Parallel Implementation of an Out-of-Core Financial Application on a GPU

Myungho Lee, Joon S. Kim, Sugwon Hong
Dept of Computer Science and Engineering
Myong Ji University
Yong In, Gyung Gi Do, Korea
myunghol@mju.ac.kr

ABSTRACT

The architecture of the latest Graphic Processing Unit (GPU) consists of a number of uniform programmable units integrated on the same chip, which facilitate the general-purpose computing beyond the graphic processing. With the multiple programmable units executing in parallel, the latest GPU shows superior performance for many non-graphic applications. Furthermore, programmers can have a direct control on the GPU pipeline using easy-to-use parallel programming environments. These advances in hardware and software make General-Purpose GPU computing (GPGPU) widespread. In this paper, we parallelize a computationally demanding financial application and optimize its performance on a latest GPU. We also analyze the performance results compared with those obtained using CPU only. Experimental results show that GPU can achieve a superior performance, greater than 190x, compared with the CPU-only case when the data fits in the graphic memory. We also address the performance issue in the out-of-core case where the data cannot fit in the device memory on the GPU. In such a case, by using streaming technique helps make up the performance gap lost due to data transfer overhead from the CPU side to the GPU DRAM.

Keywords

GPU, High Performance Computing, Monte-Carlo simulation, Shared-Memory,, out-of-core.

1. INTRODUCTION

The Graphic Processing Unit (GPU) was introduced in the late 1990s as a co-processor for accelerating the simulation and visualization of 3D images commonly used in applications such as game programs. The first GPU was introduced in the market in 1999: NVidia GeForce 256. Since then GPU has become widespread and these days it is incorporated in almost all Desktop Computers and becoming increasingly popular in the server platforms. The clock rate of the latest GPU has ramped up to 675Mhz compared with 120Mhz in the earlier models. Furthermore it has shown impressive improvement in the floating-point performance, far exceeding that of the latest CPU's, and the performance gap is widening.

In the architecture of earlier GPU's, there were separate processing units for Shader, Vertex, Pixel. However, in the latest GPU's, those units are incorporated into uniform programmable processing units on the same chip, which facilitates homogeneous parallel programming. In order to utilize the latest flexible hardware design, more user friendly programming environments have been recently developed by the GPU vendors. This lets programmers have more direct control over the GPU pipeline and memory hierarchy, whereas in the old GPU's they had to rely on specific graphics API's. The flexible GPU hardware and user friendly software have led to a number of innovative performance improvements in many application areas and more improvements are still to come.

In this paper, we deploy GPU's for the computationally demanding financial derivatives modeling applications. We use Box-Muller approach for generating random numbers following standard distribution. We then utilize the generated random numbers in modeling the Monte Carlo simulation for predicting future stock prices for many options. Using NVidia's CUDA (Compute Device Unified Architecture) parallel program development environment, we parallelize the Box-Muller code and Monte-Carlo simulation code, and conducted performance experiments on a latest GPU based computer system. Our experimental results show that GPU can perform 190 times better than the latest dual-core Intel CPU for the financial code when the data fits in the Graphic-DRAM (G-DRAM). When it doesn't fit in the G-DRAM the performance is suboptimal, where we need to hide the data copy overhead from the host to the GPU with useful computations on the GPU. Using an extra thread on the CPU which is in charge of data transfers to the device memory on the GPU, we successfully made up for the performance loss.

The rest of the paper is organized as the following: In section 2, some background information on state-of-the-art GPU technologies and General-Purpose GPU computing (GPGPU) is introduced. In Section 3, we describe financial derivatives modeling technique using Mon-Carlo simulation. In Section 4, we describe our parallelization methodologies for the financial derivative modeling applications along with the performance results. Section 5 concludes the paper.

2. GPU COMPUTING

Ever since the first Graphic Processing Unit (GPU) was introduced in the late 1990s (NVidia GeForce 256), GPU has become widespread and become main part of the modern computer systems. Lots of improvements have been made in the architecture, performance, and programmability. Recently, in order to utilize the flexible GPU architecture design, more user friendly programming environments have been developed by the GPU vendors. CUDA from NVidia, OpenCL are good examples of such software environments. Using those environments, programmers can have more direct control over the GPU pipeline and memory hierarchy.

CUDA includes compilers, libraries, debug/profiler. It is designed to operate on NVidia's Geforce 8600 GPU's or higher models. In order to execute CUDA programs on the NVidia's GPU, a hierarchy of memories is used. They are registers and local memories belonging to each thread, a shared memory used in a thread block, and Global memory accessed from the thread block arrays:

- Global memory is an off-chip GDDR of which the size ranges from 256MB to 4GB. Through the Global memory GPU can communicate with the host CPU.
- Shared memory sits within each thread block. Its typical size is 16KB. The access time closely matches with the register access time, thus it is a very fast memory.
- Registers are used for temporarily storing data used for GPU computations, similar to CPU registers.

In CUDA programs, data needed for computations on GPU is transferred from the host memory to the Global memory on the GPU, distributed to the shared memories by the programmer, then used by thread blocks and threads. Therefore, GPU computations are suitable for parallel executions of the same instruction on a large block of data rather than performing various computations on a small amount of data.

3. MONTE-CARLO SIMULATION FOR FINANCIAL APPLICATION

We first describe Monte-Carlo simulation for predicting the prices of the futures. Then we describe techniques to generate random numbers to be used for Monte-Carlo simulation, including Box-Muller approach.

3.1 Monte-Carlo Simulation

Monte-Carlo simulation is on the opposite side of deterministic algorithms where, given a set of formulae, the precise value for a variable can be calculated. Monte-Carlo simulation is used where a deterministic approach is not properly working. In such a case, we assume a random variable and also assume a certain probability distribution for the random variable. Then we generate random or pseudo-random numbers following the probability distribution. The generated random numbers are used to compute the predicted values for the random variable [12]. In this paper, the Monte-Carlo approach is used in the financial derivatives modeling, the future stock prices.

Stock prices change continuously, but unexpectedly. It is practically impossible to predict future stock prices. In order to evaluate the stock prices of the future, we need to have a model of the "uncertainty" of the stock price itself. Building a formal model for stock price changes considering numerous economic factors related is a challenging task. Furthermore the complexity to compute future stock prices based on the model will be exponential. Thus Monte-Carlo simulation is an attractive alternative. In order to obtain accurate predicted values for a given random variable using Monte-Carlo simulation, generating and using a sufficiently large number of random numbers is crucial. Due to the large problem size (random numbers), it is computationally demanding. Thus parallel execution is essential for the Monte Carlo simulation.

3.2 Random Number Generation

Random number generation is crucial for Monte-Carlo simulation. Random numbers generated using computer programs, however, are not true random numbers due to the nature of algorithms used which repeat a given sequence of steps. Those numbers could rather be called Pseudo-Random Numbers (PRN). In this subsection, we introduce three representative methods to generate PRN's with emphasis on Box-Muller approach [12].

3.2.1 Inverse Transform Method

Inverse Transform Method can easily generate a random variable by using the inverse function of a probability distribution function and a random number following uniform distribution. Therefore if it is possible to compute the inverse function of a distribution function, we can easily generate random variables following normal distribution and other probability distributions. However, it is not always possible to compute the inverse function of a distribution function [12].

3.2.2 Quasi-Random Sequences

Quasi-Random Sequence (QRS) is also called a low-discrepancy sequence. This method takes advantage of the property that the consecutive points in the sequence tend to widen the gap and fill up the gap with another number. This approach removes the clustering of numbers and thus leads to generating random numbers. This method is most preferred for Monte-Carlo simulation. However, since it is finding a sequence it naturally generates dependencies. Thus it is not suitable for the parallel execution [12].

3.2.3 Box-Muller Approach

Box-Muller approach overcomes the shortcoming of the Inverse Transform Method and most widely used. This method doesn't attempt to find an inverse function of a probability distribution function. Instead it attempts to generate random numbers following normal distribution through PRN:

1. Generate two independent random variables following uniform distribution: U1, U2, ~ U(0,1)
2. Using U1 and U2, generate two random variables X1 and X2 following a normal distribution N(0,1) using the following formulae:

$$X_1 = \sqrt{-2\ln U_1}\, cos 2\pi U_2,\; X_2 = \sqrt{-2\ln U_1}\, sin 2\pi U_2$$

In Inverse Transform Method and Box-Muller approach, we first need to generate random numbers following uniform distribution so that we can generate random numbers following standard normal distribution. The generated random numbers are, however, not perfectly random. To some extent, this can be compensated by generating more numbers. In particular, Box-Muller approach needs a simpler algorithm than other PRN generation methods because it doesn't need to compute the inverse function. Also the PRN generation steps are independent and can be easily parallelized. For these reasons, we are using this approach of generating random numbers in this paper..

4. PARALLEL EXECUTION OF MONTE-CARLO SIMULATION

In this Section, we parallelize the Monte-Carlo simulation using CUDA, and conduct performance experiments on a GPU and compare the performance of a serial code on a CPU. As explained in Section 3, we use Box-Muller approach for generating random numbers for the Monte-Carlo simulation. We also show the performance effect of built-in variables, gridDim and blockDim.

4.1 Machine Platform

Experiments were conducted on a Desktop system employing Intel Dual-Core E6750 chip and NVidia GeForce 8600. GeForce 8600 operates at 540Mhz and includes 32 thread processors. It is connected to 256MB GDDR2 memory off the GPU chip but on the same graphic card [7]

4.2 Experimental Results

4.2.1 Box-Muller Approach

Using Box-Muller approach, random numbers are first generated, then a sequence of simple operations is performed on the generated random numbers to form a standard normal distribution. Generated random numbers are stored in a one-dimensional array. A sequence of operations are performed on the data loaded from the array, then stored back to the array afterwards. Thus there is not much data transfer. Using a serial code, we first measure the run time on the CPU. Then we measure the run time of the parallel CUDA code on the GPU. The generated random numbers are "float" type. Data sizes were varied between 3 million to 24 million points. <Table 1> below show that GPU run times are 152~173 times faster than a serial code run on a CPU.

Table 1. Performance of Box-Muller Approach

Data Size	Serial Run Time (CPU only)	Parallel Run Time (CPU+GPU)	Speedup
3 Million	272.22ms	1.78ms	153
6 Million	548.85ms	3.59ms	152
12 Million	1,095.15ms	6.42ms	170
24 Million	2,183.90ms	12.57ms	173

4.2.2 Monte-Carlo Simulation

We conducted performance experiments using Monte-Carlo simulation for predicting future stock prices. Random numbers generated using Box-Muller approach was used for the simulation. Unlike the Box-Muller approach, this simulation exhibits a lot of data movements. It also uses the code to find the minimum and maximum values a lot, which can be easily parallelized. More program characteristics as follows:

- Exp() operations and summations are intensively performed. These operations are performed on the generated random numbers using Box-Muller approach. Therefore they can be easily parallelized.
- Unlike in the Box-Muller where there is a lot of data transfers between the GPU's Global memory and the GPU registers, the data array which stores the variable for the standard normal distribution is divided into small chunks (1KB~8KB) and copied to the shared memory. This leads to a reduced access time for the thread processor to the data array.
- The outputs from the threads are stored in two arrays. Reduction operations are applied to these arrays to compute the local sum for each thread. The computed local sums are stored in g_odata and g_odata2 in Global memory. Since these two arrays are also partial sums, they are added to form the final sum. The final result is transferred to the host.

As shown in (Fig 1), the parallel run times are faster than the serial times by 190~193x. (Both run times are denoted in msec.) The high performance gains were made possible by efficiently exploiting the shared memory on the GPU.

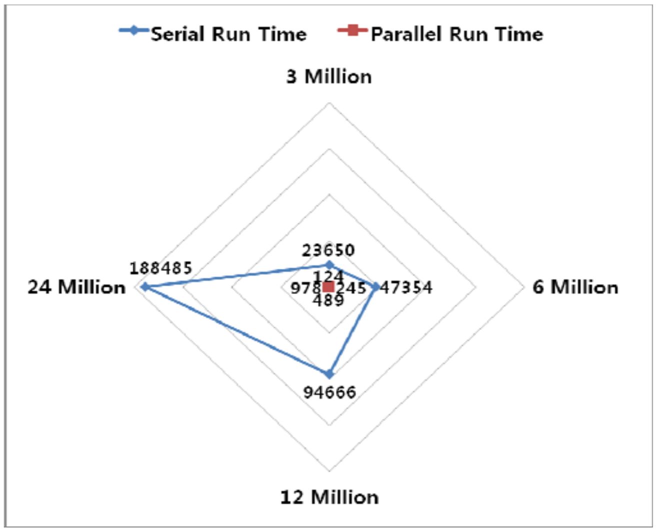

Figure 1. Run Time of Monte-Carlo Simulation

4.2.3 Out-of-Core Case

When the data size is big and it doesn't fit in the Graphic-DRAM (G-DRAM), we need to partition the data, send a portion of the data to the GPU in the G-DRAM (using CUDA's memcpy), perform computations on the GPU, receives the results from the GPU. This code transformation results in dependencies among partitioned data. Thus a lot of data movement overhead is involved and the performance drops significantly. (The speed-up drops to only 10x level).

In order to overcome the overhead, we've created two OpenMP threads on the host CPU, one thread for transferring data to the GPU, another thread for controlling CUDA kernel programs. In this way, the computations on the GPU can be overlapped with the data transfers. We've conducted experiments for a larger size Monte-Carlo simulation where the data size (1.3GB) exceeds the G-DRAM size. For the experiments, we've used the latest GPU, GeForce GTRX285 with 1GB of G-DRAM. The performance results are as follows:

- Using one CPU core only case, we've observed a run time of 7980 sec.
- Using two CPU cores for overlapping data transfer and GPU computations, we've observed a run time of 120 sec.
- Thus the resulting speed-up is 66.5x. This is about 6.6x improvements compared with no data partitioning.

5. CONCLUSION

In this paper, we parallelized a computationally demanding financial application on a GPU and conducted a performance studies by comparing the performance on the GPU with the CPU run time:

- For generating random numbers, we used Box-Muller approach which requires relatively small amount data traffic between the host CPU and the GPU. The parallelized program using CUDA showed greater than 170x performance gain compared with the serial run time on a CPU.
- The Monte-Carlo simulation for predicting future stock prices was parallelized using CUDA. As the simulation requires a lot of data movement, we exploit the shared memory to reduce the traffic to the Global memory. This led to greater than 190x performance gain compared with the serial run time on a CPU.
- We also conducted performance studies for the out-of-core case where the data size is bigger than the G-DRAM. We've created two OpenMP threads so that one thread is responsible for transferring data to the GPU while another thread is controlling the kernel CUDA programs on the GPU responsible for the computations. We've obtained a significant improvement by the overlapping of computation and data movement.

6. ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the ministry of Education, Science, and Technology (Grant No: 2010-0012059).

7. REFERENCES

[1] "AMD ATI Radeon™ HD 4800 Series", http://ati.amd.com/products/radeonhd4800/index.html

[2] V. Volkov and J.W. Demmel,"Benchmarking GPUs to Tune Dense Linear Algebra", Proceedings of the SuperComputing 08, pp. Art. 31:1-11, Nov 2008.

[3] K. Fatahalian , J. Sugerman , P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication", Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, August 29-30, 2004, Grenoble, France.

[4] K. Fatahalian, M. Houston, "A Closer Look at GPUs", Comm. of ACM, Oct 2008.

[5] B. He, N. Govindaraju, Q. Luo, B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors", Proceedings of the SuperComputing 07, pp. 175-186, Nov 2007..

[6] "NVidia gtx280", http://kr.nvidia.com/object/geforce_family_kr.html

[7] "NVidia CUDA", http://developer.nvidia.com/object/cuda.html

[8] Matt Pharr et. al., "GPU Gems 2", Addison Wesley, 2004.

[9] NVidia CUDA Programming Guide", http://kr.nvidia.com/object/cuda_develop_kr.html

[10] Sam S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Bradley P. Sutton, Zhi-Pei Liang, "Accelerating advanced MRI reconstructions on GPUs", J. Parallel Distrib. Comput. Vol. 68, No 10, pp.1307-1318, 2008.

[11] K. Lee, Y. Kwon, J. Shin,"Final Derivatives Modeling Using MATLAB", Ah-Jin Press, 2007.

[12] D.P. Playne, et. al., "Benchmarking GPU Devices with N-Body Simulations", Massey University, Technical Report CSTN-077, 2009.

[13] S. Tomov, et. al., "Towards dense linear algebra for hybrid GPU accelerated manycore systems", Parallel Computing, Vol 36, Issue 5-6, 2010.

[14] W. Kim, J. Kim, "A Parallel Monte Carlo Simulation for Financial Derivative Pricing", Korea Computer Congress, Oct. 2008

[15] Y. Li, et. al., "A Note on Auto-Tuing GEMM for GPUs", Lecture Notes on Computer Science, Vol 5544, 2009.