Distributed Time, Conservative Parallel Logic Simulation on GPUs

Bo Wang
Institute of Microelectronics
Tsinghua University
wangb06@mails.tsinghua.edu.cn

Yuhao Zhu
Department of Computer Science
Beihang University
zhuyh@cse.buaa.edu.cn

Yangdong Deng
Institute of Microelectronics
Tsinghua University
dengyd@tsinghua.edu.cn

ABSTRACT

Logical simulation is the primary method to verify the correctness of IC designs. However, today's complex VLSI designs pose ever higher demand for the throughput of logic simulators. In this work, a parallel logic simulator was developed by leveraging the computing power of modern graphics processing units (GPUs). To expose more parallelism, we implemented a conservative parallel simulation approach, the CMB algorithm, on NVidia GPUs. The simulation processing is mapped to GPU hardware at the finest granularity. With carefully designed data structures and data flow organizations, our GPU based simulator could overcome many problems that hindered efficient implementations of the CMB algorithm on traditional parallel computers. In order to efficiently use the relatively limited capacity of GPU memory, a novel memory management mechanism was proposed to dynamically allocate and recycle GPU memory during simulation. We also introduced a CPU/GPU co-processing strategy for the best usage of computing resources. Experimental results showed that our GPU based simulator could outperform a CPU baseline event driven simulator by a factor of 29.2.

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. Performance, Verification

Keywords. Discrete event simulation, CMB algorithm, Gate-level simulation, GPU

1. INTRODUCTION

Today we are still witnessing a rapid increasing in the VLSI design complexity as the main-stream fabrication process moves to the 45nm technology node. For instance, NVidia recently released its next generation graphics processing unit (GPU) chip, which consists of 3 billion transistors [1]. The large scale of VLSI system certainly poses significant challenges to the verification process. In addition, the integration capacity now allows the deployment of all components of a system onto a single chip. The heterogeneity of system building blocks as well as the complicated software running on the system make the verification process extremely time consuming. In fact, currently the verification process would take over 70% of the total design turnaround time in a typical System-on-Chip (SoC) design project [2]. Among various function verification tools, logic simulation remains to be the major mechanism due to its accuracy and capability to expose arbitrary internal details. Unfortunately, logic simulation can be a lengthy process for large designs. For example, the logic simulation of a billion-transistor design could take over one month to finish [3].

As a result, it is essential to accelerate the logic simulation process to shorten the design turn-around time. In the past 30 years, a large body of research has been devoted to developing fast simulation algorithms [4]. Among different acceleration techniques, the parallel logic simulation is of key importance, because ultimately only such an approach could guarantee the scalability for future VLSI circuits [5]. Today as the single CPU performance is saturating, it is urgent to develop multi-core or many-core based parallel simulation solutions so that the momentum of IC functionality increase can be maintained. Especially, general purpose computing on many-core GPUs is recently rising as an exciting new trend to accelerate general purpose applications. For applications with proper algorithmic flow and data structures, modern GPUs could often outperform CPUs by an order of one or two magnitudes [6]. It is thus appealing to investigate how to unleash the power of GPUs to help the logic simulation.

Typically, logic simulation is performed with a discrete event driven approach on a sequential computing platform. Within such a framework, the simulation events, a.k.a., internal state updates, happen at discrete time steps. These events will be stored into a queue ordered by their timestamps. At each evaluation step, the simulator takes the event with the earliest timestamp (i.e., the event at the head of the queue), performs the logic evaluation, and then inserts newly created events into the queue. The event driven simulation algorithm is simple and highly efficient. It can be implemented on parallel machines by using multiple processors to handle different events triggered at the same time step. Of course, the efficiency of such a parallel scheme depends on the available number of simultaneous events.

Recently GPUs are emerging as a cost-efficient platform for high performance computing. Accordingly, there are already a few works on using GPUs to speed up the circuit simulation. Gulati and Khatri first developed a fault simulator on GPUs in [7] and opened the path to this new direction. In [8], the authors presented a GPU based logic simulator by implementing an oblivious algorithm in which all gates have to be evaluated at each simulation step. In [9], a novel GPU based event driven simulator was proposed and an average speed-up of 10 folds was attained.

On the other hand, a digital circuit actually offers a larger degree of parallelism than that can be directly extracted from simultaneous events. In fact, the Chandy-Misra-Bryant, or CMB, algorithm [10] and [11], defines a different parallel approach to solve the logic simulation problem. The key idea of the CMB algorithm is to decompose the simulated system into many interactive components. Such a component is represented as a logic process (LP) and maintains its local simulation time. A LP would evaluate its local input events and advance its local time provided that the global causal relations can be maintained. In other words, two or more events, which might not happen at the same global time step, can still be evaluated in parallel as long as they are independent.

Intuitively, CMB algorithm would expose a higher level of parallelism from a simulated VLSI circuit . However, CMB could also incur deadlock during simulation [12]. Researcher have developed various mechanisms to avoid or recover from deadlocks. Unfortunately, such mechanism proved to be costly in terms of CPU time. In fact, a comprehensive evaluation of the CMB based logic simulation on an ideal multiprocessor computer indicated that the overhead of deadlock prevention through sending null messages [11] generally overweighed the extra parallelism [13].

Modern GPUs offer a large number of parallel processing elements (PEs). In addition, communication among these PEs only brings upon a small overhead, which can be several orders of magnitude faster than the inter-processor communication on previous multiprocessors. Accordingly, we believe modern GPUs provide a vehicle for a truly efficient CMB implementation. In this paper, we present a fine-grain, gate-level, discrete event simulator on NVidia GPUs. Our simulator implements the original idea of distributed simulation and employs null messages to avoid deadlock. Experiments showed that our GPU based CMB simulator could outperform a sequential event driven logic simulator by a factor of around 29X. We summarize our contributions as follows.

- To the best of our knowledge, this is the first work of implementing a GPU based CMB algorithm for logic simulation.
- We developed GPU-friendly data structures to store messages passed among different logical processors.
- A memory management mechanism is proposed for GPU processing. The dynamic memory allocation and recycling framework guarantees efficient memory usage in spite of the irregular memory usage pattern that is prevalent in logic simulations. To our best knowledge, this is the first built-in memory manager for GPU based applications and it can be deployed in other applications or GPU runtimes.
- Our simulator integrates both GPU and CPU for concurrent computations. Through asynchronous computation and zero copy techniques [14], GPU and CPU could cooperate in a closely-coupled manner with little overhead.
- We developed a hierarchical optimization strategy to efficiently orchestrate memory accesses. We systematically utilized such techniques as memory coalescing, memory caching, as well as direct CPU-GPU data copying to achieve a high memory throughput. Our optimization scheme would be useful for general GPU applications.

The remaining of this paper is organized as follows. In Section 2, we introduce the background of both GPU computing and the CMB algorithm. The details of our GPU based logic simulator is explained in Section 3. Section 4 discusses the optimization techniques. We present experimental results of the simulator in Section 5. Section 6 concludes the paper and outlines future research directions.

2. PRELIMINARIES

In this section, we briefly describe the NVidia GPU architecture and its programming model, CUDA [14]. A general introduction to the CMB algorithm is followed.

2.1 GPU Computing

Recently, GPUs are becoming a high performance computing platform that everybody could afford. While traditionally based on a graphic pipeline with fixed functional units, modern GPUs are composed of hundreds of unified processing elements to support efficient data parallel processing. Meanwhile, major GPU vendors all released programming tools to ease the GPU programming practices.

In this work, we use NVidia GPUs and its CUDA programming environment[14]. The CUDA-enabled GPUs all have a similar architecture but a varying number of cores installed. The latest commercially released NVidia GPU, GTX 280, installs 30 multiprocessors, each comprising 8 streaming processors. CUDA follows a single program, multiple data (SPMD) execution model and could launch up to tens of thousands of threads concurrently. Note that the number of threads is usually much larger than the number of basic processors so that the hardware resource can be efficiently used even in the case of long memory stalls. During execution, every 32 threads on a multiprocessor are organized into a warp and follow exactly the same instruction schedule. If threads in a warp take diverge execution branches at a conditional statement, both branches have to be executed sequentially. Such an overhead can seriously drag down program performance in the cases of complex branch structures.

GPUs are backed up by a memory hierarchy with varying latency and capacity at each layer. It takes 400~600 cycles to access the off-chip global memory with a capacity of up to 4GB, while only 1 cycle to on-chip registers and shared memory with relatively lower capacity. In current NVidia GPUs, there is no cache for the global memory and thus the long latency has to be carefully managed. Fortunately, GPUs are enhanced with a memory coalescing mechanism. If a half-warp of threads access data residing in the same 128-byte segment of the memory space, these memory requests can be merged into a single memory operation and all required data can be made available after one latency. Meanwhile, the texture memory and constant memory, both located in the off-chip memory, can be cached to hide the latency. Besides the latency, the bandwidth between the host and device memories could also become a performance bottleneck. The corresponding data transfer between them should be maintained at a reasonable level. Furthermore, it could deliver relatively higher bandwidth by leveraging *pinned* or *page-locked* memory.

2.2 Discrete Event Simulation and the CMB Algorithm

The discrete event simulation model assumes that the simulated system only changes its states at discrete time points. Such a change of system state is designated as an *event*. There are two major mechanisms for parallel discrete event simulation, the *conservative* and *optimistic* approaches [4]. In this paper, we choose the conservative parallel simulation strategy, or CMB algorithm, which was firstly proposed by Chandy, Misra [10] and Bryant [11] in two independent works.

According to the CMB algorithm, a simulated system is modeled as a group of interacting logic processes (LPs). Different LPs maintain their own local simulation time and do not share a global clock. LPs communicate with each other to transfer events via messages. A *message* is composed of a logical *value* indicating the content of an event and a *timestamp* indicating when this event would happen. In this paper, the concepts of a message and a event are used interchangeably hereafter. A LP has several inputs and an output. At each simulation step, a LP may receive several new messages from its inputs and generate one message at its output. In digital circuits, it can be guaranteed that the messages received by the same input are chronological, i.e., a later received message would always happen later than previous ones. When evaluating a LP, the simulator extracts the messages with the latest timestamp from all input channels and identifies the smallest one, T_{min}, among them. Then the local time of this LP will be updated to T_{min} and we can safely evaluate all messages received before T_{min}.

Care must be taken for the CMB algorithm to prevent deadlocks. For instance, a cyclic dependency among LPs may lead to deadlock. The most commonly used deadlock avoidance technique is through the introduction of *null* messages.[10] At a certain time step, a LP, *A*, would send a null message with timestamp T_{null} to another LP, *B*. Upon receiving the message, LP *B* would know that there is no other message with timestamp smaller than T_{null} to be sent from *A* to *B*. With null messages, it could be formally proved that CMB algorithm would correctly proceed until completion [10].

An implementation of the CMB algorithm needs the following data structures for each LP: (1) *state variables*, (2) a *message priority queue* containing all pending messages, (3) the *local simulation time* and *the timestamps* of latest received messages in each input channel.

3. GPU BASED LOGIC SIMULATOR

To simulate a circuit, our simulator loops through three consecutive phases: *primary input update*, *input pin update* and *gate evaluation*. In the *primary input update* phase, the primary input signals are extracted from the primary input queue and inserted into the message queues of the pins on first-level gates. In the *input pin update* phase, the output signals generated by each gate are fetched and inserted into the input pins driven by that gate. Finally, in the *gate evaluation* phase, the earliest events in the input pin FIFOs are extracted and then the new states of gate outputs are calculated according to the gate types and input pin values. Similar to [7], the logic evaluation can be done efficiently through looking up a truth table of all gates already stored in the constant memory. These phases are organized as three GPU kernel functions coordinated by the CPU.

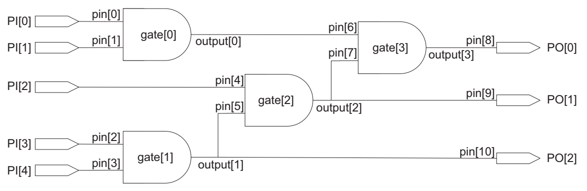
Since GPUs are designed for data-level parallel execution, we mapped the CMB algorithm to GPU in the finest granularity. In the *primary input update* phase, one thread is responsible for handling a primary input. In the *input pin update* phase, one thread is assigned to manage an input pin. In *the gate evaluation* phase, a thread is employed to evaluate a different gate. Figure 1 presents the pseudo-code of our algorithmic flow, in which the "*for each"* primitive indicates that the following operation can be executed in parallel. The advantages of such a fine-grained mapping strategy are twofold. First, the computation of each thread can be simplified and thus prevents the memory accessing overhead and redundant computations introduced by complex logic. Secondly, divergent branches would be minimized due to a higher level of structural regularity in the fine-grained objects. In the remaining of this section, we explain the details of the simulator.

```
while not finish
    // kernel 1: primary input update
    for each primary input(PI) do
        extract the first message in the PI queue;
        insert the message into the PI output array;
    end for each

    // kernel 2: input pin update
    for each input pin do
        insert messages from output array to input pin;
    end for each

    // kernel 3: gate evaluation
    for each gate do
        extract the earliest message from its pins;
        evaluate the message and update gate status;
        write the gate output to the output array;
end while
```

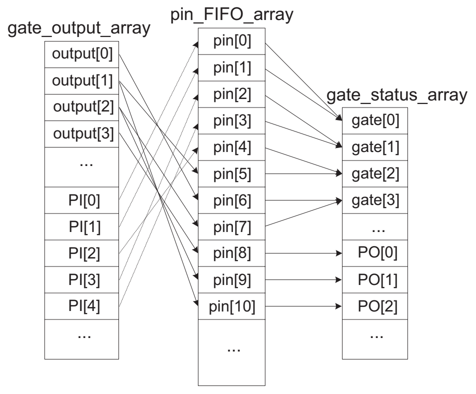
Figure 1. Processing flow of GPU based logic simulation

3.1 Fundamental Data Structures

In our simulator, we store the simulation status in three linear arrays, namely *gate_output_array*, *pin_FIFO_array* and *gate_status_array*. The *gate_output_array* array stores messages generated by each gate. The *pin_FIFO_array* array keeps track of the messages received on each pin. The *gate_status_array* array stores related information, including the current logic value of a gate and the time of the last received message among input pins. Figure 2(a) is a simple circuit and its corresponding data structures are shown in Figure 2(b). It is worth noting that we do not use a priority queue for each gate, which is required in the traditional implementation of CMB. The data stored in the priority queue are distributed to the *pin_FIFO_array* array. The reason for such a data organization will be discussed in Section 3.3.



(a) A sample circuit



(b) The data structure corresponding to the sample circuit

Figure 2. Fundamental data structures for simulation

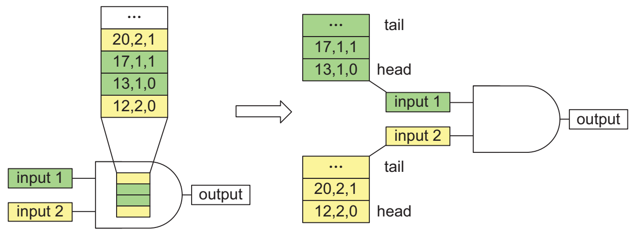


Figure 3. The mapping from a gate-wise priority queue to distributed FIFOs. Each message consists of a timestamp, a pin index, and a value.

3.2 Message Passing on GPUs

Simulated objects (gates in our case) rely on messages to interact in a distributed simulation process. However, current GPUs are only equipped with a shared memory mechanism. Accordingly, we designed a mapping strategy to convert the message passing pattern into a shared memory pattern through a global array with fixed entries for every gate output. During each simulation iteration, the message transmitted by one gate is written to its corresponding entry in the array. Then in the update kernel, the message is read from that position and inserted into the message FIFO of corresponding input pins.

3.3 Message Queue Management

As described in Section 3.1, a priority queue for each gate is required in the original CMB algorithm. Generally speaking, a priority queue can be realized as a minimum heap [15]. However, maintaining a heap involves unpredictable branches and loops, which hinder the efficient mapping onto GPU's SIMD execution model. To overcome this problem, we separated the priority queue of a gate into multiple lightweight FIFOs so that each pin has its own FIFO. These FIFOs store messages in a distributed manner. Figure 3 is an illustration.

With this transformation from a gate-wise priority queue to a group of distributed FIFOs, the insert operation for a priority queue is no longer needed, because messages arrive at the same input pin are automatically chronological. The newly arrived message can thus be safely placed at the end of the FIFO. Meanwhile, if a newly arrived message has the same logical value as the latest one in the FIFO, it will not be added into the FIFO. This mechanism greatly reduces the number of messages during the simulation and is proved to be essential in the experiments.

The proof of correctness for the transformation can be outlined as follows. In the traditional CMB algorithm, a message extracted for the evaluation of a gate has the smallest timestamp among all the messages received by the gate. After transformation, we now pick up a message with the smallest timestamp among all messages at the head of each pin FIFO. Because messages delivered to the same pin are arranged chronologically, the message at the head of FIFO always has the smallest timestamp. This "minimum of minimum" approach automatically guarantees the correctness.

```
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;
```

Figure 4. FIFO Structure

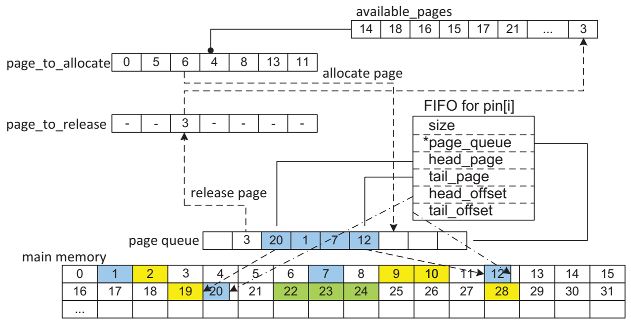


Figure 5. Illustration of a sample memory layout

3.4 Dynamic GPU Memory Management

The distributed FIFO structure in Section 4.1 well fits the SIMD architecture of GPU. However, it is difficult to decide the most appropriate FIFO size for each pin before simulation. Small FIFOs suffer from frequent overflows, while large ones result in excessive usage of the GPU memory. In our experiments, we observed considerable irregularity in the message distribution at different pins. The number of messages of those *"hot"* input pins can be orders of magnitude more than that of the *"cold"* input pins. For a more efficient memory usage, we introduced a paging mechanism to allocate and release GPU memory dynamically.

Our dynamic memory management functions as follows. First, a large bulk of memory is allocated on GPU and uniformly divided into a large number of small pages of the same size, e.g., 128 bytes in this work. A page would be the minimum unit of memory management. Each FIFO for an input pin can have up to *MAX_PAGE_NUM* pages. During runtime, the available pages are assigned to each FIFO upon request. An assigned page is recycled when it is empty. The FIFO structure is defined in Figure 4.

Before the simulation starts, each FIFO is pre-allocated with a single page. The indexes of all other empty pages are inserted into a global *available_pages* FIFO. When the pre-allocated page *i* in FIFO *j* is full during the simulation, the GPU kernel will check the *j*-th element in the *page_to_allocate* array to find a new page and mark this page as used. When an allocated page *k* in FIFO *j* is empty, it will be released to the global *available_pages* FIFO by writing the index *k* to the *j*-th element in *page_to_release* array.

When one iteration finishes, the control flow returns to the host thread, which would then pick out all used elements in *page_to_allocate* array, fetch new empty pages from *available_pages* and write their indexes into the corresponding entries marked at used in *page_to_allocate*. The host thread also checks the *page_to_release* array and recycles the released pages. Newly released pages are re-inserted into the *available_pages* FIFO.

A dynamic memory management mechanism is crucial for GPU computing, because all modern real-world applications depend on runtime memory manipulations. Accordingly, our implementation can be also deployed into other GPU applications or a GPU runtime to improve the efficiency of memory usage. Although we independently proposed and developed the GPU memory manager, we found that NVidia also developed a similar memory allocation system during writing of this paper. In our future work, we are going to compare the performance of these two memory management implementations.

3.5 Adaptive Issuing of Input Patterns

In most cases, the message generation speed is higher than the message consumption rate. For large designs with long sequences of input patterns, the GPU memory might be fully occupied. To prevent such a situation, the stimuli are not issued in a single shot, but in multiple passes. When to pause the issuing is determined by the occupation of the *available_pages* FIFO. The issue of stimuli is paused if the number of available pages in the *available_pages* FIFO is lower than the pause threshold,.

While the stimuli issuing are paused, the primary inputs will send *null* messages with the timestamp the same as the last message from this input. As the simulation continues, some occupied pages may be released gradually. When the number of available pages is higher than the resume threshold, the issuing process resumes.

4. PERFORMANCE OPTIMIZATION

With techniques presented in Section 3, we can build a function-correct parallel simulator. However, it still requires extensive optimizations to guarantee a desirable simulation performance. In this section, we describe our strategies to improve our simulator for a higher throughput.

4.1 CPU and GPU Co-Processing

In our simulation flow, CPU thread needs to update the *page_to_release*, *page_to_allocate*, and *available_pages* FIFOs. It is worth noting that the input pin update phase only needs memory allocation operations, while the gate evaluation phase only requires memory release actions. As a result, we can overlap the updating of *page_to_allocate* array with the gate evaluation, and the updating of *page_to_release* array with the input pin update. The overlapped processing can be realized using CUDA's *stream* mechanism. By asynchronously initializing GPU kernels in a stream, CPU can work simultaneously.

To further reduce CPU processing time, we adopted a *group flag* strategy. Pins of gates are distributed into small groups of 32 (i.e., warp size) according to the pin index. Each group is then marked with a *flag*. A flag is set to be true if any pins in this group require new pages or release free pages within the current simulation iteration. When processing the *page_to_allocate* array or the *page_to_release* array on CPU, only the group whose flags are true will be examined and processed. This strategy delivers around a 10x speed-up of the CPU operation in the experiment.

4.2 Low Overhead Dynamic Memory Management

The dynamic GPU memory management is essential for large circuits simulation. However, the CPU/GPU co-processing requires explicit memory copy of the *page_to_allocate* and *page_to_release* arrays between the host and device in each iteration. Though these two arrays are not very big, the explicit memory transferring incurs a large overhead. To work against the inefficiency, we took advantage of a new GPU feature, *zero copy* [14], which enables the access to host memory be automatically overlapped with kernel execution. The timing of zero-copy is illustrated in Figure 6.

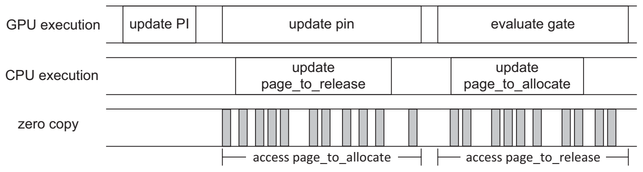


Figure 6. CPU/GPU co-processing

4.3 Memory Optimization

Uncoalesced accesses to the global memory are time-consuming and should be avoided as much as possible. For more efficient memory accesses, we stored all input pin FIFO descriptive variables (i.e., *head*, *tail* pointers and *size* value) as a structure of arrays (SOA). Coalesced accesses are guaranteed because pins are assigned to threads in order.

We also store the read-only data in the texture memory and constant memory, because they are cached. The data for truth tables and gate delays, which can be determined offline, are located in constant memory. The circuit topological information extracted at runtime are saved in texture memory. This storage pattern significantly reduces the latency of irregular memory accesses.

5. EXPERIMENTAL RESULTS

We selected a group of open-source IC designs publicly available from OpenCores.org[16] and ITC99 Benchmarks (2nd release)[17] as the test cases. These designs cover a wide range of typical circuit styles, ranging from random combinational logic, regular datapaths, to control logic. The characteristics of these circuits are listed in Table 1. The ITC99 circuits are released as gate level netlists. The other designs are downloaded as RTL Verilog/VHDL code. We then synthesized the designs by Synopsys Design Compiler with a 0.13um TSMC standard cell library. The testbenches used in experiments are released with the designs. The only exception is b18, we use randomly generated stimuli to check the average behavior of input patterns.

Table 1. Designs for simulation

Design	# Gates	Description
AES	13118	AES encryption core
DES3	53131	DES3 encryption core
R2000	8284	MIPS 2000 CPU core
M1	14850	3-stage ARM core
JPEG	117701	JPEG image encoder
NOC	64095	mesh routing switches for a NOC
SHA1	5616	Secure Hashing algorithm core
b18	72712	2 Viper processors and 6 80386 processors

5.1 Irregular Distribution of Pin Activities

In our simulation experiments, we observed significant variation in the peak FIFO sizes at different pins. In fact, the maximum number of messages in a FIFO could vary by several orders of magnitude over a design. In Table 2, we list the distribution of pins with regard to their peak message numbers, which are divided into 5 levels shown in the first column. The data were collected on 5 randomly selected designs after 50,000 simulation iterations. It can be seen that most pins receive less than 1000 messages during the simulation, but there do exist *hot* pins that have over 10,000 messages in its FIFO. Such an observation clearly justifies the indispensability of our dynamic memory allocation mechanism.

Table 2. Pin Count of Peak Message Number

Peak number of messages	DES3	R2000	M1	JPEG	NOC
0-9	68170	15747	24788	178728	157891
10-99	63895	11567	16506	117820	23297
100-999	3960	53	663	2913	590
1000-9999	2253	2	3	202	0
10000-50000	85	0	4	0	15

5.2 Efficiency of Dynamic Memory Management

Our dynamic memory management makes it possible to recycle FIFO memory in the GPU global memory space. Such recycling behaviors include page allocation and page release, both executed under the control of a host side arbitrator detailed in Section 3.4. As a result, we are able to free unused pages for future allocation and thus keep GPU memory usage under control. Table 3 shows the number of allocation calls, the number of release calls, and the ratio of the latter to the former in all 8 designs. On average, our mechanism would recycle 47.64% of the allocated pages, which can be directly translated into the same portion of reduction in GPU memory usage.

Table 3. Memory release and allocation calls

Design	Allocation	Release	Ratio
AES	26646	11321	42.4%
DES3	9998	8004	80.1%
SHA1	2745	3	0.11%
R2000	222377	179445	89.7%
JPEG	26594	8350	31.4%
NOC	571550	20545	3.6%
M1	137657	118657	86.2%
b18	1069742	978	< 0.1%

5.3 Performance Evaluation

Using the benchmark circuits, we compared the simulation performance of our GPU simulator against a baseline CPU simulator. Rather than the CMB algorithm, the CPU simulator implements a centralized-time, event-driven algorithm, because the CMB algorithm on CPU is generally slower [13]. The results were gathered on a 2.66GHz Intel Core2 Duo server with a NVIDIA GTX 280 graphics card. The GPU program was compiled with CUDA 2.2, and the baseline simulator was compiled by gcc 4.2.4 with -O3 optimization. Our baseline CPU simulator outperforms Synopsys VCS simulator [18] by a factor of 1.4~2.0X because VCS handles complex verification features that are not implemented by us.

The simulation performance is reported in Table 4. The GPU simulator outperforms the CPU simulator on all test cases. On one half of all designs, the GPU simulations are at least one order of magnitude faster than their CPU counterparts. On average, our work brings about a speedup of **29.2X**.

Obviously, the speedup values vary considerably among different designs. Such an irregularity is mainly caused by the structure of the stimuli applied to circuits. The stimuli for R2000, M1, JPEG, and b18 have large gaps between neighboring patterns. Such a sparse stimuli results in a low activity ratio. Therefore, many threads have to be idle during gate evaluation. The current scheduling mechanism of CUDA cannot intelligently adapt to such a computation pattern. On the other hand, those stimuli with much shorter intervals between input patterns exhibit more parallelism and can efficiently take advantage of GPU hardware.

Table 4. Simulation performance

Design	Simulated cycles	CPU simulation time (s)	GPU simulation time (s)	Speedup
AES	42,935,000	109.90	4.45	**24.70**
DES3	30,730,000	183.11	4.50	**40.66**
SHA1	2,275,000	56.66	0.41	**138.20**
R2000	28,678,308	9.20	3.15	**2.92**
JPEG	26,132,000	136.33	43.09	**3.16**
NOC	1,000,000	5389.42	347.95	**15.49**
M1	99,998,019	118.48	22.43	**5.28**
b18	19,125,000	37.30	11.49	**3.25**

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed the first GPU based, conservative parallel logic simulator using the CMB algorithm. The algorithmic mapping is realized at the finest granularity to best match the hardware pattern. We developed new data structures and algorithmic flows to efficiently handle messages. A novel GPU memory manager is developed to dynamically recycle memory pages. As a generic technique, this memory management system can be used by many GPU applications with intensive memory operations. We also adopted a CPU/GPU co-processing strategy as well as other memory optimization techniques to enhance the performance. In our experiment, an average speed-up of 29.2X is achieved against a CPU baseline event-driven simulator.

In the future, we will study the scalability of our simulator on large, industry-strength circuits. Especially, we will also explore new techniques to combine task-level and data-level parallelism by using 4-GPU Tesla workstations and Fermi GPUs. We are also going to apply our techniques to solve problems such as system level SystemC-based simulation and network simulation.

ACKNOWLEDGEMENT

This work was partially supported by Tsinghua CUDA Center of Excellence and an Intel University program. The authors also appreciate hardware donation from NVidia.

REFERENCES

[1] NVidia White Paper, NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi.

[2] Rashinkar, P.,Paterson, P., Singh, L. System-on-a-Chip Verification: Methodology and Techniques. Springer, Ed 1, 2000.

[3] ESNUG Industry Discussion, http://www.deepchip.com/items/0421-01.html.

[4] R. M. Fujimoto, Parallel and Distributed Simulation Systems. Wiley-Interscience. 2000.

[5] M. L. Bailey, J. V. Briner Jr., and R. D. Chamberlain, Parallel logic Simulation of VLSI Systems. ACM Computing Surveys. Vol. 26, No. 3, pp. 255-294, Sep. 1994

[6] D. Blythe,. Rise of the Graphics Processor. Proceeding of IEEE, Vol. 96, No. 5, pp. 761– 778 , 2008

[7] K. Gulati and S. Khatri. Towards Acceleration of Fault Simulation Using Graphics Processing Units. Proc. DAC, 2008.

[8] D.Chatterjee, A.DeOrio, V.Bertacco. GCS: High Performance Gate-Level Simulation with GPUs. In Proc. DATE'09

[9] D.Chatterjee, A.DeOrio, V.Bertacco. Event-Driven Gate-Level Simulation with GPUs. in Proc.DAC, pp.557-562,2009.

[10] K.M.Chandy, J.Misra, Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Trans. on Software. Eng., SE-5, No.5, pp.440~452, Sep.1979.

[11] Bryant, R.E. Simulation of Packet Communications Architecture Computer System. MIT-LCS-TR-188, MIT, 1977

[12] K. M. Chandy, J. Misra, and V. Holmes, Distributed Simulation of Networks. Comput. Netw. 3, pp. 105-113, 1979

[13] L.Soule, A.Gupta, An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation. ACM Trans. On Modeling and Computer Simulations. Oct. 1991.

[14] NVidia, CUDA Programming Guide 2.2.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990.

[16] OpenCores, http://www.opencores.org/

[17] ITC99 Benchmarks, http://www.cad.polito.it/tools/itc99.html

[18] Synopsys VCS simulator (Apr. 2008 release).