GPUs. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'09)*. 1332–1339.

CHATTERJEE, D., DEORIO, A., AND BERTACCO, V. 2009b. Event-Driven gate-level simulation with GP-GPUs. In *Proceedings of the 46th IEEE/ACM International Conference on Design Automation (DAC'09)*, ACM, New York, 557–562.

CUDA 2.3. 2011. NVidia, CUDA programming guide 2.3. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

FUJIMOTO, R. M. 2000. Parallel and Distributed Simulation Systems. Wiley-Interscience.

GTX 280. 2011. GeForce GTX280. http://www.nvidia.com/object/geforcefamily.html.

GULATI, K. AND KHATRI, S. 2008. Towards acceleration of fault simulation using graphics processing units. In *Proceedings of the 45th IEEE/ACM International Conference on Design Automation (DAC'08)*. ACM, New York, 822–827.

HOLMES, V. 1978. Parallel algorithms on multiple processor architectures. Doctoral disserta- tion, University of Texas at Austin, Austin, TX.

IEEE SYSTEM C. 2011. IEEE Std. 1666-2005, Standard for SystemC. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=10761.

ITC99. 2011. ITC99 benchmarks. http://www.cad.polito.it/tools/itc99.html.

JEFFERSON, D. R. 1985. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3, 404–425.

JEFFERSON, D. AND SOWIZAL, H. 1985. Fast concurrent simulation using the time warp mechanism. *Distrib. Syst. 19*, 3, 183–191.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2, 39–55.

LUBACHEVSKY, B. D. 1988. Bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*. 183–191.

MPI. 2011. MPI. http://www.mpi-forum.org/docs/.

NANJUNDAPPA, M., PATEL, H., JOSE, B. A., AND SHUKLA, S. 2010. SCGPSim: A fast systemc simulator on gpus. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASPDAC'10)*. IEEE, 145–154.

NICOL, D. M. 1991. Performance bounds on parallel self-initiating discrete event simulations. *ACM Trans. Model. Comput. Simul. 1*, 1, 24–50.

NVIDIA. 2009. NVidia white paper: NVIDIAs next generation cudatm compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

OPENCORES. 2011. OpenCores, http://www.opencores.org/.

PARK, H. AND FISHWICK, P. A. 2008. A fast hybrid time-synchronous/event approach to parallel discrete event simulation of queuing networks. In *Proceedings of the 40th Conference on Winter Simulation (WSC'08)*. 795–803.

PARK, H. AND FISHWICK, P. A. 2009. A GPU-based application framework supporting fast discrete-event simulation. *SIMUL. 86*, 10, 613–628.

PEACOCK, J. K., WONG, J. W., AND MANNING, E. G. 1979. Distributed simulation using a network of processors. *Comput. Netw. 3*, 1, 44–56.

PERUMALLA, K. S. 2006a. Parallel and distributed simulation: Traditional techniques and recent advances. In *Proceedings of the 38th Conference on Winter Simulation (WSC'06)*. 84–95.

PERUMALLA, K. S. 2006b. Discrete-Event execution alternatives on general purpose graphical processing units (gpgpus). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*. 74–81.

RASHINKAR, P., PATERSON, P., AND SINGH, L. 2000. System-on-a-Chip Verification: Methodology and Techniques 1st Ed. Springer.

REIHER, P. L., FUJIMOTO, R. M., BELLENOT, S., AND JEFFERSON, D. 1990. Cancellation strategies in optimistic execution systems. *Proc. SCS Muitlconf. Distrib. Simul. 22*, 1, 112–121.

RYBACKI, S., HIMMELSPACH, J., AND UHRMACHER, A. M. 2009. Experiments with single core, multi-core, and gpu based computation of cellular automata. In *Proceedings of the 1st International Conference on Advances in System Simulation (SIMUL'09)*. IEEE, Los Alaminos, CA, 62–67.

SOULE, L. AND GUPTA, A. 1991. An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation. *ACM Trans. Model. Comput. Simul. 1*, 4, 308–347.

WILSON, W., FUNG, L., SHAM, I., YUAN, G., AND AAMODT, T. 2007. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. IEEE, Los Alaminos, CA, 407–418.

XU, Z. AND BAGRODIA, R. 2007. GPU-Accelerated evaluation platform for high fidelity network modeling. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*. IEEE, Los Alaminos, CA, 131–140.

Received January 2010; revised January 2011; accepted March 2011