Accelerating Financial Applications on the GPU

Scott Grauer-Gray, William Killian, Robert Searles, John Cavazos
Computer and Information Sciences, University of Delaware
Newark, DE 19716
{sgrauerg,killian,rsearles,cavazos}@udel.edu

ABSTRACT

The QuantLib library is a popular library used for many areas of computational finance. In this work, the parallel processing power of the GPU is used to accelerate QuantLib financial applications. Black-Scholes, Monte-Carlo, Bonds, and Repo code paths in QuantLib are accelerated using hand-written CUDA and OpenCL codes specifically targeted for the GPU. Additionally, HMPP and OpenACC versions of the applications were created to drive the automatic generation of GPU code from sequential code. The results demonstrate a significant speedup for each code using each parallelization method. We were also able to increase the speedup of HMPP-generated code with auto-tuning.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; J.1 [**Administrative Data Processing**]: Financial

General Terms

Performance

Keywords

Computational Finance, GPGPU, GPU, HMPP, OpenACC, CUDA, OpenCL, Optimization, auto-tuning

1. QUANTLIB

QuantLib [3] is a open-source library written in high-level C++ used for quantitative finance. The application includes codes to evaluate a variety of option types, bonds, and swaps. It also contains models for yield curves, interest rates, and volatility as well as a number of methods including analytic formulae, tree methods, finite difference methods, and monte-carlo. The library is well-developed and contains many parameters and options within each module.

GPGPU-6 March 16 2013, Houston, TX, USA

This work focuses on applying GPU acceleration on particular paths of Black-Scholes, Monte-Carlo, Bonds, and Repo financial applications in the QuantLib library.

2. GPU ACCELERATION OF QUANTLIB

In recent years, the graphics processing unit (GPU) has evolved from specialized hardware to a powerful parallel processor that can be used as a co-processor to the CPU to accelerate particular applications [17]. The development of CUDA and OpenCL [14] environments allows programmers to compile and run parallel kernels targeted for GPUs without needing to map the kernels to a graphics environment such as OpenGL. Furthermore, the development of directive-based GPU programming allows the programmer to target the GPU by simply placing pragmas in sequential code with the compiler generating the code to drive parallelism. This work looks at using CUDA and OpenCL as well as OpenACC and HMPP directives to accelerate QuantLib code paths.

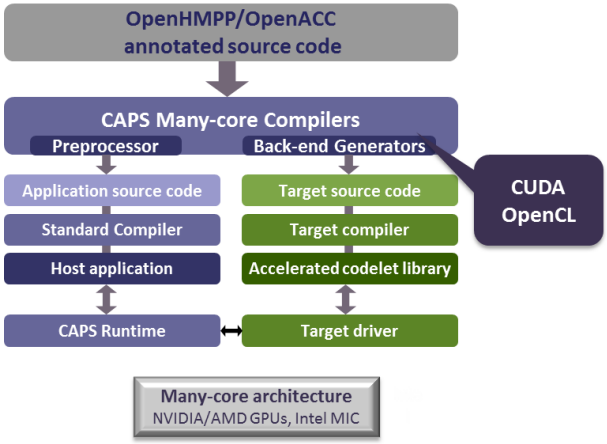
2.1 CUDA/OpenCL Environments

The QuantLib library is written in object-oriented C++ that contains many layers of abstraction and supports a large range of parameters for computation. With a goal of GPU acceleration of particular code paths in the library, the target applications are run line-by-line using a debugger to determine the areas of code that must be implemented in CUDA/OpenCL. Code flattening, or the removal of high-level code abstraction, is applied to the high-level C++ code. The original code is translated into a set of lower-level structures and functions implemented outside of the class environment. This code modification allows the program to run in the more limited GPU environment.

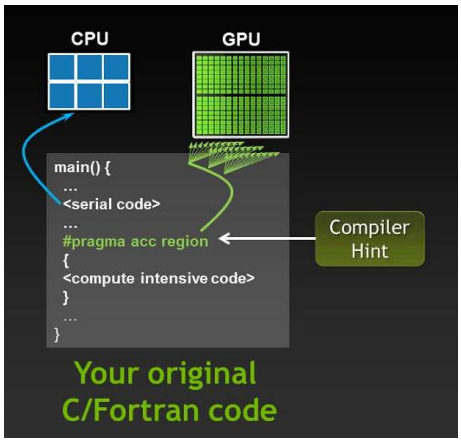
In order to fairly compare the GPU implementations to a sequential CPU run, the same implementation without the abstraction in QuantLib is written for the CPU. The modified code is validated by comparing the outputs of the modified code to the output values of the original high-level code when using the same input values and parameters.

2.2 Directive-Based GPU Acceleration

The HMPP Workbench 3.2.1 compiler from CAPS Entreprise [10] is used to compile and run the directive-based parallel versions of the financial applications. Both the OpenACC and HMPP directive-based languages are used to drive GPU code generation. The HMPP workflow from annotated source code to binary executable is shown in Figure 1a. OpenACC is an open standard that was developed by



(a) HMPP compiler workflow



(b) Illustration of OpenACC with 'region' pragma

Figure 1: Directive Compilers targeting GPU Accelerators

NVIDIA, CAPS, PGI, and Cray. OpenACC has a similar syntax to OpenMP to drive parallelism on the GPU [22]. An illustration of OpenACC with a sequential region and a parallelized loop using the 'region' pragma is shown in Figure 1b. Meanwhile, HMPP was developed by CAPS and provides advanced options including optimizations such as loop unrolling and permutation.

The stripped-down sequential CPU implementation described in Section 2.1 is used as a basis to generate the directive-based parallel implementations, as HMPP Workbench does not currently support high-level C++. Both OpenCL and CUDA can be used as target architectures for OpenACC and HMPP.

3. FINANCIAL APPLICATIONS

The experiments described in this section use the financial applications described in Table 1 and are run on an NVIDIA Fermi-based C2050 compute GPU card composed of 448 cores [19] and a Kepler-based K20 composed of 2,496 cores [21]. The CPU used in all collective experiments is a set of two Quad-Core Intel Xeon X5530 CPUs clocked at 2.40GHz. The results for each algorithm show the speedup using hand-written CUDA and OpenCL as well as OpenACC and HMPP targeting CUDA and OpenCL, with a multi-core CPU implementation using OpenMP is included for additional reference. The hand-written GPU codes are relatively naive and straightforward parallel implementations that do not take advantage of more advanced GPU features such as shared memory. Complete source code for each QuantLib application described in this work is available at `http://sourceforge.net/projects/quantlib-gpu/`.

Some OpenMP codes show speedups over the sequential CPU implementation of more than 8 times when run on applications using double-precision floating-point numbers. This speedup is possible due to the compile flags used for compilation. Specifically, GCC 4.7.0 is used to compile the OpenMP code with the flags set to `"-O3 -march=native -fopenmp"`. The sequential CPU codes are compiled with GCC 4.7.0 and the flags set to `"-O2"`. Analyzing the OpenMP code shows generated vector instructions with the more aggressive compiler flag options.

Application	Description
Black-Scholes	Option pricing using Black-Scholes-Merton Process
Monte-Carlo	Pricing of a single option using QMB (Sobol) Monte-Carlo method
Bonds	Bond pricing using a fixed-rate bond with a flat forward curve
Repo	Repurchase agreement pricing of securities which are sold and bought back later

Table 1: Description of each financial application

3.1 Black-Scholes

The first QuantLib application that is accelerated using the GPU is the Black-Scholes-Merton process with the Analytic European Option engine. This application is from the European Option test in the QuantLib test suite. The GPU implementation is structured such that each option pricing is run in parallel in a separate thread on the GPU. Single-precision 32-bit floating-point numbers are used for the Black-Scholes implementation.

Figure 2 shows the speedup graph when using multi-core CPU and GPU acceleration on the C2050 using the single-core CPU time as the baseline. Figure 2a shows the speedup using a CUDA target and Figure 2b shows the speedup using an OpenCL target on the NVIDIA C2050. The speedup for all GPU implementations is over 100 times over the sequential CPU implementation and over 10 times over the multi-core OpenMP implementation when running over 20,000 options. The directive-based acceleration implementations when targeting CUDA performs around 50% slower on average compared to the hand-written CUDA implementation. When targeting OpenCL, HMPP and OpenACC perform about the same, with the hand-written OpenCL implementation being nearly twice as fast. The manual OpenCL implementation is over 40% faster than the hand-written CUDA implementation. This difference in speedup could be because of differences in how the code in each environment is compiled.

Figure 5 shows the speedup graph when using multi-core CPU and GPU acceleration on the K20 using the single-core

CPU time as the baseline. Figure 5a shows the speedup using a CUDA target and Figure 5b shows the speedup using an OpenCL target on the K20. The speedup for all GPU implementations on the K20 is over 300 times over the sequential implementation and over 35 times over the OpenMP implementation when running more than 10,000 options, with the hand-written CUDA and OpenCL implementations performing a little better than the directive-based implementations.

3.2 Monte-Carlo

GPU acceleration is next used to accelerate Monte-Carlo for financial computation. The code path from the EquityOption example in QuantLib, which prices a single option using the QMC (Sobol) Monte-Carlo method, is modified to run in parallel on the GPU. The experiments are run using single-precision 32-bit floating-point numbers, 250 timesteps, and a varying number of samples. Each sample is generated and run in parallel on the GPU.

To generate random numbers on the GPU, the cuRAND library [1] is used in the hand-written CUDA implementation, while the Mersenne Twister algorithm code from Nishimura and Matsumoto [16] is adapted to generate random numbers in the directive-based and OpenCL implementations. In addition, the Thrust library [4] is used in the hand-written CUDA implementation to sum the output data on the GPU and retrieve the average output of all samples. This summation is performed on the CPU in the other implementations.

Figures 3a and 3b show the speedup graph when using multi-core CPU and GPU acceleration on this application when targeting the C2050 with CUDA and OpenCL code targets, respectively; the single-core CPU time is used as the baseline. This speedup corresponds to the computation of the output value for each sample and does not include the summation and averaging of the outputs. All GPU results using over 5,000 samples show a speedup of at least 75 times over the single-core CPU implementation and at least 10 times over the multi-core CPU implementation. HMPP and OpenACC implementations give a speedup of around 80 times compared to sequential CPU when targeting CUDA and a speedup of around 95 times when targeting OpenCL. Directive-based acceleration targeting OpenCL performed better than manual OpenCL for problem sizes greater than 5,000 by 17% on average. Hand-written CUDA outperformed HMPP and OpenACC in most experimental runs, often by a large factor. A possible explanation is the use of the cuRAND library for random number generation in the hand-written CUDA implementation.

Figures 6a and 6b show the speedup graph when using multi-core CPU and GPU acceleration for Monte-Carlo on the K20 with CUDA and OpenCL code targets, respectively, with the single-core CPU time used as the baseline. All K20-accelerated results using at least 2,000 samples show a speedup of at least 145 times over the single-core CPU implementation and at least 18 times over the multi-core CPU implementation. Notably, there is a speedup of over 1,000 times compared to the sequential implementation when running the hand-written CUDA implementation with 50,000 samples.

The only time when the hand-written CUDA implementation does not perform as well as the OpenACC and HMPP implementations is when the number of samples is set to 500,000; there is a sudden drop-off in the performance of the implementation at this problem size. The CUDA profiler [2] is used to analyze kernel execution details, and the output indicates that cache misses during the random number generation component of the application caused the decrease in speedup with a greater number of samples.

An extension of this implementation allows the parallel pricing of multiple options. This is particularly useful when there are not enough samples in each option to fully take advantage of the massive parallelism available on the GPU.

3.3 Bonds

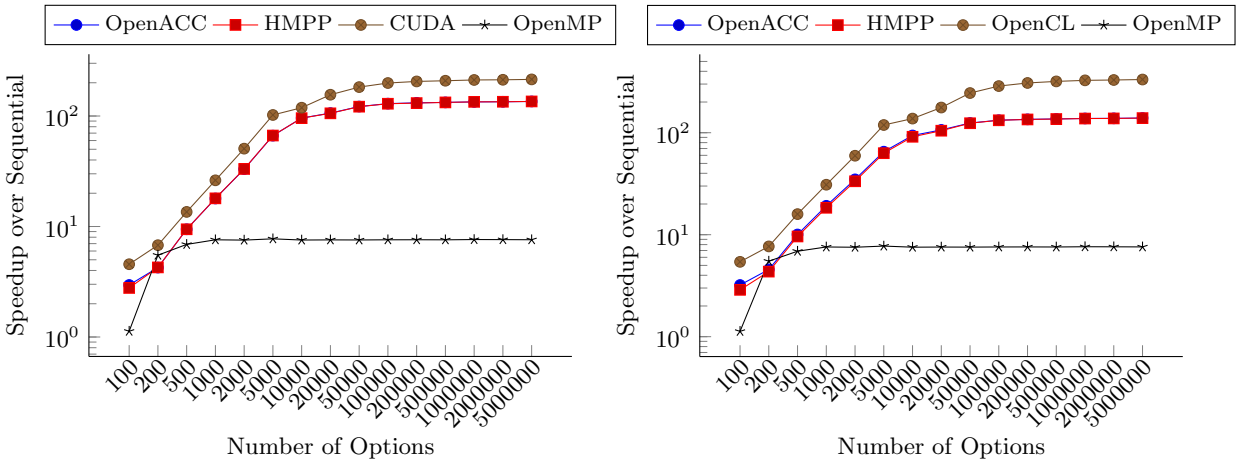
Bonds is the next QuantLib application for GPU acceleration. A bond is a form of loan between an issuer and a holder. The bond issuer is obligated to pay the holder interest (called the coupon) at particular intervals and/or pay back the principle at the maturity date in the future. A bond's yield curve refers to the relation between the bond yield and maturity date. The QuantLib library contains a number of bond-related computations including zero-coupon, fixed-rate, and floating-rate bonds as well as flat, depo-bond, and depo-swap curves. A greater precision is needed for computations involving small decimal values in the application. The necessity for greater precision requires the use of double-precision 64-bit floating-point numbers.

The following experiments show the acceleration of a fixed-rate bond with a flat forward curve using the GPU. The issue date, maturity date, and coupon rate for the bond are varied across experiments, therefore allowing a number of bond configurations to be computed simultaneously. This implementation could also be used to compute the accrued interest at multiple dates. The results using manual CUDA as well as HMPP and OpenACC using a CUDA target on the C2050 and K20 GPU are shown in Figures 4a and 7a, respectively. These results show a speedup of over 40 times compared to the sequential CPU implementation on the C2050 and a speedup of over 80 times on the K20, as well as a speedup over the OpenMP implementation. This speedup is significant but less than the speedups for Black-Scholes and Monte-Carlo. Profiling this application using the NVIDIA Visual Profiler shows that the computation kernel has a lot of divergent branches during computation and a large number of loads from and stores to slower local memory. These characteristics can significantly increase the run-time of a GPU kernel and are likely explanations for the lesser speedup.

Unfortunately, the directive-based codes did not run as expected using an OpenCL target, returning a 'build program failure' error when using HMPP Workbench 3.2.1 while the same input code with a CUDA target worked correctly.

3.4 Repo

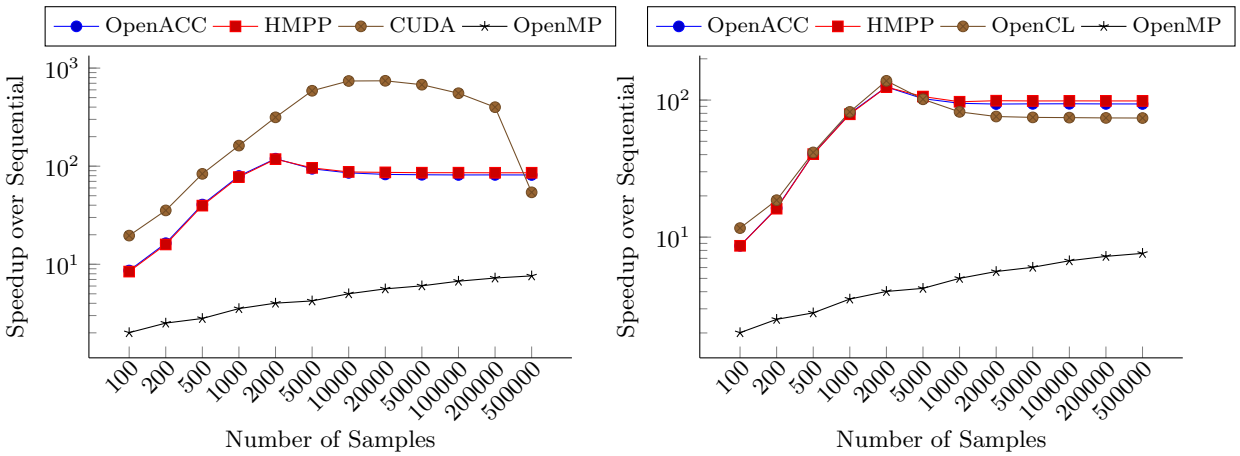
The last financial application for GPU acceleration is the Repo example in the QuantLib library. A repo is a repurchase agreement, specifically the sale of securities with an agreement that the seller will later buy the securities back at a greater price. The difference between the repurchase price and the sale price represents interest and is called the repo rate. The experiments look at the results of varying the repo purchase date, repo sale date, the repo rate, underlying fixed-rate bond rate, and underlying fixed-rate bond date. These variables affect whether or not the buyer and/or seller gains or loses money as a result of the agreement. Similarly



(a) Speedup of Black-Scholes with CUDA target

(b) Speedup of Black-Scholes with OpenCL target

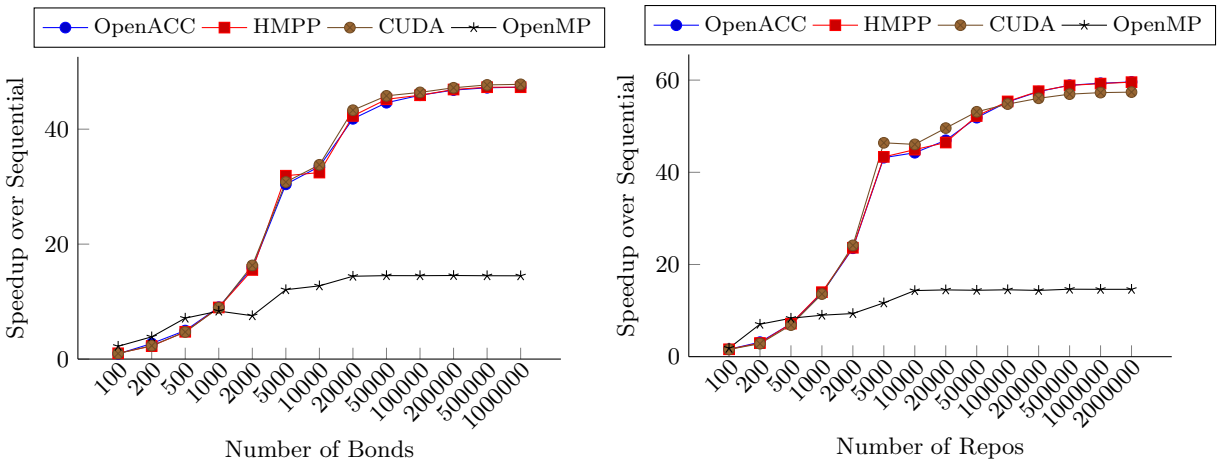
Figure 2: Graphs showing Black-Scholes Speedup on NVIDIA C2050 with Intel Xeon X5530 (8 core OpenMP) for comparison.



(a) Speedup of Monte-Carlo with CUDA target

(b) Speedup of Monte-Carlo with OpenCL target

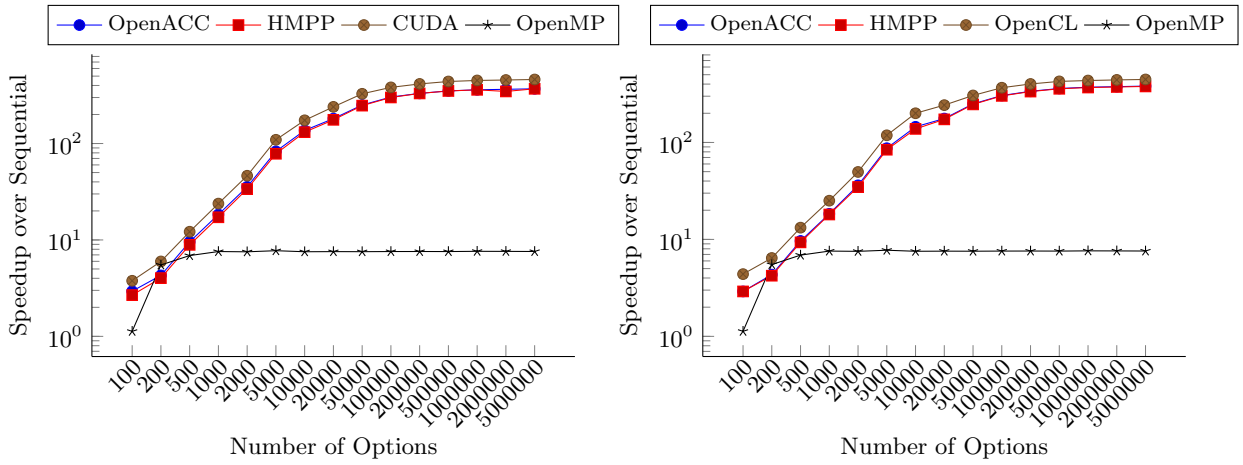
Figure 3: Graphs showing Monte-Carlo Speedup on NVIDIA C2050 with Intel Xeon X5530 (8 core OpenMP) for comparison.



(a) Speedup of Bonds with CUDA target

(b) Speedup of Repo with CUDA target

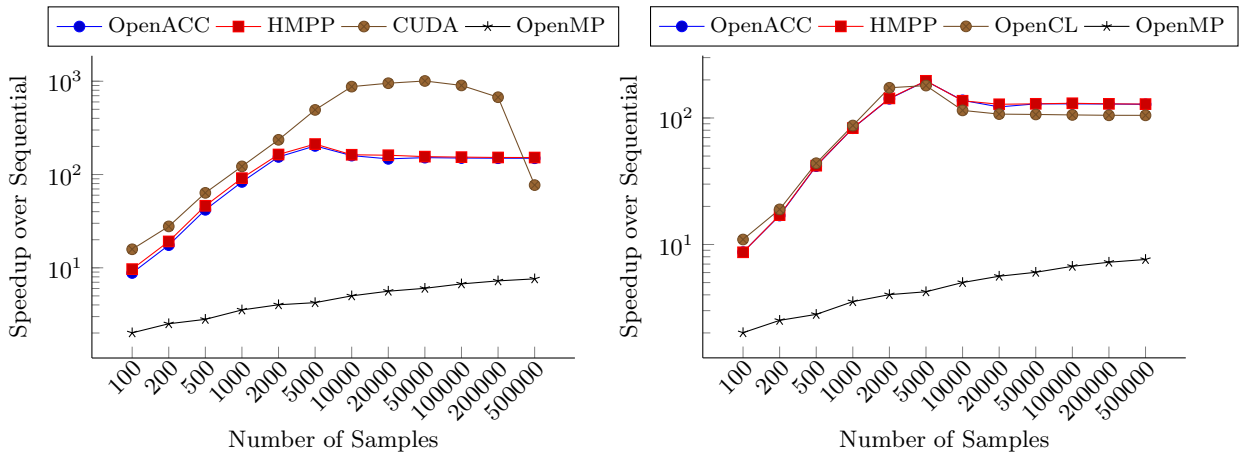
Figure 4: Graphs showing Bonds and Repo Speedup on NVIDIA C2050 using CUDA target with Intel Xeon X5530 (8 core OpenMP) for comparison.



(a) Speedup of Black-Scholes with CUDA target

(b) Speedup of Black-Scholes with OpenCL target

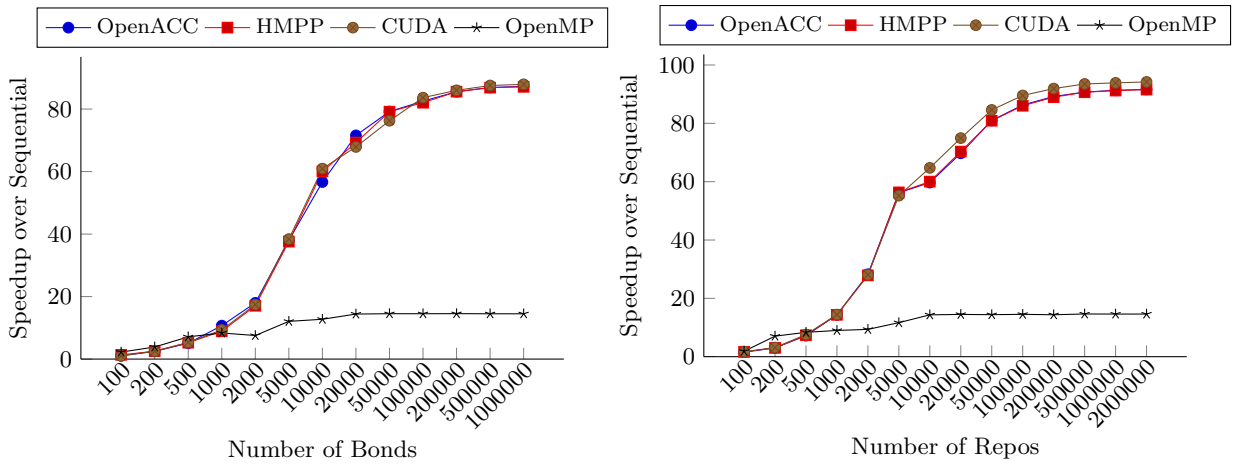
Figure 5: Graphs showing Black-Scholes Speedup on NVIDIA K20 with Intel Xeon X5530 (8 core OpenMP) for comparison.



(a) Speedup of Monte-Carlo with CUDA target

(b) Speedup of Monte-Carlo with OpenCL target

Figure 6: Graphs showing Monte-Carlo Speedup on NVIDIA K20 with Intel Xeon X5530 (8 core OpenMP) for comparison.



(a) Speedup of Bonds with CUDA target

(b) Speedup of Repo with CUDA target

Figure 7: Graphs showing Bonds and Repo Speedup on NVIDIA K20 using CUDA target with Intel Xeon X5530 (8 core OpenMP) for comparison.

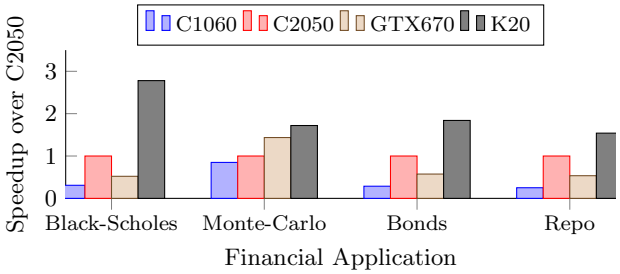


Figure 8: Default HMPP CUDA speedup of financial applications on C1060 (Tesla architecture), C2050 (Fermi), GTX 670 (GK104 Kepler), and K20 (GK110 Kepler) with C2050 as the baseline

to Bonds, Repo requires the use of double-precision 64-bit floating-point numbers because of the need for precise computations using decimal values near zero.

The speedup results using the C2050 and K20 GPUs with the CUDA target compared to the sequential CPU implementation are in Figures 4b and 7b alongside multi-core CPU results. The graphs show that both the hand-written and directive-based GPU implementations are over 50 and 80 times faster than the single-core CPU results on the C2050 and K20 GPUs, respectively, when running at least 50,000 repos in parallel. The GPU target codes are also faster than the multi-core CPU results when running over 1000 repo computations. We were not able to generate OpenCL results due to the same error during compilation as Bonds (described in previous section) when compiling/running the directive-based codes.

4. RESULTS ON OTHER ARCHITECTURES

To compare performance across GPU architectures, the HMPP implementation of each financial application using the CUDA target and a particular input size is run on multiple NVIDIA GPUs. Specifically, each application is run on the Fermi-based C2050 [19], which has 448 cores; the Tesla-based C1060 [18], which has 240 cores; the GK104 Kepler-based GTX 670 [20], which has 1,344 cores; and the GK110 Kepler-based K20 [21], which has 2,496 cores. The financial applications are run on each GPU using 5,000,000 options for Black-Scholes, 400,000 samples for Monte-Carlo, 1,000,000 bonds for Bonds, and 1,000,000 repos for Repo. The experiments for the C1060, C2050, and GTX 670 are compiled and run using CUDA 4.2, while the K20 experiments use CUDA 5.0 because the architecture does not support previous CUDA releases.

The results are shown in Figure 8 in terms of speedup/slowdown using the C2050 results as a baseline. The K20 has the best results for each application, with the performance increase over the C2050 ranging from 1.54 times on Repo to 2.78 times on Black-Scholes. Surprisingly, the older C2050 with less cores than the GTX 670 outperforms the GTX 670 on Black-Scholes, Bonds, and Repo, while the GTX 670 performs better than the C2050 on Monte-Carlo. One possible explanation is that the GTX 670 is not as optimized for GPGPU computing as the C2050, as the K20 is the processor in the Kepler family that is most intended for GPGPU computing.

Pragma	Experimental Parameter Values in Codes
blocksize	Thread block dimensions 32x2, 32x4 (default), 32x6, 32x8, 32x12, 32x16, 16x16
unroll	Unroll factors 1 through 8, 16, 32 using 'contiguous' and 'split' options.
tile	Tiling factors 1 through 8, 16, 32.
remainder / guarded	Used each option with loop unrolling. 'Remainder' option allows generation of remainder loop. The 'guarded' option avoids this via guards in unrolled loop.

Table 2: Transformations applied to each input kernel for optimization.

5. HMPP AUTO-TUNING

Next, experiments are performed on each application using auto-tuning with HMPP on CUDA, first on the C2050 and then on alternate architectures. In particular, block size and unrolling/tiling optimizations provided in HMPP are applied to each HMPP run with a CUDA target. These experiments focus on adjusting the thread block shape, as well as unrolling and tiling transformations on loop(s) in the application code. The set of transformations and the experiment parameters for the experiments described in this work are shown in Table 2 and the best results for each application is shown in Table 3. The auto-tuning framework here is similar to the framework described by Grauer-Gray et al. in [11].

With Black-Scholes using 5,000,000 options, the loop optimizations are performed on the 'main' parallelized loop. The best results on the C2050 show that we are able to speed up the performance by 1.01 times when using thread block dimensions of 32x6 rather than the HMPP default of 32x4 with no tiling or unrolling on the main loop. A likely reason for this speedup is that the change to 32x6 thread block dimensions increased the multiprocessor occupancy of the computation kernel from 0.667 to 0.875. While the speedup is not too significant on this application with the given parameters on this GPU, this general framework provides flexibility to optimize on any HMPP-supported architecture with a number of input parameters. Some configurations are likely to give a greater speedup.

In the Monte-Carlo application, auto-tuning is performed using 400,000 samples and optimizations are performed on two loops: the 'main' loop as well as the loop of the 250-step 'path' of the Monte-Carlo computation. The best results on the C2050 give a speedup of 1.72 times over the default HMPP configuration for Monte-Carlo. This optimal result occurs when using thread block dimensions of 32x2 and tiling the main (parallelized) loop with a factor of 8. Inspection of the optimized output code shows that tiling the main loop changes the parallel computation space from a 1-D row to a 2-D grid, which seems to contribute to a significant speedup in this application. Specifically, output from performance counters via the CUDA profiler reveals that the hit rate of the L2 cache for read requests from L1 cache is 65.7 percent in the optimized configuration and 6.17 percent for the default configuration, resulting in more accesses to slower DRAM in the default configuration.

For the Bonds application, the loop optimizations are performed on the 'main' parallelized loop and the results on

App	Size	GPU	Best Auto-Tuned Configuration		Speedup over default
			Thread Block Dim.	Loop Tiling/Unroll Parameters	
Black-Scholes	5,000,000 options	C1060	16x16	Unroll 'main' loop w/ factor 6 using 'contiguous' and 'guarded' options	1.24
		C2050	32x6	No tiling/loop unrolling	1.01
		GTX 670	32x2	Unroll 'main' loop w/ factor 6 using 'contiguous' and 'remainder' options	1.17
		K20	32x4	No tiling/loop unrolling	1.00
Monte-Carlo	400,000 samples	C1060	32x4	Unroll 'main' loop w/ factor 5 using 'split' and 'remainder' options	1.02
		C2050	32x2	Tile 'main' loop w/ factor 8	1.72
		GTX 670	32x2	Unroll 'main' loop w/ factor 3 and 'path' loop w/ factor 4, both with 'contiguous' and 'guarded' options	1.14
		K20	32x2	Tile 'main' loop w/ factor 8 and 'path' loop w/ factor 2	1.74
Bonds	1,000,000 bonds	C1060	16x16	No tiling/loop unrolling	1.02
		C2050	32x2	No tiling/loop unrolling	1.03
		GTX 670	32x2	No tiling/loop unrolling	1.01
		K20	32x2	No tiling/loop unrolling	1.03
Repo	1,000,000 repos	C1060	16x16	Tile inner 'cash flows' loop w/ factor 2	1.07
		C2050	32x2	No tiling/loop unrolling	1.02
		GTX 670	32x2	Unroll inner 'cash flows' loop w/ factor 2 using 'contiguous' and 'remainder' options	1.01
		K20	32x2	Unroll inner 'cash flows' loop w/ factor 2 using 'split' and 'guarded' options	1.07

Table 3: CUDA results using auto-tuning on HMPP on NVIDIA C1060 (Tesla), C2050 (Fermi), GTX 670 (GK104 Kepler), and K20 (GK110 Kepler) architectures.

the C2050 show a speedup of 1.03 times over the default HMPP configuration when running 1,000,000 bonds. The best speedup occurs when using a thread block size of 32x2 without any tiling or loop unrolling.

Finally, auto-tuning experiments are performed on the Repo application with input size 1,000,000 using loop optimizations on the 'main' parallelized loop as well as an inner kernel loop that sets the cash flows associated with the current repo. The results show a speedup of 1.02 times over the default on the C2050 when using thread block dimensions of 32x2.

5.1 Auto-Tuning Alternate Architectures

The next experiments involved auto-tuning each financial application on alternate NVIDIA GPUs in addition to the C2050, specifically the Tesla-based C1060, GK104 Kepler-based GTX 670, and GK110 Kepler-based K20. The best configuration and speedup of each alternate architecture is shown alongside the C2050 results in Table 3.

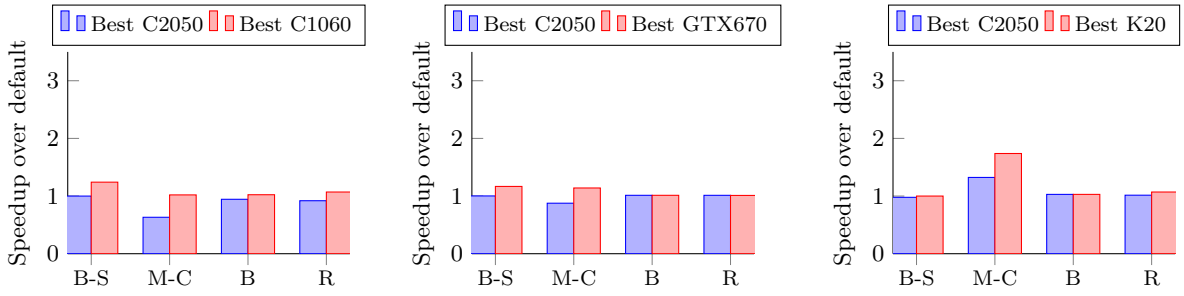
Notably, the results show that the speedup using auto-tuning on Black-Scholes is over 1.15 times on the GTX 670 and C1060, which is much higher than the 1.01 speedup factor on the C2050 in the initial results. The speedup when auto-tuning Monte-Carlo is over 1.70 times on the C2050 and K20 and over 1.10 times on the GTX 670, but the speedup on the same application is only 1.02 times on the C1060. The speedup for Bonds is the highest on the C2050 and K20. Finally, the best found speedup for Repo on the C1060 and K20 is 1.07 times, higher than the best auto-tuning speedup for the application on the C2050 and GTX 670. These results show that the effects of auto-tuning can differ across varying architectures from the same vendor.

The speedup of the best C2050 optimizations when used on the other architectures are shown alongside the best auto-tuned speedup on the architectures in Figures 9a, 9b, and 9c, and the best K20 optimizations when used on the other architectures are shown alongside the best auto-tuned speedup on the architectures in Figures 10a, 10b, and 10c. The results show that the best optimization configuration on a particular architecture may not result in the best possible speedup on another architecture. For example, using the best C2050 or K20 configuration for Monte-Carlo on the C1060 results in a significant slowdown compared to the default while the best auto-tuned C1060 configuration results in a small speedup, using best C2050 configuration for Repo on the K20 results in a small speedup of less than 2 percent while the best auto-tuned configuration gives a larger speedup of around 7 percent, and using the best K20 configuration for repo on the C2050 gives a slowdown of almost 10 percent while the best auto-tuned configuration gives a small speedup. The best auto-tuned configuration is likely to vary across architectures, i.e., it is likely to be specific to a particular architecture.

6. CODE ANALYSIS

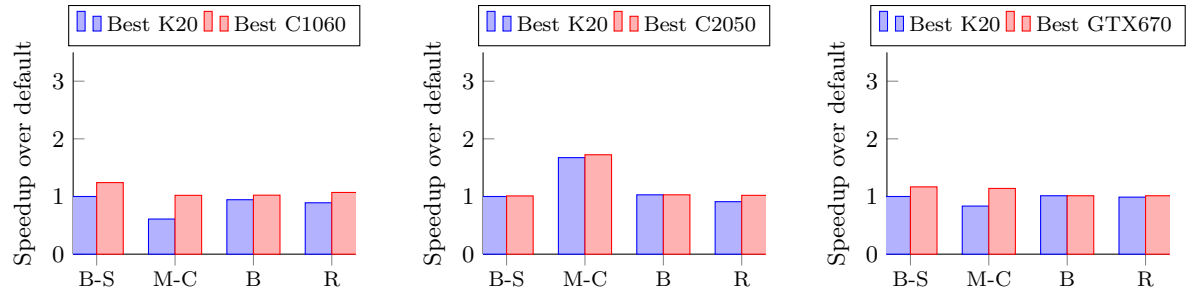
The process of porting the original QuantLib code to different architectures produces many different versions of the same application. In total, six versions of code are created for each application: CPU, CUDA, OpenCL, HMPP, OpenACC, and OpenMP. Table 4 shows the code size in terms of line count for each application and target. The count corresponds to the number of lines for application execution and any data transfer that occurs between the device and host. Input data generation and output printing are not included.

Targets employing pragma-based directives have the lowest line counts compared to the CPU target. Applications with the CUDA target have more lines due to data transfer between the host and device. Applications with the OpenCL target have a significant increase in line count due to the initialization of OpenCL, initialization of the kernels, and



(a) C1060 Speedups (b) GTX 670 Speedups (c) K20 Speedups

Figure 9: Speedup of best C2050 (Fermi) auto-tuned configuration for each application run on C1060 (Tesla), GTX 670 (GK104 Kepler), and K20 (GK110 Kepler) compared to best auto-tuned configuration for each architecture.



(a) C1060 Speedups (b) C2050 Speedups (c) GTX 670 Speedups

Figure 10: Speedups of best K20 (GK110 Kepler) auto-tuned configuration for each application run on C1060 (Tesla), C2050 (Fermi), and GTX 670 (GK104 Kepler) compared to best auto-tuned configuration for each architecture.

Type	Black-Scholes	Monte-Carlo	Bonds	Repo
CPU	208	152	1243	949
CUDA	219	172	1289	1009
OpenCL	313	291	1344	1150
HMPP	216	163	1251	1001
OpenACC	213	156	1249	995
OpenMP	212	156	1246	952

Table 4: Code size, measured as number of lines, for targeted architectures and financial applications

transferring data between the host and device.

6.1 Serial CPU and OpenMP Implementations

Because QuantLib is highly object-oriented, a debugger is used to step through the code paths of each application and see what lines of QuantLib code are executed for each application. Code flattening is manually applied to convert the QuantLib code path corresponding to each application to lower-level C code. Results from QuantLib are compared to the lower-level code to confirm the same results within a minimal epsilon value.

The other target codes are created from this serial CPU implementation. For the OpenMP implementation, a parallel region specifying the number of threads is added as well as a `parallel for` directive.

6.2 CUDA Implementation

The CUDA code is created from the C-based CPU implementation. Additional lines of code are needed for device memory allocation, freeing of memory, device/host memory transfer, and synchronization. In particular, code must be added for each array allocated on the GPU due to device memory allocation, device memory deallocation, and possible host-to-device/device-to-host transfer. In addition, a CUDA thread synchronization call is often necessary after a kernel call.

6.3 OpenCL Implementation

The OpenCL code is created from the C-based CPU implementation using the developed CUDA code as a point-of-reference. OpenCL requires additional setup work to configure the OpenCL target, load the OpenCL source file, compile the source file, and report any compilation errors. On top of the mandatory code for each application, the number of additional lines required is dependent on the number of memory objects used for arrays on the GPU and the number of arguments required for the computation kernel(s).

6.4 HMPP and OpenACC Implementations

The directive-based GPU acceleration codes require a minimal addition of code. The C-based CPU code is left nearly identical for both the HMPP and OpenACC targets. HMPP requires that the codelet (the portion of the code that is parallelized and runs on the GPU) be in its own function while OpenACC does not have that limitation. Directives are used to specify the region of the code that is to be run on the GPU, define times to transfer data from the host to the accelerator (and vice versa), and state how to parallelize the data-independent loop(s) in the code.

A limitation of HMPP and OpenACC is that a structure of arrays may not be passed as an argument. The Repo

and Bonds applications are implemented using an array of structures in the original code, but the codelets had to be adjusted to take the input arrays as individual arguments.

7. RELATED WORK

There is a body of related work of GPU acceleration of computational finance. Zhang and Oosterlee [29], Podlozhnyuk [24], Abbas-Turki and Lapeyre [5], Egloff [9], Joshi [12], and Dang et al. [7] look at Black-Scholes pricing on the GPU, while Podlozhnyuk and Harris [25], Tian et al. [28], Rees and Walkenhorst [26], Dixon et al. [8], Pages and Wil-bertz [23], Bernemann et al. [6], and Murakowski et al. [15] look at GPU acceleration of Monte-Carlo for financial computation. Additional related work by Thomas [27] describes acceleration of Monte-Carlo using FPGAs.

This work differs from the related work by using directive-based pragmas to drive GPU acceleration in addition to hand-written CUDA and OpenCL code while most of the previous GPU work focuses on CUDA. In addition, the focus of this work is on accelerating multiple code paths in the QuantLib library while much of the previous work focuses on a single application. These aspects make this work more general as it can be easily modified to run on alternate parallel architectures with the same directives and also expanded to include the acceleration of additional codes in QuantLib.

There is also related work in using directive-based pragmas. Both Lee and Vetter [13] and Grauer-Gray et. al. [11] show results of benchmarks using pragmas with available optimizations to drive parallelization. This work differs because of the focus on financial applications rather than more general benchmarks.

8. CONCLUSIONS

This work looks at GPU acceleration of code paths in the popular, open-source QuantLib library used in computational finance. Manually written CUDA and OpenCL code as well as directive-based HMPP and OpenACC languages are used to drive GPU acceleration for code paths in QuantLib.

The results show acceleration of Black-Scholes, Monte-Carlo, Bonds, and Repo codes from QuantLib. Results indicate significant speedups using the GPU with each experimental method. Subsequent experiments look at the speed-ups across different NVIDIA GPU architectures and investigate auto-tuning on HMPP-generated code. Particular configurations from auto-tuning show increased speedup as compared to the default configuration. We plan to continue this work by accelerating additional QuantLib codes using the GPU and other parallel architectures that support directive-driven GPU acceleration.

9. ACKNOWLEDGMENTS

This work was funded in part by JPMorgan Chase as part of the Global Enterprise Technology (GET) Collaboration.

10. REFERENCES

[1] cuRand. [Online]. http://developer.nvidia.com/cuRAND.

[2] Profiler User's Guide. [Online]. http://docs.nvidia.com/cuda/profiler-users-guide.

[3] Quantlib home page. [Online]. http://quantlib.org.

[4] Thrust. [Online]. http://thrust.github.com.

[5] L. Abbas-Turki and B. Lapeyre. American options pricing on multi-core graphic cards. In *Business Intelligence and Financial Engineering, 2009. BIFE '09. International Conference on*, pages 307–311, july 2009.

[6] A. Bernemann, R. Schreyer, and K. Spanderen. Pricing structured equity products on gpus. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–7, nov. 2010.

[7] D. M. Dang, C. C. Christara, and K. R. Jackson. Pricing multi-asset american options on graphics processing units using a pde approach. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–8, nov. 2010.

[8] B. T. C. J. Dixon, Matthew F. and K. Keutzer. Monte carlo-based financial market value-at-risk estimation on gpus. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[9] D. Egloff. Pricing financial derivatives with high performance finite difference solvers on gpus. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[10] C. Entreprise. *HMPP Basics - Introduction to Concepts*. CAPS Entreprise, 2012.

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. *2012 Innovative Parallel Computing: Foundations And Applications of GPU, Manycore, and Heterogeneous Systems (InPar2012)*, 2012.

[12] M. S. Joshi. Graphical asian options. In *Wilmott J.*, volume 2, pages 97–107, 2010.

[13] S. Lee and J. S. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 23:1–23:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[14] A. Munshi, editor. *The OpenCL Specification*. Khronos OpenCL Working Group, 2011.

[15] D. Murakowski, W. Brouwer, and V. Natoli. Cuda implementation of barrier option valuation with jump-diffusion process and brownian bridge. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–4, nov. 2010.

[16] T. Nishimura and M. Matsumoto. A c-program for mt19937. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c`.

[17] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[18] NVIDIA. Nvidia geforce gtx 200 gpu architectural overview, second-generation unified gpu architecture for visual computing. Technical report, May 2008.

[19] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. Technical report, 2009.

[20] NVIDIA. Gtx 680 kepler whitepaper - geforce.

Technical report, 2012.

[21] NVIDIA. Nvidia's next generation compute architecture: Kepler gk110. Technical report, 2012.

[22] OpenACC. *The OpenACC Application Programming Interface*. OpenACC-Standard.org, 2011.

[23] G. Pages and B. Wilbertz. Parallel implementation of quantization methods for the valuation of swing options on gpgpu. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–5, nov. 2010.

[24] V. Podlozhnyuk. Black-scholes option pricing. In *CUDA SDK*, 2007.

[25] V. Podlozhnyuk and M. Harris. Monte carlo option pricing. In *CUDA SDK*, 2008.

[26] S. J. Rees and J. Walkenhorst. Large-scale credit risk loss simulation. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[27] D. B. Thomas. Acceleration of financial monte-carlo simulations using fpgas. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–6, nov. 2010.

[28] Y. Tian, Z. Zhu, F. C. Klebaner, and K. Hamza. Option pricing with the sabr model on the gpu. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–8, nov. 2010.

[29] B. Zhang and C. Oosterlee. Option pricing with cos method on graphics processing units. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, may 2009.