CPU-GPU Hybrid Parallel Binomial American Option Pricing

Nan Zhang, Eng Gee Lim, Ka Lok Man and Chi-Un Lei

***Abstract*—We present in this paper a novel parallel binomial algorithm that computes the price of an American option. The algorithm partitions a binomial tree constructed for the pricing into blocks of multiple levels of nodes, and assigns each such block to multiple processors. Each of the processors then computes the option's values at its assigned nodes in two phases. The algorithm is implemented and tested on a heterogeneous system consisting of an Intel multi-core processor and a NVIDIA GPU. The whole task is split and divided over and the CPU and GPU so that the computations are performed on the two processors simultaneously. In the hybrid processing, the GPU is always assigned the last part of a block, and makes use of a couple of buffers in the on-chip shared memory to reduce the number of accesses to the off-chip device memory. The performance of the hybrid processing is compared with an optimised CPU serial code, a CPU parallel implementation and a GPU standalone program.**

***Index Terms*—Parallel computing, option pricing, binomial method, graphics processing unit, heterogeneous processing**

I. Introduction

An American call/put option is a financial contract that gives the contract buyer the right, but not the obligation, to buy/sell a unit of certain stock, whose current price is S_0, at any time until a future expiration date T at a strike price K agreed at the time the contract is established. If the buyer of the contract chooses to exercise the right the option seller must sell/buy a unit of the stock to/from the buyer at the strike price. Since such a contract gives the buyer a right without any obligation, the buyer of the contract must pay the seller a certain amount of premium for this right. The problem of option pricing is to compute the fair price of the contract to both the seller and the buyer.

Black, Scholes and Merton studied this problem, and published their work [1], [2] in 1973. They deduced closed-form formulae for calculating the prices of European call and put options. These options can only be exercised at the expiration date T. However, for American options, because of the early exercise feature, no closed-form formula has been found for computing their prices. Instead, their price can be computed using numerical procedures, such as the binomial or trinomial methods, the finite-difference methods, Monte Carlo simulations, etc.

Option pricing is a crucial problem for many financial practices and so is to be completed with minimal delay. Nowadays, as parallel computers become widely available, many new developments have been advanced in applying parallel computing to the problem of option pricing. Some researchers developed parallel algorithms for various option pricing problems on shared- and distributed-memory multi-processor computers [3]–[6], and some developed algorithms for option pricing on GPUs [7]–[9].

In this paper, we present a parallel algorithm for pricing American options on a heterogeneous system which hosts both a shared-memory multi-core processor and a NVIDIA GPU. The algorithm computes the price of an American option on a recombining binomial tree. The computation is split and divided over the CPU cores and GPU. The implementation was tested on a laptop system with an Intel dual-core P8600 and a NVIDIA Quadro NVS 160M. The performance of the algorithm was tested and analysed.

The contributions of our work are twofold. First, a novel parallel binomial algorithm for option pricing was designed and implemented. The algorithm is suitable for parallel CPU processing and for CPU-GPU hybrid processing. Second, a standalone GPU binomial pricing algorithm was developed and tested. The algorithm improves the binomial option pricing method found in NVIDIA's CUDA SDK 4.0 by removing the restrictions for certain parameter values.

Organisation of the rest of the paper: Section II presents a brief literature review on the application of parallel computing to option pricing. Section III discusses the valuation of an American option on a recombining binomial tree. Section IV presents the hardware configuration of the system used in this work. Section V presents the parallel binomial algorithm that we designed for pricing American options. Section VI presents a CPU implementation of the parallel algorithm and its performance. Section VII shows a GPU binomial pricing algorithm and its implementation. Section VIII presents the hybrid implementation of the parallel algorithm on the CPU and the GPU, and its performance comparison. Finally, conclusions are drawn in Section IX.

II. Related work

Gerbessiotis [3] presented a parallel algorithm that computes the price of a European (or an American) option on a recombining binomial tree. The algorithm partitions a binomial tree into blocks of multiple levels. Each block is further divided and assigned to distinct processors. A processor partitions the sub-block of nodes that has been assigned to it into two regions. Computation in one of the two regions depends on results from nodes out of the region,

Manuscript received December 10, 2011; revised January 31, 2012. This work was supported in part by the XJTLU Research Development Fund under Grant 10-03-08.

Nan Zhang is with the Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University (XJTLU), China. Email: nan.zhang@xjtlu.edu.cn

Eng Gee Lim is with the Department of Electrical and Electronic Engineering, Xi'an Jiaotong-Liverpool University, China. Email: enggee.lim@xjtlu.edu.cn

Ka Lok Man is with the Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, China, Myongji University, South Korea and Baltic Institute of Advanced Technology, Lithuania. Email: ka.man@xjtlu.edu.cn

Chi-Un Lei is with Department of Electrical and Electronic Engineering, the University of Hong Kong, Hong Kong. Email: culei@eee.hku.hk

while that in the other does not have such external dependency. In such a scheme, each block is processed in parallel by multiple processors. The assignment of sub-blocks to processors is fixed from the beginning of the computation. The performance of the algorithm is analysed following the bulk-synchronous parallel model. The implementation of the algorithm is tested in a cluster of PC workstations under a message-passing interface (MPI) and a non message-passing interface. A similar parallel trinomial option pricing algorithm was presented by Gerbessiotis in [4].

Peng et al. [5] presented a similar parallel option pricing algorithm based on the binomial tree method. The parallel program was implemented in C via MPI, and was tested on a cluster of 16 Intel Xeon processors.

Zubair et al. [6] discussed two cache-friendly binomial option pricing algorithms. These algorithms made ample allowance for exploiting the benefit brought about by the memory hierarchy available in today's computers so as to maximise locality for data access. These algorithms were implemented on a single processor and a shared memory multi-processor. A journal extension of this work is found in [10].

All the above-mentioned algorithms parallelise a binomial tree along the axis that represents the stock's price. However Ganesan et al. [11] presented an algorithm where the processing of a binomial tree was parallelised along the time-axis. The algorithm was implemented on a GeForce 8600GT GPU. Since the way they implemented the algorithm and the 16KB shared memory limitation, the implementation can only work with a tree of maximumly 1024 time steps.

Solomon et al. [7] presented a GPU-based trinomial algorithm for pricing European options and a binomial algorithm for pricing American look-back options. The algorithms were implemented and tested on a NVIDIA GTX260 GPU using the CUDA [12] programming model. In pricing the American look-back option on a binomial lattice, the authors implemented a hybrid method where a threshold was preset. As the backward computation proceeds from the leaf nodes to the root, the level of parallelism reduces during the course. So in their algorithm when the backward induction passes over the threshold the computation was taken over by the CPU. The assumption was that the CPU could likely perform the later calculations faster than the GPU.

Dai et al. [8] presented an option pricing algorithm via the solving of backward stochastic differential equations. The equations were solved using a theta method plus Monte Carlo simulations. The algorithm was implemented on a NVIDIA Tesla C1060, and the performance of the GPU implementation was compared against a CPU case.

Surkov [9] presented algorithms for GPU that prices single- and multi-asset European and American options with stock prices following exponential Lévy processes. The algorithms were based on the Fourier space time-stepping method.

For CPU and GPU to work side by side for hybrid parallel processing several challenges have to be solved. Tomov et al. [13] discussed ideas in this respect with the development of a hybrid LU factorisation algorithm where the computation was split over a multi-core and a graphics processor. Of such challenges the synchronisation between CPU and GPU or even between different thread blocks of a GPU execution is foremost. At the moment NVIDIA's CUDA programming environment does not provide any means by which inter-block coordination can be easily handled. So when algorithms are designed for NVIDIA GPUs, the computation task must be decomposed in such a way that each thread block is executed independent of other blocks. This restriction has caused problems to the applications where explicit inter-block synchronisation has to be employed. Some researchers have been looking into this issue. For example, Xiao et al. [14] developed three inter-block synchronisation schemes for NVIDIA GPUs. As in the CUDA programming model the execution of a thread block is non-preemptive, in the schemes, they use an one-to-one mapping between thread blocks and multi-processors (also known as streaming processors). However, a very simple solution to this problem is to stop the GPU kernel at a point where synchronisation is needed and later re-start it. But this stopping and re-launching of GPU kernels is a high-cost operation which hurts performance. This is the synchronisation method we adopted in the implementation of our hybrid algorithm.

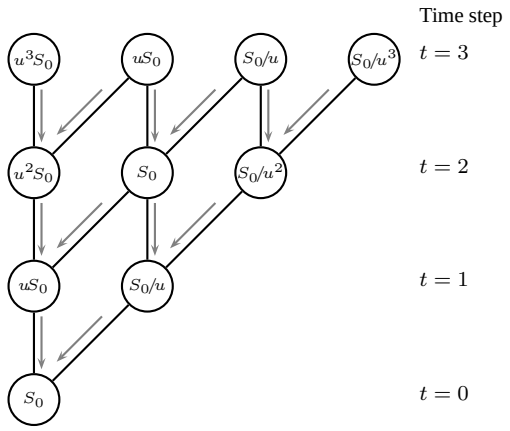


Fig. 1: A recombining binomial tree of 3 time steps and 4 levels. The price at each node is shown in the node. Note that in such a tree at any level $t = n$, the number of nodes in that level is $n + 1$.

III. Binomial American option pricing

The binomial tree model [15] is a widely-used numerical solution to various problems in computational finance. A recombining binomial tree models the dynamic change of a stock price within the time frame from 0 to T. For a binomial tree of N time steps there are $N+1$ node levels, corresponding to the $N + 1$ time spots where $t = 0, 1, 2, \ldots, N$. Any interior node (for example, denoting stock price S) has two successors – an up-move node (denoting price uS) and a down-move node (denoting price dS). If the annual volatility of the stock price is σ, we set u to be $u = \exp(\sigma\sqrt{T/N})$ and $d = 1/u$, so that the tree describes the discrete version of the continuous price change. An example of a recombining binomial tree of 3 time steps and 4 levels is shown in Fig. 1.

To compute the price of an American option (expiration T, strike price K) on a stock (current price S_0), assuming annual continuous compound interest rate R, we start by calculating the option's payoff P_T ($\max(S_T - K, 0)$ for a call, and $\max(K - S_T, 0)$ for a put) at each leaf node. We set the option's value π_T at each leaf node to be the option's payoff P_T at that node. For an interior node (assuming the

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

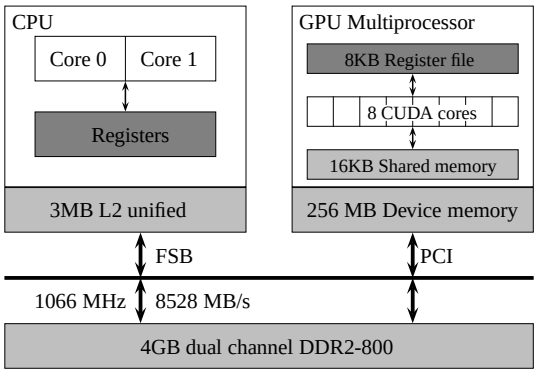


Fig. 2: The CPU-GPU heterogeneous system with an Intel P8600 and a NVIDIA Quadro NVS 160M.

price at which is S_t) we calculate the discounted expected option value $r^{-1}\mathbb{E}(\pi_{t+1}|S_t)$ at the node S_t as:

$$r^{-1}\mathbb{E}(\pi_{t+1}|S_t) = r^{-1}(p\pi^u_{t+1} + (1-p)\pi^d_{t+1}), \qquad (1)$$

where $r = \exp(RT/N)$ is the one time-step interest rate, $p = (r-d)/(u-d)$ is the risk-neutral probability of the up-move, and π^u_{t+1} and π^d_{t+1} are the option's values at the successive up-move and down-move nodes, respectively. Then we set the option's value π_t at node S_t to be the maximum of the discounted expectation and the immediate payoff P_t. So we have:

$$\pi_t = \max(P_t,\ r^{-1}\mathbb{E}(\pi_{t+1}|S_t)). \qquad (2)$$

We apply these steps to all the interior nodes of the binomial tree in the backward induction manner until we get π_0 at the root node.

IV. The CPU-GPU heterogeneous system

The hardware platform (Fig. 2) we used in our work was a laptop system that equipped with a dual-core Intel P8600 (2.4GHz) and a NVIDIA Quadro NVS 160M. The NVIDIA GPU has a single multi-processor that integrates 8 CUDA cores. Their clock speed is 1.45GHz. On-chip the graphics processor has 8KB registers and 16KB shared memory. Off-chip the processor has 256MB device memory installed, which serves as the local, global, constant memories, etc. Accessing the on-chip shared memory is much faster than accessing the device memory. According to NVIDIA's manual [12] the Quadro NVS 160M is of compute capability 1.1, and so it only supports single-precision floating point arithmetic. Eight single-precision floating point operations can be performed per clock cycle per multi-processor.

V. The parallel American option pricing algorithm

To compute the price of an American option on a binomial tree of N time steps ($N+1$ time spots) the parallel algorithm partitions the tree into blocks of multiple levels of nodes. Each block is further divided into equal-sized (except the last one) sub-blocks. The blocks are processed in a sequential order backwards from the leaf nodes. However, within each block the sub-blocks are processed in parallel by distinct processors. The parallel processing of a block consists of two phases. In phase one each processor computes at the

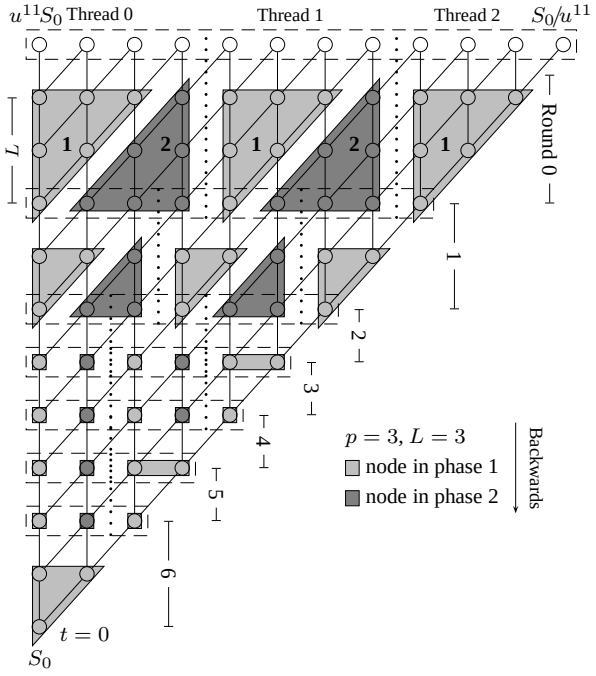


Fig. 3: The parallel algorithm on a binomial tree of 11 time steps.

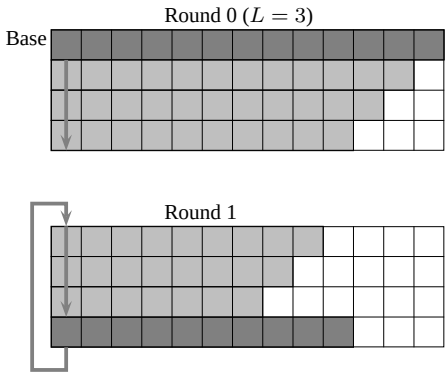


Fig. 4: The modulo wrapping around manner of the local buffer kept by each processor.

half (region A) of the sub-block which has no dependency on nodes out of the region. Once all the processors finish computing nodes in their region A, phase two begins in which each processor computes at the nodes in the remaining half (region B) of the sub-block. After phase two is completed, all the processors move onto the next block. The parallel processing on a binomial tree of 11 time steps is shown in Fig. 3.

In the algorithm we defined a parameter L which specifies the maximum number of levels that a block can have. However, the actual number of levels in a block is also determined by the number of nodes that each processor gets in the base level. To save all the intermediate results each processor maintains a local buffer. The buffer is of $(L+1)$ rows and $(N+2)$ columns. To avoid excessive memory transactions the buffer is used in a modulo wrapping around manner. Fig. 4 shows such an example where in round 0 the base level nodes are saved in row 0 of the buffer, and after the block is processed the nodes saved in row 3 become the base level nodes of the next round. At this point, they are not copied back to row 0.

The synchronisation scheme used in the algorithm is

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

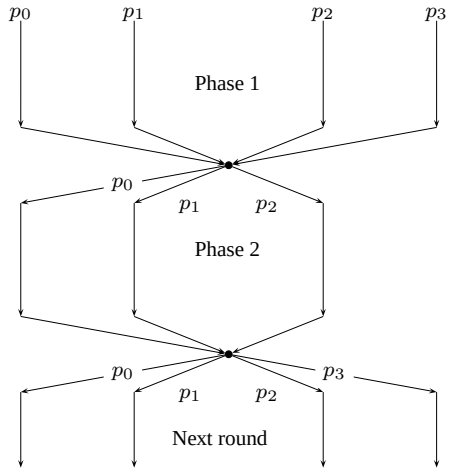


Fig. 5: The synchronisation between multiple threads.

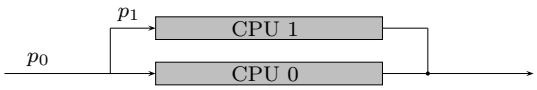


Fig. 6: Binding two threads onto the two cores of the CPU.

shown in Fig. 5. It is always the case that the last thread does not have nodes to process in phase two.

For a p-way parallelism (p distinct processors are used) on an N-step binomial tree, because processor p_0 roughly processes $1/p$ of the total nodes in the tree, the parallel runtime $T_{\rm P} = O(N^2/p)$, the parallel speedup $S = T_{\rm S}/T_{\rm P} = O(p)$ ($T_{\rm S}$ is the serial runtime), the parallel efficiency $E = S/p = O(1)$, and the cost $pT_{\rm P}$ of the parallel algorithm is $pT_{\rm P} = O(N^2)$. So the parallel algorithm is cost-optimal in that the cost has the same asymptotic growth rate as the serial case.

VI. Performance testing on the P8600

On the dual-core Intel P8600 we implemented the parallel algorithm and compared its performance with an optimised serial implementation of the binomial pricing. We used an American put option in the tests, where the parameters were set as: current stock price $S_0 = 100$, strike price $K = 100$, option expiry date $T = 0.6$, annual continuous compound interest rate $R = 0.06$ and annual volatility $\sigma = 0.3$. The number N of time steps varied from 4×10^3 to 56×10^3, with an increment of 4000 in each test. In the parallel implementation we used two threads and explicitly bound them onto each of the two cores of the CPU, as Fig. 6 shows.

In computing the stock's price at a node we did not use S_0. We know that the stock's price S_n^0 at the first node (at column 0) of a certain level where $t = n$ is, as it is shown in Fig. 1, $S_n^0 = u^n S_0$. So the price S_n^j at the j-th column in that level is $S_n^j = u^{-2j} S_n^0$. So for the nodes in any level $t = n$ we computed S_n^0 once and re-used its value at the remaining nodes of the level. By this way we avoided repeatedly evaluating the same mathematical expression. This optimisation made noticeable improvement to the performance of the implementation.

We used single-precision floats in the programs so that we could make comparisons between the performances on the CPU and on the GPU. The operating systems used was Ubuntu Linux 10.10 (64-bit version). The compiler used was Intel's icpc 12.0 for Linux with optimisation options -O3 and -ipo switched on. The POSIX thread library used was NPTL (native POSIX thread library) 2.12.1. The parallel speedups in all the tests are plotted in Fig. 7. In the tests we observed super-linear speedup in some of the tests. This must have been caused by the caching effect and the more efficient use of the system bus. In this group of tests the maximum number L of levels in a block was set to 20.

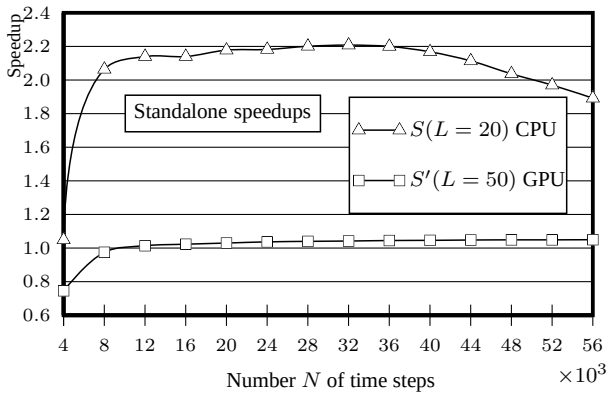


Fig. 7: The speedups of the CPU parallel implementation and the GPU implementation.

VII. The GPU algorithm and its performance

Programming the same binomial American option pricing problem on the Quadro NVS 160M is very different from working with the Intel P8600, because of the SIMT (single instruction multiple threads) execution model of the NVIDIA GPU. The NVIDIA CUDA 4.0 SDK comes with an example where thousands of European calls are priced using the binomial method. In the example, a single one-dimensional thread block is used to price a single call option. The algorithm used in the pricing of a single call is briefly explained in [16]. To avoid frequent access to the off-chip global memory but to make use of the on-chip shared memory as much as possible, the algorithm partitions a binomial tree into blocks of multiple levels. The partition pattern is very similar to the one shown in Fig. 3, except that the NVIDIA's algorithm requires that all the blocks have the same number of levels and this number must be a multiple of two. The algorithm also uses two buffers in the shared memory. The algorithm begins by allocating an one-dimensional buffer in the global memory. All the threads in the thread block compute the option's payoffs at the leaf nodes and save them into the buffer. When processing a block of interior nodes, the threads first load the computed option values from the global buffer into one of the two shared buffers. Then the computation is carried out between the two shared buffers. After this the results are copied back to the appropriate positions in the global buffer. The threads then move to the next part of the block to repeat the same processing.

The algorithm we implemented on the Quadro NVS 160M modified the NVIDIA's algorithm by allowing arbitrary number of levels in a block. A run of our algorithm for NVIDIA GPUs is shown in Fig. 8.

The performance comparison between this GPU implementation and the CPU sequential program is plotted in Fig. 7, where the same American put option example was

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

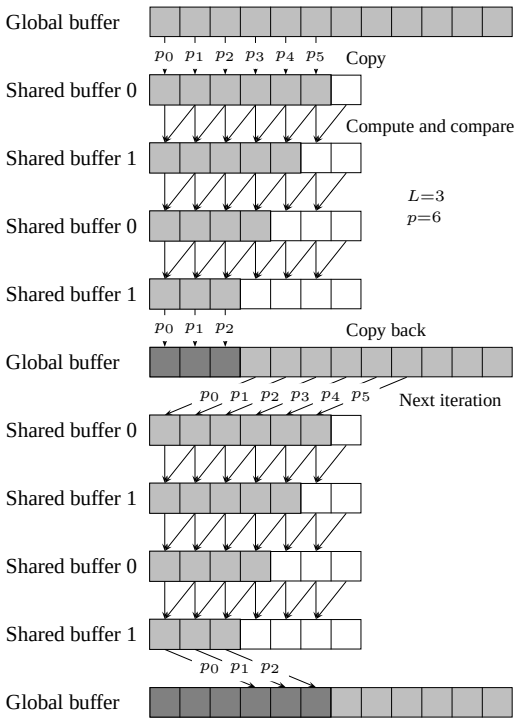


Fig. 8: GPU binomial option pricing with double buffers in the on-chip shared memory. Note that in this example we have 6 threads.

used. From the results we can see that the performance on the GPU was almost the same (or, slightly better in some cases) as that on a single core of the CPU. Without the double-buffer memory access optimisation the GPU's performance was far worse. In all the GPU tests the parameter L (the maximum number of levels in a block) was set to 50, much increased from the CPU parallel tests where L was 20. This was to reduce the number of times that the GPU threads have to access the buffer in the global memory.

VIII. The CPU-GPU hybrid processing

To compute the parallel binomial algorithm (illustrated in Fig. 3) using both the CPU and the GPU in the laptop system (Fig. 2) we assigned the GPU the last sub-block (for example, the part processed by thread p_2 in Fig. 3) in each round, because the GPU algorithm is not suitable for a sub-block that has nodes in region B. Moreover, as the GPU's performance (Fig. 7) on this pricing problem was almost identical to a single core of the CPU, initially the workload was divided equally among the two cores of the CPU and the GPU.

To coordinate the GPU with the two cores of the CPU we have to use one of the two cores for the communication and the synchronisation. Since the launch of a kernel on the GPU is asynchronous, that is, control is returned to the CPU before the task on the GPU is completed, we did not leave the coordinating core of the CPU idle while the kernel is executed on the GPU. We assigned an equal part of the total workload to the coordinating core. This distribution of workload on the CPU and GPU is illustrated in Fig. 9.

The parallel algorithm (Fig. 3) requires all the threads working at block i to finish before the processing of block $i+1$ to start. So at the end of each round the GPU kernel had to be ended and a new kernel was launched at the start

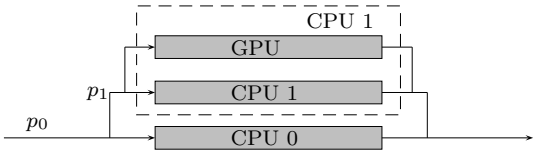


Fig. 9: Using a CPU core to coordinate with the GPU. An equally-sized workload is assigned to that core.

of each new round. Launching a new kernel every round is a high-costly operation, but this is the price that one has to pay in order to make the CPU and the GPU work side by side. Algorithm 1 shows the steps performed by this coordinating core of the CPU.

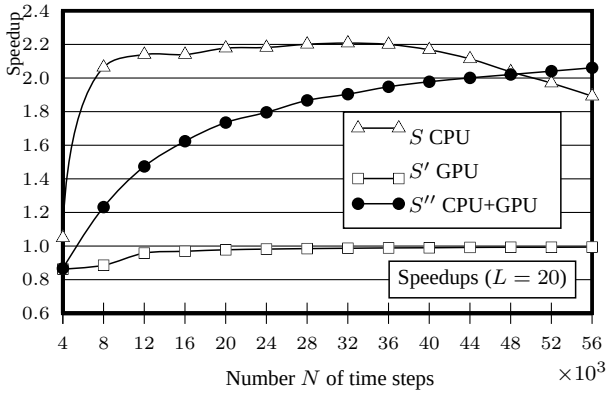
Algorithm 1: Computational steps performed by the coordinating core.

```
begin
    // Initialisation
1   Compute option's payoffs at the end-level nodes assigned to the
    core and the GPU;
    // Backward induction
2   while there is a next round do
        // Phase 1
3       if GPU is needed then
4           Launch kernel for the part assigned to the GPU;
5       Compute at region A of the sub-block assigned to the CPU
        core;
6       if GPU is needed then
7           Wait for the GPU to finish;
8           Copy data from the GPU buffer to the CPU buffer;
9       Synchronise with the other CPU cores;
        // Phase 2
10      if there is region B then
11          Compute at region B of the sub-block assigned to the
            CPU core;
12          if GPU is needed then
13              Copy data from the CPU buffer to the GPU buffer;
14      Synchronise with the other CPU cores;
        // For next round
15      Update variables and parameters;
end
```

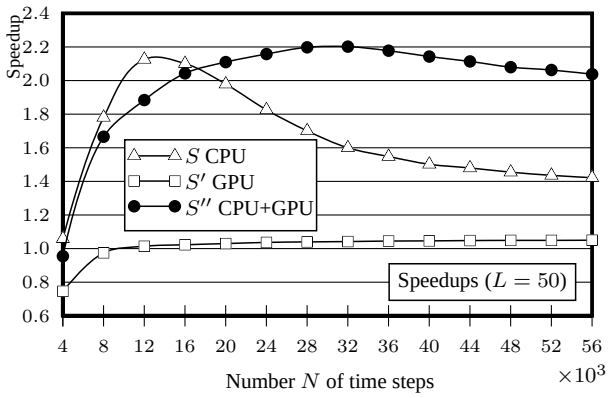
To see the performance of the hybrid algorithm we did two groups of tests where L, the maximum number of levels in a block, was set to 20 and 50, respectively. The tests were made using the same American put option with the same parameter setting. The GPU kernel were launched with a single thread block of 512 threads. The speedups are plotted in Fig. 10.

From the results we can see that when $L = 20$ the CPU parallel implementation with 2 working threads outperformed the hybrid processing, but when $L = 50$ the opposite situation was observed. The reason for the first observation was that the repetitive launching of GPU kernels reduced the performance of the hybrid processing. When $L = 50$, the number of launchings was reduced so that the performance of the hybrid processing was improved. However, when $L = 50$ the CPU parallel code became poorly performed. The reason was that the local buffer that saved the intermediate results at the CPU side became less efficient for caching when L became large. According to the theoretical analysis the speedup S of this parallel algorithm is $S = O(p)$. However, the performance of the hybrid processing after adding the GPU did not show significant enhancement over the CPU parallel code. We believe that this was due to the coordination

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)



(a) $L = 20$



(b) $L = 50$

Fig. 10: Speedup plots of the CPU parallel implementation and the hybrid implementation.

overhead between the CPU and the GPU.

IX. Conclusion

We have presented a parallel algorithm that computes the price of an American option on a recombining binomial tree. The tree is partitioned into blocks of multiple levels of nodes. A block is divided into sub-blocks and these sub-blocks are assigned to distinct processors to be processed in parallel. The processing of a block by multiple processors consists of two phases. In phase one, the processing is carried out on the nodes at which the computation has no external dependency, and in phase two, the nodes are processed where such dependency exists and has been resolved in phase one. The parallel algorithm dynamically adjusts the assignment of sub-blocks to processors since the level of parallelism decreases as the computation proceeds from the leaf nodes to the root.

The parallel algorithm was implemented on the dual-core CPU. In some of the test cases super-linear speedups were observed against an optimised serial CPU code. A GPU binomial pricing algorithm was then discussed, where double buffers in the on-chip shared memory are used to reduce the number of accesses to the off-chip device memory.

The parallel algorithm was then adapted to the dual-core CPU and the GPU. The partition of a binomial tree is in such a way that the GPU is always given the last sub-block to compute. To coordinate the GPU with the CPU we had to use one of two CPU cores to repeatedly launch the GPU kernel and then stop it at the synchronisation point. This has caused much overhead and reduced the performance of the hybrid processing.

As a future work, the proposed algorithms in this paper may be implemented as a customised integrated circuit (IC) using the FPGA technology. Such an IC will be expected to improve significantly the runtime performance of the pricing algorithms.

References

[1] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *The Journal of Political Economy*, vol. 81, no. 3, pp. 637–659, 1973.

[2] R. Merton, "Theory of Rational Option Pricing," *Bell Journal of Economics and Management Science*, vol. 4, no. 1, pp. 141–183, 1973.

[3] A. V. Gerbessiotis, "Architecture Independent Parallel Binomial Tree Option Price Valuations," *Parallel Computing*, vol. 30, pp. 301–316, 2004.

[4] ——, "Parallel Option Price Valuations with the Explicit Finite Difference Method," *International Journal of Parallel Programming*, vol. 38, pp. 159–182, 2010.

[5] Y. Peng, B. Gong, H. Liu, and Y. Zhang, "Parallel Computing for Option Pricing Based on the Backward Stochastic Differential Equation," *Lecture Notes in Computer Science*, vol. 5938, pp. 325–330, 2010.

[6] M. Zubair and R. Mukkamala, "High Performance Implementation of Binomial Option Pricing," *Lecture Notes in Computer Science*, vol. 5072, pp. 852–866, 2008.

[7] S. Solomon, R. K. Thulasiram, and P. Thulasiraman, "Option Pricing on the GPU," in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, Melbourne, Australia, sep 2010, pp. 289–296.

[8] B. Dai, Y. Peng, and B. Gong, "Parallel Option Pricing with BSDE Method on GPU," in *Proceedings of the 9th International Conference on Grid and Cloud Computing*, Nanjing, China, nov 2010, pp. 191–195.

[9] V. Surkov, "Parallel Option Pricing with Fourier Space Time-stepping Method on Graphics Processing Units," *Parallel Computing*, vol. 36, no. 7, pp. 372–380, jul 2010.

[10] J. E. Savage and M. Zubair, "Cache-Optimal Algorithms for Option Pricing," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, pp. 7:1–7:30, jan 2010.

[11] N. Ganesan, R. D. Chamberlain, and J. Buhler, "Accelerating Options Pricing Calculations via Parallelization along Time-axis on a GPU," in *Proceedings of the 1st Symposium on Application Acceleration and High Performance Computing (SAAHPC '09)*, Urbana-Champaign, Illinois, jun 2009.

[12] *NVIDIA CUDA C Programming Guide (version 4.0)*, NVIDIA Corporation, 2011.

[13] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, jun 2010.

[14] S. Xiao and W. chun Feng, "Inter-block GPU Communication via Fast Barrier Synchronization," in *Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, apr 2010, pp. 1–12.

[15] J. C. Cox, S. A. Ross, and M. Rubinstein, "Option Pricing: A Simplified Approach," *Journal of Financial Economics*, vol. 7, no. 3, pp. 229–263, sep 1979.

[16] C. Kolb and M. Pharr, "Options Pricing on the GPU," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr and R. Fernando, Eds. Addison-Wesley, 2005, ch. 45.

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)