Massively Parallelized Monte Carlo Simulation and its Applications in Finance

by:

Ashkan Ziabakhshdeylami
BASC in Electronic Engineering - Simon Fraser University

Lauren Looi
BBusSc in Finance - Drexel University

PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF FINANCIAL RISK MANAGEMENT

In the
Faculty
of
Business Administration

Financial Risk Management Program

SIMON FRASER UNIVERSITY

Approval

Name: Ashkan Ziabakhshdeylami
Lauren Looi

Degree:

Title of Project: Massively parallelized Monte Carlo simulation and its applications in finance

Supervisory Committee

Dr. Andrey Pavlov
Associate Professor
Academic Chair, Master of Financial Risk Management

Dr. Dave Peterson
Chief Financial Engineer
Markit

Date Approved

ABSTRACT

In this paper, we propose, develop and implement a tool that increases the computational speed of exotic derivatives pricing at a fraction of the cost of traditional methods. Our paper focuses on investigating the computing efficiencies of GPU systems. We utilize the GPU's natural parallelization capabilities to price financial instruments. We outline our implementation, solutions to practical complications arising during implementation and how much faster GPU systems are. Each step that we explore has a significant impact on the efficiency and performance of GPU pricing. Rather than speaking in theoretical, abstract terms, we detail each step in an attempt to give the reader a clear sense of what's going on.

Efficiency is one of the pillars of financial calculations. With the volume of risk calculations mandated by prudent risk management practices, even moderate improvements in calculation efficiency can translate into material changes in trading limits or savings in regulatory capital. This can make the difference between a growing, successful trading operation or an also-ran.

Unfortunately, a decent algorithm written in VBA cannot calculate option prices at the same speed as a farm of computers, particularly if we must price the trade in less than 150 milliseconds using 10 million simulation paths.

Fast forward from one trade to a book of several hundred thousand trades, many of which are exotic products. Not only is it necessary to price each trade, but we must do so in each of thousands of different market scenarios in order to calculate even basic risk measures such as Greeks and Value-at-Risk (VaR). At the end of the paper, we discuss how GPUs are currently used in the industry and their various advantages, including cost, time, accuracy and calculation frequency. In addition, we discuss the implementation challenges of GPU systems and the attention to detail that is required for memory allocation.

Keywords: GPGPU, Derivative Pricing, Monte Carlo Simulation, High Frequency trading

ACKNOWLEDGEMENTS

Special thanks to Dr. Dave Peterson, who gave us the opportunity to explore what we are passionate about in a meaningful way. Thank you.

Many thanks to Dr. Andrey Pavlov, for his continued support and his encouragements.

Many thanks to Dr. Anton Theunissen, for continuously pushing us to expand our horizons and explore our financial creativity.

We are grateful to QuIC Financial Technologies for providing us with our test platforms.

Table Of Contents

Abbreviations

CUDA	Compute Unified Device Architecture
LCG	Linear Congruential Generator
MC	Monte Carlo
MT	Mersenne Twister
PFE	Potential future exposure
RNG	Random Number Generator
SIMT	Single-Instruction, Multiple-Thread
SM	Streaming Multiprocessors
VaR	Value-at-Risk
ZCB	Zero Coupon Bond

1 INTRODUCTION

We developed a tool that calculates the pricing of certain financial instruments on GPUs, thus utilizing performance advantages over standard CPUs. This translates into other key advantages, such as improvements in accuracy. Computational efficiency is such a significant concern for portfolio-level credit risk measures that certain liberties are often taken with the details of the derivative pricing models employed. However, attempts to achieve greater model realism bear a substantial cost in terms of computing time. Consequently, simpler pricing models that do not capture all relevant market features are typically used instead. The resulting "broad brush" results are deemed sufficient, recognizing the steep performance versus realism trade-off. However, recent developments in high performance computing technology allow for computationally efficient implementations of realistic pricing models. The use of such models would lead to greater confidence in the integrity of these portfolio-level risk measures.

Previous studies have focused on pricing basic options via Monte Carlo simulation or very complicated products using a Partial Differential Equation approach. Our focus is to price a slightly more complicated barrier option using Monte Carlo that would be useful for an organization to implement. Our analysis includes a more detailed look in various aspects of GPU computing including the practicality and use of the increased computing speed in the financial industry.

With technology improving at a lightning fast rate, the finance industry is taking advantages of the extra speed for their risk analytics, algorithmic trading and option pricing through GPU (Graphical Processing Unit) programming. Many financial goliaths have explored these technologies through pilot programs with JP Morgan and BNP Paribas' publicly announcing their GPU-based risk-analysis. JP Morgan recently was awarded a prize for innovation in banking technology by the Banker magazine (Turner, 2011).

Our test hardware includes 2 NVidia GPUs: Tesla C2050 and Quadro 2000. Visual Studio 2010 express as our IDE, and programmed the models using C++ to interface with the Compute Unified Device Architecture (CUDA). During our tests, we predominantly utilized the Tesla C2050 card for our calculations. As a baseline test, we used MATLAB, which was programmed on an Intel-based MacBook Pro running a Core i7 processor. The details of the systems used to run the CPU and GPU tests are in Table 1.

TABLE 1 SYSTEM DETAILS

CUDA	MATLAB Tests
Graphics Card 1: Tesla C2050 Graphics Card 2: Quadro 2000	Processor: 2.66 GHz Intel Core i7 RAM: 4 GB

To test the calculation efficiencies produced by GPUs, we reproduced the stochastic pricing, risk calculation and hedging possibilities of two financial products; (1) default risky bonds and (2) barrier bond options. They were tested against MATLAB running on Windows running on an Intel-based MacBook Pro. In addition to the ability to parallelize the calculations, we also achieved mathematical efficiencies through the generation of the random variables for the Monte Carlo simulation in parallel. We tested three different random number generators (RNG) namely Linear Congruential Generator (LCG), XORWOW and Mersenne Twister (MT). Ultimately, the XORWOW method yielded the best results and we thereby simulated our testing using this methodology.

We faced a few thoughtful issues, as the pricing of each of these products requires attention to memory management. Since graphics cards operate differently from regular processors, careful consideration towards planning, allocation and usage of the CUDA is required for the most efficient use of this powerful technology. This paper gives a brief overview of the architecture to better facilitate that understanding of the implementation.

We conclude the paper with discussing how GPUs are currently being used in the industry and its various advantages including cost, time, accuracy and frequency.

2 LITERATURE REVIEW

The speedups of GPUs have been analyzed in various papers. Niramarnsakul, Chongstitvatana and Curtis found that an overall speedup of 900 times was achieved by implementing a Monte-Carlo method to price a European option on an Nvidia GEforce8600GT (Niramarnsakul, Chongstitvatana, & Curtis, 2011). Other work has been done on pricing an American lookback option through a binomial and trinomial lattice which achieved speedups of 101 times (Solomon, Thulasiram, & Thulasiraman, 2010). Binomial trees have also been used to price a convertible bond which achieved speedup factors of 511 – 668 depending on the number of bonds on a 960-core Tesla C1060 (King, Cai, Lu, Wu, Shih, & Chang).

More advanced instrument pricings have been investigated by Dang, Christara and Jackson in pricing of exotic cross-currency interest rate derivatives. However, these instruments were priced using Partial Differential Equations (PDE) and achieved a 30 to 60 times speedup based on using two GPU cards (Dang, Christara, & Jackson, 2011).

Past implementations have mainly been in the pricing of options. However, we were interested in investigating the speedup in pricing of more exotic instruments such as a barrier bond or those that can be immediately useful to an organization. An additional focus was to gain efficiencies in multiple areas, which includes the testing of various random number generators and memory management. We wanted to build instruments that would be practical and of immediate benefit to the industry and have thus also incorporated potential opportunities. This paper aims to give a full perspective of pricing financial instruments on GPUs.

3 Simulation of Default Risky Bond

3.1 Model

3.1.1 Default Risky Bond

Initially we tested the speed of the GPUs by simulating a portfolio of one hundred corporate bonds with the possibility of default. The calculations that we used were fairly simple in nature as a first test to get some rough speed results. With a simple problem, the implementation will be more or less similar on different platforms, yielding a more comparable result across various platforms.

Our goal is to calculate the value of a portfolio consisting of 100 corporate bonds modeled as Zero Coupon bonds (ZCB). The ZCB bond pays $1 if it does not default and $0 if it does default. The probabilities of default within a week are estimated as follows, where x is the number of independent defaults in a week for the whole portfolio:

$$P(x) = \begin{cases} 0.9, & x = 0 \\ 0.1, & x = 1 \\ 0.0, & x \geq 2 \end{cases} \quad (1)$$

We used a Monte Carlo simulation to calculate the 90% Value-at-Risk (VaR) for the portfolio. Each simulation has52 time steps to represent each week in the year. At each step if a bond does default, no other bond can default for that week. The calculations were done in a series of steps:

1. Generate a random number, y, for each period for the corresponding number of simulations
2. Default occurs if $y < 0.1$
3. Count the number of times the bond defaults and arrange the number from smallest to biggest

3.1.2 VaR

Calculation of the VaR of the portfolio at the confidence level, $\alpha = 99\%$, is given by the smallest number, x, such that the probability that the loss, L, in excess of x is not larger than $1 - \alpha$ or 1%.

The VaR result for this calculation is shown in Figure 1, where the 99% VaR is at bond 9 defaulting.

$$VaR_{99\%}(L) = -\inf\{l \in \Re : P(L > l) \leq 1\%\} \quad (2)$$

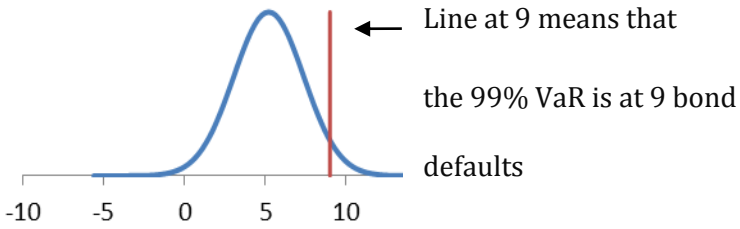

FIGURE 1PORTFOLIO OF DEFAULT RISKY BONDS VAR REPRESENTATION

3.1.3 DERIVATIVE ON A PORTFOLIO OF THESE BONDS

We'd like to also calculate the price of a derivative with the following payment structure:

$$D(T,X) = \begin{cases} \frac{e^X}{500}, & if\ X \geq 5 \\ 0, & otherwise \end{cases} \quad (3)$$

Where:

X: number of defaults during the year

We value this option by taking the present value of its expected payoff under the risk neutral measure, with risk free rate at 5%

$$D(t,X) = e^{r_f(T-t)}E[D(T,X)], \qquad r_f = 0.05 \quad (4)$$

3.2 MC

Monte Carlo simulation is the natural choice for this given problem. We can simulate a portfolio of bonds, and record the end result in a result grid. The Matlab implementation of this is given as in Table 2.

TABLE 2 BOND PORTFOLIO DEFAULT SIMULATION – MATLAB IMPLEMENTATION – SPEED OPTIMIZED

```
clc;clear; format compact; format short;
tic;

numPeriods          = 364/7;
numSim              = 1000000;

%Default simulation
toss= rand(numPeriods,numSim);
toss=toss<.1;
numDefaultsAtT =sum(toss);

%VaR
numDefaultsAtT=sort(numDefaultsAtT);
VaR= (100-numDefaultsAtT(floor(numSim *0.01))) - 90;   % VT-V0

%Derivative Payoff
discountedDerivativePayoffs = exp(-0.05)*exp(numDefaultsAtT)/500 .* (numDefaultsAtT>5);

s=summary(numDefaultsAtT);
d=summary(discountedDerivativePayoffs);
toc;
s
d
```

Note in this implementation that we are sacrificing memory for speed. We could save memory by sacrificing efficiency by simulating the defaults as outlined in Table 3.

TABLE 3 BOND PORTFOLIO DEFAULT SIMULATION – MATLAB IMPLEMENTATION – MEMORY OPTIMIZED

```
%Default simulation
for i=1:numPeriods
    toss= rand(numSim,1);
    toss=toss<.1;
numDefaultsAtT = numDefaultsAtT + toss;
end
```

The second implementation runs 26% slower on our machine. One should note that this sacrifice of efficiency for memory is reversed on the CUDA devices. On CUDA, the lower amount of global memory accessed results in a more efficient code. We discuss this in more details in its corresponding section.

A very simple intuitive parallelization of this code is to simulate each path in parallel to the other paths. We discuss one way to implement this on a GPU in the following section.

3.3 GPU implementation

3.3.1 Implementation

On the abstract level, we need a parallelization mechanism to compute the paths of default and a data container to store the data. One possibility is to use low level kernel implementation, i.e. use CUDA kernels. This technique is very effective and yields the most amount of flexibility. However, one drawback of writing low level code is the need for a larger number of code lines and the associated costs of development. Here for this example, we used the Thrust library instead of writing CUDA kernels. Thrust is a flexible compile time library that provides us with parallelism mechanics and data containers necessary to work with the GPU efficiently. We will employ CUDA kernels in the next section where we simulate and price a barrier bond.

As the authors of Thrust put it, "Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible high-level interface for GPU programming that greatly enhances developer productivity. Develop high-performance applications rapidly with Thrust!" (Jared Hoberock, Thrust Code at the speed of light, 2011)

The general idea is depicted in Figure 2, where the threads are simulating a portfolio of bonds and then saving the portfolio values in a simulation grid in the global memory.

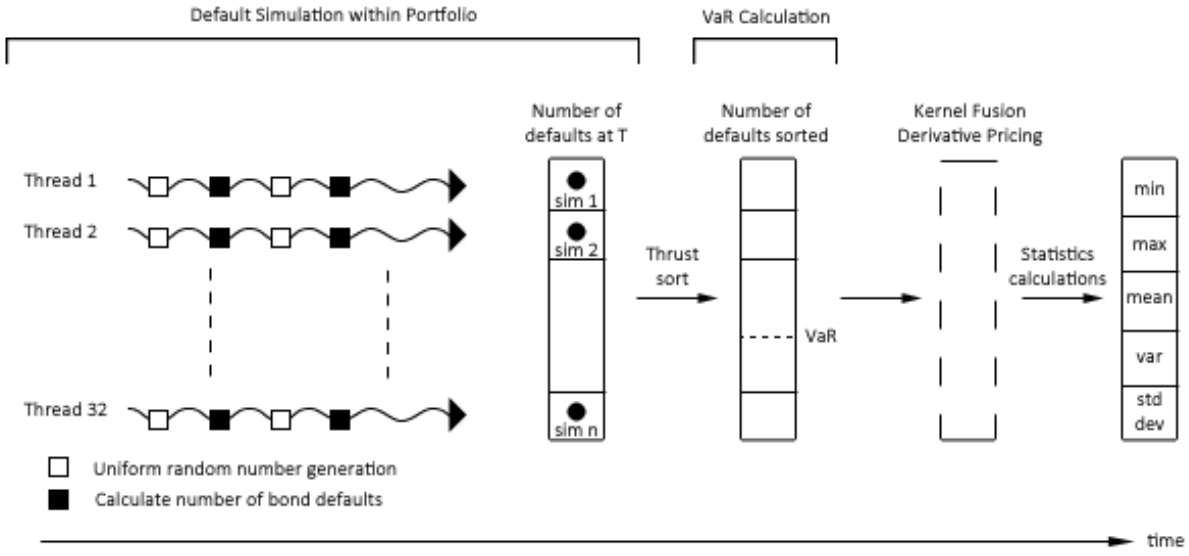

FIGURE 2 MC OVERVIEW – GPU IMPLEMENTATION

Our GPU implementation code might look different to the Matlab code on the surface but rest assured that it is accomplishing similar task. The heart of the code is given in Table 4, as follows:

TABLE 4 BOND PORTFOLIO DEFAULT SIMULATION –CUDA THRUST IMPLEMENTATION

```
thrust::device_vector<int>defaultsAtT(numSim);
printf("\n [%f(ms.)]   Memory Allocated on device",  cpuTime.StopTimer());

thrust::transform(     thrust::counting_iterator<int>(0),      // Start from memory 0
                       thrust::counting_iterator<int>(numSim), // # of simulation paths
                       defaultsAtT.begin() ,   // memory location for storing the result
                       bond_Simulate());       //bond_Simulate() simulate portfolio over its life time

printf("\n [%f(ms.)]   Bond simulation finished.",  cpuTime.StopTimer());


thrust::sort(defaultsAtT.begin(), defaultsAtT.end());
printf("\n [%f(ms.)]   Sorting finished.",  cpuTime.StopTimer());

//VaR
intValueAtRisk=100-defaultsAtT[(int)(numSim*0.01)]-90;            //Value at Risk(VT-V0)


// setup arguments
//Get all types of information on the derivative payoff distribution ( including the mean)
printf("\nStats on derivative payoff:");
stat_summary2( thrust::make_transform_iterator(defaultsAtT.begin() ,derivative_payoffPV()) ,
               thrust::make_transform_iterator(defaultsAtT.end() ,derivative_payoffPV())     );

printf("\n [%f(ms.)]   Computed Stats On DerivativePayoff",  cpuTime.StopTimer());
```

```
// compute summary statistics
printf("\n**DONE** Whole Program: %f(ms.)\n\n",  cpuTime.StopTimer());
```

As it can be seen the code written using Thrust is very readable and modular. We invite the first time reader to first read "An Introduction to Thrust" (Jared Hoberock, An Introduction to Thrust) and get familiarized with concepts such as thrust containers and kernel fusion before reviewing this code.

Thrust uses the bond_Simulate() function, not shown here, to simulate paths in parallel. So at one time, many instances of this function are run in parallel and their results are recorded in the simulation result container, which is a device_vector called "defaultsAtT". The rest of the code sorts the results to calculate the VaR, the derivatives payoff and finally the statistics summary.

Within this code, we have used Thrust kernel fusing concepts to minimize global memory access and to increase efficiency. This advanced technique should be employed whenever possible.

One fundamental issue for MC simulations is the employment of the random number generators (RNG). Here we present our result with different RNG techniques. Later on in this paper, we provide an in-depth analysis of random number generation techniques for MC simulations.

3.3.2 Numerical results

The calculation of the default risky bonds were calculated across two platforms; Matlab and CUDA while the CUDA platform was testing using three types of RNG. As can be seen in Table 5, each platform generated numbers within 4% of each other.

TABLE 5 NUMERICAL RESULTS FROM SIMULATION OF DEFAULT RISKY BONDS

	Matlab	Mersenne Twister	Thrust (Disjoint Seq)	XORWOW
Simulation Paths (#)	1 M	200K	10M	10M
Defaults: (#)				
Minimum	0	0	0	0
Maximum	18	18	19	19
Mean	2.1638	5.20098	5.20011	5.20402
Std	2.1638	2.16448	2.16356	2.16375
VaR ($)	9	9	9	9
Option Price ($)	7.4192	7.06221	7.13648	7.35462
Minimum	0	0	0	0
Maximum	124,915	124,915	339,555	339,555
Mean	7.4192	7.06221	7.13648	7.35462
Std	288.6925	180.613	281.603	297.137
Simulation Time (ms)	1,387.9	316.427	196.406728	119.83
Simulation Time per path (ms)	1.39	1.58	0.02	0.01

3.3.3 Performance Comparison

As the results in Table 5 show, the GPU programs runs faster than the CPU irrelevant of the RNG used, with the exception of MT. One must note that a high quality RNG such as MT takes much longer than the other two RNGs used. Also a random number generator that produces all the numbers from the same seed will take much longer, since it needs to synchronize threads (thread will become sequential more or less), or discard numbers (there is a time penalty involved with higher number of discards).

Writing the Matlab code is much easier for a person without C++ experience. Also another challenge is writing CUDA optimized C++ code. This requires some expertise in the field. Two codes might achieve the same thing, but in a very different way. If efficiency is desired, the coding must be left to the experts in the field.

4 Simulation of Barrier Bond

4.1 Model

Merton models corporate debt as a riskless bond plus a short position in a put option on the firm's assets. Equity, on the other hand, is a call option on the firm's assets. (Merton, 1973)

Assuming that the market value of the firm's assets follows the stochastic process:

$$dA = (\alpha A - \delta)dt + \sigma A\, dZ \tag{5}$$

A put on the assets of the company can then be written as

$$\begin{aligned} &P(A(t), t) = value\ of\ debt\ at\ time\ t \\ &D: face\ value\ of\ debt \\ &P(A(T), T) = \min\{D - A(T)\,, 0\} \end{aligned} \tag{6}$$

An analytical solution is then obtained from the famous Black-Scholes formula:

$$\begin{aligned} &P(A(t), t) = \Phi(-d_2)Ke^{-r(T-t)} - \Phi(-d_1)S \\ &d_1 = \frac{\ln\left(\frac{A(t)}{D}\right) + \left(r - \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}}, \qquad d_2 = d_1 - \sigma\sqrt{T-t} \\ &\Phi \sim \mathcal{N}(0,1), \qquad \sigma = \text{volatility of asset A} \end{aligned} \tag{7}$$

As a side note, this simple example with its analytical solution can be used as a control variate in more complicated problems and for accuracy measurement.

Moving onto a more interesting instrument, let the bond default with recovery value R, at the first timeτ $(0 < \tau \leq T)$ that the values of the assets of the firm $A(t)$ fall below a trigger level L. This barrier bond is what we will be simulating and pricing using Monte Carlo simulations.

4.2 MC

The Matlab code for the simulation of this bond is given in Table 6 as:

TABLE 6 BARRIER BOND PRICING– MATLAB IMPLEMENTATION

```
clc; clear; format short;

%dA = (alpha A - div ) dt + sig * dZ

tic;

% MC Simulation Parameters
N=1000000;
T=2;                  % Time to maturity (All units in Years)
dt = .05;             % Time steps
alpha = .05;          % alpha
sigma = .3;           % volatility of assets
div = 10;             % continuous dividend yield
rf = .05;             % risk free rate
A0 = 150;             %Initial asset price

% Simulation Setup
A = ones(N,1) * A0;

%Bond
PVd=zeros(N,1);         %Present value of the bond for each path
defaultTime = zeros(N,1);
barrier = 100;
faceValueD = 100;
RecoveryPercentage = .8;

% Simulation
for t=0:dt:T
    dA = (alpha * A(:,1)- div ) * dt+ sigma * A(:,1) * sqrt(dt) .* randn(N,1)  ;
    A(:,1) = A(:,1)+dA;

    for n=1:N               %If default occur at this time step record it
        if((A(n,1)< barrier)&&(defaultTime(n,1) == 0))
            defaultTime(n,1) = t;
            PVd(n,1) = RecoveryPercentage * faceValueD * exp( -rf * (T-t));
        end
    end
end

%For the bonds which didn't default:
for n=1:N               %If default occur at this time step record it
    if(defaultTime(n,1) == 0)
        PVd(n,1) = faceValueD * exp( -rf * (T));
```

```
    end
end

% bond price at initial time 0
BondPrice=summary(PVd);
toc;

BondPrice
```

4.3 GPU implementation

Our GPU implementation this time is solely based on CUDA kernels and we do not use Thrust. We have created a Monte Carlo engine that is optimized for running on CUDA. The Monte Carlo engine will simulate the stochastic differential equation (SDE) of the assets of the company using a simulation grid. During this simulation it will probe an object that defines the cash flows of a barrier bond. The cash flows are then discounted back using the risk-free ZCB. This result will be stored in a simulation grid which is a piece of memory on the global memory space of the GPU.

Visually we can represent our implementation as in Figure 3.

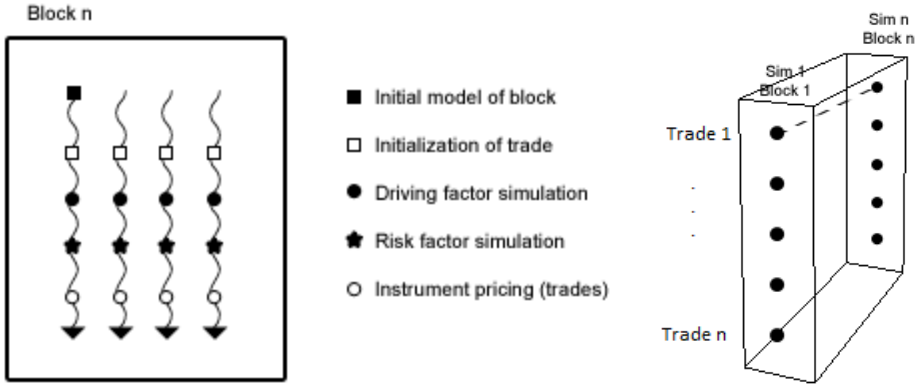

FIGURE 3 MC BARRIER BOND – BARRIER BOND – SIMULATION BLOCK AND SIMULATION GRID

The MC Engine is implemented as a CUDA kernel, represented in Figure 3. The model numbers are copied from the CPU memory into the GPU memory and it is then simulated using the engine.

When executing the kernel, each thread block is responsible for one set of simulations. Within the thread block, each thread is responsible for simulating one instrument.

For each thread block, we load the model from global memory of the GPU into the shared memory of the thread block. All the threads within the block will be sharing that specific simulation path. For each thread, we load its corresponding instrument into thread memory. These memory access optimizations will reduce the execution time of the kernels.

Table 7 shows that the Barrier Bond has a simple implementation in C++.

TABLE 7 BARRIER BOND – GPU IMPLEMENTATION

```
structBarrierBond                                       // An example of an instrument
{
//Contract definitions
        size_triskFactorNum;
        GQDate maturity;
        GQDate start;

        float R;                    //Recovery
        float barrier;              //Barrier
        floatfaceValue;             //faceValue
        floatcouponRate;            //Coupons by the bond if not defaulted.

// Simulation of this contract through time
        BarrierBond()
        {
                bondHasDefaulted=false;
                couponRate=0;
        }

        __device__ inlinefloatdoTimeStep(GQDate t, float A)   //Asset price
        {
                if(bondHasDefaulted==true)//In case of default or after maturity the bond
                                          //doesn't produce any cash flow
                        return0;

                if( t>(maturity+0.0025)  )//Floating point issues, adding .5 of a day is
                                        //the accuracy
                        return0;

                if(gqequalf( maturity, t, 0.001))         // Bond matured
                        returnfaceValue;
```

```
                if( A<barrier )                                         // Bond defaults or not
                {
                        bondHasDefaulted=true;
                        return(faceValue*R);
                }

                if(couponRate==0)                               // Bond doesn't pay coupon
                        return0;

                return(faceValue*couponRate);                   // Bond pays coupon
        };
private:
        bool  bondHasDefaulted;
};
```

The main program, detailed in

```
host_Bonds[i].start     =0;                //starts from 0 day
host_Bonds[i].maturity  =2;                 //300 days from inception

host_Bonds[i].faceValue=100;
host_Bonds[i].barrier   =100;
host_Bonds[i].R         =0.8;               //Recovery amount in case of default

host_Bonds[i].riskFactorNum=1;      //It's a call on the first asset.
```

After this setup, we ran a grid of CUDA kernels responsible for pricing, where lg is the lognormal model of assets, dev_bonds are the bonds to be priced on device, and sim_grid is the memory allocated for the results as show in Table 9.

Table 9 is responsible for extracting and initializing instances of barrier bond.

TABLE 8 BARRIER BOND CONTRACT SETUP

```
host_Bonds[i].start     =0;                //starts from 0 day
host_Bonds[i].maturity  =2;                 //300 days from inception

host_Bonds[i].faceValue=100;
host_Bonds[i].barrier   =100;
host_Bonds[i].R         =0.8;               //Recovery amount in case of default

host_Bonds[i].riskFactorNum=1;      //It's a call on the first asset.
```

After this setup, we ran a grid of CUDA kernels responsible for pricing, where lg is the lognormal model of assets, dev_bonds are the bonds to be priced on device, and sim_grid is the memory allocated for the results as show in Table 9.

TABLE 9 BARRIER BOND KERNEL INVOCATION

```
LognormalMC<<<numSimulations,numBonds>>>(lg, dev_Bonds, sim_grid);
//MC simulation that prices all the Bonds
```

4.3.1 Numerical results

As an example, we simulate assets of one company with the assumptions detailed in Table 10.

TABLE 10 BARRIER BOND MARKET SETUP

```
// Market Setup
LognormalModellg;
lg.alpha        =.05;
lg.sigma        =.3;
lg.delta        =10;            //Annual Continous div yield

lg.A0    =150;
lg.start=0;                   //unit:Years
lg.end=2;                      //unit:Years
lg.dt=0.05;                   //unit:Years
```

Note that, the simulation time goes from 0 to 2, in 0.05 time steps. We value one barrier bond with characteristics given as Table 8.

The numerical result and timing from the Matlab implementation and the GPU are presented in Table 11.

TABLE 11 MATLAB GPU TIMING COMPARISON – BARRIER BOND

1 Trade, 1Million Simulations	Matlab	GPU
Simulation Time (ms)	1,755.081	4,923.64
Calculated Price ($)	84.5910	84.6047
Std Price	7.1026	1.85098
Std/#Simulations	7.1026E-06	1.85098E-06
Simulation time per Bond (ms)	1,755.081	4,923.64

As it can be seen the Matlab code running on a CPU is running faster than the GPU. The underlying reason is that we are not fully utilizing the power of our Tesla GPU when we simulate only one trade. In order to improve this, we can start by pricing multiple trades in parallel. Table 12 summarizes a comparison of simulation time versus trades to be priced with a single run.

TABLE 12 BARRIER BOND GPU TIMING

Num Grids	Sim/Grid	Bonds	Simulation Time(ms)	Simulation Time/Bond
65000	15	**1**	4,923.64	4,923.64
65000	15	**32**	4,981.03	155.66
65000	15	**33**	5,767.60	174.78
65000	15	**128**	8,304.05	64.88
65000	15	**256**	15,927.68	62.21
65000	15	**512**	42,415.55	82.84

4.4 Performance comparison

When we utilize the full power of GPUs and price multiple trades in parallel, we can achieve performances that are far superior to what can be achieved with CPUs. For example, pricing 256 trades in parallel on the GPU, we spend roughly 16seconds. This translates into 62ms per trade. However, per trade execution time is fixed at its best in Matlab. Therefore, pricing 256 trades in Matlab takes roughly 256*1,755.081=449,280ms or 7 minutes.

Pricing 1 trade vs.32 trades practically takes the same amount of time in CUDA. This is explained by the fact that threads are run in warps of 32 parallel threads on the Streaming Multiprocessors(SM). Therefore, running 1 or 32 threads takes practically the same amount of time.

One other note is that the more trades we price, i.e. the more resident blocks we have on the SM, the faster the code runs. Therefore, pricing more trades in parallel is beneficial to us. However, the upper limit of parallel trades to be priced is dictated by the maximum number of threads per thread

block. Also, there is a limit on global memory where the simulate prices are stored. These limits should be taken into consideration when designing an efficient MC engine.

5 CUDA

5.1 Architecture

A simple graphic that compares the CPU and GPU architecture is depicted in Figure 4 (NVIDIA, 2011):

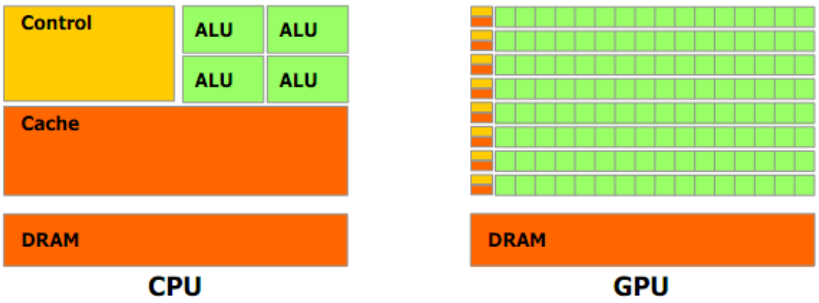

Figure 4 CPU GPU Architecture Comparison (NVIDIA, 2011)

Any CPU or GPU consist of two major parts, one is the processing units (ALUs) and the other one is the Memory Units which are responsible for storing the data. There are different types of Memory Units and there is a speed associated with each type. On a quad-core CPU, there are 4 ALUs, and two types of memory, the DRAM and Cache. The Cache is a smaller but faster memory that is close to the ALUs. Any data processed by the ALU should go through the Cache.

The CUDA is built around a scalable array of multithreaded Streaming Multiprocessors (SMs), something similar to the ALU part of the CPU, but different in detail. A SM is designed to execute hundreds of threads concurrently. The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism. Each SM has its own fast on chip memory (Local memory), which is essential for speed considerations.

5.2 PARALLELISM ON GPU

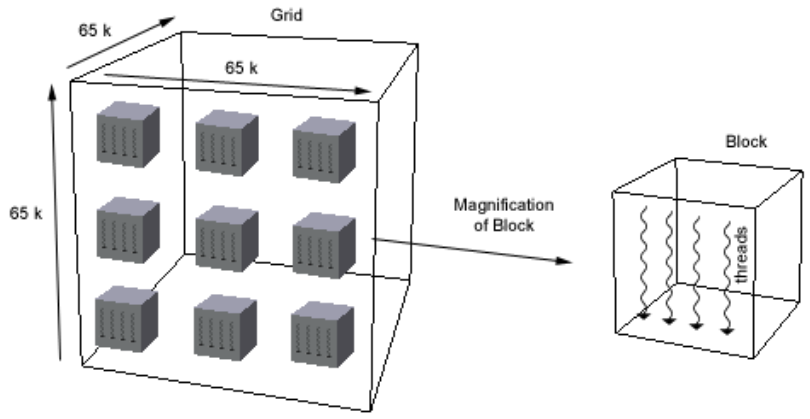

FIGURE 5 PARALLELISM VISUALIZATION IN CUDA

5.2.1 THREADS

In the CUDA world, a thread is the smallest possible unit of work as seen in Figure 5. A collection of threads run in parallel, and collectively aim to solve a problem. Usually, many replications (if not thousands) run in parallel. The function that defines a thread is sometimes called a kernel. So, we run a thread, by invoking its kernel.

One must remember that one kernel is limited to 2 Million instructions. Therefore, a bigger project is divided into different kernels and each kernel accomplishes one task. So, for example one kernel might filter a stock price to get log returns and another kernel executes a regression on the log returns.

5.2.2 WARPS

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads within a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. However, this branching slows the program down and should

be avoided. On the programing level a lower number of if statements in the kernel code translate into less branching.

As a programmer you do not have control on the number of Warps within a thread block, but you do have control over the number of threads within a Block, and number of blocks within a grid.

5.2.3 Blocks

After warps, the next unit of thread grouping is a block, as seen in Figure 5. Threads are grouped into blocks of threads. A Block conceptually organizes the threads into a maximum 3dimensionalspaces(2D space on devices with compute capability < 2). The maximum number of threads in a block is dictated by the CUDA hardware architecture. On the Arch2, there is a maximum of 1024 threads per block. (NVIDIA, 2011).

5.2.4 Grids

A number of thread-blocks are organized into a Grid. A grid conceptually organizes the blocks in a 3 dimensional space. So, each block can be identified by a X,Y,Z address.

5.3 Memory

Memory is usually the bottleneck of GPU programs. Therefore, attention to memory management is probably the most important part of CUDA programming. Table 13 provides a brief type and speed comparison of memories (NVIDIA, 2011).

Table 13 Memory types within CUDA enabled systems

Memory	Scope	Location	Cached	Size(Compute 2)
Registers	One thread	On-chip	No	32K / MP
Local	Thread	Device Memory	No	512 K /Thread
Shared	Thread Block	On-chip	N/A	48K /MP
Global	All Threads + Host	Device Memory	No	
Constant Memory	All Threads + Host	Device Memory	Yes	64 KB /MP
Texture and Surface Memory	All Threads + Host	Device Memory	Yes	

5.4 Precision & Performance

When we dive into the world of numerical calculation, precision becomes an issue. On a very basic level, the data containers and standards used by the hardware become important. Fortunately, CUDA devices like many other processors follow IEEE Standards for Binary Floating-Point Arithmetic (ANSI/IEEE, 1985). Within this standard, a floating point number depending on its accuracy is represented in Table 14.

Table 14 Floating and Double data type bit lengths

	Total Bit Size	sign	exponent	Fraction
float	32 bits	1	8	23
double	64 bits	1	11	52

Let us motivate the issue with an example. We need to do Monte Carlo simulation for 2 years, with time steps of 0.05 (T=5, dt= 0.05). The counter for simulation is t. There is an option that monitors the counter, and whenever its maturity arrives (t= 2), it produces $1. The sudo code for this can be as simple as outlined in Table 15.

Table 15 Option that Never Matures

```
T=2;    t=0    dt=.05;      option =0;
While t< T
{
    If ( t==2)
        Option =1;
    t+=dt
}
```

The problem is that Option will never equal to 1. Underlying reason for this is that 0.05 cannot be represented perfectly in either float or double container. Table 16 shows how it can be represented.

TABLE 16 OPTION THAT NEVER MATURES

```
printf ("\n (double) .05 %.24f,  (float) 0.05  %.24f" , (double) .05 , (float) 0.05 );
```

Output:

(double) 0.05 0.050000000000000003000000 (float) 0.05 0.050000000745058060000000

After 40 iterations depending on the data type used, the result is shown in Table 17.

TABLE 17 THE TIME COUNTER AFTER 40 ITERATION

Float	Double
1.9999991655349731000	2.0000000000000009000

Neither of these equals to 2. To avoid problems as such, one must remember to never use == sign for floating point numbers and always use an error bound range. There are many ways to do so, each of which with some advantages, and disadvantages. Matthias Rupp provides a very short and concise report on different techniques in his paper titled "Comparison of Floating Point Numbers" (Rupp, 2007). Bruce Dawson also has written a very comprehensive article on Comparing floating point numbers (Dawson), where he proposes a fast 2s complement method for comparison of floats.

For our purposes, we used the Relative error bound technique proposed by Donald Knut in the book "The Art Of Computer Programming" (Knuth, 1997). We based our CUDA floating compare on a public C implementation of this by Theodore Belding (Belding, 2009).

One must mention that iterative addition of dt will yield more error than multiplication of dt by a factor. For example, adding dt to t 40 times will produce more error than adding 40*dt to t. For illustrative purposes, our curious reader can run the code illustrated in Table 18 on her platform.

TABLE 18 TESTING OF NUMERICAL DEFICIENCIES OF FLOAT AND DOUBLE

```
template<class T>
voidnumericTesters(size_t iterations, T step)
{
    T a;
    a=0;
    for(size_t n=0;n<iterations;n++)
        a=a+step;
    printf("Iterated answer:%.19f,  Multiplied Answer:%.19f\n", a, iterations*step);
}

int main(intargc, char*argv[])
{
    numericTesters<float>(40, .05);
    numericTesters<double>(40, .05);
}
```

Output:

```
Iterated answer:1.9999991655349731000,  Multiplied Answer:2.0000000298023224000
Iterated answer:2.0000000000000009000,  Multiplied Answer:2.0000000000000000000
```

One might think about this problem, in the world of Matlab or excel. In Matlab, the equality sign is in fact an error bounded if statement rather than a primitive equality check like ==. Essentially, the authors of Matlab have already implemented this for the user, and the problem is concealed to some extent.

Excel solves this issue by showing only certain number of decimal points, and doing the if statement on the same basis. However, if you start adding 0.05 to 0 repetitively, after 73 repetitions or so, the error becomes significant.

5.5 PARALLEL NUMBER GENERATORS

5.5.1 MC MOTIVATION

An essential part to Monte Carlo simulation is generation of random numbers. Starting with an Ito process as:

$$dX = \mu(X,t)dt + \sigma(X,t)dW \qquad (8)$$

We would like to value an option$f(X,t)$. One way to do this is to derive the PDE of $f(X,t)$ and then solve it either analytically or numerically using finite difference techniques. One other way is to write the present value of $f(X,t)$ in terms of expectations, and then generate random numbers for dW, and calculate the expectations for a large number of samples. This is essentially possible through the infamous Feynman-Kac formula, and can be written as:

$$f(x,t) = E\left[e^{-\int_t^T V(X_\tau)d\tau}\psi(X_T)\middle|X_t = x\right] \qquad (9)$$

As Peter Jaeckel puts it "Feynman-Kac connects the solutions of a specific class of partial differential equations to an expectation which establishes the mathematical link between the PDE formulation of the diffusion problems we encounter in finance, and Monte Carlo simulations." (Jackel, 2002)

One way to solve this expectation is to do integration over the sample space. However, in the MC method we draw lots of samples from the sample space, and then do the average on our limited number of realizations. Since this sampling is done randomly, the quality of RNG directly affects the quality of calculated expectation.

Since RNG is a very important topic for scientists, many papers have been written on this topic. Here, the focus of this paper is not the RNG, but since it is fundamentally important for efficiency, timing, quality and pricing of any instrument, we spent considerable time to compare the possibilities. We briefly discuss some of the more famous and useful ones here in this paper.

5.5.2 CUDA AND RNGS

Random number generators fall into two categories: pseudorandom and quasirandom. A pseudorandom sequence is generated by a deterministic algorithm, but it satisfies most of the statistical properties of a truly random sequence. From the tests that have been proposed, BigCrush test is a famous one. CUDA RNG library uses a BigCrushuniform(0,1) test platform (Simard., August 2007). Only a small number of generators pass all of the BigCrush tests. For

example the widely-respected Mersenne twister (Nishimura, January 1998) consistently fails two of the linear complexity tests.

We'll be discussion the following types of RNG:

1. Linear Congruential Generators (LCG)
2. XORWOW
3. Mersenne twister

5.5.3 Linear Congruential Generator

Linear Congruential Generator (LCG) is one of the oldest and most well understood RNG, however there are disadvantages to this methodology, particularly with serial correlation. For example, a point in an n-dimensional space will lie on, at most, $m^{\frac{1}{n}}$ hyperplanes if that point is chosen by an LCG (Marsaglia's Theorem, developed by George Marsaglia) because of serial correlations between successive values of the sequence.

An additional disadvantage is that if m is set to a power of 2, then the lower-order bits of the generated sequence have shorter cycles than the sequence as a whole. The nth least significant digit in the base b representation of the output sequence repeats with at most period b^n, where $b^k = m$ for some integer k. Substituting $2n$ for the modulus term demonstrates the shorter cycles.

This methodology is beneficial when the amount of memory available is limited. However, as mentioned above, the lower-order bits are not reliable when m is set to a power of 2, as it will produce alternatively odd and even results (Kato, 1996).

5.5.4 Mersenne Twister

Makoto Matsumoto and Takuji Nishimura developed the Mersenne Twister (MT) to fix the flaws from older algorithms. It is based on a matrix linear recurrence over a finite binary field F_2 for fast

generation of high-quality pseudorandom numbers. The name originates from the fact that the length of the period is a Mersenne prime (Matsumoto M. a., 1998).

There are two variants of the algorithm, the newer 32-bit word length MT19937 and the older 64-bit word length MT19937-64. They differ only by the size of the Mersenne primes used. The numbers are generated with an almost uniform distribution in the range $[0, 2^k - 1]$.

The most common 32-bit word length MT19937 has the following advantages:

- Very long period $2^{19937} - 1$
- K-distributed to 32-bit accuracy for every $1 \leq k \leq 632$. This means that the following holds for the pseudorandom sequence x_i of 32-bit integers of period P:

Let $trunc_{32}(x)$ denote the number formed by the leading 32 bits of x, and consider P of the $32k$ -bit vectors

$$\left(trunc_{32}(x_i), trunc_{32}(x_{i+1}), \dots, trunc_{32}(x_{i+k-1})\right) \quad (0 \leq i < P)$$

All-zero combinations occurs once less often than each of the 2^{32k} possible combinations of bits.

- Passes most of the statistical randomness tests including Diehard and TestU01 Crush randomness tests.

5.5.5 XORWOW

The xorshift RNG is based off of a repeated use of a simple computer construction: exclusive-or (xor) a shifted version of itself (Marsaglia, 2003). This produces a sequence of $2^{32} - 1, 2^{64} - 1$ or $2^{96} - 1$ sequences for integers x, pairs x, y, or triples x, y, z respectively. The advantages of using such operations with various shifts and arguments are speed, simplicity and the high quality of randomness.

In C, these basic operations can be used:

y^ (y<<a) for shifts to the left

y^ (y>>a) for shifts to the right

The simple C procedure demonstrates the power and effectiveness of xorshift operations. Only three xorshift operations are needed to provide $2^{128} - 1$ random 32-bit integers using four random seeds w, x, y, z:

tmp = (x^ (x<<15)); x = y; y = z; z = w; return w = (w^ (w>>21)) ^ (tmp ^ (tmp>>4));

Marsaglia proposed the XORWOW generator, which has been tested using the TestU01 Crush randomness test in addition to the full suite of NIST pseudorandomness tests. The most rigorous TestU01 test has been the BigCrush, which executes 106 statistical tests over the course of five hours. The XORWOW generator is noted to "pass all of the tests on most runs, but does produce occasional suspect statistics." (NVIDIA, January 2011)

5.5.6 Our Tests

For our purposes we tested 7 techniques of random number generations. In the first three techniques we generate the random numbers on the CPU and transfered them to the GPU. We tested these on the context of valuing a portfolio of default risky bonds. A brief overview is presented in Table 19.

Table 19 Random Number generation timing comparison – On the CPU

	rand()	MT	Thrust min_std
Memory Allocated on device and CPU	92.058467	91.857008	111.313523
Random Numbers generated	714.157580	500.139225	
Random Numbers transferred to device	769.857255	557.298299	415.055440
Bond simulation finished.	905.593321	692.318780	547.274307
Sorting finished.	910.726453	696.417244	552.635485
Computed Stats On Data	910.975817	698.288970	554.005934

* Tests run on Quad Core Intel Xeon CPU 2.40GHz

In the first column of Table 19, the C++ rand() implementation is presented. This implementation is one of the worst ones possible, time wise and quality wise.

The second implementation is the famous Mersenne Twister MT19937 implementation by Makoto Matsumoto and Takuji Nishimura (Matsumoto T. N., 2002). This implementation is probably one of the best and widely used RNGs. It runs faster and produces high quality random numbers.

The third implementation is Thrust's default RNG engine min_std, which produces reasonable quality numbers very fast.

In the next phase, we explored the generation of random numbers in parallel. Going from sequential to parallel has its own advantages and challenges. The biggest challenge is the tracking of states between different threads.

A pseudorandom number generator starts with a seed and creates a predictable sequence of numbers that has statistical properties of numbers coming from a random distribution. As the RNG moves along the sequence, it keeps an internal state that is used to generate the next number. One can either share and update the states between the threads or alternatively start from the same seed in each thread, and discard the first n numbers. There are penalties associated with either of these two techniques. In first technique, we run the hazard of our programing becoming sequential due to sequential memory read and writes. In second technique, there is a cost associated with discarding first N numbers in thread N. This may be proven to be drastically in efficient.

The third alternative way is to have different threads start with different seed numbers. One might notice that, a sequence of N numbers generated from N seeds, will not yield the exact same properties as N numbers generated from the same seed. Never the less, if we choose a smart way to choose these seed sequences we may be able to produce random numbers in parallel with reasonable quality.

One way to choose the seeds for each thread that produces reasonable numbers is using thread number in conjunction with Robert Jenkin's 32 bit integer hash function.

The C++ code is reflected in Table 20.

TABLE 20 ROBERT JENKIN'S 32BIT INTEGER HASH FUNCTION

```
uint32_t hash(uint32_t a)
{
   a =(a+0x7ed55d16)+(a<<12);
   a =(a^0xc761c23c)^(a>>19);
   a =(a+0x165667b1)+(a<<5);
   a =(a+0xd3a2646c)^(a<<9);
   a =(a+0xfd7046c5)+(a<<3);
   a =(a^0xb55a4f09)^(a>>16);
   return a;
}
```

Since, the focus of our paper is not Random numbers; we don't fully test RNGs for their statistical properties. What we look for is whether they produce reasonable option prices with reasonable number of paths and execution time.

We tested 3 parallel number generators. First is the CUDA random number generator which is based on min_std implementation. The numbers start from the same seed and are created using a skip-ahead functionality. Then we test the RNG that comes with CUDA Library, the XORWOW. In this method, we hash the thread number using Jenkin's 32bit hash, and use that number as the seed number. At the end, we tested an implementation of Mersenne Twister by Makoto Matsumoto and Mutsuo Saito specifically designed for GPUs (Mutsuo Saito, 2011). However with this third implementation, we are not able to do kernel fusing and we needed to store the generated random numbers in the global memory which reduces efficiency.

TABLE 21 RANDOM NUMBER GENERATION TIMING COMPARISON – ON THE GPU

	Thrust RNG	CUDA RNG	MT RNG
Method	min_std	XORWOW	Mersenne Twister
Number Of Paths	10 M	10 M	2M
Memory Allocated	55.006395	45.664290	-
Memory Allocated on device with random numbers			110.789124
Bond simulation finished	55.192030	45.988511	296.744956
Sorting finished.	180.262713	157.273804	301.221176
Derivative Payoff Statistic computed	194.770543	184.723107	313.132670

The summaries of the observations are as follows as well as been captured in Table 21:

1. Thrust RNG (based on minstd_rnd)
    a. 10M Paths takes 190ms
    b. Does not need much memory since we generate the numbers to do the calculation and discard them.
2. XORWOW
    a. We do hashing on seeds (Robert Jenkins' 32 bit integer hash function), since skip-ahead cost is not justifiable. The numbers are created in parallel and not stored, since we do not need them to be stored for pricing of the bonds
    b. 10M Paths takes 180ms.
    c. Does not need much memory since we generate the numbers, do the calculation and then discard them.
3. MT
    a. The numbers should be created and stored in global memory, even though we do not need them to be stored.
    b. 2M Paths takes about 300ms.
    c. Requires a large amount of the GPU's global memory, since we cannot do kernel fusion. We can generate numbers using the MT code and use the random numbers stored in the global memory for the simulation. Storage in global memory (on the GPU) is very costly overall and also makes our bond pricing code more complex.

5.5.7 Comments on RNG

The choice of the best RNG is based on both cycle periods and the amount of memory required. From a memory perspective, MT requires a significant amount of memory (624 words) that needs to be updated if used on the CUDA framework where tens of thousands of threads share it. In contrast, XORWOW requires only four words to generate $2^{128} - 1$ random 32-bit integers. This is beneficial from a performance perspective while passing the standard tests of RNG quality.

We suggest using the CUDA RAND library for the purposes of random number generation. If seed numbers are shared between threads XORWOW is a reasonable choice quality wise. For more speed, one can resort to hashing tables and using thread numbers as the key and outputting the hash table as the seed number. If even more speed is desired, an implementation of LCG is favourable.

6 Real world applications and advantages

The ability to use GPUs in finance have a significant economic and performance benefit. There have already been a few large companies that have utilized the power of parallel processing and over time, the power will be utilized in a much great capacity in the future. Bloomberg went live in 2009 running 48 server or Tesla GPU pairs to calculate pricing for 1.3 million hard-to-price asset-backed securities, instead of 1,000 servers (Crosman, 2009). BNP Paribas has also adopted a small cluster containing 2 NVIDIA Tesla units that replace 250 dual-core CPUs. Ultimately, this conversion helped them get a 10x savings in power (Giles, 2010). The world's largest investment bank, JP Morgan, recently adopted a combination of CPUs and GPUs to accelerate performance by 40X allowing the bank to calculate risk across a wide spectrum of products in minutes rather than

Table 22 Organizations that Adopted GPUs

	CPU Only	GPU and CPU
Bloomberg	1,000	48 Tesla
BNP Paribas	250	2 Tesla
JP Morgan	40 x faster	

overnight providing a significant market advantage. Calculations can now be run as needed, enabling flexibility, increasing speed, improving accuracy and allowing for more complex scenario calculations (Marketwire, 2011). Table 22 summarizes the list of companies who have adopted GPU technologies.

Organizations such as Bloomberg and JP Morgan are pioneers in the large-scale adoption of GPUs and are just beginning to explore the benefits of the technology. Utilizing GPUs can provide companies with a high-performance, cost-effective and energy savings solutions to processes that require computational-intensive services. King, Cai, Lu, Wu, Shih and Chang created a system that may offer real-time pricing for traders to better arbitrage or hedge their positions and for the creation of better trading strategies. (King C. L., 2010).

The surrounding industries have started to support computational finance on GPUs, including support by MATLAB and C++ Accelerated Massive Parallelism (C++ AMP) amongst others. The libraries are continuously growing to support future development in GPU coding. In addition to this, independent software vendors (ISVs) are beginning to offer software based on CUDA making it easy for clients to take advantage of this technology. Murex claims that its exotic derivatives models are 60 – 250 times faster, SciComp can run Monte Carlo simulations 7 – 300 times faster and Quantifi Solutions has achieved a 100 times faster calculation of Brace-Gatarek-Musiela (BGM) interest rate derivative pricing model (Davidson, 2010).

There are many advantages for traders and risk managers to increase the speed of their calculations and improve accuracy in their modelling. This will give them a competitive edge against competitors at significantly reduced costs. With the supporting market, enabling the development in this area and overcoming the programming difficulty that there once was, the future of derivative pricing is likely to quickly take advantage of the new speed and power.

7 CONCLUSION

Based on our tests, we have seen significant performance enhancements of 28 times– 140 times faster when using a GPU. The calculations of the bond default time improved by 140 times while the calculation of the barrier bond improved by 28 times when moving from MATLAB to CUDA via C++. This can provide a large time saving advantage for companies who calculate risk exposures on large portfolios. Additionally, this is beneficial for traders who would like to run real-time derivative pricing and risk models. GPUs provide advantages on three fronts: cost reduction, performance enhancement and calculation accuracy. However, special attention needs to be paid to memory management and algorithmic programming in order for the whole system to be efficient. This skill needs to be learned as it varies from the traditional form of programming.

It is for this reason that schools are beginning to offer GPU programming courses which will help to accelerate further development in this area in the near future. The memory management, floating point and specific algorithmic details (including RNG) have been discussed in our paper. These are only a few of the issues that have been outlined, which pose initial challenges for adoption of GPU technology in finance. However, once we built the libraries and programmed solutions, the implementation portion of the product pricings became easier. As is true with most things, we focused on building solid foundations and building blocks that we can utilize in other projects.

The adoption of GPU technology required us to learn three different subjects: (1) the CUDA, (2) memory management and (3) financial pricing. Once the understanding for grids, blocks, warps and threads were understood, we could assimilate how to increase efficiency in the calculations and control the way the data was handled. For each of physical pieces, there are memory devices that vary in speeds depending on which one is used. The coding needed to be designed so that the most efficient steps, in the correct order were taken to minimize slow transfers of data. Finally, we

utilized our financial understanding to price various instruments that are present in the market starting from the simpler default risky bond and then pricing barrier bond via Monte Carlo simulation. Various risk measures can be calculated on these prices including Value-at-Risk (VaR), PFE and Greeks. VaR has already been calculated on the default risky bond, but other risk measures will be left to another time.

Our goal was to demonstrate to increased performance gained by using GPUs in the calculation of pricing derivatives. As we can see, there are significant advantages and these methodologies are trickling through the financial market. With Bloomberg, BNP Paribas and JP Morgan adopting the faster technologies, many companies are likely to follow suit. Therefore, GPUs are becoming a vital participant of computational finance.

8 Appendix - Codes

8.1 Random Number Generators

Here we use the Thrust RNG, disjoint Sequences techniques in pricing of a portfolio of bonds.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/random.h>
#include <thrust/sort.h>
#include <thrust/extrema.h>
#include <cmath>
#include <limits>

#include <cuda.h>

#include <curand_kernel.h>


#include "HRTimer.h"
#include "random.h"
#include "stats.h"



#include <iostream>
#include <iomanip>
```

```
struct bond_Simulate : public thrust::unary_function<unsigned int,float>
{
 __device__
 float operator()(unsigned int thread_id)
 {
            float defaultsAtT = 0;  //          defaults at the end of the year.

            unsigned int numWeeks = 52;     //          Weeks per year

  // note that M * N <= default_random_engine::max,
  // which is also the period of this particular RNG
  // this ensures the substreams are disjoint

  // create a random number generator
  // note that each thread uses an RNG with the same seed
            thrust::default_random_engine rng;

  // jump past the numbers used by the subsequences before me
  rng.discard(numWeeks * thread_id);

  // create a mapping from random numbers to [0,1)
  thrust::uniform_real_distribution<float> u01(0,1);


  // take N samples in a quarter circle
  for(unsigned int i = 0; i < numWeeks; ++i)
  {
   // draw a sample from the unit square
   float toss = u01(rng);
             defaultsAtT += toss>.1 ? 0 : 1;
  }

            return defaultsAtT;
 }
};


struct derivative_payoffPV : public thrust::unary_function<float,float>
{

            __device__
  float operator()(float numDefaultsAtT )
  {
                        float ret;
                        ret= numDefaultsAtT>4? (float)exp(-0.05)*exp( (float) numDefaultsAtT)/500.0 : 0.0 ;
                        //ret = numDefaultsAtT;

                        return ( ret );
  }
};

int main(void)
{
            HRTimer cpuTime;     //Timer


            printCUDAdevices();
            cudaSetDevice(0);         // Set device 1 (the tesla card)  as current
```

```
  int numSim = 10000000;                                    // number of simulations, ie. rows

            printf("\nProgram start with %d paths", numSim );
            cpuTime.StartTimer();


  // allocate storage for row sums and indices
  thrust::device_vector<int> defaultsAtT(numSim);
            printf("\n [%f(ms.)]   Memory Allocated on device",  cpuTime.StopTimer());

            thrust::transform(      thrust::counting_iterator<int>(0),              // Start from memory 0
                                                                  thrust::counting_iterator<int>(numSim),     // # of simulation paths
                                                                  defaultsAtT.begin() ,                                     // memory location
for storing the result
                                                                  bond_Simulate());                                         //
bond_Simulate() simulate one bond over it's life time of one year

            printf("\n [%f(ms.)]   Bond simulation finished.",  cpuTime.StopTimer());


            thrust::sort(defaultsAtT.begin(), defaultsAtT.end());
            printf("\n [%f(ms.)]   Sorting finished.",  cpuTime.StopTimer());

//VaR
            int ValueAtRisk = 100-defaultsAtT[ (int)(numSim*0.01) ] -90;                  //Value at Risk(VT-V0)


//derivative pricing     (code for statistics copied from summary_statistics.cu)
            // setup arguments

            //Get all types of information on the derivative payoff distribution ( including the mean)
            printf( "\nStats on derivative payoff:");
            stat_summary2( thrust::make_transform_iterator( defaultsAtT.begin() ,derivative_payoffPV() ) ,
                                                      thrust::make_transform_iterator( defaultsAtT.end() ,derivative_payoffPV() )   );

            printf("\n [%f(ms.)]   Computed Stats On DerivativePayoff",  cpuTime.StopTimer());

 // compute summary statistics
            printf("\n**DONE** Whole Program: %f(ms.)\n\n",  cpuTime.StopTimer());


            printf ( "\n Yearly 99%% Value at Risk (VaR) is %d .", ValueAtRisk);

            printf( "\nStats on defaults:");
            stat_summary( defaultsAtT  );

            std::cin >> ValueAtRisk;;

  return 0;
}
```

Here we use XORWOW RNG in pricing of Portfolio of bonds, using Thrust library.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/random.h>
#include <thrust/sort.h>
#include <thrust/extrema.h>
#include <cmath>
#include <limits>

#include <cuda.h>

#include <curand_kernel.h>


#include "HRTimer.h"
#include "random.h"
#include "stats.h"



#include <iostream>
#include <iomanip>

__host__ __device__
unsigned int hash(unsigned int a)
{
   a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;
}







struct bond_Simulate : public thrust::unary_function<unsigned int,float>
{
 __device__
 float operator()(unsigned int thread_id)
 {


            float defaultsAtT = 0;  //            defaults at the end of the year.

   unsigned int seed = hash(thread_id);
            unsigned int numWeeks = 52;     //            Weeks per year

   curandState s;

   // seed a random number generator
   curand_init(seed, 0, 0, &s);

   // take N samples in a quarter circle
```

```
   for(unsigned int i = 0; i < numWeeks; ++i)
   {
    // draw a sample from the unit square
    float toss = curand_uniform(&s);
                defaultsAtT += toss>.1 ? 0 : 1;
   }

                return defaultsAtT;
 }
};


struct derivative_payoffPV : public thrust::unary_function<float,float>
{

                __device__
   float operator()(float numDefaultsAtT )
   {
                                float ret;
                                ret= numDefaultsAtT>4? (float)exp(-0.05)*exp( (float) numDefaultsAtT)/500.0 : 0.0 ;
                                //ret = numDefaultsAtT;

                                return ( ret );
   }
};

int main(void)
{
                HRTimer cpuTime;     //Timer


                printCUDAdevices();
                cudaSetDevice(0);          // Set device 1 (the tesla card)  as current

   int numSim = 10000000;                                       // number of simulations, ie. rows

                printf("\nProgram start with %d paths", numSim );
                cpuTime.StartTimer();


   // allocate storage for row sums and indices
   thrust::device_vector<int> defaultsAtT(numSim);
                printf("\n [%f(ms.)]   Memory Allocated on device",  cpuTime.StopTimer());

                thrust::transform(      thrust::counting_iterator<int>(0),               // Start from memory 0
                                                                                thrust::counting_iterator<int>(numSim),     // # of simulation paths
                                                                                defaultsAtT.begin() ,                                                   // memory location
for storing the result
                                                                                bond_Simulate());                                                                       //
bond_Simulate() simulate one bond over it's life time of one year

                printf("\n [%f(ms.)]   Bond simulation finished.",  cpuTime.StopTimer());


                thrust::sort(defaultsAtT.begin(), defaultsAtT.end());
                printf("\n [%f(ms.)]   Sorting finished.",  cpuTime.StopTimer());

//VaR
                int ValueAtRisk = 100-defaultsAtT[ (int)(numSim*0.01) ] -90;                    //Value at Risk(VT-V0)
```

```
//derivative pricing     (code for statistics copied from summary_statistics.cu)
            // setup arguments

            //Get all types of information on the derivative payoff distribution ( including the mean)
            printf( "\nStats on derivative payoff:");
            stat_summary2( thrust::make_transform_iterator( defaultsAtT.begin() ,derivative_payoffPV() ) ,
                                                            thrust::make_transform_iterator( defaultsAtT.end() ,derivative_payoffPV() )   );

            printf("\n [%f(ms.)]   Computed Stats On DerivativePayoff",  cpuTime.StopTimer());

 // compute summary statistics
            printf("\n**DONE** Whole Program: %f(ms.)\n\n",  cpuTime.StopTimer());


            printf ( "\n Yearly 99%% Value at Risk (VaR) is %d .", ValueAtRisk);

            printf( "\nStats on defaults:");
            stat_summary( defaultsAtT  );

            std::cin >> ValueAtRisk;;

   return 0;
}
```

Here we use XORWOW Disjoint Sequences RNG in pricing of Portfolio of bonds, using Thrust library.

```
/*
For the highest quality parallel pseudorandom number generation, each experiment should
be assigned a unique seed. Within an experiment, each thread of computation should be
assigned a unique sequence number. If an experiment spans multiple kernel launches, it is
recommended that threads between kernel launches be given the same seed, and sequence
numbers be assigned in a monotonically increasing way. If the same configuration of
threads is launched, random state can be preserved in global memory between launches to
avoid state setup time.

Source: CURAND_Library.pdf

*/

#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/random.h>
#include <thrust/sort.h>
#include <thrust/extrema.h>
#include <cmath>
#include <limits>

#include <cuda.h>
```

```
#include <curand_kernel.h>


#include "HRTimer.h"
#include "random.h"
#include "stats.h"




#include <iostream>
#include <iomanip>

__host__ __device__
unsigned int hash(unsigned int a)
{
   a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;
}






struct bond_Simulate : public thrust::unary_function<unsigned int,float>
{
  __device__
  float operator()(unsigned int thread_id)
  {


              float defaultsAtT = 0;  //            defaults at the end of the year.

   unsigned int seed = 1234;        //hash(thread_id);
              unsigned int numWeeks = 52;     //            Weeks per year

   curandState s;

   // seed a random number generator
   curand_init(seed, 0, 0 , &s);

   // take N samples in a quarter circle
   for(unsigned int i = 0; i < numWeeks; ++i)
   {
     // draw a sample from the unit square
     float toss = curand_uniform(&s);
                defaultsAtT += toss>.1 ? 0 : 1;
   }

              return defaultsAtT;
  }
};
```

```
struct derivative_payoffPV : public thrust::unary_function<float,float>
{

        __device__
  float operator()(float numDefaultsAtT )
  {
                float ret;
                ret= numDefaultsAtT>4? (float)exp(-0.05)*exp( (float) numDefaultsAtT)/500.0 : 0.0 ;
                //ret = numDefaultsAtT;

                return ( ret );
  }
};

int main(void)
{
        HRTimer cpuTime;     //Timer


        printCUDAdevices();
        cudaSetDevice(1);         // Set device 1 (the tesla card)  as current

  int numSim = 200000;                                  // number of simulations, ie. rows

        printf("\nProgram start with %d paths", numSim );
        cpuTime.StartTimer();


  // allocate storage for row sums and indices
  thrust::device_vector<int> defaultsAtT(numSim);
        printf("\n [%f(ms.)]   Memory Allocated on device",  cpuTime.StopTimer());

        thrust::transform(      thrust::counting_iterator<int>(0),              // Start from memory 0
                                                        thrust::counting_iterator<int>(numSim),     // # of simulation paths
                                                        defaultsAtT.begin() ,                                   // memory location
for storing the result
                                                        bond_Simulate());                                               //
bond_Simulate() simulate one bond over it's life time of one year

        printf("\n [%f(ms.)]   Bond simulation finished.",  cpuTime.StopTimer());


        thrust::sort(defaultsAtT.begin(), defaultsAtT.end());
        printf("\n [%f(ms.)]   Sorting finished.",  cpuTime.StopTimer());

//VaR
        int ValueAtRisk = 100-defaultsAtT[ (int)(numSim*0.01) ] -90;                    //Value at Risk(VT-V0)


//derivative pricing     (code for statistics copied from summary_statistics.cu)
        // setup arguments

        //Get all types of information on the derivative payoff distribution ( including the mean)
        printf( "\nStats on derivative payoff:");
        stat_summary2( thrust::make_transform_iterator( defaultsAtT.begin() ,derivative_payoffPV() ) ,
                                                thrust::make_transform_iterator( defaultsAtT.end() ,derivative_payoffPV() )   );

        printf("\n [%f(ms.)]   Computed Stats On DerivativePayoff",  cpuTime.StopTimer());

 // compute summary statistics
        printf("\n**DONE** Whole Program: %f(ms.)\n\n",  cpuTime.StopTimer());
```

```
            printf ( "\n Yearly 99%% Value at Risk (VaR) is %d .", ValueAtRisk);

            printf( "\nStats on defaults:");
            stat_summary( defaultsAtT  );

            std::cin >> ValueAtRisk;;

  return 0;
}
```

8.2 INSTRUMENTS

```
// C headers
#include <cmath>
#include <limits>
#include <iostream>
#include <iomanip>
#include <stdint.h>                                    //The definition of the basic data types used in this project


//Cuda Headers
#include <cuda.h>
#include <curand_kernel.h>



/*
//Thrust Headers
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/random.h>
#include <thrust/sort.h>
#include <thrust/extrema.h>
*/
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/reduce.h>



// Our Classes
#include "HRTimer.h"
#include "GQDate.hpp"
#include "GQDVector.hpp"
#include "cudaSafeCall.hpp"
#include "cudautilities.hpp"
```

```
//#include "random.h"
#include "stats.h"


#define urand ((float) rand()) / (float) RAND_MAX


__host__ __device__
unsigned int hash(unsigned int a)
{
   a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;
}


#define DrivingFactor float
#define RiskFactor float
#define NULL            0



/*
In case of default, ie. the bond hit the barrier ( ie, the underlying company has defaulted)
the bond holders get the recovery amount R * faceValue;

Otherwise, the bond holders get the faceValue at time T.
*/

struct BarrierBond                                              // An example of an instrument
{
//Contract definitions
        size_t riskFactorNum;
        GQDate maturity;
        GQDate start;


        float R;                        //Recovery
        float barrier;                  //Barrier
        float faceValue;        //faceValue

        float couponRate;               //Coupons by the bond if not defaulted.

// Simulation of this contract through time
        BarrierBond()
        {
                bondHasDefaulted=false;
                couponRate=0;
        }

        __device__ inline float doTimeStep(GQDate t, float A)  //Asset price
        {
                if (bondHasDefaulted==true )                    //In case of default or after maturity the bond doesn't produce any cash flow
                        return 0;
```

```
                        if ( t> (maturity+0.0025)  )          //Floating point issues, adding .5 of a day is the accuracy
                                return 0;

                        if ( gqequalf ( maturity, t, 0.001) )                                                   // Bond matured
                                return faceValue;

                        if ( A<barrier )                                                                        // Bond defaults or not
                        {
                                bondHasDefaulted = true;
                                return (faceValue*R);
                        }

                        if ( couponRate == 0 )
                                return 0;

                        return ( faceValue * couponRate);
        };
private:
        bool  bondHasDefaulted;
};



/*
Simulation of
        [(dS_i)/S_i ]_N=[μ_i]_N dt+[A]_NM [dZ]_M
*/
struct LognormalModel                   //Data to be transfered to a Lognormal MC Kernel.
{
        float A0;

        float alpha;
        float sigma;
        float delta;


        GQDate start;
        GQDate end;
        GQDate dt;
};



__global__ void LognormalMC(LognormalModel lg, GQDVector<BarrierBond> callOptions,  GQDVector<float> simGrid)
{
        // The first thread in the block does the allocation
   // and then shares the pointer with all other threads
   // through shared memory, so that access can easily be
   // coalesced. 64 bytes per thread are allocated.
        const size_t MC_LOOPS_PER_KERNEL        = 1;
                        //Number of simulations per Grid Point
        const size_t seed = threadIdx.x + (blockIdx.x + blockDim.x*blockIdx.y);

        const size_t simGridPos         =       threadIdx.x+blockDim.x*blockIdx.x;
        const size_t Ti = threadIdx.x;



//Block Specific
        __shared__ float A;
        __shared__ float alpha;
```

```
	__shared__ float delta;
	__shared__ float sigma;

	float dz;

	if ( Ti==0 )                                        //First thread allocate the memory for the block specific data
	{
		alpha= lg.alpha;
		delta= lg.delta;
		sigma= lg.sigma;
	}



//Thread Specific

	curandState s;
	curand_init( hash(seed) , 0, 0, &s);                                    // seed a random number generator

	//First Thread in the block initialize the random number generator for this block
	double simPrices[MC_LOOPS_PER_KERNEL];                                  //Holds the prices calculated within this
thread; Each thread will have it's own, for the Instrument it's responsible for



//	if ( Ti < INSTRUMENT_SIZE )                                                                 //Simulation of one
instrument in this block
//		__shared__ Instrument thisInstrument = Instruments[Ti];

	for(size_t mc_counter=0; mc_counter<MC_LOOPS_PER_KERNEL; mc_counter++)          //# of simulation per thread
	{
		BarrierBond c=callOptions[Ti];                                          //The trade that this thread is
responsible for.

		GQDate startTime = lg.start;
		GQDate SimulationEndTime = lg.end;
		GQDate dt = lg.dt;



		simPrices[mc_counter]		= 0;

		if (Ti==0)
			A=lg.A0;

		GQDate t=startTime;

		while (  t < (SimulationEndTime+dt) )                                   //One Path Simulation
		{
			if (Ti==0)   //First thread do the calculation, then all the other trades get priced
			{
				// dZ simulations
				float dA;
				dA = ( alpha * A - delta ) * dt + sigma * A * curand_normal(&s) * sqrt(dt) ;
				A += dA;
			}
			__syncthreads();                                                                    //Make
sure all the riskFactors are loaded


			//Simulation of call options
```

```
                                        //if ( Ti < INSTRUMENT_SIZE )
                                                float cashflow = c.doTimeStep(t, A);
                                                if ( cashflow != 0)
                                                        simPrices[mc_counter]+= cashflow * exp(- 0.05 * (t-startTime) ) ;

                                        //__syncthreads();                                                                      //Make
sure all the INSTRUMENT_SIZE are loaded

                                        t = t+ dt;
                        }

                }


                float avgPrice=0;
                for ( size_t i=0; i<MC_LOOPS_PER_KERNEL;i++)
                        avgPrice += simPrices[i] /MC_LOOPS_PER_KERNEL ;


                simGrid[simGridPos] = avgPrice;




                //__syncthreads();
}

double CUDA_Simulate_Bond(size_t numSimulations, size_t numBonds)
{
                /* These are inputs to the function now

                        // size_t devNumber = 0;                                        //            The device that these simulations will be
run on
                        // const size_t numSimulations = 65000;     //            max 65K, Number of simulations per trade to be runned, would
be the number of blocks
                        // const size_t numBonds = 33;              //            There is one thread per block for each call, multiple of 32 is the
best
                */

                //printCUDAdevices();
                cudaSetDevice(0);



// Instrument Setup
                BarrierBond * host_Bonds;
                cudaHostAlloc( &host_Bonds, numBonds * sizeof(BarrierBond),cudaHostAllocDefault );
                //= (BarrierBond*) malloc(numBonds* sizeof(BarrierBond) );

                for ( size_t i=0; i<numBonds; i++)
                {
                        host_Bonds[i].start             = 0 ;                                   //starts from 0 day
                        host_Bonds[i].maturity          = .05 * (rand()%40);    //Random maturities between 0 and 2 years. only on steps .05

                        host_Bonds[i].faceValue         = 100;
                        host_Bonds[i].barrier = 100;
                        host_Bonds[i].R                                 = 0.8;                                  //Recovery amount in case of default

                        host_Bonds[i].riskFactorNum = 1;                        //It's a call on the first asset.
                }
```

```
	HRTimer cpuTime;                                                        //Timer start
	cpuTime.StartTimer();
	//cpuTime.dEcho( "Program Started" );


	// Transfering the call options to be priced to the GPU
	GQDVector<BarrierBond> dev_Bonds(host_Bonds,numBonds,0);   // Allocate memory and transfer 1024 options to the device0


	// Market Setup
	LognormalModel lg;
	//lg.alpha  = .15;
	lg.alpha     = .05;
	lg.sigma     = .3;
	lg.delta     = 10;                          //Annual Continous div yield

	lg.A0         = 150;
	lg.start = 0 ;                                          //unit:Years
	lg.end = 2;                                             //unit:Years
	lg.dt = 0.05;                                           //unit:Years

	//lg.mu.print();

// Simulation Grid memory allocation
	GQDVector<float> sim_grid(0, numBonds*numSimulations);

	cudaDeviceSynchronize();
	LognormalMC<<<numSimulations,numBonds>>>(lg, dev_Bonds, sim_grid );   //MC simulation that prices all the Bonds


	cudaError_t cudaErr = cudaDeviceSynchronize();
	if(cudaErr != cudaSuccess){
			std::cout << "problem " << cudaErr << "\n";
	}


	//cpuTime.dEcho( "Bond Simulation Ended" );
	//sim_grid.print();
	for ( size_t i=0; i<numBonds; i++)
	{
		thrust::device_ptr<float> dev_ptr ( sim_grid.dptr + i   );
		thrust::device_ptr<float> dev_ptr_end(sim_grid.dptr + i +  numSimulations );
		stat_summary<float> ( dev_ptr , dev_ptr_end );
	}
	/*thrust::device_ptr<float> dev_ptr ( sim_grid.dptr  );
	thrust::device_ptr<float> dev_ptr_end(sim_grid.dptr + sim_grid.arraySize );
	stat_summary<float> ( dev_ptr , dev_ptr_end );
	*/

	//cpuTime.dEcho( "Bonds Priced" );

	double endTime = cpuTime.Lap();

	//delete host_Bonds;
	cudaFreeHost ( host_Bonds);
	dev_Bonds.kill();

	return (endTime ) ;
```

```
}

size_t simCounter=0;

int main(void)
{
	double simulationTimes[6*5];


	for ( size_t numSimulations = 10000; numSimulations<60001; numSimulations+=10000 )
	{
		for ( size_t numTrades=32; numTrades<513; numTrades=numTrades*2 )
		{
			simulationTimes[simCounter] = CUDA_Simulate_Bond( numSimulations,numTrades );
			simCounter++;
			printf("Simulation:%d\n", simCounter );
		}
	}


  char c;
  scanf("\n\n Press any key to exit... %c", &c);

  return 0;
}
```

9 BIBLIOGRAPHY

Marketwire. (2011, August 4). Retrieved August 8, 2011, from NVIDIA:

http://www.marketwire.com/press-release/-1546009.htm

ANSI/IEEE. (1985). IEEE Standard for Binary Floating-Point Arithmetic. *American National Standards Institute*, p. 754.

Belding, T. (2009, 07 17). *FCMP*. Retrieved from http://fcmp.sourceforge.net/:

http://fcmp.sourceforge.net/

Crosman, P. (2009, September 24). Retrieved August 8, 2011, from Wall Street and Tech:

http://www.wallstreetandtech.com/articles/220200055

Dang, D. M., Christara, C. C., & Jackson, K. R. (2011). GPU pricing of exotic cross-currency interest rate derivatives with a foreign exchange volatility skew model. *JCCPE*, 16.

Davidson, C. (2010, November 01). *Balancing the benefits and costs of GPUs.* Retrieved August 8, 2011, from Risk Magazine: http://www.risk.net/risk-magazine/feature/1741590/balancing-benefits-costs-gpus

Dawson, B. (n.d.). *Comparing floating point numbers*. Retrieved from Comparing floating point numbers: http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm

Giles, M. (2010, March 8). Using GPUs for Computational Finance. United Kingdom.

Jackel, P. (2002). *Monte Carlo methods in finance.* Wiley.

Jared Hoberock, N. B. (2011, 07 25). *Thrust Code at the speed of light*. Retrieved from http://code.google.com/p/thrust/

Jared Hoberock, N. B. (n.d.). *An Introduction to Thrust.* Retrieved from thrust.googlecode.com/files/An%20Introduction%20To%20Thrust.pdf

Kato, T. (1996). On a Nonlinear Congruential Pseudorandom Number Generator . *Mathematics of Computation* , 227-233.

King, C. L. (2010). A High-Performance Multi-user Service System for Financial Analytics Based on Web Service and GPU Computation. *International Symposium on Parallel and Distributed Processing with Applications*, 7.

King, G.-H., Cai, Z.-Y., Lu, Y.-Y., Wu, J.-J., Shih, H.-P., & Chang, C.-R. (n.d.). A High-Performance Multi-user Service System for Financial Analytics Based on Web Service and GPU Computation. *International Symposium on Parallel and Distributed Processing with Applications*, 7.

Knuth, D. (1997). Seminumerical Algorithms. In D. Knuth, *volume 2 of The Art Of Computer Programming.* Addison Addison.

Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software*, 1--6.

Matsumoto, M. a. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 3--30.

Matsumoto, T. N. (2002, 1 26). Retrieved from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c

Merton, R. C. (1973, December 28-30). On the Pricing of Corporate Debt: The Risk Structure of Interest Rates. *The Journal of Finance, Vol. 29*(No. 2), pp. 449-470.

Mutsuo Saito, M. M. (2011, 6 20). *Mersenne Twister for Graphic Processors*. Retrieved from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/index.html

Niramarnsakul, C., Chongstitvatana, P., & Curtis, M. (2011). Parallelization of European Monte-Carlo Options Pricing on Graphics Processing Units. *Eighth International Joint Conference on Computer Science and Software Engineering*, 3.

Nishimura, M. M. (January 1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, pp. 8(1):3–30.

NVIDIA. (2011). *NVIDIA CUDA C Programming Guide.* NVidia.

NVIDIA. (January 2011). *CUDA Toolkit 4.0 CURAND Guide.* NVIDIA.

Rupp, M. (2007). *Comparison of Floating Point Numbers.* Frankfurt am Main, Germany.

Simard., P. L. (August 2007). TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), Retrived From: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf.

Solomon, S., Thulasiram, R. K., & Thulasiraman, P. (2010). Option Pricing on the GPU. *12th IEEE International Conference on High Performance Computing and Communications*, 8.

Turner, R. (2011, July 01). *Ovum.* Retrieved July 26, 2011, from http://about.ovum.com/app/jpmorgan-goes-public-about-using-gpus-and-inspires-use-in-capital-markets/

Wilmott, P. (2007). *Paul Wilmott Introduces Quantitative Finance .* Wiley.