Cache Simulator based on GPU Acceleration

Wan Han, Gao Xiaopeng, Wang Zhiqiang
School of Computer Science and Engineering
Beijing University of Aeronautics & Astronautics
+86-010-82338059
wanhan@cse.buaa.edu.cn, {gxp, wangzhiqiang}@buaa.edu.cn

ABSTRACT
Cache technology plays a fundamental role in modern computer systems as it serves the purpose of matching the speed gap between processor and memory. Trace-driven simulator has been widely adopted in the process of design and evaluation of cache architectures. However, as the cache design moves to more complicated architectures, size of the trace is becoming larger and larger. Traditional simulation methods, which can only execute simulation operations in sequence, are no longer practical due to their long simulation cycles. In this paper, we explore both set-parallelism and search-parallelism in cache simulation process, and map our parallel algorithm to GPU-CPU platform. And we propose a trace-driven cache simulator on GPU using Compute Unified Device Architecture (CUDA). Our experimental result shows that the new algorithm gains 2.5x performance improvement compared to traditional CPU-based serial algorithm.

Categories and Subject Descriptors
B.3.2 [**Memory Structure**]: Design Styles– *associative memories; cache memories;* B.3.3 [**Memory Structure**]: Performance Analysis and Design Aids– *simulation;* C.4 [**Computer Systems Organization**]: Performance of Systems–Modeling– *Techniques*

General Terms
Algorithms, Design, Measurement, Documentation, Performance, Experimentation,

Keywords
Multi-core, GPGPU, Trace-driven, Cache Simulator, CUDA

1. INTRODUCTION
Trace-driven Cache simulator [1] is a vital tool in the performance analysis and design space exploration in the area of computer architecture research. Compared to execution-driven cache simulator [2] and model analysis [3], trace-driven simulator has the advantage of yielding better accuracy and being of more flexibility. However, traditional trace-driven cache simulators only analyze trace in sequence, and lack the ability of utilizing multi-core processing power by exploiting inherent parallelism of modern platforms. As the cache architecture grows in complicity, exponentially increased trace size makes traditional simulation methods not applicable due to their extremely long simulation cycles. The need for more effective simulation method is therefore raised and brings much research attention. In this paper, we introduce a novel parallel method to accelerate the simulation of single-level cache, which utilizes the computation ability of GPU. Extension of our proposed method can be applied to GPU based multi-level and multi-core cache simulator.

For implementation, we map our parallel trace-driven simulation algorithm to the SIMD computation model in GPU. We developed trace-driven simulator for single-level cache on a Geforce 8800GTX with Compute Unified Device Architecture (CUDA). With different parallel granularities, we implemented five parallel algorithms for our experiment. The most efficient algorithm shows 2.5x speedup compared to traditional CPU-based serial algorithm.

Parallel trace-driven cache simulation has been a research hot spot, and there have been many excellent researches in this area. For example, one single pass simulation [4], trace reduction [5], time-parallel simulation [6] and SIMD massive parallel simulation [7]. One single pass simulation is able to compute statistics for different sizes of cache within a single pass. But it is confined to certain range of parameters and may create large overhead as flexibility increases. Trace reduction technique can greatly reduce trace length but cannot guarantee the accuracy of performance metrics. Compared to the time-parallel simulation, the method proposed in this paper exploits both set-parallelism and search-parallelism in the trace-driven cache simulation. Our method can simulate the behavior of the cache accurately without extra processing for simulation result correction. Furthermore, our algorithm is of more flexibility as it is not limited to LRU simulation or other acceleration condition.

The remainder of the paper is organized as follows: concepts of Graphics Processing Unit and parallel simulation algorithms are introduced in next section; Section 3 presents our CUDA based parallel algorithm as well as related techniques; implementation of simulation algorithms is explained in detail in section 4; and Section 5 elaborate the results of experiments. In the end, we summarize and conclude the paper.

SIMUTools 2009, March 2-6, 2009, Rome, Italy

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5562
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5562

2. BACKGROUND

2.1 Trace-driven Cache Simulator

Sequential simulation algorithm in traditional trace-driven cache simulator can be described as follows:

For each memory reference address, the cache simulator computes its set number and tag information according to cache parameters such as the block size and associativity. Then simulator checks corresponding set to find out whether there is a cache line has the same tag as current memory reference. And finally set status and metrics will be updated accordingly.

2.2 General-Purpose Computation on GPUs

In the last decade, GPU performance has been increasing so fast that even out paces the speed of integrated circuit predicted by Moore's Law. This rapid increase in GPU performance takes advantage of the highly parallel nature of visual computing. State of the art graphic architectures provide tremendous memory bandwidth and computational power. Besides performance improvement of the hardware, the programmability also has been significantly increased. These improvements make GPU a compelling platform for general-purpose computing as well as visual computing.

Advanced GPU architecture offers significant level of parallelism with relatively low cost. The operations executed in GPU are similar to the well known vector processing model, which is also known from Flynn's taxonomy as Sing Instruction Multiple Data or SIMD. Therefore it can be predicted that many applications use to be hosted on verctor superconputers in the past can be deployed on GPU platform. With the ever increasing programmability, specific powerful programming tools (e.g. CUDA [8] and CTM [9]) can be used for implementation of algorithms. For example, GPU has been utilized as a math co-processor in special games and physics simulations in [10]; [11] introduces a GPU based implementation of Reyes-style adaptive surface subdivision; [12] presents fast algorithms for scan and segmented scan on GPUs; [13] develops a programming framework on the graphics architectures and applies it to a variety of problems (e.g. matrix multiplication); [14] introduces a framework for the implementation of linear algebra operators on GPUs; And in [15], Fast Fourier Transform is implemented on NVIDIA graphics architecture.

Among the various applications of GPU programming, the major challenge is how to map the algorithm to units of graphics architecture. As shown in the GPGPU technique, applications need be partitioned into independent parallel sections. And each section needs to be implemented as a kernel executes on a processing unit. While input and output of a kernel are stored in the memory of GPU.

2.3 Parallel Simulation Algorithm

It has been observed that the simulation process of each memory reference shows a weakly partial order. Whether current reference is a cache hit is dependent on cache status, which is modified by the memory references that have been simulated. This observation implies that memory references belonging to the same cache set should be simulated jointly while simulation of different sets need to be carried separately.

During cache simulation, the following operations are performed on an address: (1) fetch address from the trace; (2) break address into tag, block number, block offset; (3) calculate set number; (4) search blocks in corresponding set; (5) update the set status and metrics. Among all five steps, step 4 and step 5 are the two most time-consuming steps, which can be performed independently on different sets. This observation leads to exploit of set-parallelism (i.e. trace-driven cache simulation can be performed in parallel on a set base). Parallel simulation algorithm first classifies trace by set numbers, and then implements simulation kernel. In addition to set-parallelism, searching in the step 4 can also work in parallel.

In a coarse granularity, multi-configuration can be parallelized using the computational resource on GPU. Once trace file is read into memory, the simulator can generate metrics for cache with different parameters within a single pass. Together with set-parallelism and search-parallelism in the process of cache simulation are explored, the cache simulator acceleration utilizing GPU is feasible.

3. PARALLEL ALGORITHM BASED ON CUDA AND KEY TECHNIQUES

3.1 CUDA

CUDA issues and manages computations on GPU as a data-parallel computing device without the need of mapping computation to a graphics API. When programming CUDA, programmers take GPU as a processing device capable of executing a large number of threads in parallel. In this case, GPU behaves as a coprocessor to the main CPU (host), and hosts parallel portions of applications' workload. These portions are independent when they process different data set, and can be isolated and partitioned into several functions (kernels). Well established functions can be built with CUDA instruction set and downloaded to the device for execution.

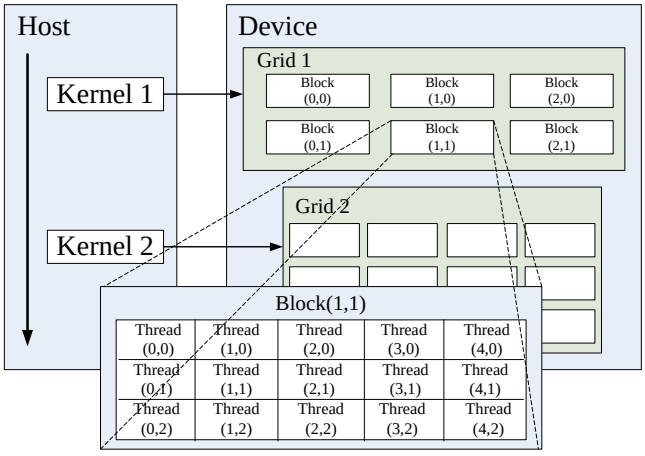

Figure 1 CUDA Programming Model [8]

As shown in figure 1, host can request services of the device via specific programming interfaces. Each kernel is executed as a batch of threads which are organized as a grid of thread blocks. A thread block is a cluster of threads that communicate with each other efficiently via fast shared memory and synchronize their

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5562
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5562

execution for memory access competition. Synchronization points are specified in the kernel. Once such a point exists, threads in a block will be suspended until they all reach the synchronization point. There is a limitation on the maximum number of threads that a block can host. However, blocks of same dimensionality and size that execute the same kernel can be clustered into a grid of blocks. Therefore the total number of threads that can be launched in a single kernel invocation is much larger than single block limitation. But threads in different thread blocks from the same grid cannot communicate and synchronize with each other.

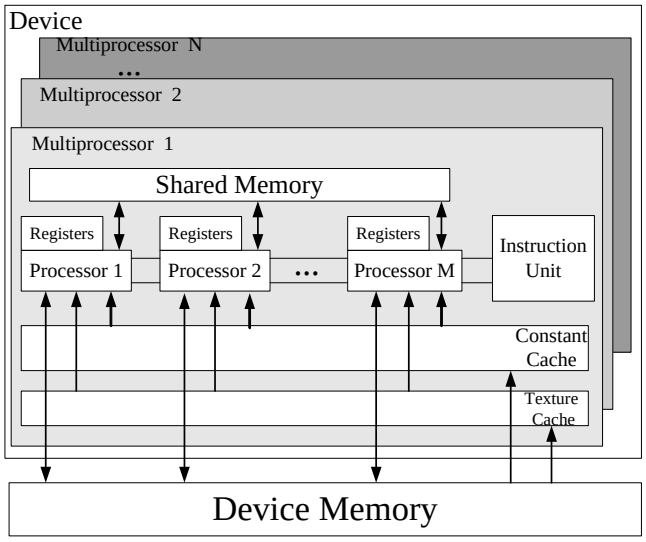

Figure 2 Hardware Model in CUDA [8]

The device is implemented as a set of multiprocessors is shown in figure 2. Each processor has one set of 32-bit local registers. Shared memory is shared among processing cores of a multiprocessor. A read-only constant/texture cache is shared by all the processors and serves the purpose of speeding up reads from the constant/texture memory. The local and global memory spaces are implemented as read-write regions of device memory and are not cached.

3.2 Parallel Simulation Algorism based on CUDA

As analyzed in section 2.3, we can utilize the GPU parallel architecture to optimize the following computations:

1. The simulation processes on different cache sets are independent. And the computations on different thread blocks are also independent. Therefore the parallel simulation on different cache set can be implemented by distributing the simulation process of each cache set to a separate thread block in GPU.

2. Searching process is time-consuming when large associativity presents. As this process is inherently parallel, given that threads in one block can communicate with each other in the same block, the search-parallelism can be exploited by distributing the search operations to several threads.

3.3 Key Techniques

3.3.1 Bucket Sort

The sort process classifies the trace data according to the cache set, and holds the original sequence of the same cache set. It also sets up several buckets which numbers are in accordance to the set numbers. And ultimately, the sort process matches memory references to buckets according to the sequence in cache sets.

3.3.2 Ping-pong Buffer

In general, as cache lines are stored in static arrays in memory, they need to be reordered after searching process is over. Ping-pong buffer is therefore adopted to parallelize the reordering process.

1. The information about one cache set is stored in two buffers: Buf0 and Buf1. Assume initial input is Buf0;

2. After each searching process, all information stored in Buf0 is copied to Buf1 according to the replacement policy and updating policy, then set the input as Buf1 and output as Buf0;

3. After each memory reference is simulated, exchange the position of input and output buffer until the whole trace file is simulated. And the input buffer at the time is the final status.

3.3.3 Memory Model on the GPU

Different memories on GPU varies greatly in terms of bandwidth, which has a significant impact on the performance of the simulator. The memory model adopted here is:

1. Since the trace data is large, it can be only stored in the global memory;

2. The information about the cache set, such as the cache lines, tag, status and metrics, which needs high memory access speed, is stored in the shared memory.

3.3.4 Stream Management

In order to improve performance, we use two streams to parallelize cache simulation process and bucket sort process. As trace data can be divided into multiple segments, one stream prepares the bucket sort for next phase, while the other stream executes the simulation process on the sorted stream. Parallelizing the two streams, we can achieve approximate 2x speed-up compared to one stream implementation.

There are specific limitations about the feature of supporting overlapped memory copy concurrently with kernel execution using new stream management interface. Overlapping is only allowed on 1.1 architectures (g84/g86/g92), and it will revert to serial operation on 1.0 architectures (g80).

4. ALGORITHM IMPLEMENTATION

When programming CUDA, users need to allocate device memory first, then transfer data from host to device, and finally get results back from device when computation is finished. The five parallel algorithms we implemented are decribed as follows:

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5562
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5562

Parallel simulation algorithm 1: trace data is not sorted, but simply stored in an array. The simulation is parallelized at both set-parallelism level and search-parallelism level.

For each memory reference {

If (the memory reference is mapped to this cache set)

Handle it;

Else

Discard it; }

In this algorithm, one thread block is dedicated to the simulation of one cache set, and each thread in the block is in charge of searching in one cache line. For every memory reference, each thread block needs to compute whether this data belongs to the cache set it is simulating, and simulates the cache function in case the memory reference does. Since there are multiple threads executing the search process and they work independently, synchronization is needed to find out if the memory reference is hit, and then to update status and metrics.

Parallel simulation algorithm 2: perform bucket sort on trace data. One thread block is dedicated to the simulation of one cache set. And each thread in the block is in charge of searching in one cache line.

Parallel simulation algorithm 3: perform bucket sort on trace data. One thread block is dedicated to the simulation of one cache set. Only one thread in each thread block is in charge of searching.

The difference between algorithm 2 and 3 is the searching process. While algorithm 2 does searching in parallel, algorithm 3 does it in sequencial. This improvement to algorithm 3 is based on the consideration that the thread synchronization comsumes lots of computation cycles.

Parallel simulation algorithm 4: similar to algorithm 2, but using the ping-pong buffer to accelerate the update process.

Parallel simulation algorithm 5: the simulation of multi-configuration is implemented in a single pass. Since the bucket sort process takes lots of cycles, sorted trace data is used to simulate caches with different degrees of associativity. In this algorithm, several thread blocks are dedicated to the simulation of one cache configuration, and each thread block only has one thread to execute the simulation on one cache set.

Another consideration is to use a thread block to simulate one configuration, and all threads in it to implement the simulation on one cache set. The problem is that the shared memory for one thread block is only 16KB, but information such as cache line and tag for each configuration is larger than 16KB. Therefore it is necessary to store some information in the uncached local memory. As the simulation for each memory reference needs to access local memory in this case, the experimental results show that the small bandwidth of local memory has a negative impact on parallel computation performance.

5. PERFORMANCE

We use DineroIV sequential uniprocessor cache simulator for our evaluation. The experiments were conducted on a testbed equipped with an Intel core 2 E6550 and GeForce 8800 GTX graphics card.

Our experiments consist of the simulation on various input traces which are obtained from the NMSU Trace Base facility. The traces are collections of memory references from programs in the SPEC 92 benchmark suite.

5.1 Measurement of Time

The CPU simulation time does not include the time that read trace file from disk to memory. While the GPU simulation time includes the bucket sort time, the time of data transfer from CPU to GPU, GPU simulation time, and the time of results feed back from GPU to CPU.

5.2 Single Configuration Simulation

We simulate one trace file of 10MBytes to get the GPU simulation time distribution as shown in table 1. The cache is configured with: 64 sets, cache block size is 16 and associativity is 4, using LRU and write back policy.

Table 1 GPU Simulation Time Distributions

Process	Average Time (ms)	Percentage
Bucket sort	40.09	36.609%
Data download	7.05	6.438%
Kernel executing	62.35	56.935%
Data upload	0.02	0.018%

It can be observed from the results that: bucket sort process occupies a great amount of time in simulation; Data download time is bounded to the memory bandwidth between CPU and GPU; And data upload is very fast and not significant compared to the others.

As we discussed in section 3.3.4, bucket sort process and kernel simulation process are executed in sequential due to the limitation of the GeForce 8800 GTX hardware. Although bucket sort process has an large impact on the overall performance, single pass multi-configuration can reduce the impact of bucket sort process by increasing computation task on GPU.

The rest experiments and analysis adopt alvinn.din as an example.

5.2.1 Increasing the Associativity

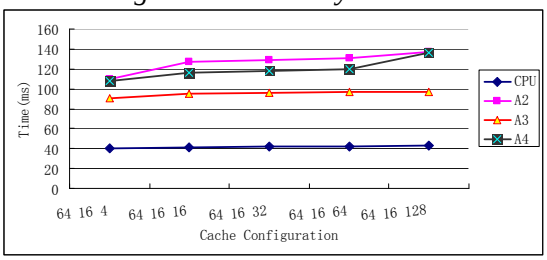

Figure 3 Simulation Time Curve with Increasing Associativity

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5562
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5562

As shown in figure 3, A2-A4 represent parallel algorithm 2-4 respectively. And we choose LRU as replacement policy, write-back as updating policy in this experiment. X-axis represents cache configuration in following format: number of sets (64), block size (16) and associativity (4\16\64\128).

We found that the time using by A1 is several dozen times longer than others so we do not show it in this figure. The reason behind this is that the unsorted trace data is large and stored in the global memory. Thus A1 experiences long memory access latency.

As the degree of associativity increase:

1. Serial simulation time increased slowly. It benefits from the principle of locality. Hit rate is high since the memory reference is located centralized in an address range. When associativity is increased, hit rate would not increase dramatically, and thus the simulation time is not improved significantly.
2. A2 and A4 simulation time increased. Since the thread number is increased when associativity is increased in A1, this results in much more time spending on synchronization.
3. A1's simulation time increased evidently. Since each thread block needs to access all memory references, synchronization time is increased.
4. A3 simulation time increased slowly. As each thread block has only one thread, the sequential searching time is increased as the associativity increase. When hit rate increased, searching time increment is negligible.

5.2.2 Increasing Set Number

Figure 4 shows simulation time curves with increasing set number when the cache block size and associativity are fixed.

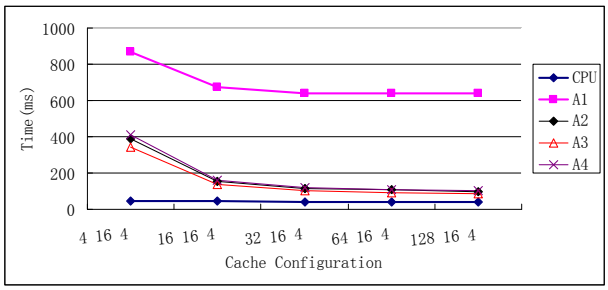

Figure 4 Simulation Time Curve with Increasing Set Number

As set number increase:

1. Sequencial simulation time reduces slowly. Hit rate increases as the set number increased. Since the degree of associativity is 4, the replacement and update execution on CPU is fast.
2. A1, A2, A3 and A4 simulation time first reduces fast and then slows down. Hit rates are low when there are few sets. And hit rate increases as set number increases.

5.2.3 Single Configuration Simulation Summary

Results have shown that A1-A4 algorithms cannot speed up the simulation of single cache configuration:

1. Too much time is spent on bucket sort process;
2. Fetching data from the global memory of GPU suffers from long latency and the computation density on the GPU is small.
3. The computation capability of CPU is much larger than that of the single processor in the GPU.

5.3 Multi-configuration Simulation in Single Pass

Figure 5 shows time curves of multi-configuration simulation in single pass with increasing set number. Cache block size is set to 16KB and the number of simulated configurations is 128. Hit rate improves when the set number increases, which result in less replacement, thus the CPU sequential simulation time is reduced. However, both the number of the thread blocks and the number of threads in each thread block affect simulation time on GPU. When the number of set increases to a certain number, the thread synchronizations become the major portion of simulation time.

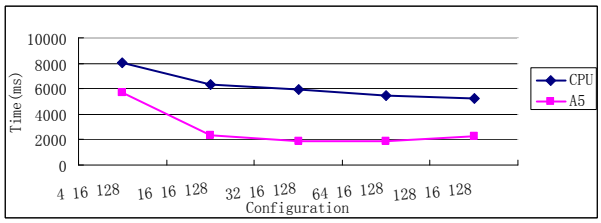

Figure 5 Simulation Time of Multi-configuration in Single Pass Curve with Increasing Set Number

Figure 6 shows the speedup of A5 algorithm with increasing number of set. And it can be observed that the average speedup comes to 2.5. X-axis represents the configurations in the following format: number of sets, block size and number of the simulated configurations.

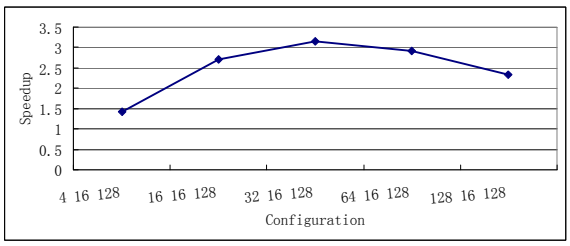

Figure 6 Speedup of A5 Algorithm Curve with Increasing Set Number

6. CONCLUSIONS and FUTURE WORK

In this paper, we suggest that the trace-driven cache simulator can be accelerated on the GPU. Our experiments show that this method is fast, low-cost and easy to use. Performance is analyzed according to the GPU architecture and CUDA programming model.

The functionality and performance of our GPU based simulator could be easily improved by incorporating the following changes:

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5562
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5562

1. When simulate multi-level cache, it needs to dispatch more thread block to simulate the extra cache levels, and to parallelize the simulation of different levels of cache;
2. When simulate cache coherency in multi-core system, it needs to execute cache simulation of different cores on different threads within one thread block and use shared memory to simulate the coherency;
3. Use two streams to parallelize cache simulation process and bucket sort process on 1.1 graphics hardware;
4. Use Pin to generate the trace, pipelined the trace generation process and simulation process.

7. ACKNOWLEDGMENTS

Our thanks to the support provided by the National High Technology Research and Development Program (2007AA01Z183).

8. REFERENCES

[1] R. A. Uhlig. and T.N.Mudge. “Trace-driven Memory Simulation: A survey” [J]. ACM Computing surveys, Vol.29, 1997.

[2] Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L. Evaluation techniques for storage hierarchies.

[3] Puzak, T. Analysis of cache replacement algorithms. Ph.D. dissertation, University of Massachusetts, 1985.

[4] Yuguang Wu and Richard Muntz, Stack evaluation of arbitrary set-associative multiprocessor caches, IEEE Tram on Parallel and Distribured Systems, 6(9), pp. 930-942, Sep.1995.

[5] Milenkovi´c A. and Milenkovi´c, M. An efficient single-pass trace compression technique utilizing instruction streams. ACM Transactions on Modeling and Computer Simulation, Vol. 17, No. 1, Article 2, Publication date: January 2007.

[6] R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.Approximate Time-parallel Cache simulation. Proceedings of the 2004 Winter Simulation Conference. 2002.

[7] T. Kiesling and S. Pohl. Time-Parallel Simulation with Approximate State Matching. In Proceedings of the 18th Workshop on Parallel and Distributed Simulation, 2004.

[8] NVIDIA CUDA Programming Guide, http://developer.nvidia.com/cuda

[9] ATI CTM Guide， http://ati.de/companyinfo/researcher/documents.html

[10] Marcelo P. M. Zamith, Esteban W. G. Clua, Aura Conci, Anselmo Montenegro, Regina C. P. Leal-Toledo, Paulo A. Pagliosa, Luis Valente, Bruno Feijo. A game loop architecture for the GPU used as a math coprocessor in real-time applications. Computers in Entertainment (CIE), 2008, 1-19.

[11] Anjul Patney, John D. Owens. Real-time Reyes-style adaptive surface subdivision. ACM SIGGRAPH Asia 2008 papers, 1-8.

[12] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, John Manferdelli. Fast scan algorithms on graphics processors. Proceedings of the 22nd annual international conference on Supercomputing, 2008, 205-213.

[13] Chris J. Thompson, Sahngyun Hahn and Mark Oskin. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis[A]. In Proceedings of International Symposium on Microarchitecture, Istanbul, 2002. 306-317.

[14] PJens Kruger, PRudiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. ACM SIGGRAPH 2005 Courses. Jul. 2005. 234-es.

[15] Sergio Romero, Maria A. Trenas, Eladio Gutierrez, Emilio L. Zapata. Locality-improved FFT implementation on a graphics processor. Proceedings of the 7th WSEAS International Conference on Signal Processing, Computational Geometry & Artificial Vision. Aug. 2007. 58-63.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5562
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5562