Adaptive Partition-Based Logic Simulation Using GPGPU

Meng Zhang, *Member, IEEE,* Yuxuan Zhang, Wei Yang, Yaowen Kai, Tingcun Wei, and Xiaoya Fan

***Abstract*—With the improvement of the gate complexity, the verification overhead becomes more decisive for VLSI design cost. In order to reduce the simulation time, a adaptive partition based parallel method of VLSI logic simulation with GPGPU is addressed in this paper. The numerous arithmetic blocks of GPGPU is utilized simultaneously for disparate circuit macros. The partition strategy we proposed shows a sufficient flexibility to balance the different work load in parallel threads and fit the feature of GPU architecture. To explore the parallelism and locality of logic simulation further, the circuit macro is organized as stream data. The data dependency between the input and output nets in one individual logical path is handled with the shared memory of GPGPU. As for different logical paths, the dependency is processed by threads synchronization. To illustrate the performance, a serial experiments is implemented in Intel CoreDuo workstation with Nvidia GTX465 GPU board. Four typical digital circuits (LDPC, DES3, OpenRISC 1200 and OpenSPARCPARC T1) are considered as the benchmark. The result of experiments demonstrate a significant speed-up is achieved by using GPGPU parallel method, comparing with the CPU serial logic simulation. In maximal case (OpenS T1), the GPGPU parallel acceleration computes 21 times faster than serial program.**

***Index Terms*—Logic Simulation, Stream Computing, GPGPU, CUDA, EDA**

I. Introduction

IN the past years, logic simulation has greatly promoted the development of circuit designing, especially in modern ASIC and FPGA designs with millions of gates. It verifies the correctness of circuit designs in the early phase of designing. The logic simulation is usually a process of software program with intensive computation, and much time consuming. With the improvement of sophisticated VLSI designing and manufacturing, efficient simulation solution is required urgently. A series of researches and works had been done to accelerate the logic simulation in recent decades.

Conventional accelerating solution of logic simulation is based on parallel computing platforms, by means of shared-memory systems or distributed systems. Simulation on shared-memory systems like multiprocessors benefits from light-weighted parallelism and communication [1], [2]. Parallel solutions on distributed systems achieve the acceleration by multiple threads and processes executing on distributed-memory computer or workstation clusters [3], [4]. There are also some special purpose simulators, speeding up the simulation with an array of special-purpose processors [5].

Meng Zhang, Yuxuan Zhang, Wei Yang, Yaowen Kai, Tingcun Wei and Xiaoya Fan are with School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, P.R.China (e-mail: zhangm@nwpu.edu.cn).

The research in [6] listed a series of aspects mostly influence the performance of parallel simulation including circuit structure, target architecture and partition. Among these aspects, the crucial problem is the strategy of partitioning and mapping, which generate separate simulation units to match the target computing architecture. A great deal of effort has dedicated in partition methods include cones [6] and clusters [3]. There are also optimization methods for partitioning with heuristic algorithms [7], [8]. However, the survey generalized that simulation performance varies dramatically from one circuit to the other due to the difference between circuits structures [9].

Recently, along with the mature of GPGPU (General Purpose Graphic Processing Unit) technology, with numerous arithmetic units integrated on single chip, low-cost high performance parallel solution for scientific computing become possible. Researchers tend to explore the potential of GPU for circuits simulation [10]–[13]. The first GPU based logic simulation was presented in [10], however the simulation performance was reduced by the barrier of data transfer between host memory and GPU off chip memory, which the result was even much lower than CPU serial program. While, later works including the cluster [11] and cone [13] both promoted the simulation on GPU. And an event-driven simulator in [12] showed a considerable performance, with macro-gate partitioning in average layers. However, previous partition algorithms ignored the limitation of GPU platform, effective utilization of on-chip resources and inherently structural feature of circuit designs. Therefore, performance of logic simulation on GPU is still expected to be improved.

The rest of this paper is organized as follow. Some essential information about GPU architecture and CUDA, which we used to completed acceleration in this paper, is introduced in Section II. In Section III, We describe the simulation model and parallel method with GPU. In order to take full advantage of parallel resources in GPU, an adaptable partition strategy is presented in Section IV. Section V presents experiments which demonstrates the significant speedup of simulation. Finally the conclusion is drawn in Section VI.

Nvidia GPU integrates hundreds of cores on single chip, and supplies parallel programming models. Fig.1 shows the CUDA architecture. The basic execution units are organized as SPs (Stream Processor). Each SM (Stream Multiprocessor) consists of several SPs. Nvidias CUDA [14] (Computing Unified Device Architecture) is the hardware and software architecture that enables general purpose programming on GPU. CUDA organizes the parallel programming with hierarchies of threads and memories. In the threads hierarchy model, threads are

defined as the finest dispatch units on SPs; each block consists of a batch of threads, corresponding to a SM block. Parallel programs are executed in SIMT (Single Instruction Multiple Thread) mode. The memory hierarchy model contains register, shared memory and global memory, and maps the former two to the on-chip memory and the later one to off-chip memory.

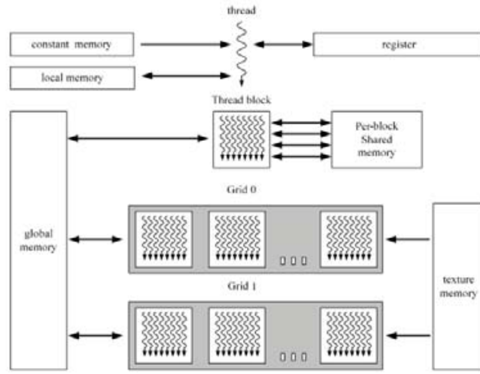


Fig. 1. CUDA Programming Model

The combinational logical part of a sequential circuit is extracted through ranking and partitioning. Finally it is mapped to computing resources. In the simulation phase, separate partitions are dispatched as tasks to the multiprocessors of GPU. We proposed the adaptable partition strategy targets the GPU platform, which is flexible and efficient. In this strategy, a variable partition of circuit design is instructed by related factors. These factors involved several crucial indicators of partitioning and mapping, including the capacity of a partition unit, the memory limitation of computing platform and the overlap rate indicator of partitioning. We also analyse and optimize these factors by a series experiments. Parallel logic simulation with adaptable strategy achieved considerable performance on Nvidia Geforce GTX465.

The rest of this paper is organized as follow. In Section II, We describe the simulation model and parallel method with GPU. As a parallel computing platform, GPU is of the architecture specified. CUDA GPU has a stream computing liked programming model. Some essential consideration for parallel logic simulation to fit the hardware features is presented in section III. In order to take full advantage of parallel resources in GPU, an adaptable partition strategy is presented in Section IV. Section V presents experiments which demonstrate the significant speedup of simulation. Finally the conclusion is drawn in Section VI.

II. The Simulation Framework

Modern GPU provides a numerous of arithmetic cores, and supports abstract programming model to utilize these resources for general purpose computing. In this section, we analyzed the programming model of GPU, the common procedure of logic simulation, and the parallel simulation model on GPU.

A. Basic Simulation Model

Firstly the gate-level netlist of digital circuit design is generated by a commercial synthesis tool. Secondly, the preprocessing routines start to compile the netlist, ranking and partitioning the circuit design to match the GPU platform. Finally, structured data of the simulation object is transfer to the off-chip memory of GPU. The simulation program is driven by inputs stimulus and exports simulation outputs iteratively.

The synthesis library in our simulation experiment is a reduced subset of a standard library, including typical dual-input gates, and D flip-flop.

B. Parallel Programming Methodology

The parallel programming always adopts four steps to organize the computation: Partitioning, communication, agglomeration and mapping [16]. As for the method we proposed,the circuit information from the target netlist is extracted and the whole circuit is partitioned into basic element as much as possible firstly. According to the partition information, massive parallel threads are generated. For GTX465 GPU, we allocate at least 128 threads in one SM block. The logical gate is divided to individual thread with its correlation between input and output. However, this correlation is intrinsic feature of circuit topology. The thread executes independently in a period in which the correlation only involves with those logical gate allocated in one SM block. The communication between different SM block is triggered at a unified synchronisation time to insure the threads process do not disturbed by a high latency memory accessing. The logical gates in one logical path which is chained without branches are agglomerated to one thread. With this mapping strategy, one logical path is broke at the logical branch point by threads synchronizing. Therefore, the thread can execute without disturbation by correlative data accessing.

C. Parallel Simulation with CUDA

To explore the parallelism and capture the data locality, we adopt macros [11] as coarse-grain tasks, partitioned from combinational logical. They are generated by propagation of connected gates in rank order. Each macro can be seen as a small combinational logic unit, in which the gates are connected with local nets. Between macros, there are global nets connected for data communication. Fig.2 shows the macro partition of a circuit design.

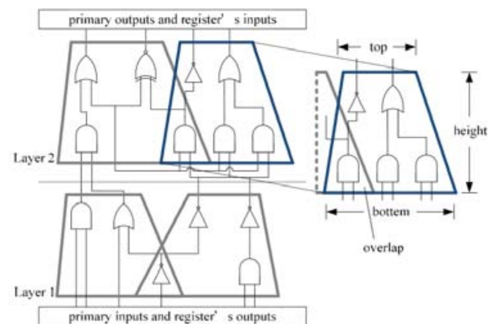


Fig. 2. Partitioning of Macros

To match the hierarchies of threads and memories in CUDA programming model, macros are scheduled and executed on

blocks, with internal gates naturally mapped to threads in a block as fine-grain tasks, and local signals captured by on-chip shared memory. Gates are evaluated individually by threads according to LUT (Look-Up Table) of truth-tables.

Fig. 3 shows the parallel simulation with macro partitioning. Each macro is an individual driven object. In oblivious mode, macros are always reevaluated, while in the event-driven simulation mode, a macro is reevaluated only when it is stimulated.

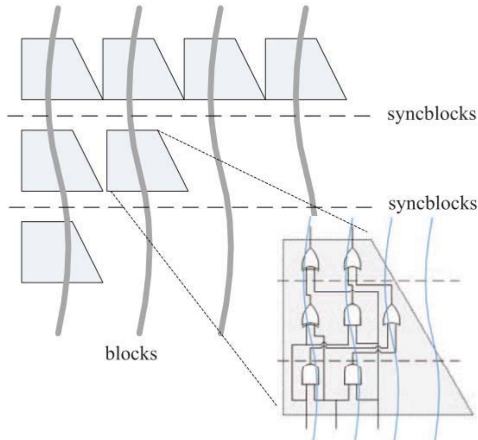


Fig. 3. Parallel Simulation Model with CUDA

The related data are organized as follow: gates and macros are mapped to the global memory; Look-up table is mapped to shared memory for high speed access; Nets are mapped to shared memory and global memory separately based on their property. Texture binding does not perform well to cache data, even worse, so we prefer to optimize the coalesced access of global memory. We also rearrange and reorganization the data to optimize both on-chip and out-chip memory access. The detailed optimization will be described in next section. In addition, global block synchronization [15] is adopted in the simulation to accomplish blocks synchronization in kernels.

III. Stream Organization

CUDA is a kind of typical program model for stream computing [14]. The parallelism of stream computing is denoted in three levels.

- **Data Level** in which the parallel computing is the most fine gain. Eg. a set of instructions in a loop executes repeatedly.
- **Kernel Level** in which the different threads run simultaneously. As for GPGPU, plenty of threads can be executed in a CUDA SM block at the same time, even though the number of threads exceeds execution units'. The Local register file in a SM block is used to store the context to support threads switch with low latency.
- **Task Level** in which the disparate algorithmic tasks can be proceed simultaneously. The individual memory space is located for each task proceed in this level. Task parallel is still not involved with the GPU hardware.

We optimized threads and data organization with the former two features which are performed in one CUDA SM block. In one macro, the logic gate is arrayed with different SYNCBLOCK as Fig. 3. Each thread in the block process one logical path. According to the partition strategy which is described in next section, the logic result of gates in one SYNCBLOCK level are independent with each other although those in different SYNCBLOCK levels are always correlative. Then these threads should be synchronized at every SYNCBLOCK.

The inputs and outputs value is a intermediate result for every gate logical simulation. A LNVT (Local Nets Value Table) is used to store this intermediate data, which LNVT is arrangement with the Netlist relationship index. For example, in Fig. 4, a AND gate is evaluated. The value of Input0 and Input1 is found in LNVT with index 11 and 12, which is store as the outputs of last level simulation. The index of logical simulation LUT is composed of these two input values and logical type AND. The result from LUT is stored as output in LNVT with index 45. In an SM block, A thread is allocated to one gate simulate.

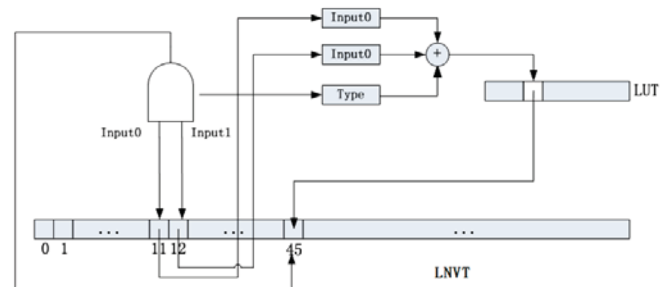


Fig. 4. Logical Simulation for a Gate

Since the CUDA GPU has a hierarchy memory scheme and provides a private low latency shared memory for every SM block, the efficient of parallel logical simulation is depended on the data locality in shared memory. In CUDA GPU, constant value can be stored in a cacheble constant memory. Only when all threads access the same unit, the large latency of off-chip memory access can be hidden. However LUT is indexed and accessed randomly by threads. The constant memory will lose the low latency, if LUT is mapped to them. Therefore LUT is loaded to shared memory when every kernel initials. Due to the similar accessing property, we also allocate a individual space in shared memory for LNVT. Obviously the amount of whole netlist is much large than shared memory. It is more convenient that the scale of netlist in shared memory is shrunk to macro level. The LNVT is constructed with a circuit macro netlist.

According to our simulation platform, logic gate calculating inside the macro is mapped to thread. The LNVT is composed of the intermediate values (I, 0, x, z) produced and consumed by the logic gate computing. The constant input can be handled with the different procedure that the constant value is decided by netlist information without searching LNVT. A sample solution is to locate the constant value by program branch. However every branch in a GPGPU warp is processed serially which will slow down the performance. An alternative method is proposed in this paper, which the constant values is stored

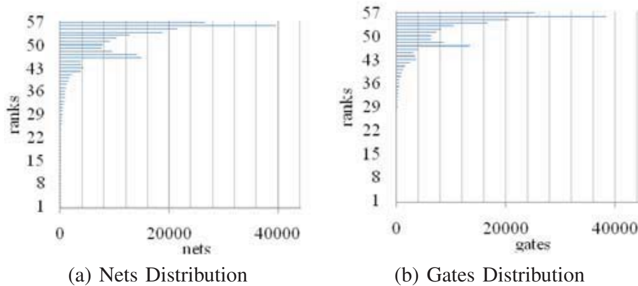
at the end of LNVT and can be found at the LNVT as well instead of processing branches. With this method, the logic simulation is mapped in a unified mode which the wire intermediate values and the constants are both stored in shared memory.

IV. Adaptable Partition Strategy

To achieve acceleration of simulation on GPUs, the challenges are exploring parallelism of tasks and locality of data. We analyze the structural character of circuit design and the feature of GPU platform, to design an adaptable partition strategy for promoting the efficiency of logic simulation.

A. Statistics and Analysis

As mentioned above, partitioning is based on the ranking of circuit design in the first step. Actually, ranks also imply the critical route of the design. While, the distribution of elements always tend to accumulate in several ranks, which is largely determined by the structural characteristic of circuit design. As an example, Fig. 5a shows the ranking statistics of OpenSPARC T1 [17]. The distribution situation of nets and gates are shown in Fig. 5a and Fig. 5b respectively, rank with ALAP (As Late As Possible). The result confirmed that most nets and gates are gathered in some intensive ranks nearby the outputs of the combinational logic part.



(a) Nets Distribution (b) Gates Distribution

Fig. 5. Statistic of the Ranking

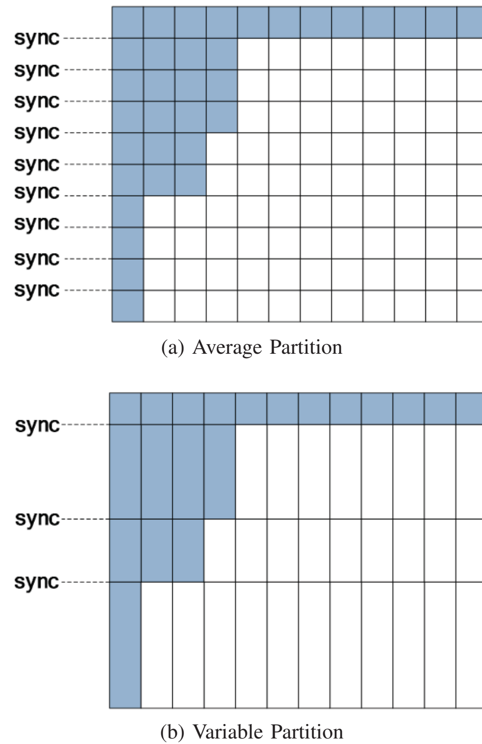
B. Optimization on Partitions

The challenges to explore simulation performance on CUDA can be attributed as:

- load balance between simulation tasks;
- opportune utilization of limited on-chip resources;
- efficient data communication, and synchronization of parallel simulation tasks.

Previous works partition macro in average-height layers, which separated ranks into equal-height layers. As the distribution of elements in ranks, macros were also scattered intensively in some of these layers, and sparsely in the retired layers. In such cases, the partitioning result could not map the target architecture perfectly. Most of those blocks are merely idle to wait for synchronization in cycles as in Fig. 6a. It wastes the computing resources and limits the performance.

Although the frequency of elements in ranks reflects the circuits structure, it is complex and unreliable to model the partition of ranks with statistic information directly. And optimization on ranking method is difficult when the size of circuit is large. To achieve variable partition as in Fig. 6b, we introduce an adaptable partition strategy.



(a) Average Partition

(b) Variable Partition

Fig. 6. Illustration of Partition

C. Adaptable Partition Strategy

The adaptable partition strategy targets on constructing opportune simulation partitioning to inspire the computing potential of the platform. It is a dynamical partitioning process with probed adjustment instructed by constraints, to optimize the partitioning in the following aspects:

1) Load Balance: Unbalanced partitions form a critical path of the parallel processing, which limit the efficiency and performance. To achieve global load balance between parallel tasks, we introduce the parameter C, represents the appointed capacity of each macro, as a slack parameter to instruct the generation of a macro. In the partitioning of each macro, the amount of gates inside is expected approach to the capacity, by adjusting the top of this macro gradually. The capacity should be selected prudently considering the balance between the computation ability of a single SM on GPU and the cost of data communication and synchronization.

2) Platform Limitation: In a macro, the simulation signals of internal gates are attached on connected local nets. And as an extreme situation determined by the connection of gates in ranks, exponential explosion of nets might occur in a macro with a certain probability. So this is the premier premise that, the amount of nets in each macro should be restricted by a threshold quantity T, to ensure that 1) on-chip shared memory could able to accommodate all the local nets; 2) the size of shared-memory is enough for switches between blocks in a SM to hide long latency operations.

3) Overlap Rate: As traditional partition and simulation algorithms, to reduce data communication in parallel tasks, a gate might be overlapped among different related macros. To promote parallelism in such decoupling method, the price is the increment of redundant computing load. And with the rise of overlap rate, the redundant computing load finally hamper the overall performance.

Meanwhile, the overlap rate is also a phenomenon reflects the partitioning. With the impaction of the factor C and L we discussed above, the overlap rate in a layer is in direct proportion to the height of this layer. For example, a small value for height leads to fat macros partitioned in the layer. However, with the growth of height, the algorithm shrinks the tops of macros. As a result, macros tend to narrow tops even degrade to cones, with a sharply increment of the quantity and the overlap in this crowed layer.

If the overlap rate is lower than a given threshold, the performance will benefit from it throughout. Therefore, we introduce the threshold R, represents an acceptable upper bound of overlap rate of the partition in a layer, to control the partitioning, and also to prevent the degradation of partitioning from macros to cones. In this strategy, we attempt to search the optimal point of overlap rate for the tradeoff between parallelism and the redundant load.

4) Patition Algorithm: The partition algorithm separates ranks into layers from rank 0 to the last rank successively. When the algorithm turns to generate a new layer, the height of this layer is initialized maximally, that is, the probe attempts a maximum height primarily. Then, by continuous adjustment and probing, the height decreasingly approach to the optimization. The probing and partitioning is guided by following constraints:

- The amount of gates in each macro should approach the appointed quantity C, for load balance.
- The amount of nets in each macro should be lower than the value of threshold T, considering the limitation of platform.
- The overlap rate of partitioning in a layer should be lower than threshold R, to achieve the optimization of performance.

If the layer could not content all of the constraints, the height chosen for this layer will be decrease, then, the probing continues. Once all of the constraints are contented, the layer and macros partitioned are confirmed. Then, the algorithm turns to generate the next layer, until all ranks are separated in layers. Fig. 7 shows the algorithm flow.

The algorithm instructed by these constraints, to partition the circuit design based on its structural character and platforms limitation, for inspiring the performance of simulation on GPU.

V. Experimental Results

In this section, we introduce our experimental method. Then, we analyze and optimized the factors of adaptable strategy by a series experiments. Finally, the parallel solution on GPU is implemented and evaluated.

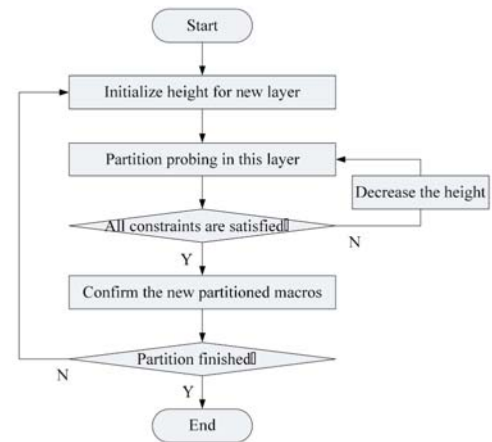


Fig. 7. Adaptable Partition Algorithm Flow

TABLE I
Parameters of Experimental Platform

Proc.	Type	Frequency	Memory	Description
CPU	Intel Core T2400	1.83GHz	DDR2 2GB	Dual Cores
GPU	NVIDIA GTX 465	1.20GHz	GDDR5 1GB	11 SMs, 32 SPs/SM

A. Experimental Platform

The simulation experiments, including the optimization and the parallel solution, have been conducted on Nvidia Geforce GTX 465. And the serial simulation is tested on an Intel Core Duo T2400 as a comparison. The specification of the experimental platform is shown in Table I. The experiments involved on a variety of circuit designs from purely combinational logic like LDPC, complicated sequential logic like DES, to processors as OpenRISC 1200 and OpenSPARC T1 [17], [18]. The synthesis result of these designs is shown in Table II In order to simplify the simulation, several essential modification on the circuits under-tested. Additionally, since the memory module simulation is exhausted and worthless for the data path circuit logic verification, which is always simulated individually, the memory modules is removed from the RTL (Register Transfer Level) designs to avoid the synthesis and simulation of RAM instead of extracting RAM interfaces on top module. The testbench which is coded in Verilog language is dumped from the commercial simulator (Synopsys VCS 7.1.2). The dumped testbench file is transformed to binary file as the input data of the GPU simulation experiment. Simulating inputs in our experiments were generated based on main clock frequency.

TABLE II
Synthesis Information of Circuits

Circuits	Nets	Gates	DFFs
LDPC	62475	60752	0
DES3	86254	52372	6984
OpenRISC 1200	28297	25924	1891
OpenSPARC T1	223864	189587	28376

The cycle accurate simulation is implemented in our platform. The information of testbench for these circuit designs is shown in Table III. The simulation cycles of later two testbenches depend on the executable programs respectively. The numbers of simulation cycles are different, however the simulation time consumed does not dependent on the input size completely because the logical paths of each are actually quite distinct.

TABLE III
TESTBENCH INFORMATION

Circuits	Testbench Type	Simulation Cycles
LDPC	Random	25001
DES3	Random	205002
OpenRISC	OpenRISC Program Testbench	150012
OpenSparc	OpenSparc Program Testbench	41052

B. Optimization

The selection of value for the partition factors has an important impact on the performance. They are also influenced with each other in the partitioning. We design the experiments to select the optimal group of values for these parameters, by evaluating the performance of simulation with corresponding adaptable partitioning.

Firstly, threshold L in the experiment is naturally selected as multiple lower than the shared-memory limitation.

Secondly, we designed the experiment to search the optimal value for R in a successive interval. The value for parameter C was set 800 in this experiment. The statistic of operating time is shown in Fig. 8. Experimental result shows in the interval between (0.2, 0.5), the partition for simulation could achieve stable and efficient performance.

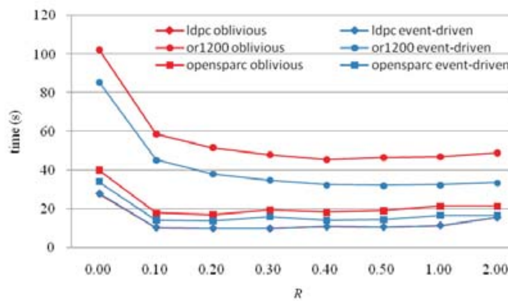


Fig. 8. Simulation Performance Influenced by Parameter R

Thirdly, we also cared about the selection of C by testing the performance of simulation partitioned with a series value for macros capacity C, with 0.2 set for R. The result in Fig. 9 manifest that the performance of parallel simulation is inversely proportional to the value select for C. When set 800 for C, the parallel simulation achieved the minimal operation time. It demonstrated the importance of data locality contribute to the overall performance. However, with the growth of C, the practical capacity of macros is restricted by other factors like parameter L, and the parameter C loses control of the

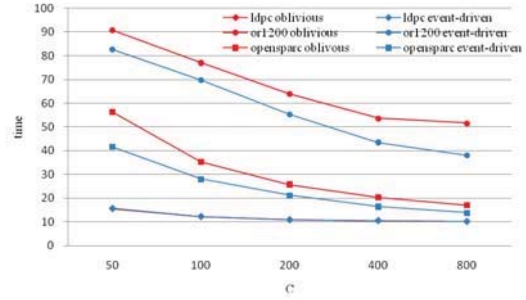


Fig. 9. Simulation Performance Influenced by Parameter C

partitioning. Therefore, the operation time of simulation tends to plane gradually as shown in this chart.

The optimized partition result is shown in Fig. 10, with 0.2 for R and 800 for C. The experimental result indicated that, the adaptable partition strategy is flexible to respond different circuit designs and structure diversity between these circuit designs. Meanwhile, it has advantages in efficient utilization of computing resources, and also in saving expensive communication and synchronization between blocks map to the CUDA platform.

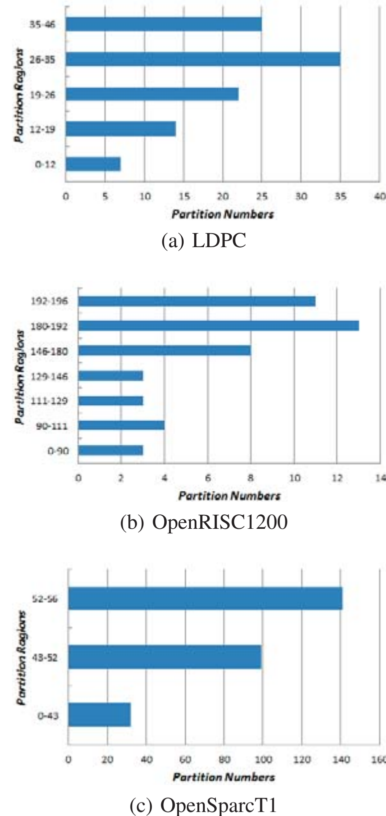


Fig. 10. Variable Partition Results

C. Performance

The parallel solution with adaptable partition strategy has been implemented and tested. The performance of parallel simulation on GPU is compared with corresponding serial

simulation on CPU. The experimental results of oblivious and event-driven simulation are illustrated in Fig.11.

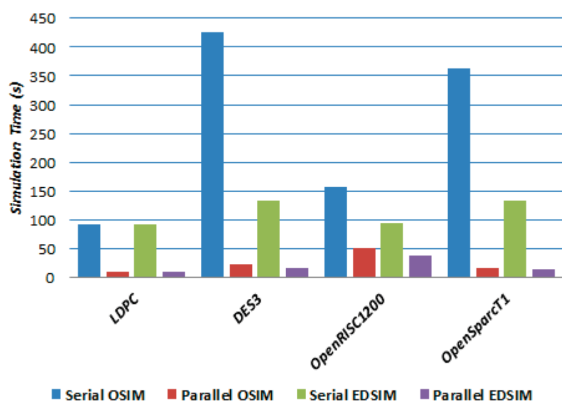


Fig. 11. Simulation Performance

A significant speed-up of the parallel simulation on GPU comparing with serial simulation is shown in Table IV. The parallel event-driven simulation, benefited from acceleration with GPU, performed the fastest execution speed. While, oblivious simulation generally achieved a better acceleration. The major aspect affects the simulation acceleration is the structure of circuit designs. A regular layout, with balanced distribution of ranks and elements, leads to balanced partition, which is critical for parallel simulation. Therefore, some of these simulations have stable performance, like LDPC. And although in the similar design type as OpenRISC1200 and OpenSparc T1, the performances can be categorically different from each other.

TABLE IV
SIMULATION SPEED-UP

Circuits	Oblivious Speed-up	Event-Driven Speed-up
LDPC	9.032	8.985
DES3	18.447	7.876
Orl200	3.041	2.480
OpenSparc T1	21.382	9.686

D. Data Locality Analysis

Since the memory of GPGPU is organized with a kind of hierarchical model, the data locality should be taken full advantage of to acquire high simulation performance. In order to monitor the locality of the GPU simulation program, we evaluate our simulation program with Compute Visual Profiler provided by Nvidia. The kernel run time and the off chip memory accessing time of OpenRISC and Opensparc are visualized as Fig.12. The report of Profiler demonstrates that:

- For OpenRISC1200, kernel time is 97.84% of GPU time, global memory copy time is 0.20%;
- For OpenSparcT1, kernel time is 95.33% of total GPU time, global memory copy time is 4.06%.

In this visualized result, there is no time overlap between memory accessing and kernels on GPU for each simulation experiment. Fig.12 also illustrate the global memory copy time is much less than kernel run time, which indicates that the kernel achieves a high utilization factor of the shared memory and L2 cache as expected so that it does not need to access the global memory frequently.

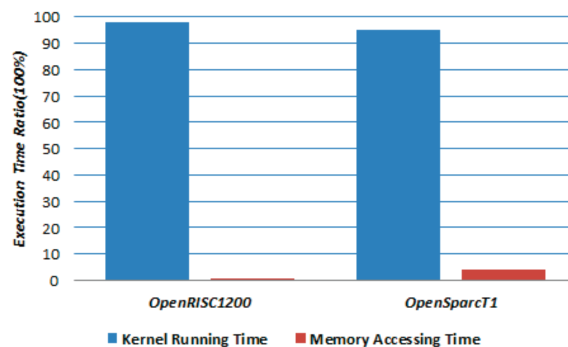


Fig. 12. Execution Time Ration

VI. CONCLUSION

A parallel solution of logic simulation on GPUs is proposed in this paper. By extracting and partitioning the combinational logical part of the design, macros are generated as course-grain tasks to map to the parallel platform firstly. To develop further performance, an adaptable partition strategy is presented, which targets at the CUDA platform. In this strategy, a variable partitioning of circuit design is controlled by related factors. These factors are crucial indicators of partitioning and mapping include the capacity of a partition unit, the memory limitation of computing platform and the overlap rate indicator of partitioning. We analysed and optimized the selection of these factors by a series experiments. Finally, we illustrate the strategy on the simulation of some typical circuit designs. The experimental result indicated a considerable improvement of performance. The parallel simulation solution by using GPGPU achieves 21x speed-up maximally comparing with serial simulation on CPU.

ACKNOWLEDGMENT

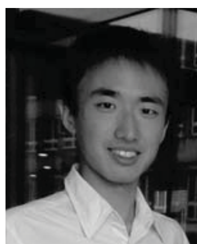
This research is complemented in Engineering Research Center of Embedded System Integration, Chinese Ministry of Education and School of Computer Science and engineering, Northwestern Polytechnical University. The work is supported by the National High-technology Research Development Project: "Actively Adaptable Architecture Research for Stream Computing" (No. 2009AA01Z110). Additional support has been provided by the Innovation Fund of Northwestern Polytechnical University "Parallel logic simulation based on GPU" (No. Z2011119), and NSFC of China (No. 60773223, No. 61003037, No. 60736012).

REFERENCES

[1] L. Soule, T. Blank, "Parallel Logic Simulation on General Purpose Machines", in *25th ACM/IEEE Design Automation Conference*, 1988, pp. 166–171.

[2] D. A. Reed, A. D.Malony, B. D. McCredie, “Parallel Discrete Event Simulation Using Shared Memory”, *IEEE Transactions on Software Engineering*, vol. 14, no. 4, pp. 541–553, 1987.
[3] C. Sporrer, H. Bauer, “Coralla Partitioning for Distributed Logic Simulation of VLSI-Circuits”, in *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 1993, pp. 85–92.
[4] Y. Matsumoto, K. Taki, “Parallel Logic Simulation on a Distributed Memory Machine”, in *Proceedings. European Conference on Design Automation*, 1992, pp. 76–80.
[5] Y. Hur, S. A. Szygenda, “Special Purpose Array Processor for Digital Logic Simulation”, in *Proceedings of the 28th Annual Simulation Symposium*, 1994, pp. 297–302.
[6] M. L. Bailey, J. V. Briner, R. D. Chamberlain, “Parallel Logic Simulation of VLSI Systems”, *ACM Computing Surveys*, vol. 26, pp. 255–294, 1994.
[7] K. Hering, G. Runger, S. Trautmann, “Modular Construction of Model Partitioning Processes for Parallel Logic Simulation”, in *International Conference on Parallel Processing Workshops*, 2001, pp. 99–105.
[8] S. Patil, P. Banerjee, C. D. Polychronopoulos, “Efficient Circuit Partitioning Algorithms for Parallel Logic Simulation”, in *Proceedings: Supercomputing 89*, 1989, pp. 361–370.
[9] R. D. Chamberlain, “Parallel Logic Simulation of VLSI Systems”, in *Proceedings of the 32nd Design Automation Conference*, 1995, pp. 139–143.
[10] A. S. Perinkulam, “Logic Simulation Using Graphics Processors”, Ph.D. dissertation, University of Massachusetts, 2007.
[11] D. Chatterjee, A. DeOrio, V. Bertacco, “GCS: High-Performance Gate-Level Simulation with GP-GPUs”, in *2009 Design, Automation and Test in Europe Conference and Exhibition*, 2009, pp. 1332–1337.
[12] D. Chatterjee, A. DeOrio, V. Bertacco, “Event-Driven Gate-Level Simulation with GP-GPUs”, in *46th ACM/IEEE Design Automation Conference*, 2009, pp. 557–562.
[13] A. Sen, B. Aksanli, M. Bozkurt, M. Mert, “Parallel Cycle Based Logic Simulation using Graphics Processing Units”, in *9th International Symposium on Parallel and Distributed Computing*, 2010, pp. 71–78.
[14] NVIDIA, “Fermi Compute Architecture Whitepaper”, Tech. Rep., 2009.
[15] D. A. R. Polanco, “Collective Communication and Barrier Synchronization on NVIDIA CUDA GPUs”, Ph.D. dissertation, University of Kentucky, Sep. 2009.
[16] I. T. Foster, “Desinging and Building Parallel Program ”, New York, Addison-Wesley Publishing Company, 1994.
[17] OpenSparc, “OpenSparc.” [Online]. Available: http://www.opensparc.net
[18] Opencores, “OpenCores.” [Online]. Available: http://www.opencores.org

Meng Zhang received the BSc and MSc degrees from Northwestern Polytechnical University, Xi’an, China, in 2001 and 2004 respectively. He received the PhD degrees in computer science and engineering from Northwestern Polytechnical University in 2010. Since 2004, he has been on the faculty of the School of Computer Science and Engineering, Northwestern Polytechnical University. He also serves as an engineer in Engineering Research Center of Embedded System Integration, Chinese Ministry of Education. He worked on 16 short vector cores stream processor which was supported by Chinese National High-technology Research Development Project. His work and research interests are in parallel programming, multicore computer architecture.



Yuxuan Zhang received the B.S. degree in Computer Science from Northwestern Polytechnical University, Xian, China, in 2009. He is currently a master candidate of Computer Architecture in Northwestern University. His research interests are Computer Architecture, High-performance Computing, Digital Design, and EDA technology.

Wei Yang received the BSc degree in microelectronics from Northwestern Polytechnical University, Xi’an, China in 2010. He is a master candidate in School of Computer Science and Engineering, Northwestern Polytechnical University. He works on GPGPU parallel programming. His research research interests are in parallel programming, computer architecture.

Yaowen Kai received the B.S. degree in Computer Science from Northwestern Polytechnical University, Xian, China, in 2009. He is currently a master candidate of Computer Architecture in Northwestern University. His research interests are Computer Architecture, VLSI, EDA technology, Digital System Design and Verification.

Tingcun Wei received the PhD degree from Tohoku University, Japan, in 1999. He has been on the faculty of the School of Computer Science and Engineering, Northwestern Polytechnical University since 2003. He had been with TOPPAN PRINTING Co., Ltd., Japan for five years: he served as the senior engineer of microelectronic circuit. He is a reviewer of Chinese Journal of Semiconductors and Chinese Journal of Liquid Crystals and Display, a Special commentator of Chinese Physical Letter. His research interests include CMOS hybrid-voltage process, mixed-signal SoC/VLSI design methodology, Low voltage and low power analog VLSI design, Front-end readout and signal process IC for biomedical imaging system and Power management and digital power IC design.

Xiaoya Fan received the PhD degree from Northwestern Polytechnical University, Xi’an, China, in 1989. He has been on the faculty of the School of Computer Science and Engineering, Northwestern Polytechnical University since 1989, and heads its Computer Architecture Laboratory. He is a senior member of China Computer Federation, the member of Computer Architecture Technical Committee, China Computer Federation. He is a reviewer of Chinese Journal of Aeronautics and Journal of Computer-Aided Design & Computer Graphics. He is particularly interested in parallel and distributed architectures for information systems, including stream based high performance computing systems, many-cores architecture for Cloud Computing.

Calculation of First-, Second-Order and Multiparameter Symbolic Sensitivity of Active Circuits by Using Nullor Model and Modified Coates Flow Graph

Irina N. Asenova

***Abstract*—A new method of first-, second-order and multiparameter symbolic sensitivity determination based on the nullor model of active devices and modified Coates flow graph is presented. Rules for a symbolic reduction of nullor circuit complexity are described. An algorithm performs symbolic sensitivity analysis with respect to various circuit parameters appeared not only at one location in the modified Coates flow graph. Advantages of the method suggested are that, the matrix inversion is not required and the main drawback of some methods based on the adjoint graph, i.e. the necessity to analyze the corresponding graph twice, is avoided. Illustrative examples on symbolic sensitivity analysis are given.**

***Index Terms*—analogue circuits, flow graphs, nullor model, symbolic sensitivity analysis.**

I. INTRODUCTION

SENSITIVITY analysis plays an important role in determining the critical design variables in analog circuit analysis and synthesis [1], [2]. Sensitivity analysis is used in a wide range of areas such as prediction and evaluation of change in the characteristics of a network due to the change in the parameters, and optimization design of the network [3]. According to the classical formulae, the calculation of the first- and second-order transfer function sensitivities needs in the first place to find the corresponding derivatives. This is the main problem sensitivity analysis and its investigation is an object of some special methods, described in the literature [4], [5]. Coates flow graph (CFG) is useful and often used in the network theory and in the linear system theory [6]. On the other hand, nullor-based models have been generated taking into account the ideal behavior of the active devices [7]. However the input-output resistance and capacitance, gain, input offset voltage or current and the frequency response are all finite. This is the reason to include these effects in the nullor-based models [8]. In this manner, any analog network can be modeled with nullors and impedances, and the equivalence between them is introduced in [8]-[11]. In this paper, the equivalent nullor model of the active circuit is a starting point for the sensitivity analysis. On the base of nullor models using some network partial transfer functions, the CFG is used for the first-order sensitivity analysis of active networks [12]. This method was improved and simplified in [13], [14] using the modified Coates flow graph (MCFG). The symbolic equations generated by symbolic analysis help not only understand the first-order functional behavioral of an analog circuit, but also provide insight into second-order effects in the circuit. In some network-optimization schemes, it is desirable to know the dependence of first-order sensitivity on the elements of the network [4], [15]. In [16] the nullor model is combined with the MCFG aiming at the calculation of the multiparameter sensitivity (MS) in a symbolic form.

In this paper the process of obtaining first-, second-order and multiparameter symbolic sensitivity is automated and allows obtaining of all symbolic sensitivities simultaneously. The remaining work in this paper has been organized as follows. A detailed description of symbolic sensitivity analysis method, based on nullor model and modified Coates flow-graph, is presented in Section 2. In Section 3, the proposed method been applied to the nullor model of the STAR network for calculation of its first-, second-order and multiparameter symbolic sensitivities. Simulation results for the symbolic sensitivities of the voltage transfer function for the second-order high-pass filter are obtained. In Section 4, the conclusions are discussed.

II. NULLOR-MODIFIED COATES FLOW GRAPH SYMBOLIC SENSITIVITY ANALYSIS METHOD

A. Reduction of the Nullor Circuit Complexity

This section analyses a case when more than one parameter are likely to vary in a given circuit. Suppose that p parameters exist having very small fractional perturbations from their nominal values. According to [8]-[11] an equivalent nullor circuit N is composed by a designer. Let us assume that there are $m+n+R+1$ nodes, and R nullors in N. In accordance with [13], [14], the nodes, numbered from 1 to m represent network sources, nodes from $m+1$ to $m+n$ are inner nodes, that all or some of them can be considered as output nodes, and the node $m+n+1$ is the common node for the nullor circuit. The sequence of the nodes in the nullor circuit is determined as follows:

I.N. Asenova is with the Electrical Engineering Department, University of Transport, Sofia, Bulgaria (e-mail: irka_honey@yahoo.com).

Copyright © 2011 by Department of Microelectronics & Computer Science, Technical University of Lodz