SCIENCE CHINA

February 2010 Vol. 53 No. 1: 1–18
doi:

Efficient Monte Carlo-based Options Pricing on Graphics Processors and Its Optimizations

Liu Li[1]*, Zhang YouHui[1], Liu Li[1], Yang Guangwen[1] & Zheng WeiMin[1]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084*, China,*

Received August 22, 2009; accepted February 10, 2010

Abstract Options pricing is a critical problem and one of the fundamental building blocks in mathematical finance. Monte Carlo (MC) simulation is the most widely used solution for options pricing. In most cases options pricing must be performed in real time. Although today's multi-core CPUs can provide a high computing power, the options pricing on today's multi-core CPUs is far from responding in real time. In this paper, we use modern graphics processors (GPUs), which provide a much higher computing power than multi-core CPUs, to perform MC-based options pricing. The challenge is that GPUs only provide a small on-chip scratchpad while MC-based options pricing requires a large memory space and accesses the memory space irregularly. To make MC-based options pricing efficient on GPUs, we propose a recycling approach which compacts the data to shrink the memory space, and a crossing-path layout which reorganizes the data to make memory accesses GPU-friendly. We use real market data and benchmarks to evaluate our optimization approaches. The experimental result shows that our MC-based options pricing on the latest GPU is 43-fold faster than the latest multi-core CPU for single precision computation (with Intel C++ Compiler 11.1); compared with GCC 4.2.4, the speedup is as high as 145.

Keywords GPU, stream architecture, Options Pricing, Monte Carlo Simulation

1 Introduction

In mathematical finance, options pricing is one of the fundamental problems and receives more and more attentions these years [1][2]. According to the exercising dates and conditions, people propose a variety of quantitative pricing models [2][3], and use complicated numerical methods to price these models, which usually takes hours or even days to get a convergent solution. Monte Carlo simulation is widely accepted due to its broad applicability to those pricing models with complex boundary conditions [4][5].

In this paper we choose MC to solve the Black Scholes model considering its natural massive parallelism. Although we use BS as the pricing model, the methods proposed can be applied to all other mathematical pricing models since they are MC-oriented but not pricing model oriented.

"A Monte Carlo model generates random price paths of the underlying asset, each of which results in a payoff for the option. The average of these payoffs can be discounted to yield an expectation value for the option." [6] According to the law of large numbers, the more price paths are sampled, the more accurate the expectation value will be.

*Corresponding author (email: irving02@gmail.com)

However, the amount of price paths that can be simulated to quickly respond to the dynamic market in real-time heavily relies on the data processing capability of the processor. For these computation intensive applications, a stream processor has an absolute advantage over the traditional CPU [7]. Many previous works[7][8] compared the peak performance growth of Intel and NVIDIA classical processors from Jun 2003 to Jun 2008: compared with the annual increasing rate of 1.4x for CPU performance, known as Moore's Law, GPUs computational capability has been compounding at an annual rate of 1.7x (pixel/second) to 2.3x (vertices/second).

But the large memory requirement and irregular memory access pattern of MC simulation [9] usually lead to quite poor performance, especially for the modern stream processor which has a very small on-chip scratchpad. Such an example is NVIDIA Tesla C1060, which has merely 16KB on-chip shared memory for each multiprocessor [7].

In this paper, we present a suit of architecture conscious optimization methods to compress the working set, reorganize the data layout, and eliminate the above mentioned performance penalties. Compared to implementations with OPENMP on Intel 8-core Xeon processor @ 2.0GHz (with Intel C++ Compiler 11.1), our method achieves a 43-fold speedup with CUDA on Tesla C1060 for single precision computation.

2 Related work

Before the prevailing of the modern GPU architecture, people began use GPU to accelerate options pricing. In ref.[10], Craig et al. gave a simple implementation for Black-Scholes Model and Binomial Model with Cg, and got a 10x speedup compared with CPU implementation. Stanimire[11] implemented two probability-based simulation models, Ising model and Percolation model, and received a 3x speedup compared to CPU.

To make GPU programming more user-friendly, people devoted a lot of efforts on the development of new programming frameworks, among which RapidMind is a well-known one. In ref.[12], McCool et al. published their experimental results on the implementation of European options pricing via quasi-Monte Carlo sampling and reported a 30x speedup over the CPU implementation, which proves the high efficiency of RapidMind.

With the rapid evolution of computer architecture, NVIDIA released their new product with the Computing Unified Device Architecture (CUDA) in 2007, which provides much better support for general purpose computation and greatly relieves the pain of traditional graphics programming. Among those works based on CUDA, Jauvion et al. [13] implemented a Trinomial Option Pricing Model which is 31.9 times faster than the CPU implementation.

Also, NVIDIA released their own work [16] with CUDA SDK, an MC-based solution for European options pricing, which achieved a peak performance of 10 billion path samples per second. However, it oversimplified the pricing model, and deemed the whole maturity period of the option as one single time unit, regardless of however long it is. For an option pricing model of real usage, we should divide the duration into time slices, on each of which we let the underlying asset price randomly walk in Brownian motion style.

Besides, none of the above works mentioned any detailed optimization techniques close to the GPU architecture. In this paper we present our optimization work and development experiences on GPU, and most of the techniques can be generally applied to all GPU applications.

3 Option Pricing Algorithm and Challenges on GPU

3.1 MC-based Black Scholes options pricing model

The Black Scholes model [3] is a major breakthrough in mathematical finance in the 1970s, which explicitly describes the relation between the price of an option and its underlying stock price by a stochastic differential equation. MC-based methods do not directly solve the equation, instead they generate different price paths of the underlying stock, each of which starts from price S, follows a random walk of

Brownian distribution, and finally reaches a normally distributed price S_t. By averaging and discounting all the payoffs calculated from S_t of different paths, we can get an expectation value for the option. The algorithm is illustrated in Figure 1(a).

3.1.1 Random number generation

The random number generator we used to produce log-normally distributed price changes is a parallel version of L'Ecuyer's Combined Recursive Generator with a 32-element Bayes-Durham shuffle-box [14], which has good statistical properties and generates independent Mersenne Twisters[15][16]. Ahead of the simulation, the initial parameters are computed off-line.

3.1.2 Parallelization

We implement pricing algorithms for Exotic options and Vanilla options separately. A Vanilla option is just a straight call or put option whose pricing is relatively easy and takes quite short time. However, Vanilla options are often traded at a large amount, and banks will price thousands or even millions Vanilla options with different maturity periods in the same time, which usually takes hours.

An Exotic option is much more complex: the payoff at maturity does not depend just on the value of the underlying index at maturity, but on its value at several times during the contract's life. Therefore, the pricing for an Exotic option is more complicated and it needs to sample millions of price paths to reach convergence, which also requires quite a long time.

The parallelization is straightforward: we simply assigns independent price paths (for Exotics) or different options (for Vanilla) to different threads. According to how each thread completes its price paths, MC method can be categorized into two execution modes. If a thread simulates the price paths one after another, we call it the path mode execution; alternatively, if a thread completes the current step simulation for all price paths before continuation with the next step, we call it the slice mode execution.

So in this paper we use four kinds of benchmarks to evaluate the performance, which are path mode for Exotic option (EP), slice mode for Exotic option (ES), path mode for Vanilla option (VP) and slice mode for Vanilla option (VS).

3.2 Challenges for GPU implementation

In this section we briefly introduce the main challenges to efficiently implement MC on GPU.

3.2.1 Large working set

As discussed in subsection 3.1, the more price paths are sampled in MC-based solution, the more accurate the result will be, but the more memory space will be needed. When the memory requirement exceeds the shared memory size, we have to access global memory for each single step, which leads to long memory-access latency. Therefore, to minimize global memory accesses, we should compress the working set of the algorithm as much as possible.

3.2.2 Numerous bank conflicts

To achieve a higher memory bandwidth, shared memory is divided into equally-sized memory modules, called banks. If two memory accesses fall in the same bank, there is a bank conflict and the accesses have to be serialized [7], so in GPU programming we should reorganize the data layout in such way that memory requests from the 16 threads in the same half warp should access different banks.

But for our MC-based solution, each thread requires *numSteps* consecutive words for price paths simulation. In the worst case, if *numStep* is a multiple of 16. Then all memory accesses within the same half warp will lead to bank conflicts, and available bandwidth is only 1/16 of the peak value. For general cases, suppose R is the remainder of *numSteps* divided by 16, and LCM is the least common multiple of R and 16, then maximum simultaneous non-conflict accesses will be LCM/R, and available bandwidth will be decided by :

$$available\_bandwidth = peak\_bandwidth * (LCM/R)/16 \quad (1)$$

```
for(i=0; i<numPaths; i++)
{
    for(j=0; j<numSteps; j++)
    {
        Simulate sample path i for underlying
        stock price, from step j to step j+1;
    }
    Check stock price in each step against
    strike price;
    Compute corresponding option payoff
    for path i;
}
Average all payoffs and discount the average
value to yield the option price
```

```
for(i=0; i<numPaths; i++)
{
    for(j=0; j<numSteps; j++)
    {
        Simulate sample path i for underlying
        stock price, from step j to step j+1;
        Check stock price of current step
        against strike price and set failure flag;
    }
    Compute corresponding option payoff for path i;
}
Average all payoffs and discount the average value
to yield the option price
```

Figure 1: Program structure before and after working set compression

Therefore, to make full use of the on-chip memory bandwidth, we should reorganize the data layout so that threads within the same half warp will access shared memory in prime-interval way, i.e., R will not be a factor of 16.

3.2.3 Uncoalesced global memory accesses

In CUDA, there are 32-bit, 64-bit and 128-bit global memory access instructions, and the runtime will try to combine simultaneous accesses to neighboring addresses into one instruction, i.e., global memory coalescence.

However, in MC-based solution, different threads will access their respective paths iteration space and we cannot use global memory coalescence. Therefore, again we should reorganize the data layout and make threads in one half warp access neighboring addresses as much as possible.

3.2.4 Low multiprocessor occupancy

The multiprocessor occupancy is defined as the ratio of active warps to the maximum number of warps supported by a multiprocessor. On GPU, the more active threads there are on GPU, the better computation and memory access are overlapped. But because of the limited hardware resources, there are two constraints deciding how many active threads GPU can support simultaneously: register file and shared memory.

In our MC-based solution, active threads number is severely limited by its huge requirement of registers, and the occupancy is only 0.5.

Besides, there are many variables spilled into local memories in each thread. To minimize the negative effect of local memory accesses, and maximize the simultaneous active threads on each multiprocessor, we should cut the register requirements of each thread to 20%∼30% of the original implementation.

4 Our Optimization methods

4.1 Working set compression

To compress the working set, we propose to make necessary modifications to the original algorithm presented in Figure 1(a).

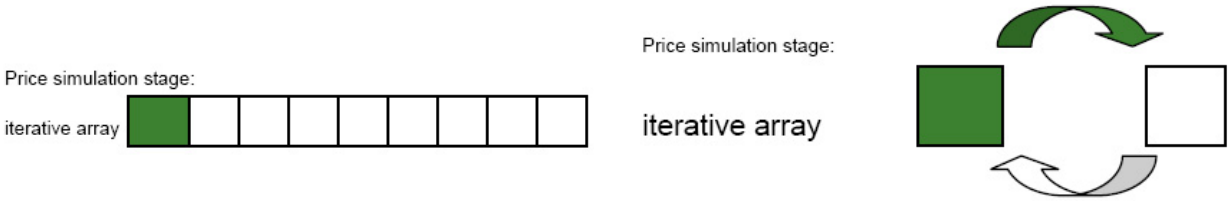

Figure 2: Working set of stock price simulation stage before and after compression

Figure 3: Memory layout before and after bank conflict elimination

In the options payoff calculation stage, stock prices of different time steps in the current path will be checked against the strike price, and if any one of them is smaller than the strike value, the current path is invalid, and payoff value from this path will not be cumulated into the final option price. Therefore, in this algorithm, prices of all steps along the current path have to be kept in memory for this further check, which consumes large memory space.

We move the stock price checking phase from the payoff calculation stage to the stock price simulation stage (Figure 1(b)). In this way, each simulation path only needs two elements to complete the whole price random walk: in step j, we use the price value stored in (j%2) to calculate price of step j+1, and store it in (j+1)%2, and check whether it is greater than the strike price, if not, we mark this path as failing, and its ending period price will not be counted for the final payoff calculation. In step j+1, we use price value stored in (j+1)%2 to calculate price of step j+2, and store it back to (j+2)%2 (reuse the memory space of step j%2) (Figure 2).

4.2 Bank Conflicts Elimination

As discussed in subsection 3.2.2, it is necessary to make 16 threads within a half warp access shared memory in a prime-interval pattern. We propose the crossing path memory layout to remove bank conflicts: we do not allocate $numOfSteps$ consecutive elements for each price path, instead we allocate $numOfPaths$ consecutive elements for all the first elements of each path, and immediately following this memory slice we allocate another $numOfPaths$ consecutive elements for all the second elements of each path, and so on, until we reach the $numOfSteps$ element. (Figure 3)

4.3 Global memory coalescence

As discussed in subsection 3.1.2, different from the path mode, in the slice mode each thread completes the current step for all the paths before continuation with the next step, so we cannot reuse the compressed two elements for all the paths assigned to one thread, and we need to keep two elements for each path, so the slice mode requires much larger memory space than the path mode. When this exceeds the shared memory size, we have to use global memory.

Therefore we need to maximize global memory coalescence for slice mode. Here we can take the same memory layout as the price path array in shared memory (Figure 3). In other words, the proposed crossing path memory layout not only helps eliminate all the bank conflicts for shared memory accesses, but also helps maximize global memory coalescence.

4.4 Multiprocessor occupancy maximization

As mentioned in subsection 3.2.2.4, due to the large register file consumed by each thread, we did not achieve the maximum active thread number on each multiprocessor; moreover, there were also lots of variables spilled into local memory.

For this problem, we decided to carefully divide the kernel into multiple smaller sub-kernels, each consuming less registers, so that we can get a balance between the usage of shared memory and registers. Also since each sub-kernel consumes less registers, the variables spilled into local memory can be stored in on-chip registers.

Certainly this optimization comes at an expense of more global memory accesses, since we need to write back many intermediate results from one sub-kernel into global memory, for the future use by another one. So this is a trade-off between processor occupancy plus less local memory accesses and more global memory accesses, and the principle is to find the point where we split the kernel into halves (usually after a reduction operation), so that on one hand, less registers are consumed by each sub-kernel, and there are not much data passing between sub-kernels on the other.

To reduce the register usage and thus improve the occupancy of a CUDA kernel, a generally taken method is to work on the temporal variables in the program: try to declare variables in the function scope and reuse them in any sub scopes, instead of redeclaring variables in each sub scope. Besides, it is also feasible to adjust the order of arithmetic operations to combine operators of identical precedence to the end. However, for our problem, these methods have little effect, because merely in the random number generator, we need an array of 32 integers for each thread to keep intermediate results, which is spilled into local memory in the original implementation. Our method effectively solves this problem at a cost of some extra global memory accesses, on which we can do many optimizations.

5 Performance Evaluation

In this chapter, we compare our implementation on GPU with the optimized implementation on CPU, for single and double precision separately, and for each of them we evaluate the performance of Exotic option pricing and Vanilla options pricing in path mode and slice mode. For CUDA implementation we test the performance on NVIDIA Tesla C1060; for OPENMP, we test the performance on three platforms: Intel Pentium(R) 4 @ 2.4 GHz, AMD $Opteron^{TM}$ Processor 2212 (dual core) @ 2.00GHz, and Intel(R) Xeon(R) E5335 (4 dual cores) @ 2.00GHz. For all the test cases we are using real market data from some international financial organization.

For the implementation on CPU we use OPENMP for parallel programming, and all the price paths (Exotic option pricing) or options (Vanilla option pricing) are statically assigned to different threads. We unroll the loop over different time slices (inner loop in Figure 1) by a factor of 2, and we use SSE cache prefetch instructions to improve the cache performance, we test the performance with different prefetch distances, and the best one is reported here. Due to the numerous branch instructions in the random generator, we cannot parallelize different pricing paths at SIMD level with SSE . For compilation, we use both Intel C++ Compiler 11.1 and GCC 4.2.4 with "-O3 -fopenmp" flags and report the better performance for all the figures in this chapter. We have also tested the CPU performance with different thread number settings and give the best results here.

5.1 Single Precision

In this subsection we evaluate the single precision performance of GPU. Figure 4 presents the running times on the GPU, the dual-core CPU and the 8-core CPU, and all results are normalized to the speedup relative to the 1-core CPU.

For EP (Figure 4(a)), GPU achieves a speedup of 43 in comparison to the 8-core CPU using Intel C++ compiler (for GCC, this speedup is 145). Because of the numerous branch instructions in the BS algorithm and its random number generation algorithm, we cannot fully explore the CPU's 4-width vector

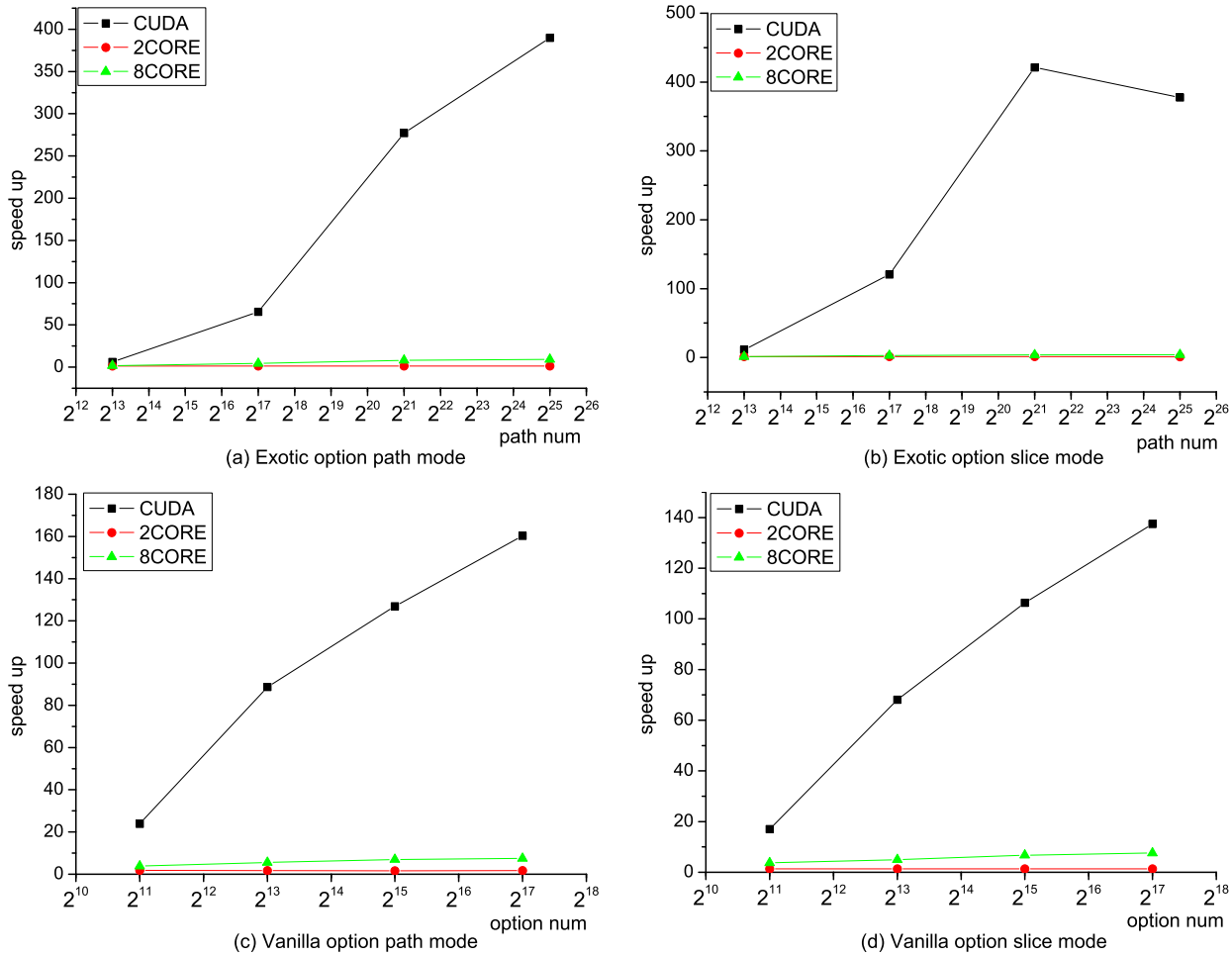

Figure 4: GPU speedup relative to CPU for single precision

processing capability, while for GPU, due to its conditional memory access, we can make full use of its 8 processors in each MP and completely explore its vector processing capability.

For ES(Figure 4(b)), when the path number is greater than 2^{20}, shared memory will not be enough to hold all computations, and we need to access global memory, which leads to a performance penalty. For pricing without global memory accesses, GPU gets a speedup factor of 111.8 compared to the 8-core CPU using GCC for compilation (for ICC, it is 120.7); while for pricing with global memory accesses, GPU gets a factor of 89.9 (for ICC, it is 96.7).

For the Vanilla option pricing (Figure 4(c) and 4(d)), due to the large memory requirement, we need to access global memory for both VP and VS. Besides, the maturity period for each option is different, which leads to lots of divergent branches, so the speedup from GPU is not as good as the Exotic option pricing. For VP, GPU is 21.3 times faster than the 8-core CPU using ICC (for GCC it is 26.5); while for VS, the speedup is 18 (for GCC it is 23.4).

5.2 Double precision

Results on double precision are quite similar to those of the single precision, but GPU's peak performance (measured in FLOPS) for double is an order of magnitude worse than that of single precision, so the speedup of GPU is smaller.

For both EP and ES (Figure 5(a) and 5(b)), GCC gives slightly better performance than Intel C++ compiler. For EP, GPU achieves 6.9 fold speedup compared to the 8-core CPU. For ES, when shared memory is enough to hold all computations, GPU is 7.7 times faster than the 8-core CPU, but when

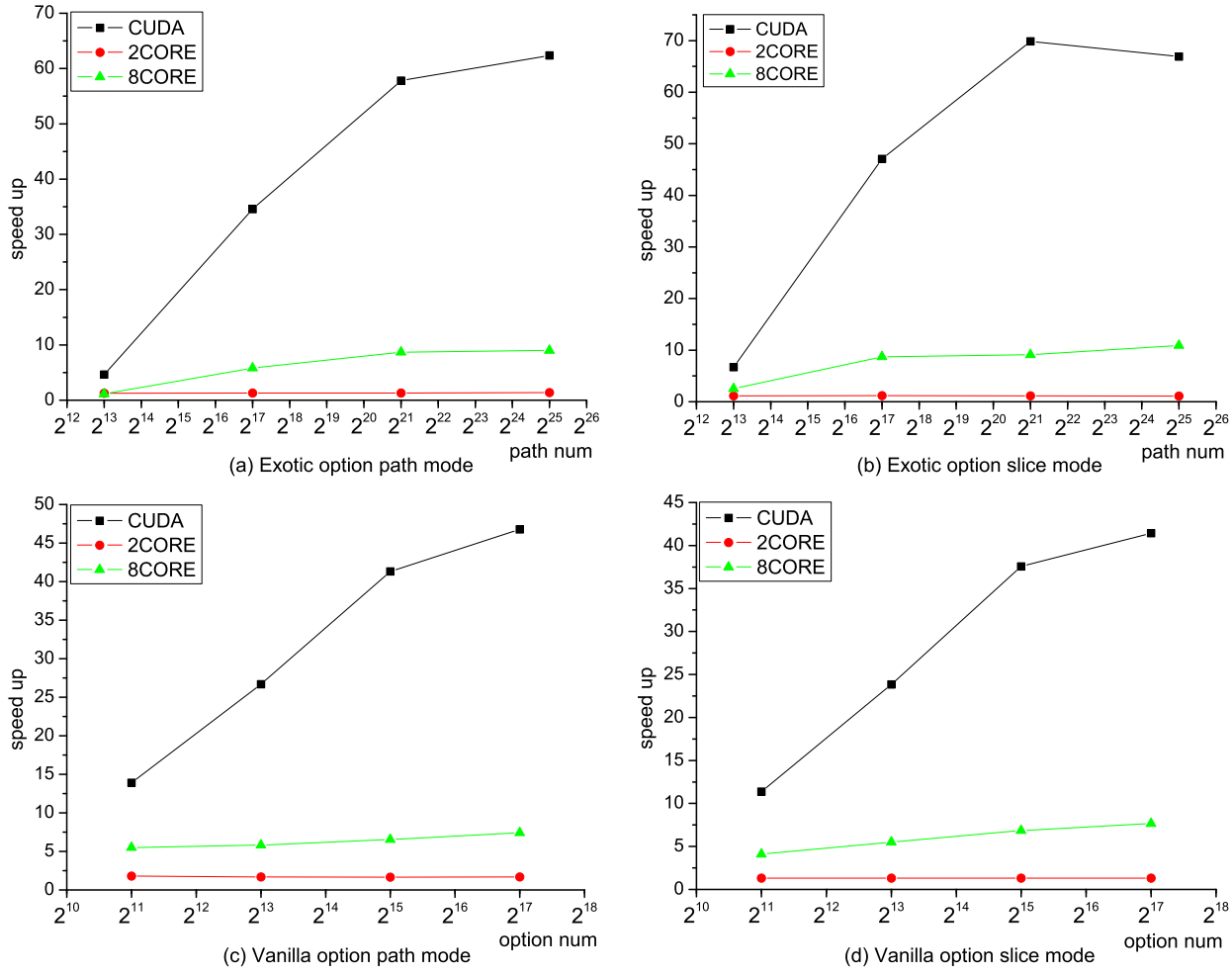

Figure 5: GPU speedup relative to CPU for double precision

shared memory is not enough, the speedup is 6.1. For the Vanilla options pricing, GPU has a speedup factor of 6.3 and 5.4 for VP and VS separately (Figure 5(c) and 5(d)) (for GCC the corresponding speedup is 7.6 or 6.0 accordingly).

5.3 Path mode vs. Slice mode

In this section we take a deep look into the performance difference between the path mode and the slice mode. Figure 6 tells us that for Exotic option pricing, EP is faster than ES; but interestingly for Vanilla options, VS is better, whether for single or double precision.

For the Exotic option pricing, the performance is almost the same as our expectation since the ES requires larger memory space and needs to access global memory when its path number reaches 2^{20}. In fact we can see from Figure 6(a) that when the path number is less than 2^{20}, the performance of EP and ES is reasonably close.

However, for Vanilla options, VP does not have any advantage over VS on memory accesses, since both of them will access global memory. Actually for the Vanilla options pricing, different options have different maturity periods, so the values of $numOfSteps$ are different for each thread, which leads to lots of divergent branches and expensive redundant computations. Therefore, it is the total number of divergent branches that decides the final performance.

Furthermore, for different options, $pathNum$ is the same while $numOfSteps$ is different. For VP, $numOfSteps$ controls the number of inner loops, so divergent branch happens for each step of the inner

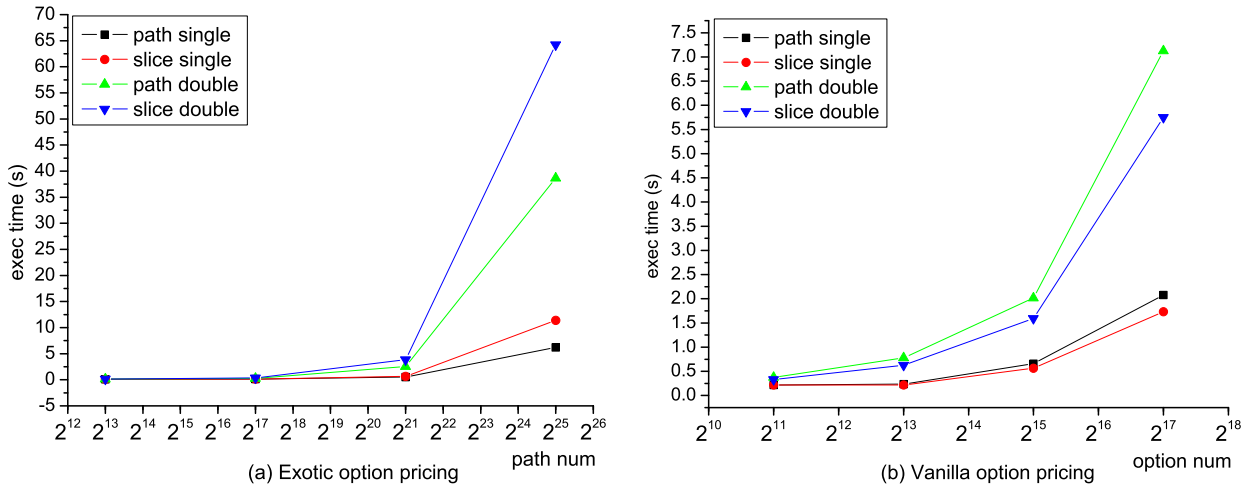

Figure 6: GPU path mode and slice mode performance comparison

loop; while for VS, $numOfSteps$ controls the outer loop, and divergent branch only happens for each step of the outer loop. That is why VS has a better behavior, since it has much less divergent branches, and hence less redundant computations. We profiled VP and VS with the CUDA profiler, and the profiling result shows that there are 9.5E+5 divergent branches for VP, which is 1000 times more than that of VS.

In conclusion, for the Exotic option pricing, especially for EP, we can hold all the computations within shared memory, and we can get a speedup of 43 on GPU for single precision. For the Vanilla option pricing, because of global memory accesses led by a large working set and numerous divergent branches, VS is 18 times faster than the 8-core CPU.

5.4 Accuracy analysis for single and double precision computation

In this subsection, we take EP as an example to compare the computation results of GPU to CPU.

In the MC-based solution for BS model, the simulation paths are independent of one another and the final result takes the average of the discounted values of all paths; therefore the relative error caused by the computation accuracy difference between CPU and GPU is also decided by the average of all relative errors from different paths.

Due to the independence of different paths, small errors from these paths will not be accumulated: for single precision, GPU has a relative error of 0.02% compared to CPU. Also, based on the same reason, the absence of ECC for GPU memory system does not have much impact on the accuracy of the final result since the MC method itself ensures its convergence to a stable value.

Also, the comparison of CPU single to CPU double does not show much difference, with a relative error of 0.01%. Therefore, for this particular application, the accuracy of single precision is good enough for most real usage.

6 Conclusion

In this paper, we present a suit of architecture conscious optimizations for MC-based option pricing algorithm on GPU. In detail, we successfully compress the working set to fit the small on-chip shared memory, reorganize the data layout to eliminate bank conflict and maximize global memory coalescence, and balance the usage of shared memory and registers to improve the multiprocessor occupancy. For the best we get a 43-fold speedup for Exotic option pricing, and 18-fold speedup for Vanilla option pricing, compared with its OPENMP implementation on the Intel Xeon 8-core CPU for single precision computation.

Acknowledgements

This work was co-sponsored by the National Natural Science Foundation of China (Grants NO. 60673144, 60673152, 60773145, 60803121), National High-Tech R&D (863) Program of China (2006AA01A101, 2006AA01A106, 2006AA01A108, 2006AA01A111, 2006AA01A117, 2006AA01Z111, 2008AA01A201), National Basic Research (973) Program of China (2004CB318000), and Tsinghua National Laboratory for Information Science and Technology (TNLIST) Cross-discipline Foundation.

References

1 Tangman D Y, Gopaul A, Bhuruth M. Numerical pricing of options using high-order compact finite difference schemes, Journal of Computational and Applied Mathematics, 2008, Vol 218, Issue 2: 270–280
2 Samuli I, Jari T. Efficient numerical methods for pricing American options under stochastic volatility, Numerical Methods for Partial Differential Equations, 2007, Vol 24, Issue 1: 104–126
3 Black F, Myron S. The Pricing of Options and Corporate Liabilities, Journal of Political Economy 81(3) 1973: 637–654
4 Phelim P B. Options: A Monte Carlo Approach, Journal of Financial Economics 4 (1977) 323–338
5 Acworth P, Broadie M, Glasserman P. A comparison of some Monte Carlo and Quasi-Monte Carlo methods for option pricing, Proceedings of the 1996 Conference on Monte Carlo and Quasi-Monte Carlo Methods. Springer, New York (1998)
6 http://en.wikipedia.org/wiki/Option_(finance)#Types_of_options
7 NVIDIA CUDA Programming Guide v2.0, 2009, Chap 1: 10–11
8 John D O, David L, Naga G, Mark H, Jens K, Aaron E Lefohn, Timothy J P. A Survey of General-Purpose Computation on Graphics Hardware, Eurographics 2005, State of Art Reports: 21–51
9 Bongki M, Joel S. Adaptive Runtime Support for Direct Simulation Monte Carlo Methods on Distributed Memory Architectures, Proceedings of the Scalable High Performance Computing Conference 1994: 176–183
10 Craig K, Matt P. Options Pricing on the GPU, GPU Gems 2, Chapter 45
11 Stanimire T, Michael M, Robert B, Gordon S, John S. Benchmarking and Implementation of Probability-Based Simulations on Programmable Graphics Cards, Computers & Graphics, 2005, Vol 29, Issue 1: 71–80
12 Michael D M, Kevin W, Brent H, Hsin Y L. Performance Evaluation of GPUs Using the RapidMind Development Platform, Proceedings of the 2006 ACM/IEEE conference on Supercomputing, NO. 181
13 Gregoire J, Ecole Centrale P, Tuan N, Arbitragis T. Parallelized Trinomial Option Pricing Model on GPU with Cuda, 2008, available at http://www3.imperial.ac.uk
14 William H P, Saul A T, William T V, Brian P F. Numerical Recipes: The Art of Scientific Computing, 3rd edition, 2007, Chapter 7
15 Makoto M, Takuji N. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Trans. on Modeling and Computer Simulation, 1998, 8(1): 3–30
16 NVIDIA computational finance in CUDA, options pricing with Black Scholes and Monte Carlo: 20–21