Facultad de Informática de A Coruña
UNIVERSIDADE DA CORUÑA

MASTER DISSERTATION

MASTER IN HIGH PERFORMANCE COMPUTING

GPU-Accelerated Pricing of Credit Value Adjustment for Interest Rate Swap under a Gaussian HJM Model

Student: Michel Agustin Herrera Sanchez

Supervisor: Dr. Diego Andrade Canosa

A Coruña, 28 de julio de 2021.

Acknowledgements

Throughout the writing of this dissertation, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Dr. Diego Andrade Canosa, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to thank Dr. Leonidas Kosmidis from Barcelona Super Computing for their valuable guidance throughout the Barcelona Supercomputing Center GPU Hackathon. You provided me with the tools and ideas that I needed to choose the right direction and successfully complete the application benchmark.

Finally, I would like to thank my family for their patience and support.

Abstract

The primary motivation for the current work is the acceleration of the Credit Value Adjustment (CVA) computation for an Interest Rate Swap portfolio written on forward contracts in a fixed income market. *Monte Carlo* method is used to simulate a set of *stochastic differential* equations which help to predict the prices of the portfolio at different times in the future. Numerical *Monte Carlo simulation* methods are *embarrassingly parallel* problems that turn out to be suitable for parallel acceleration and adapt easily to heterogeneous hardware constraints. These numerical methods converge to the true solution, require less memory and are easier to program than other numerical methods. It's well known that numerical integration with *Monte Carlo method* is a good candidate to execution speedup in *heterogeneous processors* like NVIDIA capable GPU. GPGPU technology has shown great potential in conventional derivatives pricing as well as in some risk-management applications. Three different implementations were devised on CPU, NVIDIA single-GPU and multi-GPU to evaluate the potential speed-up the heterogeneous processors technology offers. The risk factor *Monte Carlo* simulation speedup observed was up to 170x between single GPU vs a single CPU and of up to 600x when comparing four GPUs to a single CPU.

Contents

Chapter 1

Introduction

Simulation has already become an industry-approved method for estimating financial security prices for which a *simple closed-form solution* does not exist. Many problems in Mathematical Finance entail the computation of a particular integral. These integrals can be valued analytically, using numerical integrations or computed using partial differential equations. However, when the number of dimensions in the problem is large, *partial differential equations* and numerical integrals become intractable. In these cases, *Monte Carlo* simulation methods give better results at a fastest convergence rate.

Numerical *Monte Carlo simulation* methods are *embarrassingly parallel* problems that turn out to be suitable for parallel acceleration and adapt easily to heterogeneous hardware constraints. These numerical methods converge to the true solution, require less memory and are easier to program than other numerical methods. Nevertheless, is well known that they are very time-consuming and computationally intensive. That is why numerical integration with *Monte Carlo method* is a good candidate to accelerate the implementation in *heterogeneous processors* like NVIDIA capable GPU. GPGPU (General-Purpose-Graphics-Processing-Unit) technology has shown great potential in conventional derivatives pricing as well as in some risk-management applications.

The primary motivation for the current work is the acceleration of the Credit Value Adjustment (CVA) computation for an Interest Rate Swap portfolio written on forward contracts in a fixed income market. *Monte Carlo* method is used to simulate a set of *stochastic differential* equations which help to predict the prices of the portfolio at different times in the future. Three different implementations were devised on CPU, NVIDIA single-GPU and multi-GPU to evaluate the potential speed-up the heterogeneous processors technology offers.

1.1 Organization

The study of the current work is based on the acceleration of *Monte Carlo simulation* of stochastic differential equations using multiple GPUs. A problem-driven approach has been used along the study where initially the problem is stated, a sequential and parallel solutions are proposed, optimization opportunities are considered for code acceleration and a performance analysis is given at the end. The asses, parallelize, optimize and deploy methodology developed by NVIDIA is used through the whole work [NV02].

After a short introduction to OTC market, the chapter 2 explain a step-by-step explanation on how to calculate the CVA metric for a vanilla interest rate swap portfolio. Then, the concrete problem of computational challenge that represents the calculation of counterparty credit risk for financial derivatives portfolios is introduced. Chapter 3 explain the fixed income derivative interest rate swap, valuation and pricing is detailed step by step. In chapter 3.2 the Gaussian Heath, Jarrow and Morton model for forward rate simulation using Musiela parametrization stochastic differential equation is explained. Chapter 3 provides insides on how to generate the expected exposure profile and the initial implementation that led to identify which part of the code are suitable to accelerate on GPU.

Chapter 4 looks further into a basic GPU parallelization and which optimization methods could be applied to improve the speed up and the application throughput. The performance results and discussed in Chapter 5. Finally, some possible extensions of the work are suggested, and conclusions are given in chapter 6.

Chapter 2

The Credit Valuation Adjustment for an Interest Rate Swap Portfolio

Financial derivatives are products that are based on the performance of another, uncertain, underlying asset. An *Over-the-counter* (OTC) market is a decentralized market in which market participants trade derivatives directly between two parties and without a central exchange or broker. The OTC derivatives market is divided by asset class:

- Interest Rate
- Foreign exchange
- Credit derivatives
- Commodity derivatives

Interest Rate derivatives is the biggest market with a notional amount standing value of 466,494 billions of US dollars, according to Bank of International Settlements. [BIS]

Interest is the money paid for credit or similar liability. Examples are bond yields, interest paid for bank loans, and return of savings. Money worth today 1 USD might not have the same value in the future. Time value of money refers to money earned today have the potential to increase its value over a given period. Money deposited into a saving account earns an interest rate. Interest is typically calculated as a percentage of the amount owed to the lender. The percentage that is paid over a certain time period is the interest rate. Interest Rate are market prices which are determined by supply and demand. The *LIBOR* rate is a living example of such rates obtained by observing the market. Forward rates are the interest rates applicable to a financial transaction that will happen in the future.

Interest Rates can be considered fixed or floating. Fixed rates as the word implies are fixed during the duration of the whole contract and can be reset with certain frequency. Floating rates in the opposite depends on how the market behaves in the future, which gives them a certain uncertainty and its trajectory in the future can be modelled by a stochastic model. [CGR][WILL][CHY]. Trading in the interest rate market is done directly with a financial counterparty. Therefore, if any of the two-counterparty involved in the contract default the value of the contract can be considered a loss.

2.1 Credit Valuation Adjustment

Counterparty credit risk is the risk that the counterparty to a financial contract will default prior to the expiration of the contract and will not make all the payments required by the contract. Only the contracts privately negotiated between counterparties over-the-counter derivatives and security financing transactions are subject to counterparty risk. [PYT]. Counterparty risk arises from a broad class of financial products. The most traded are interest rate derivatives hence during this case of study we will be concentrating on a portfolio containing one vanilla interest rate swap derivative contract. The risk-free price of a contract must be different from a risky (counterparty might default) contract. The price of a risky derivative could be thought as the risk-free price minus the component correcting for the counterparty risk.

$$P_{risky} = P_{riskfree} - CVA$$

The latter component is called Credit Valuation Adjustment (CVA). This counterparty risk charge should be calculated in a sophisticated way to account for all aspect that define the CVA including

- The underlying contract(s) exposure
- The default probability of the counterparty
- Netting of existing transactions with the same counterparty
- Collateralization

For the sake of simplicity, the CVA calculation in the current work will be uncollaterized and no risk mitigation like netting set will be included in the scope of the current work. Since the contract value changes unpredictable over time, as the market moves only the current exposure is known with certainty, while the future exposure is uncertain [PYK]. Due to the need to investigate the future for those possible scenarios in which the counterparty can default or there is a move in the market against the positions traded in the contract it would be required to make use of *Monte Carlo* Simulations to address the issue in more than one dimension.

2.2 Computing the Credit Valuation Adjustment

Computing CVA on a portfolio of interest rate swap can be separated on several steps:

- Calibration of underlying models to simple market instruments
- Generation of forward rate, discount factors and portfolio scenarios
- Computation of the exposure profile
- Discounting exposures to current time and weighting them with the corresponding default probabilities.

Given the Exposure modelled framework explained before where exposure is simulated at a fixed set of simulation dates $\{t_k\}$ k = 1, N the computation of the CVA can be approximated as:

$$\mathrm{CVA} = LGD\ (1 - R) \sum_{k=0}^{n} DF\ (t_k, t)\ \mathrm{EE}(t_k)\ PD(t_k, t_k - 1) \qquad (1)$$

Where

- LGD is the loss fraction (of total value), given default
- R is the recovery rate
- DF (t, τ) Is the discount factor, discounting the value from time of default back to the valuation time.
- EE (t) is the Expected Exposure at time t
- PD (t_k , t_{k-1}) is the probability of default of the counterparty at time t.

The *expected exposure curve* is calculated as an average of all *exposure curves* generated and filtered out through a Mark to Market process. The *probability of default curve* ($PD[t_i]$) is the risk neutral probability of counterparty default between s and t. These probabilities were obtained from the term structure of credit default swap (CDS) spreads Sovereign Country. The JPMorgan method is generally used for the *probability of survival term structure* bootstrapping.

2.3 The Counterparty Credit Risk Computational Challenge

Calculating XVA is one of the largest challenges a bank faces daily due to the required volume of calculation to produce a figure. The generation of the *expected exposure profile* curve (*EE*) is the most complex and computation intensive step in the computation of the *CVA*. Which is the process where all *Monte Carlo simulations* happen, and we must aggregate and average the results to obtain the *expected exposure profile curve*. The number of *exposure profiles* generated depends on the number of scenarios and total number of assets.

To illustrate the volume and complexity of the problem consider an investment bank generally needs to process for the bigger counterparties and for all scenarios:

- 1,000,000 of trades
- 5000 pricing points
- 400 scenarios

This results in 200TB of simulations to be completed in a 24h or less. Derivative pricing are generally small problems compared with their scientific applications counterparts but requires a high volume of processing to be completed fasted based on the size in the number of assets held by an investment bank. [RUI]. This is when the application of High-Performance Computing, in specifically the data parallel programming implemented by NVIDIA CUDA and the heterogeneous computing hardware accelerates the calculation of the XVA figures in a fast way and with high degree of accuracy. The parallel processing allows to speed up the calculations by allowing the processing of multiple scenarios at the same time by taking advantage of throughput-oriented design of modern GPUs.

Chapter 3

Expected Exposure Profile Simulation for CVA

The Exposure Profile calculation is the most compute and memory intensive part of the CVA calculation. It has three phases applied for every trade present in the portfolio:

- Scenario Generation: Risk Factor Model Simulation
- Instrument Valuation: Mark to Market
- Aggregation: Expected Exposure Profile generation

This chapter demonstrates how the expected exposure curve calculation is implemented for an Interest Rate Swap portfolio. For each step the sequential implementation details are given with the intention to introduce later the parallelization techniques.

3.1 Vanilla Interest Rate Swap Product Valuation

An interest rate swap contract is an agreement between two counterparties to exchange fixed interest rate payments for floating interest rate payments, based on a predetermined notional principal, at the start of each of several successive periods. [BUK] There are two legs associated with each party. The fixed leg is associated with fixed interest rate payments and the floating leg is associated with future changes in a rate generally indexed by *LIBOR*.

Interest Rate Swap are used to manage exposure changes in interest rates. To illustrate this let's consider the following example:

Suppose a bank lends money to a customer A with a fixed interest of 8% to pay back in 30 years. As the interest is fixed a change in the market, if the interest rate grows, could potentially produce loss to the bank. In order to hedge this problem, the bank engages in a contract with another counterparty such that the bank receives 8% fixed interest rate payments and pays a floating interest rate based on a notional. That way every time the interest rate goes below 8% the bank protects itself from fluctuations in the interest rate market.

A vanilla interest rate swap is a contract agreed at time t and expires at time T, whereby a party exchanges a fixed against a floating interest rate (the LIBOR in the sequel) payment over n reset dates T_0, …, T_{n-1} and payment periods $T_1 - T_0$, .. $T_n - T_{n-1}$, with $T <= T_0$ on a certain notional amount. The length, $T_n - T_0$ is known as the tenor of the interest rate swap. (Figure 1)

Let L (T_{i-1}, T_i) denote the LIBOR rate applying to the period from T_{i-1} to T_i. The payoff at T_i to the counterparty paying the fixed rate δ_{i-1} ($L(T_{i-1}, T_i)$ - K) (*), where K is the fixed interest rate and δ_{i-1} is the length of the reset interval that follows a 30/360 convention. Therefore, the value at t of a forward starting interest rate swap is

$$PayOff(K; DF; L; T_1, \ldots, T_n) = N \sum_{i=1}^{n} DF_i \ \delta_{i-1} \ (L(T_{i-1}, T_i) - K) \quad (2)$$

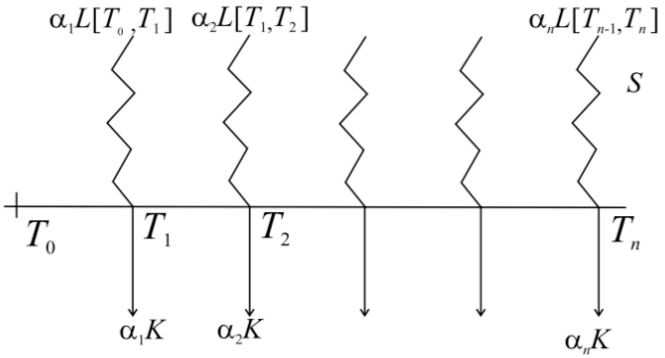

Figure 1. Interest Rate Swap Cashflows exchanges

The *Interest Rate Swap* payoff is the summation of the differences between the value of the floating leg minus the value of the fixed leg at all timepoints from 0 till expiry. Where N represents the notional principal amount, r_τ denotes the instantaneous interest rate as of time τ, and DF_i is the discount factor taken under the risk-neutral probability Q, and conditional on the information at time t. The future value of the $L(T_{i-1}, T_i)$ is unknown but can be modelled using an stochastic model. The value of $L(T_{i-1}, T_i)$ can be calculated from simulated forward rates.

3.2 Risk Factor Simulation

The underlying *forward rate* asset behaviour of the Interest Rate Swap derivative is modelled using the Heath, Jarrow and Morton Model. The stochastic models of the underlying asset are governed by a collection of *stochastic differential equations* which need to fit the model's parameters to provide rates according to the real market situation. As a result of the model parameters calibration and the simulation of the *stochastic differential equation* the term structure of bonds and forward rate curve is obtained to price the vanilla interest rate swap derivative. [GAT]

Since the stochastic differential equations modelled in the 3-factor gaussian HJM model does not have a known closed-form solution the simulation is needed to find the risk factors which are needed to find the future realization of the interest rate swap portfolio exposure. [BMM]

The value of the stochastic differential equation at each simulation step and time point represents the future realization of the forward rate. At each simulation step a new *stochastic differential equation* is evaluated on each tenor point. The forward rate is required to calculate the *LIBOR* rate and the discount factor on 51 points that represent all the cash flows exchanged across time. The number of simulations is calculated by dividing expiry(T) over the simulation time step dt giving it 2500 in total. To increase the number of simulations is required to decrease the value of dt.

At each timepoint the forward rate is calculated by solving a stochastic differential equation using Euler Maruyana method (JAECK)(HDE)

$$f(t+dt) = f(t) + m(t) * d(t) + \sum_{n=1}^{3} \left(\text{vol}_{\text{n[t]}} * \text{phi[n]} * \sqrt{dt}\right) + \frac{dF}{dtau} * dt \quad (3)$$

Where:

- $f_j(t+dt)$ is the forward rate change over time dt
- t takes values between 0 and T expiry
- dt is the size of the simulation step (0.01)
- dtau is the difference between two consecutives tenors
- m(t) the calibrated drift
- vol_i(t) the calibrated volatility i = 0..2
- phi[n] is a gaussian variate with distribution (0, 1)
- dF represents the difference between two consecutives rates. If the forward rate is not part of the right boundary (at expiry) then dF is calculated as f(t+dt) – f(t), otherwise f(t) – f(t-dt).

Although details about the model calibration will not be provided. It's worth to mention that to obtain the volatilities associated with the 3-factor risks model. A covariance matrix is created from a collection of Yield Curve from Bank of England expanded across 20 years. The covariance matrix is then decomposed using dimensional reduction method commonly known as *Principal Component Analysis* (PCA). As a result, the trend is resumed in three factors: shift, twist and butterfly [NOV] that define the shape and dynamics of all simulated forward rate curves. The result of the whole process are three volatility curves expanded from 0 to an expiry day that are used to obtain the drifts and the diffusion term on each *stochastic differential equation*. [GAT]

As mentioned before there is a realization of a different stochastic differential equation on each tenor and simulation point given the usage of different calibrated drift and three volatility factors. Each of the stochastic differential equations generated at each pricing and simulation point is used to produce the discount factors and forward rates to price the *Vanilla Interest Rate Swap* futures cash flows.

A whole round of simulation (forward rates, discount factors) is performed every time the interest rate swap product is priced. The smaller the size of simulation timesteps the higher the number of simulations to be performed. Figure 2 details the required steps to simulate the forward rates f(t,T) and calculate the discount factor Z(t, T). A high number of simulations guarantee that the price for the derivative converges or a required minimum in its variance is achieved, holding the *Law of Large Numbers* for a convergence criteria. The number of simulations is calculated by dividing expiry(T) over the simulation time step dt giving it 2500 in total. To increase the number of simulations is required to decrease the value of dt.

Simulation Simulate an evolution of the whole risk-neutral curve for the necessary length of time, from today t* to T* (0 .. 51)

- For a risk neutral forward curve use GLC data (UK Gilts)
- Realizations of the entire curve f(t, T) over time steps dt = 0.01 are in rows.
- Generate normally distributed N(0,1) pseudo random numbers phi0, phi1, phi2
- f(t,T) uses a simple forward Euler Maruyana scheme to compute: f(t) = f(t+dt) + dfbar

  where dfbar = m(t)*dt + SUM(Vol[i]*phi[i] *SQRT(dt)) + dF/dtau*dt and having i = 0..2

- The paths of forward rates for discretised tenors tau are in columns.

Discount Factors Obtain ZCB values for all required tenors up to T*. Pricing a Zero Coupon Bond with any stochastic short rate r(t) under HJM.

$$Z(t,T) = \mathbb{E}^{\mathbb{Q}}\left[\exp\left(-\int_t^T r(s)ds\right)\right]$$

Looking at the HJM output, is known that there is only an evolution of the first point on the yield curve f(t,t) = r(t).

$$Z(t,T) \quad = \quad \exp\left(-\sum r_t \Delta t\right) \qquad \text{under MC}$$

Monte Carlo Simulation is always discretized over time step dt. Therefore, integration becomes summation.

Cashflows. Calculate the value of cashflows of interest and apply discounting. Consider (FRA – K)+ payoff and discount.

The forward rate simulated at each simulation step is used as an input for the next simulation. There is a dependency across simulation timesteps and across tenors

Figure 2. Heath-Jarrow-Morton Framework with Musiela Parametrization Simulation

A simulation grid containing all forward rates (See appendix B for an example) is obtained as the result of applying the steps described in Figure 2. The top boundary points of the simulation grid are initialized with the spot rates from Bank of England and the inner points update the futures forward rates iteratively. As stated, before a different *stochastic differential equation* is evaluated at each point. Each inner point updates its *forward rate* by using the previous calculated forward rate of its neighbouring points and itself. The rate updating operation for all points in the grid needs to last long enough which means many iterations are required to get the final converged values. Figure 3 shows the calibrated input parameters volatilities and drift that are specific on each pricing point. The first row of the grid represented in figure 4 is the spot rate curve and subsequent rows represent the simulated forward rates. A different simulation grid is generated for each of the 50,000 scenarios (Figure 4). For each product pricing is required to create a grid of 51 points in the horizontal direction, representing pricing points of an interest rate swap of 25 years with cash flows exchanges every 6 months and 2500 simulations points in the vertical direction. The number of simulated rates per scenario is close to 127500 points.

Tenor →	0.00	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0	6.5	7.0	7.5	8.0	8.5	9.0	9.5	10.0	10.5	11.0	11.5	12.0	12.5	13.0	13.5	14.0	14.5
Drift	0.000000	0.000036	0.000069	0.000099	0.000128	0.000155	0.000182	0.000208	0.000233	0.000257	0.000280	0.000303	0.000324	0.000345	0.000364	0.000381	0.000397	0.000412	0.000426	0.000438	0.000449	0.000459	0.000469	0.000478	0.000487	0.000496	0.000505	0.000515	0.000526	0.000538
Vol_1	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431	0.006431
Vol_2	-0.003557	-0.003812	-0.004010	-0.004155	-0.004249	-0.004295	-0.004295	-0.004252	-0.004169	-0.004048	-0.003893	-0.003705	-0.003488	-0.003245	-0.002977	-0.002687	-0.002380	-0.002056	-0.001719	-0.001371	-0.001015	-0.000655	-0.000291	0.000072	0.000432	0.000787	0.001133	0.001468	0.001790	0.002095
Vol_3	-0.004751	-0.003909	-0.003135	-0.002427	-0.001783	-0.001201	-0.000677	-0.000211	0.000202	0.000562	0.000873	0.001137	0.001355	0.001531	0.001667	0.001764	0.001826	0.001855	0.001853	0.001823	0.001766	0.001685	0.001583	0.001462	0.001324	0.001171	0.001007	0.000832	0.000651	0.000464

Figure 3 Calibrated Drift, Volatilities

Figures 4 offer a glimpse on how the entirely forward rate evolves across timepoints and simulations. It's not difficult to appreciate the random behaviour of the forward rate and the reached maximum value of 7% and a minimum value of 2%. In Figure 5 it's possible to appreciate the shape of the simulated curves that captures the different directions or trends *(shift, twist and butterfly)* obtained by the PCA method during the calibration process for the 3-factor volatility curves.

Although convergence analysis is a mandatory step to validate the Monte Carlo simulation it's omitted on this implementation and instead a considerable high number of simulations are run assuming this can potentially reduce the standard deviation between results as the number of simulations increase.

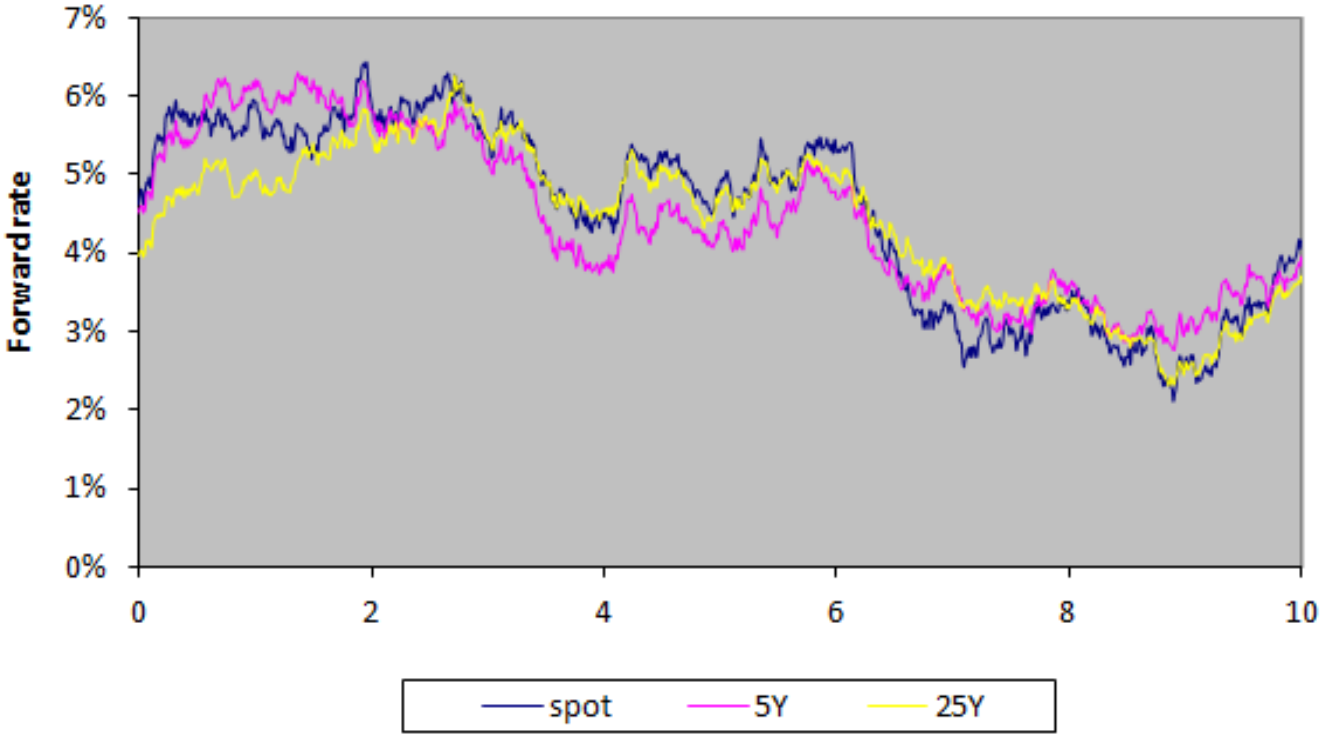

Figure 4. Forward Rates Evolution for each Tenor (Series by Row)

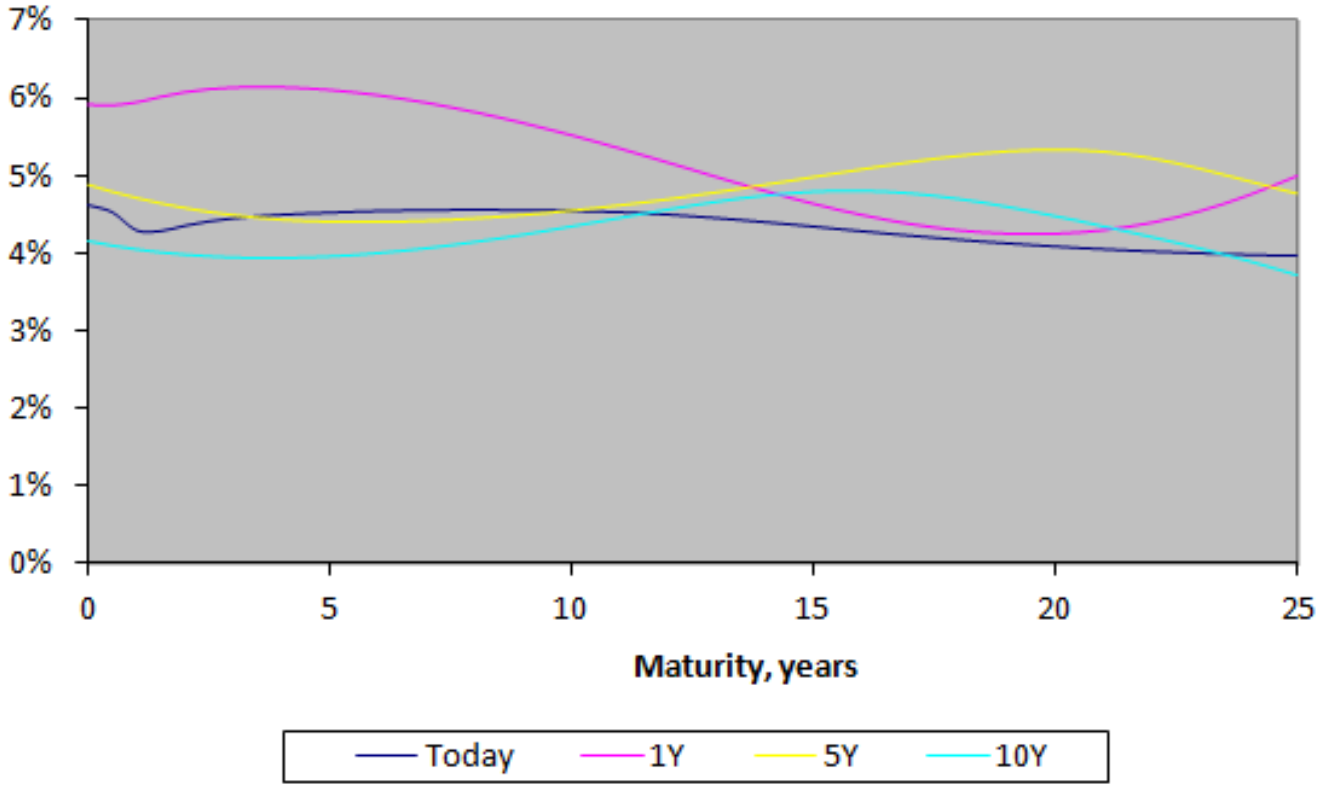

Figure 5. Forward Rates Structures – Yield Curve

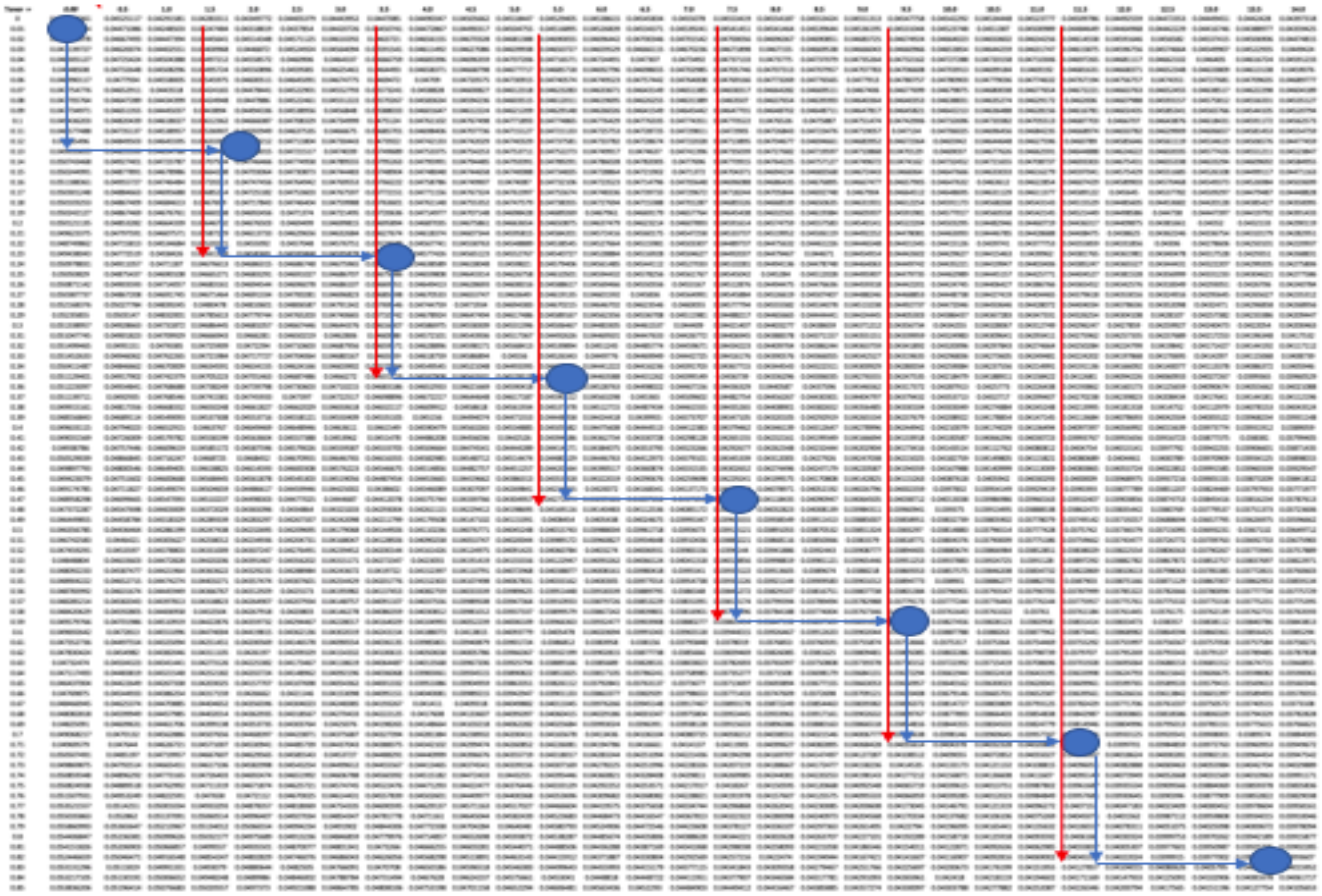

Figure 6. Forward Rate and Discount Factor 2D array Simulation Grid.

If we consider the simulation grid as a 2D array sim_grid. Simulations can be found in the Y dimension as increments of dt and a different *stochastic differential equation* is simulated at each pricing point across the X dimension. The Forward Rate A *forward rate* is generally represented as $F(t; t_1, t_2)$. The value r(t) given to calculate the $F(t; t_1, t_1)$ can be found in the simulation grid as sim_grid [t/dt, t_2], which is represented as blue dots in Figure 6. For example, the rate r(t) to calculate forward rate F(1Y; 0, 6M) can be found located at sim_grid[100, 1].

Bonds are generally represented by B(t, T), are calculated first as an exponential of the summation of all simulated rates across the whole column that represent the tenor t up to row sim_grid[t/dt] times dt. Figure 6 is represented by red arrows where summation occurs from top to bottom direction. All previous forward rates simulations are required to be considered in order to be able to calculate subsequent *discount factors*. See Figure 2 for details on how the calculation is done.

The LIBOR rate is the ratio between two continuous bonds. Its calculated nearly equivalent to the Forward Rate calculation. LIBOR rate is given by the formula:

$$L(t,T,S) = \frac{1}{tau} * \left(\frac{B(t,T)}{B(t,S)} - 1 \right) = \frac{1}{tau} * \left(\frac{e^{-\sum_{i=0}^{T} r(i)dtau}}{e^{-\sum_{i=0}^{S} r(i)dtau}} - 1 \right) \quad (4)$$

$$L(t;T,S) = \frac{1}{tau} * (e^{-r(t)*tau} - 1) \text{ where tau} = T - S \quad (5)$$

The HJM framework algorithm simulate each element r(t) of an NxM grid of all rates. There are strong dependencies between rates calculated at futures simulations times in the vertical direction although in the horizontal direction rates can be computed independently. There is $O(N^3)$ of computational complexity as shown in Appendix B. The main simulation driver needs to execute three nested loops to produce a cube of simulations rates. From the external path loop, passing for the intermediate scenario generation loop and ending the most inner loop that cover each of the points of the whole forward curve. Figure 7 shows the cube generated across all scenarios 51 points for each of the 2500 simulation steps. Each of the vertical slices of the cube (scenarios) can be calculated as separated sequential chunks which allows it to accommodate a high number of forward rate values per scenario. The implementation is very similar to a 2D heat conduction application that is a stencil-based code.

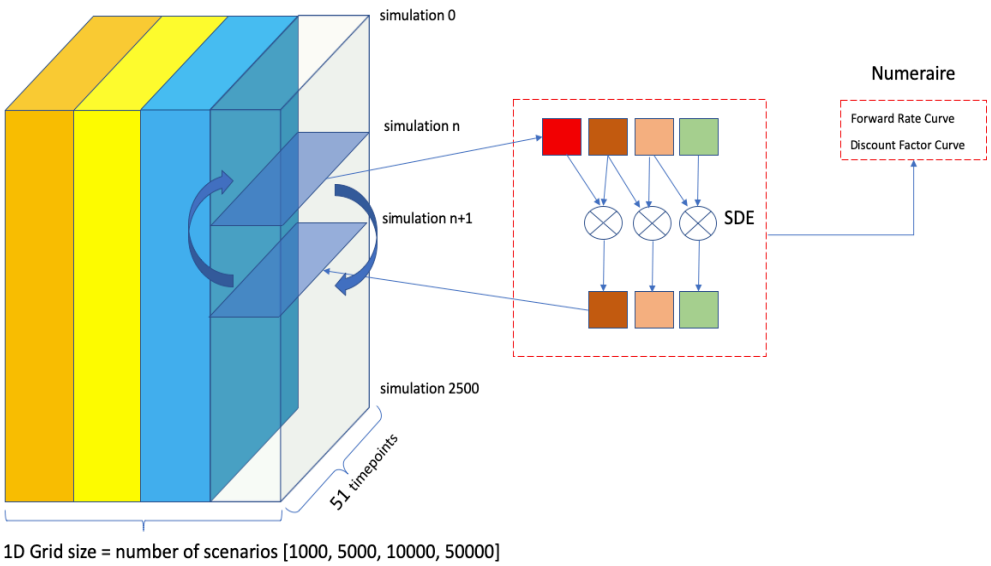

Figure 7. Simulated Risk Factor Cube / Simulation Algorithm implementation.

Figure 7 shows the implementation to generate a 2-D forward rate simulation grid with pathN (2500) and 51 points in the horizontal direction. Initially the forward rate curve is equal to the spot rate across all points of variable simulated_rates. The kernel function __generatedPaths_kernel_cpu calls the *stochastic differential equation* to simulate the stochastics rates at each simulation point which are stored in simulated_rates0 float array. After each iteration step (dt), simulated_rates0 stores the new forward rates and then swap the pointer so that the next iteration the input kernel points to the current simulated new forward rates. When all scenarios are completed, the pointers need to be swapped. On each simulation step the kernel function __generatePaths_kernel_cpu is invoked and the forward rates and their cumulative summation are stored in the arrays accum_rates and d_numeraire.

As discussed above at each simulation step of size the function __generatePaths_kernel_cpu numerically simulates 51 stochastics differential equations. At each simulation point the value of each *stochastic differential equation* is calculated independently but it always depends on values of the previous simulation step. Which somehow is like the concepts of stencil on each point. The neighbour's simulated forward rate values along the volatility, drift and gaussian variates are passed to the function to numerically simulate the *stochastic differential equation* at each of the 51 simulation points. Internally one of terms of the *stochastic differential equation* depends on the neighbour's values. If the rate can be represented by rate[i][j] then his value depends on the neighbours values rate[i][j-1] and rate[i+1][j-1] if it's an internal simulation point, at the border the neighbours values are rate[i-1][j-1] and rate[i][j-1].

3.3 Mark to Market

Once the future market scenarios are simulated for a fixed set of simulation dates using evolution models of the risk factors (PYK), the next step is to proceed to the *vanilla interest rate swap* (IRS) instrument valuation for each simulation date and for each scenario, hence simulated forward rates and discount factors.

The dynamic of the *exposure curve* is driven by the generation of multiple scenarios in the future by simulating the *discount factors* and *forward rates* stochastic models. Those numeraires are used to price the vanilla interest rate swap derivative contract on each simulation point t from zero to expiry (T). A mark to Market process is used in place to filter out our results and focus only on positive values across pricing points because we are interested in potential loss.

A central concept in *Mark to Market* portfolio calculation is the net present value (NPV) of the vanilla interest rate swap derivative at time t. Basically the interest rate swap is valuated as explained in 3.1 for every pricing point between 0 and expiry. The *cashflows* are valuated and accumulated from [t+1, T] for each pricing point t for each scenario. As a result, the exposure curve is build comprising the points from [0, T]. The Exposure is given by NPV(t)+ = max { 0, NPV(t) } as the positive *net present value* at time t of the underlying Vanilla Interest Rate Swap product contract, which is commonly known as mark to market. The result is a curve representing the portfolio exposure on each timepoint which is known as *exposure profile.*

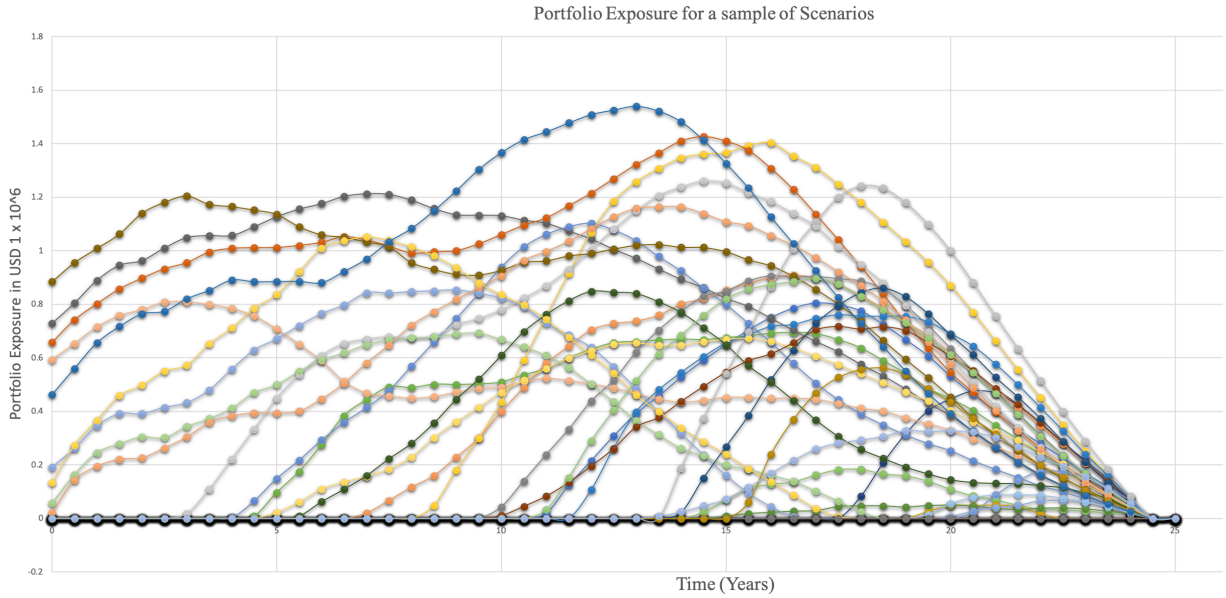

Figure 8. Portfolio Exposure Sample Scenarios Trajectories

3.4 Expected Exposure Aggregation

The *expected exposure profile* (EE) for a *vanilla interest rate swap* is a curve that expands in time from the initial day of the contract till the expiry day. The *expected exposure profile* is basically an average on each simulation point of all the realizations of the exposure profiles in the future. A small representation of the simulated exposure curves to be aggregated can be observed in Figure 8.

The problem to be tackled here is the reduction of matrix columns across all rows of a dense matrix M x N. Rows in the matrix are each of the different simulated exposures profiles (M) and N is a vector with values 1 with a dimension equals to the number of simulation points where exposure profile curve is evaluated.

It would be worth it to point out how the approach of reducing the columns of a matrix through the multiplication of a row by the same matrix can be generalized to perform the linear combination of an ensemble of vectors. The *expected exposure profile* aggregation can be implemented not by performing parallel pair-summations across all columns but instead using a linear algebra transformation that multiplies the matrix (one exposure by row) with a ones vector (d_x) of dimension equals to the number of scenarios using BLAS cblas_gemv function. This is a very efficient way as it takes advantage of the software acceleration done by Intel MKL.

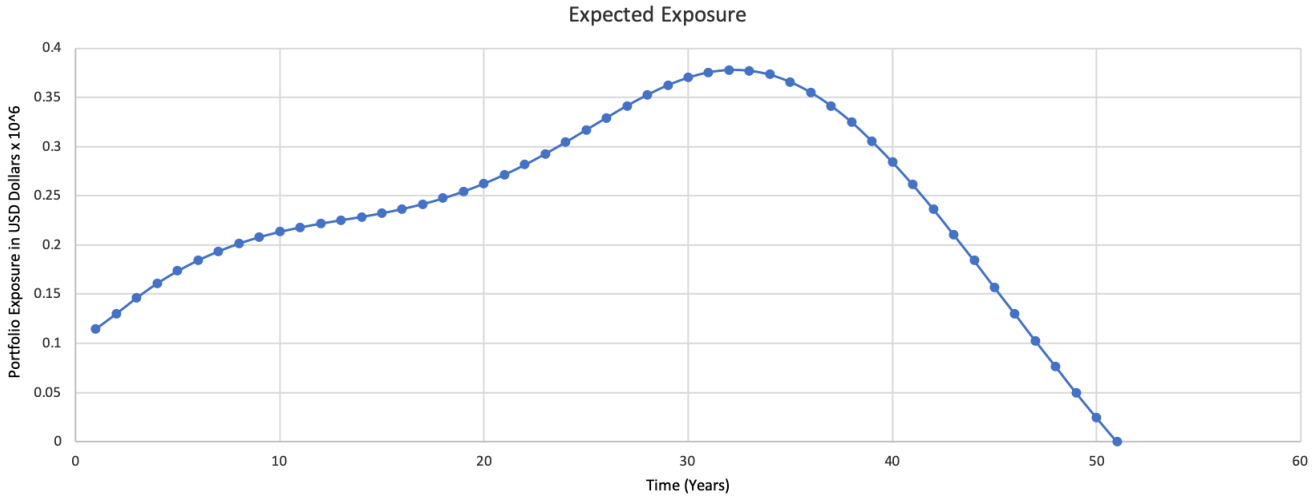

Figure 10. Expected Exposure Profile in blue colour

Observe in Figure 10 the hump shape that takes the exposure profiles for an Interest Rate Swap portfolio. The value of the exposure in the exposure profile curve increases until reaching the peak value then the value of the exposure decreases as the result of evaluating less and less cash flows so his value decreases as the product expires time is reached. The maximum loss represented by a value of 400,000 dollars from a portfolio value of 10,000,000 dollars is reached in simulation point 30, and later it decreases to 0 as the axis time approaches the expiry duration of the contract at point 51. This curve is known as the expected exposure profile.

Chapter 4

Parallelizing the Expected Exposure Profile simulation

This chapter explains how to leverage the CUDA programming model to parallelize the expected exposure profile simulation on GPU hardware. In section 4.1 the parallelization of the simulation process is explained. The mark to Market process is addressed in section 4.2 and the parallelization of the aggregation process is developed in section 4.3. The initial solution is not a fully optimized version but provided hints to take into consideration for high throughput implementation of the GPU hardware.

4.1 Parallelizing the Forward Rate Simulation with CUDA

The implementation of the Forward Rate Simulation is a stencil-based code which has been extensively implemented and accelerated on GPUs. The main flow of the application generates 50,000 scenarios, each of which requires 2500 simulation steps to build the whole *forward rate curve* of 51 points.

The external loop is used to iterate across all simulation timesteps (Appendix D line 738). All threads in the same thread group simulate the rates for a given *forward rate curve*. Basically, the same kernel __generatePaths_kernel function is called across all thread groups. Due to the capacity of GPU device to execute thousands of threads at the same time, there is a direct mapping between the size of the execution grid and the number of futures exposure scenarios to simulate. That is why the grid size has been set to the number of exposure scenarios. Although each SM of the GPU has a maximum number of active blocks and a maximum of 1024 threads per thread group, oversubscription has been put in place to take advantage of the latency-hiding technique embedded in the GPU hardware processing. To this end, all *forward rate curves* are generated in parallel by using a single loop parallelization. This way, if a block of threads cannot progress due to global memory access latency another group of threads could be scheduled for execution. This strategy increases the throughput as all the SMs of the GPU are occupied most of the time.

The main intention is to exploit the high-throughput capabilities of the GPU device. The current hardware setup can scale the execution of the kernel function __generatePaths_kernel across 40 SM. In a few words, at each simulation step represented by the variable path (Appendix D Line 740), a total number of 50,000 threads blocks are fan out or scheduled for execution across all SM. Giving it a ratio of 1250 threads blocks per SM. This oversubscription of threads allows to scale the application. This capability comes naturally for GPU devices allowing to apply loop parallelization techniques where all the independent iterations of the loop are concurrently executed across different processors. In this case, due to the *exposure profiles* being independent from each other, it was feasible to generate them in parallel across all the SM present in the GPU.

The kernel execution configuration is passed through the operator <<< >>> where the first parameter (the grid size) is set to the number of scenarios and the second parameter (the block size) is set to the number of tenors in the forward rate curve (64).

To reduce the data transfer time, data is kept on the GPU whenever is possible (Appendix D Line 756). A double buffer is used to avoid exchange data with the host between simulation steps. On simulation step (i-1) the buffer simulated_rates0 is the input and simulated_rates are the output. At the end of simulation step the role of both buffers is exchanged. As it can appreciated, because the device pointer simulated_rates0 is only used for swapping with the pointer device simulated_rates, the data exchange only happens in the memory device side, preventing data transfers between CPU and GPUs.

It could be trivial to map each of the pricing/simulation points into a 2-dimension structure of size (number of exposures profiles) x (simulation points). This design could limit the number of concurrent blocks executed per device because of GPU memory constraints. Also, it would increase memory latency as memory should be accessed every time a previous simulation result needs to be read from the previous timestep. The *CUDA* programming model can exploit *SIMD* parallelism. Thus, a group of threads can execute the same instruction at the same time if grouped on groups of 32. The internal loop used to evolve the Forward Rate across 51 points is mapped to a group of 64 threads. Each thread of a group can simulate a rate with a device function that implements the stochastic differential equation simulation. The drawback of this approach is that in a 64-thread group with only 51 threads active, 13 threads are always wasted.

As discussed before, the simulation of the Musiela Parametrization *stochastic differential equation* could be seen as a 2-dimentional grid where pricing points were represented in the horizontal direction and timestep simulations in the vertical direction. The number of simulations in the axis depends on the size of each simulation timestep. The kernel __generatePaths_kernel implements the numerical simulation of each of the different *stochastic differential equation* defined at each tenor. See *CUDA* implementation in Appendix G.

At each simulation step each thread looks for the data at his memory position and his neighbour of the previous simulation step to evaluate the *stochastic differential equation* value at a given tenor. The stochastic differential equation _musiela_sde2 could be a weighted sum of the volatilities times a random factor. Every thread from the 64-thread 1-dimensional group invokes the kernel __generatePaths_kernel at every simulation timestep. Internally the thread scheduler splits the thread group into another 32-group size called *warp*. Inside the *warp* threads execute in a lockstep fashion, executing one instruction at a time on all threads simultaneously. In this case the execution scheduler splits the thread group into two warps and schedules them for execution in the SM. For efficiency the thread group is recommended to be multiple of warp size hence multiple of 32. Since only 51 threads out of 64 threads are used to calculate the whole forward curve, a total number of 13 threads are discarded for execution based on boundary conditions. The same kernel instructions code is executed across all threads inside the warp but on different memory locations.

Each thread has a one-to-one mapping to each tenor inside the Forward Rate Curve. The memory location where the instruction will be applied is calculated (Appendix G Line 388) and stored in variable gindex as a function of blockIdx.x, total number of tenors *forward rate curve* and threadIdx.x. Each thread checked based on his index information it is still capable of accessing a valid memory location (Appendix G Line 391). A memory location is considered valid if the local thread index is less than the number of tenors of the *forward rate curve* and the global thread index is less than the linearized kernel execution configuration gridsize. The first simulated curve takes the values from the initial spot rate curve (Appendix G Line 393). For this case, data is pulled from array d_spot_rates which is stored in constant memory. The *CUDA* runtime generally stores the read only constant memory in cache which substantially reduces the number of global memory accesses by all the threads in a thread group.

Otherwise, if the simulation step is intermediate or final for the simulation block (path > 0), then all the parameters required to evaluate the stochastic differential equation at a given pricing point are pre-computed. Boundary conditions are evaluated for all tenors (Appendix G Lines 400 - 405). As all threads inside the warp need to look at *forward rates* simulated on the previous path a phenomenon known as spatial proximity is observed between threads belonging to the same warp. In that sense, instead of having each thread accessing one global memory location at a time, only one memory transaction is needed to fulfil all the memory requests for the entire warp. Access to global memory is considered a long-latency operation that could cause a whole thread group to be swapped out from the active warp execution group in order to facilitate the execution of other thread group for latency hiding.

A mapping is implemented based on the blockId, simulation path and number of volatilities to retrieve the pseudo random values for all threads in the same thread group running the same simulation step. Then the simulation of the *stochastic differential equation* is invoked across the whole thread group but is applied over different parameters located at different memory locations, see (Appendix G Lines 398 – 442). The simulated rate is stored in a register as a result of calling the device function__musiela_sde2. The usage of registers

allows the threads to use a private and faster memory location to store intermediate results. Generally, an SM has 65KB of memory capacity for registers. Once the rate is calculated it is first accumulated in the global memory variable accum_rates and captured in the vector simulated_rates0 for next simulations step executions, see (Appendix G Lines 456 – 465). To finish the execution of the simulation (Appendix G Line 382), the step condition is evaluated to verify any interest of storing the accumulated rates and the simulated rate for a given tenor as described before.

4.2 Interest Rate Swap Mark to Market

As described in section 3.3, for each simulated scenario only positive values are kept after applying the Mark to Market process. Once all the positive exposure profiles have been generated, an average reduction across all timepoints is required to generate the expected exposure profile, which is the vector input parameters for the CVA calculation.

In general, the operations performed in the mark to market process take a binary operator ⊕ and an array of elements [a0, a1, a2, ..., an], and returns the array [(a1 ⊕ a2 ⊕ … ⊕ a n-1), (a1 ⊕ a2 ⊕ … ⊕ a n-2), … , a n-1]. This transformation is a reversed prefix-sum operation which can be parallelized as explained in [MAH2]. In the case of the mark to market operation each cashflow is computed with the simulated values (forward rates (L), discount factors(DF)) generated after the scenario simulation phase is finished. As shown in formula [6] each cash flow value of the vanilla interest rate swap is computed at each timepoint:

$$\text{cashflow}(K; DF; L; T1, \ldots, Tn) = \sum_{i=1}^{n} \quad DFi \ \ \delta i\text{-}1 \ (\ L(Ti\text{-}1, Ti) - K) \quad (6)$$

$$\text{price}(i) = \sum_{k=i+1}^{n} \quad \text{cashflow}[i] \quad (7)$$

The whole prefix-sum can be parallelized [MHA3]. First, a parallel map function represented at formula [6] is applied over each component of the input vector array and produces a temporal intermediate array, then a prefix_sums method is invoked to calculate the exposure values at each timepoint of the exposure curve, as seen in formula [7]. See the Mark to Market CUDA kernel implementation in Appendix F.

4.3 Expected Exposure Aggregation

The expected exposure aggregation follows the same idea as the one explained before in the cpu implementation context. The main difference relies on that the unitary vector and the matrix where each row is represented by an exposure profile are stored in the GPU device memory. In order to avoid transferring that data to host memory to perform the final computation and averaging, the CUBLAS function cublas_gemv is used to perform the multiplication and summation inside the GPU device memory. The cublas_gemv makes extensive use of Fused Multiplies and Add (FMA) instructions which help to increase throughput with the potential cost of less accuracy.

As the number of simulated exposure profiles increases, the application is exposed to floating point summations problems. When adding a huge number of floating-point numbers using the recursive summation approach the application ends up by experiencing round-off error. This problem of floating-point accuracy leads the application to provide different results after different runs or invalid floating-point results. For details on how to address floating point issues in numerical methods see [HIGH] The pair-wise summation is a better choice than the recursive summation, which results in a summation tree just like the one we use in our parallel reduction. Thus, the parallel reduction efficient on GPUs and it can improve accuracy. [PODL]

Chapter 5

Optimization

Different optimization acceleration strategies are explained in this chapter. Aiming to improve kernel execution throughput three techniques have been depicted:

- Modify the kernel execution configuration
- Take advantage of data locality facilities
- Scale the application across multiples GPUs.

5.1 Exploiting oversubscription

The parallelized simulation performance can be improved by analysing the constraints between the software and the underlying accelerator hardware. In this case a thread block in *CUDA* could be composed of up to 1024 threads. The current implementation divides kernels by flattening a 2D problem into one dimension and splitted the processing work between different SM of the same GPU device at *CUDA* thread grid level.

It would be possible to improve performance by increasing only the thread group size as a potential processing throughput increase is expected due to more jobs per thread group. The number of total simulations increases with the number of scenarios. For instance, to obtain 3000 exposure profile realizations, we need 70,000 simulations, if we jump to 10000 exposure profile realizations implies 25000000 simulations.

The execution configuration for kernel__generatedPaths_kernel is specified as following:

```
// Kernel Execution Parameters
int N = 1;
int blockSize = N* BLOCK_SIZE;
int numberOfPoints = N * TIMEPOINTS;
int gridSize = scenarios_GPUs / N;
```

Rather than following the standard way of increasing the amount of work per thread or applying a loop tiling, the approach described here exploits oversubscription. Therefore, there is a map between one or multiple scenarios and a thread group. The *CUDA* Kernel Execution Configuration is defined by the number of blocks or grid size and the number of threads per block. The gridSize is equal to the number of scenarios to be simulated. The blockSize is basically the nearest multiple of 64 that is greater than the total number of points required to simulate the whole rates curve.

As the value of constant N increases, the block Size go up by N times which means that a smaller number of threads blocks per grids need to be distributed across the SM. This results in more work to be performed by thread blocks but constrained by the maximum amount of register per SM and the total number of blocks executed that can be active inside the device for the execution.

What has been benchmarked here is the sweet spot of the number of threads per block based on the code of the __generatePaths_kernel. The simulation of 50,000 scenarios was benchmarked using 128, 256, 512 threads per blocks. None of the increased blocks size configuration managed to improve the application throughput due to excessive global memory access. This indicates that using more threads do not necessarily imply higher throughput. [VOLK1][VOLK2]

5.2 Using Shared Memory

As shown before the CPU code executes a CUDA kernel function until all simulations paths are reached. Although the results are correct the executed kernel didn't make an efficient use of the memory due to access to global memory to fetch the previous data on each simulation step across all simulations. Hence, reducing the global memory traffic in __generatePaths_kernel would help to keep Symmetric Multiprocessors (SM) while the simulation kernel is running.

To simulate the rate with *stochastic differential equation* on each pricing point requires 18 memory access and 6 floating point operations. Calculating the compute-to-global-access-memory ratio gives a result of 0.3 which is an indication that kernel __generatedPaths_kernel can be considered as a memory bound.

The CUDA programming library allows developers to access faster memory located inside the SM. That's the case of *registers* and *shared memory*. *Register*s are private to the thread under execution, but *shared memory* is a kind of scratch path memory that can be shared by all threads that belong to the same thread block. Since *registers* and *shared memory* are inside the SM chip the memory access is faster in several orders of magnitudes with respect to the global memory.

The kernel function __generatedPaths_kernel provides the following performance sweet pots:

- Data elements are loaded multiple times for different threads inside the same thread group.
- Data is only required to be stored in global memory every 50 simulation steps.
- The size of the tile is small enough to schedule different concurrent threads blocks inside the same SM

As explained before each thread group will generate a grid of 2500 x 51 simulated rates. Rather than reserving memory for the whole grid only a portion or a tile is created and used by the thread group. Only a fraction of the memory is needed to generate the whole simulation grid for all thread groups. This type programming pattern is known as a *tiling*. Each tile belonging to one thread group basically occupies a fraction of the shared memory. The current hardware in use supports 128k shared memory size per SM. There is a trade-off with the usage of shared memory in the sense that it's faster but small when the global memory is bigger but slower. The kernel should use a tile that fits inside the shared memory and is small enough to allow other thread blocks to run in parallel. It has been proved before that shared memory is a key accelerator factor for stencil/tiled based code (WEIK).

The kernel function __generatedPaths_kernel4 addresses the potential tuning opportunities exposed in the previously discussed before. First, the total number of times the CPU invokes the CUDA kernel is reduced by a factor of 50. In case of 50,000 scenarios are needed the number of calls to invoke the CUDA kernel drop down considerably from 125,000000 to 2,500000. Therefore, less global memory traffic was required during the simulation process.

All references to code in the rest of this section will be based on the CUDA kernel implementation __generatedPaths_kernel4 shown in Appendix H.

The usage of shared memory is not only about speeding up memory access but also reusing data by different threads inside the same thread group. A float array of size 64 elements is instantiated (Line 493) and allocated in the shared memory. The CUDA directive __shared__ signals the *CUDA* compiler that the variable _ssimulated_rates[] will be stored in shared memory. Each thread in the thread group access global memory to fetch the rate generated in the previous simulation and store the data in the shared memory location array (Lines 507 – 512). To simulate the *stochastic differential equation* each thread looks at the previous intermediate simulation value stored in the shared memory array and makes a difference of two contiguous elements. It happens that for contiguous threads in the thread group this data is reused and stored in shared memory (Line 526 – 531). Bear in mind that for all intermediate simulations with path [n, n+stride], where

stride size is equal to 50 there is no need to write global memory access on every simulation step. Observe that in line (512) all threads in the thread group executes at the same time but it's mandatory to synchronize them to avoid invalid access to memory that has not been initialized.

The top compute and memory access section of the kernel __generatedPaths_kernel4 occurs between lines (515 – 571). The logic is almost the same as kernel __generatedPaths_kernel except that all memory access to simulated data is performed on shared memory. Therefore, a huge performance impact in terms of execution time is noticed as explained in the performance results section.

As elements of the same array _ssimulated_rates are reused between threads. The access to memory is also reduced by half. Allowing this an increase on the floating-point computation per global memory access. The __synchronized method to coordinate the update of array variable _ssimulated_rates[] which is stored in shared memory (Lines 562 and 569). The rest of the implementation is like __generatedPaths_kernel except that the latest simulated rate is the one to be stored in global memory. It would also be possible to improve performance by allowing each thread to process more than one rate element at a time. The section of Performance Analysis will show how the kernel __generatedPaths_kernel4 outperform the rest of the accelerated simulation implementation.

5.3 Scaling with multiple GPUs

Up to this point it have been explained the *CUDA* implementation and optimization of the application in a single GPU. However, most GPU systems include multi-GPUs devices on each node. A multi-GPU implementation is the topic covered in this chapter.

The command nvidia-smi displayed the list of GPU devices present in the multi-GPU system (Figure 11). Observe how each GPU is labelled by number from 0 up to 3. The aggregate global memory increased from 8GB to 32 GB and the throughput processing capability is also increased if work is distributed evenly across all GPU devices. In this case multiple-GPU are connected over the same PCIe bus in a single node.

The simulation of each exposure profile curve is done independently. Therefore, to take advantage of the existence of more than one GPU-device the exposure profile simulation workload needs to be partitioned in such a way that each GPU equally shares an equal amount of work to do. The calculation is self-contained per GPU and, there is no dependency between partitions, and therefore no data exchange is required across GPUs.

To provide parallelism across multiple GPU, the set of exposure profile simulations is divided into equals the number of CUDA-capable GPU installed in the system. The problem domain size is divided by the number of GPUs, processed in parallel and therefore the application improves throughput and efficiency. Once all computations are finished and output data written back to global memory per GPU device the host code transfer the partial exposures profiles from each sibling GPU to host memory and accumulates the results as a vector summation.

The application buffers initially allocated across all exposure profiles are now divided in chunks of a quarter of the total size and created on global memory for each GPU device. Analogy, constant memory data is initialized on each GPU as well. Global memory allocation operations invoke the method cudaSetDevice(GPUDevice) where GPUDevice parameters identify the GPU among others in the multi-GPU system. NVIDIA CUDA API manages and executes kernels on multiple GPUs using an asynchronous programming model. In order to execute the same kernel across multi-GPU the host code need to create a thread per each GPU device and associate it with an individual GPU CUDA context.

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release>nvidia-smi -L
GPU 0: GeForce RTX 2070 SUPER (UUID: GPU-49699fa2-d04c-ce2e-1036-63ede770f6e1)
GPU 1: GeForce RTX 2070 SUPER (UUID: GPU-9da01d6e-0ce2-764e-a1e5-b6bd89f5581a)
GPU 2: GeForce RTX 2070 SUPER (UUID: GPU-cbe94ae7-d531-10ad-c58c-2b85fe12ac37)
GPU 3: GeForce RTX 2070 SUPER (UUID: GPU-a09918d9-1f7b-39ac-44d1-c32af26e8c89)

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release>nvidia-smi
Sun Jun 06 09:27:01 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 456.71       Driver Version: 456.71       CUDA Version: 11.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce RTX 207... WDDM  | 00000000:02:00.0 Off |                  N/A |
| 48%   26C    P8    19W / 215W |    416MiB /  8192MiB |      5%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce RTX 207... WDDM  | 00000000:17:00.0 Off |                  N/A |
| 49%   26C    P8     5W / 215W |    262MiB /  8192MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce RTX 207... WDDM  | 00000000:65:00.0 Off |                  N/A |
| 44%   26C    P8    17W / 215W |    218MiB /  8192MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce RTX 207... WDDM  | 00000000:B3:00.0 Off |                  N/A |
| 48%   27C    P8    17W / 215W |    185MiB /  8192MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Figure 11. Multi-GPU system configuration.

The OpenMP with support of the Visual Studio C++ compiler allows the creation of multiples threads to parallelize the execution of the sequential code. Therefore, it was convenient in the sense that each cuda-context is required to be executed on a different thread. The number of threads to be used depends on the number of GPUs present in the system. An OpenMP parallel region with 4 threads equals to the number of GPUs is created (Appendix I Line 1079). Each OpenMP thread is assigned a unique identifier from 0 to 4 and associated with the GPUDevice number (Appendix I Line 1083). Once the host thread has been associated with the GPU device the context is established and therefore, with the help of the function riskFactorSim4 the kernel __generatePaths_kernel4 is invoked on each GPU (Appendix I Lines 1094).

Gaussian variates are generated in blocks for each GPU device independently. All pseudo random variates are generated and stored in memory using the cuRand API beforehand. A different block of gaussian variates is generated for each GPU. The offset parameter is based on the GPUDevice number times the number of random number chunk size. That way the sequence of Randoms will not be repeated across all the GPU devices. (Apendix A)

The exposureCalculation kernel is executed independently across all the GPU devices as data is locally stored on each GPU device. Initially the result of the exposure profiles is scattered across the four different GPU devices global memory location. Therefore, no data exchange occurs between host nor GPUs. The host code performs a gather operation to fetch the partial accumulated exposures profiles and accumulate them, by calling the function vsAdd, into the host variable partial_exposure.

The modifications required to implement the multi-GPU version of the code only required to introduce the multithreaded code and the scatter and gather pattern. The CUDA kernel did not suffer any change but executes in parallel across different GPUs devices with less amount data. Exploiting the nature of the algorithm that allows to partition the data in smaller chunks to be processed independently for each device was a key driver for this implementation.

Chapter 6

Performance Results

The performance results are discussed in this chapter for each of the optimization techniques introduced before. The use of shared memory and multi-GPU is expected to yield significant performance speedup. The execution time of the top consuming CUDA kernel is improved across different runtime configurations and optimization techniques. The benchmark performed during this work is based on 4 GPU cards NVIDIA RTX 2070. More details about the hardware can be found in Appendix J. The top consuming kernel __generatePaths_kernel was the focus for analysis and optimization. As it was the higher contributor to computation time latency.

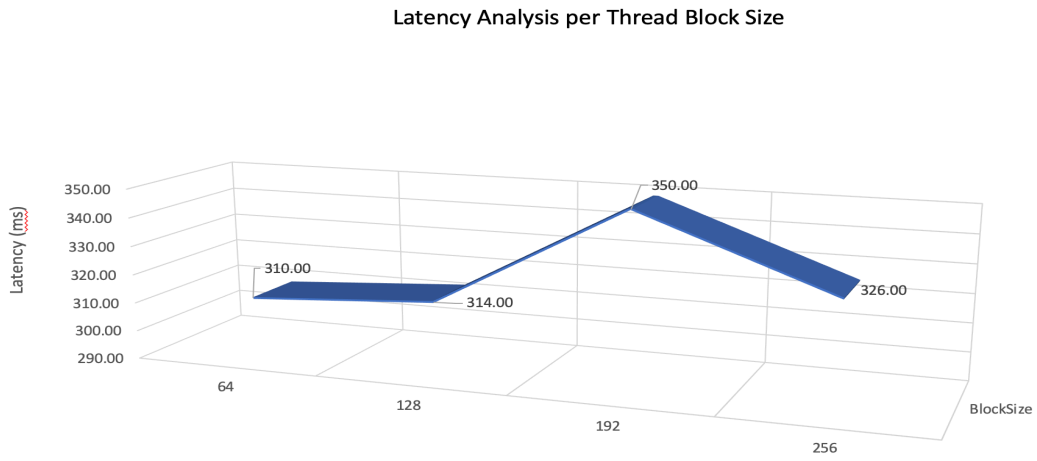

Figure 12. Latency Analysis per thread block size.

The first exploration on performance improvement was to increase the number of threads per block for the __generatePaths_kernel kernel execution configuration. As shown in Figure 12 four different blocks sizes 64, 128, 192, 256 were configured to run the 50,000 scenarios simulation test in single-precision mode. Due to the huge global memory traffic it was not possible to improve the performance. Even though 256 thread block size was the perfect size of thread block the best latency results were obtained with block size of 64 threads. In this case increasing the thread block size does not guarantee a performance improvement. [VOLK2]

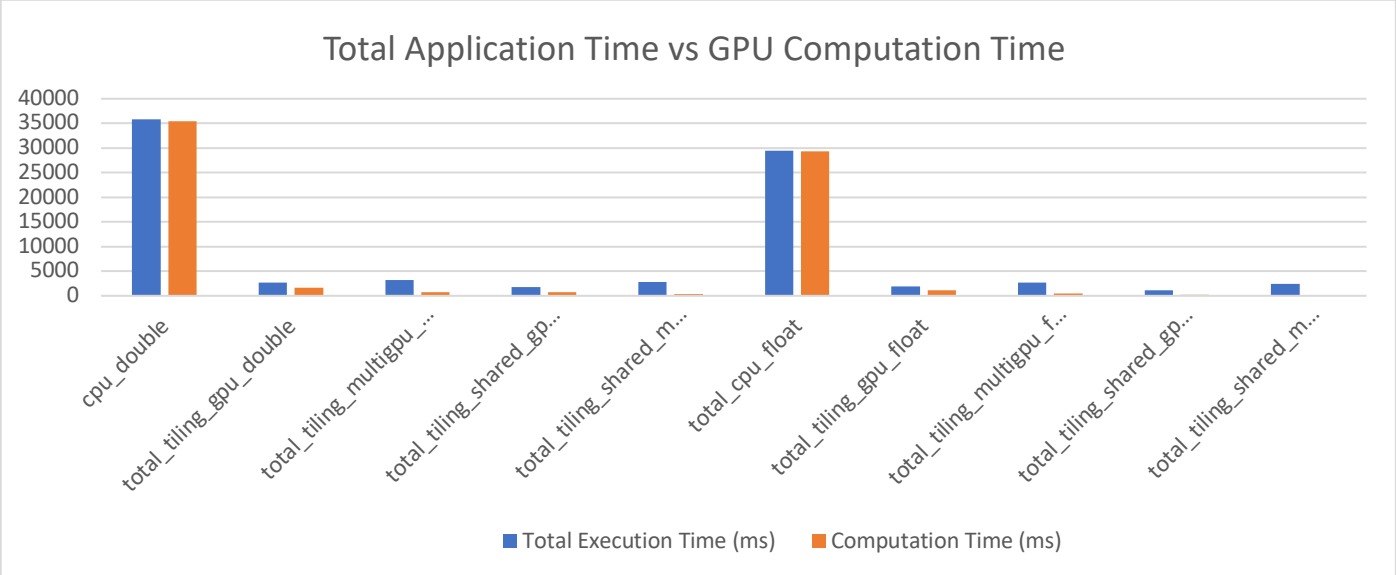

Figure 13. Latency comparison between CPU, GPU and multi-GPU solution.

Implementation	Total application execution-time (ms)	Initialization-time (ms)	Computation-time (ms)	Computation -time Speed up
cpu_double	35777		35461	
tiling_GPU_double	2642	852.49	1583	22.40
tiling_multiGPU_double	3154	1816	500	70.92
tiling_shared_GPU_double	1765	772	543	65.30
tiling_shared_multiGPU_double	2846	1824	150	236.40
cpu_float	29479		29279	
tiling_GPU_float	1916	1837	967	30.27
tiling_multiGPU_float	2674	1837	342	85
tiling_shared_GPU_float	1064	751	157.486	185.91
tiling_shared_multiGPU_float	2153	1808	46	636

Table 1. Latency CPU, GPU and multi-GPU solution for 50000 scenarios.

Figure 13 and Table 1 shows the total execution time of the application and the computation time measured in milliseconds. According to Table 1 all kernel names suffixed with double run in double precision and those kernels names suffixed with float run in single precision mode. The kernel __generatePath_kernel_cpu_double is the reference kernel to calculate the speed up for all double precision kernels. The kernel __generatePath_cpu_float is the reference kernel to calculate the speed up for all single precision kernels. All kernels implemented the tiling optimization technique and those using shared memory contains the shared word in their names. Single GPU and multiGPU specified if the application run the kernels across one or multiples GPUs. The speedup gained from parallelizing some of the applications kernels in GPU is the ratio of the single CPU execution time to the GPU parallel execution time (T). Hence, the computation time Speed up = T(cpu) / T(gpu).

The computation time was the dominant kernel when running the application in CPU. Once the simulation kernel was ported to GPUs the dominant time of the application changed due to the acceleration speed up provided by the GPU solution. It's not hard to appreciate that initialization time and random generation kernels are the major latency contributors after the computation kernel was accelerated. The test was configured to simulate 50000 scenarios in single and double precision floating point accuracy. The best combination of total application execution time and computation-time speed up is provided by the implementation tiling_shared_GPU_float which makes extensive use of shared memory and run-in single precision. Nevertheless tiling_shared_multiGPU_float resulted in the kernel with best computation time speed up.

Implementation	Rnd generation (ms)
cpu_double	1373
tiling_GPU_double	87
tiling_multiGPU_double	52
tiling_shared_GPU_double	167
tiling_shared_multiGPU_double	51
cpu_float	932
tiling_GPU_float	12
tiling_multiGPU_float	9
tiling_shared_GPU_float	13
tiling_shared_multiGPU_float	10

Table 2. Time spent in Gaussian variates generation

The single precision (float) implementation outperformed the double precision. Nevertheless, the float implementation lost accuracy in the tenor 50 of the expected exposure curve due to floating point round off. According to Table 1 and Table 2 all kernel names suffixed with double run in double precision and those kernels names suffixed with float run in single precision mode. The kernel __generatePath_kernel_cpu_double is the reference kernel to calculate the speed up for all double precision kernel. The kernel __generatePath_cpu_float is the reference kernel to calculate the speed up for all single precision kernels.

The table 3 shows the mean square error and the accuracy achieved during the benchmark. Notice how the results in double precision using shared memory are slower but more accurate.

Implementation	Mean-square errors	Error
tiling_GPU_double	0.01377044	10^-2
tiling_multiGPU_double	0.01397286	10^-2
tiling_shared_GPU_double	0	10^-6
tiling_shared_multiGPU_double	0.00059256	10^-4
tiling_GPU_float	0.01380047	10^-2
tiling_multiGPU_float	0.0143285	10^-2
tiling_shared_GPU_float	0.000006	10^-6
tiling_shared_multiGPU_float	0.003549	10^-3

Table 3. Floating Point Accuracy and mean square errors

By doing a breakdown of the computation time it can be appreciated that the multi-GPU kernel implementation combined with the usage of shared memory and single floating-point accuracy provides an acceleration of 632x. Which somehow indicates that the kernel execution scales linearly with the number of GPU present in the system. Due to the low latency of the generate_kernel_path4 of nearly 30ms per GPU device running in parallel. The point here is the trade-off between speed and accuracy. Being the latter is a strong requirement in the financial domain. Table 4 shows the *Monte Carlo* simulation time for each of the CUDA kernels analysed in this study. The programming techniques used to implement access to shared memory somehow had a positive impact in the accuracy of the recursive summation results. See also Table 3 to check the accuracy for implementation kernels tiling_shared_GPU_float and tiling_shared_GPU_double.

MC Simulation kernel name	MC simulation execution time (ms)	MC Simulation speed up
__generatePath_kernel_cpu_double	21114	-
__generatePath_kernel _double	1470	14.36326531
__generatePath_kernel _double	509	41.48133595
__generatePath_kernel4_double	540	39.1
__generatePath_kernel4_double*	164.59	128.2823987
__generatePath_kernel _cpu_float	19335	-
__generatePath_kernel _float	1030	18.77184466
__generatePath_kernel _float	397	48.70277078
__generatePath_kernel4_float	111.863	172.8453555
__generatePath_kernel4_float*	30.5459	632.9818404

Table 4. Monte Carlo Simulation execution time. (*) multi-GPU

The compute and memory access characterization of kernels __generatePaths_kernel and __generatePaths_kernel4 are compared using the metrics collected in the roofline model [WWP] for simple precision floating point accuracy. The ratio of compute to memory accesses for a kernel is called its arithmetic intensity. The estimated flop's performance value is also calculated for both kernels. In Figure 14 the arithmetic intensity is represented on the x-axis and the performance or flops on the y-axis. The thick magenta points plotted in Figure 14 represent both kernels. The magenta point with the red circle represents the kernel __generatePaths_kernel4 and the magenta point with blue circle represent the kernel __generatePaths_kernel. The straight lines through the graph represent the limit in terms of performance for double precision 0.5 teraflops and the other straight line at the top represents the limit in terms of performance for single precision 10 teraflops. The diagonal roof represents the limit on a kernel's peak flops imposed by memory bandwidth constraints for a single and double precision arithmetic intensity.

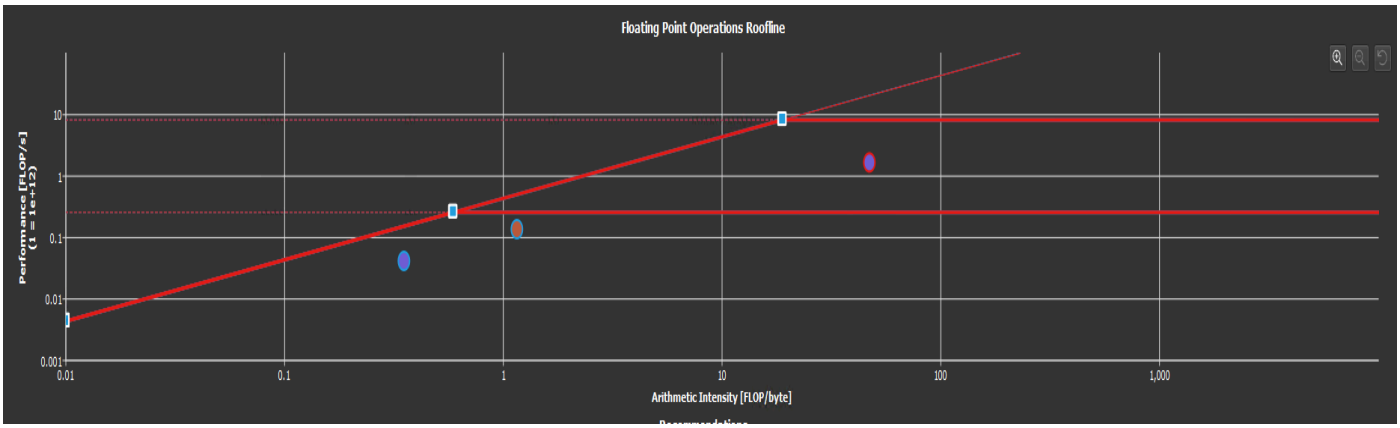

Figure 14. Roof line analysis comparison between __generatePaths_kernel and __generatePaths_kernel4

The kernel __generatePaths_kernel achieved an arithmetic intensity of 1,16 flops/bytes and the flop's performance value is 130 gigaflops. Given the fact that the __generatePaths_kernel measures were taken for single precision the observed performance is poor because the magenta dot with the blue circle represented by the __generatePaths_kernel is under the roof for double precision limits. A kernel using single floating-point precision is faster than the same version of it running in double precision. The poor performance could be related to the excessive not coalesced global memory access in all the simulation steps. In contrast, kernel __generatePaths_kernel4, represented in Figure 14 as a magenta dot with a red circle, is closer to the roof of floating-point precision limits. The improvement in arithmetic intensity for single floating-point accuracy is 47.5 FLOPs/bytes and a flops performance over 1 TB/s. The improvement in performance is in part due to the

use of shared memory to improve the application locality, data reusing between threads inside the same thread block, and partial reduces of all global memory access by using the fast on chip shared memory for intermediate simulation results.

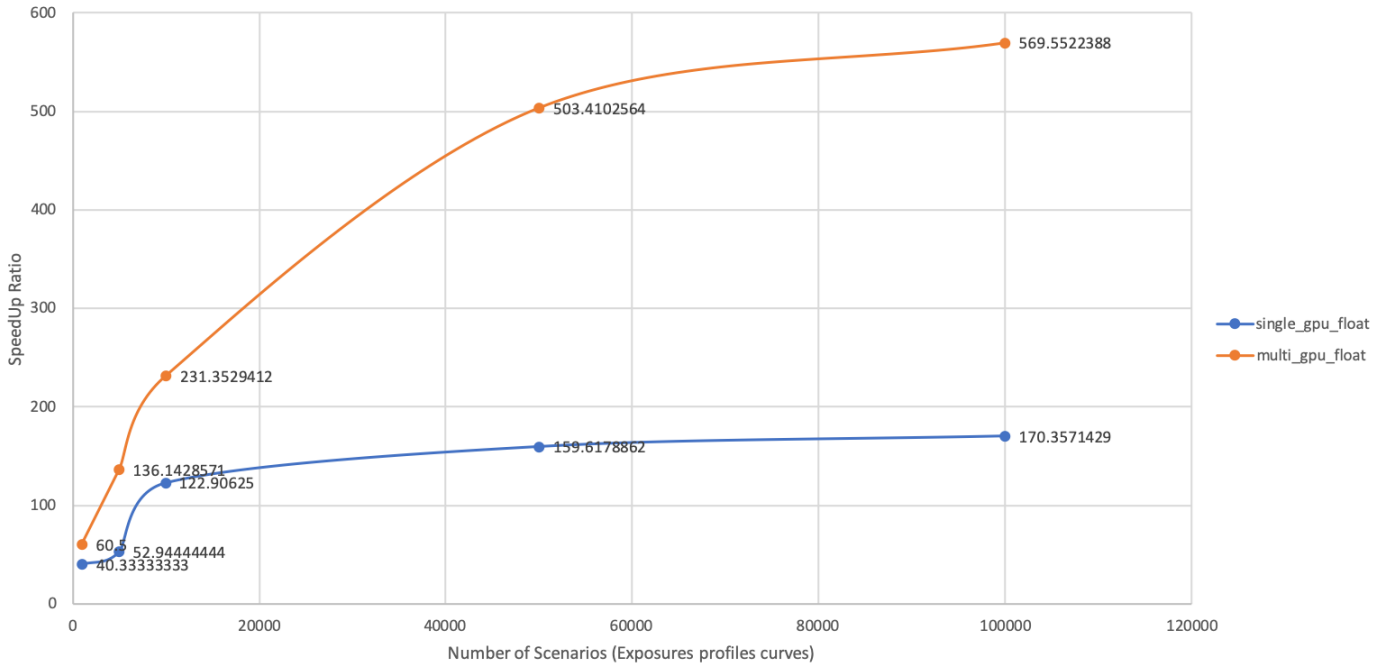

Figure 15. Speed Up ratio of kernel __generatePaths_kernel4 on NVIDIA GeForce RTX 2070 fp 32 in/out fp 32 compute

The single precision speed up ratio achieved by kernel __generatePath_kernel4 is shown in figure 15. The multi-GPU invocation of kernel __generatePath_kernel4 scales linearly as the number of scenarios (number of simulated exposures profiles) increases. The single GPU solution running in single precision flattered the scaling curve factor after reaching 50000 simulated scenarios. In contrast the multi-GPU solution increased their speed up to a value close to a ratio value of 600. This observation shows how the Monte Carlo simulation kernel scales linearly as the kernel __generatePath_kernel4 is executed across 4 different GPUs in a multi-GPU system.

Chapter 7

Conclusion

The CVA calculation can be considered one of the computation dwarfs and the problem has been shown that can be tackled using parallel heterogeneous programming. It has been shown how GPUs can be used to accelerate the calculation of CVA interest rate swap. The GPU accelerated implementation outperforms the CPU implementation in computing the expected exposure profile by using Monte Carlo simulation to evolve the risk factors. The usage of multi-GPU improved even better the final benchmarks results as a larger number of scenarios for an interest rate swap exposure profiles increased. Although the multi-GPU reduces the time spent in simulation time the generation of random numbers penalized the whole calculation time and shows a slight drop in accuracy with respect to the single GPU implementation. It has been illustrated the use of GPU acceleration on NVIDIA hardware, CUDA Programming can achieve a considerable speed up for embarrassing parallel problems like applying Monte Carlo to simulate the stochastic differential equation that represents the Gaussian Heath Jarrow Merton model.

There are several interesting extensions and areas for further work within the area of XVA calculations. First, solving stochastic differential equations numerically by using Monte Carlo simulation is not only about path generation and averaging but also increasing the convergence rate by using variance reduction methods. To improve the floating-point accuracy a fix-point arithmetic could be implemented or incorporate the usage of the predictor-corrector path generation as the number of simulation steps increased [GLAS][JAECK]. Although we achieved good performance results on the acceleration side it would be better if we could increase the amount of calculation work per thread, leading to process more than one exposure profile per thread group. For better utilization of the SM a reduced number of blocks could generate all the required exposure profiles by applying the grid-stride loop pattern [MHA4]. It would be equally interesting to implement a stream-based solution since it would be possible to launch more kernels per SM rather than launching only one kernel at a time. The expected exposure profile would be calculated by using parallel reduction for better accurate and stable results as a result of large floating-point summations [HIGH][MHA3]. An extension of the current work would be the instrumentation of *Monte Carlo* path generation to calculate path-wise sensitivities [SAV][GLAS].

I strongly believe that High Performance Computing is a strong technology that not only helps to obtain faster and accurate results but also will contribute with efficiency to achieve greener energy compliant investment banking.

Chapter 8

References

ASHL Ashley, John. (2007). Computational Finance with GPUs : what is next

BIS OTC derivatives outstanding. https://www.bis.org/statistics/derstats.htm

BUK England, Bank of. (2009) Yield Curve Terminology and Concepts

CGR Cornelis and Grzelak. (2019) Mathematical Modeling and Computation in Finance

HDE Higham, Desmond. (2001). An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations. SIAM

HIGH Higham, Nicholas J. (2002). Accuracy and Stability of Numerical Algorithms. 2nd Edition. SIAM

HOU Houston University, An OpenACC example code for c-based heat conduction code

GAT Dariusz Gaterek, Przemyslaw Bachert and Robert Maksymiuk. The LIBOR Market Model in Practice

BMM Alen Brace, Marek Musiela. A multifactor Gaus Markov implementation of Heath, Jarrow and Morton

JAECK Jaeckel, Peter. (2000). Monte Carlo Methods in Finance.

MHA Harris, Mark. (2000). An even easy introduction to CUDA.

NOV Novosyolov, Arcady (2008). Global Term Structure Modeling using Principal Component Analysis.

MHA2 Harris, Mark. (2000). Parallel Prefix Sum (Scan) with CUDA.

MHA3 Harris, Mark. (2000). Parallel Reduction with CUDA.

MHA4 Harris, Mark. (2008). CUDA Pro Tip: Write Flexible Kernels with Grid-Stride loops

PODL Podlozhnyuk, Victor and Harris, Mark. (2008) Monte Carlo Option Pricing.

PYK Pykhtin, M, and Zhu, S. (2007). A Guide to Modelling Counterparty Credit Risk

NV01 NVIDIA, Curand User Guide.

NV02 NVIDIA, CUDA C Programming Guide.

NV03 NVIDIA, CUDA Best Practices Guide.

NV05 NVIDIA, Precision and Performance Floating point IEEE754 Compliance for NVIDIA GPU

SAV Savigne, Antoine. (2018). Modern Computational Finance, AAD and Parallel Simulations.

RUI Ruiz, I. (2015). XVA Desks – A New Era for Risk Management Understanding, Building and Managing Counterparty, Funding and Capital Risk. Palgrave Macmillan

THO Thomas-Collignon, Thomas. (2018). Volta Performance architecture and performance optimization.

VOLK1 Volkov, Vacily. (2018). Understanding Latency hiding on GPU.

VOLK2 Volkov, Vacily. (2018). Better Performance at Lower Occupancy.

WEIK Wen Wei and Kirk. (2018). Programming Massively Parallel Processors. 3rd edition.

WILL Willmot, Paul. Introduces Quantitative Finance.

WWP Williams, Samuel Patterson, David. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures

Appendix A
Gaussian random variates generation

Pseudorandom sequence: a sequence of numbers, generated by a deterministic algorithm, that has most of the properties of a truly random sequence. The pseudo-random numbers will be used in the case study example to generate gaussian variates in the sde model numerically solved by using Monte Carlo Methods. The Musiela Parametrization stochastic differential equation requires 3 different pseudo random normal distributed variates on each simulation timestep.

The NVIDIA cuRAND library provides a CUDA-based library for rapid pseudo-random number generation.

Pseudo Random Generator Options:

- RNG Algorithm: An RNG algorithm with which to generate a random sequence
- Statistical Distribution: a distribution to which the returned values will adhere
- Seed: A 64-bit integer that initialize the starting state of the pseudorandom number generator
- Offset: A parameter used to sort skip ahead in the sequence. Set offset = 100 to return the 100th number in the sequence first.

Although RNGs could be generated inside the main simulation kernel it was preferred to generate a large batch of RNGs on GPU and store them in global memory for the HJM simulations. This prevents us from having early thread divergence in code as all threads in a block share the same 3 RNGs numbers on each simulation step. The number of RNGs generated was equal to the number of 3 times HJM simulations times the number of blocks.

Figure 16 shows a sample distribution of independent pseudo random number variates for 2500 simulation time steps using the NVIDIA cuRAND library.

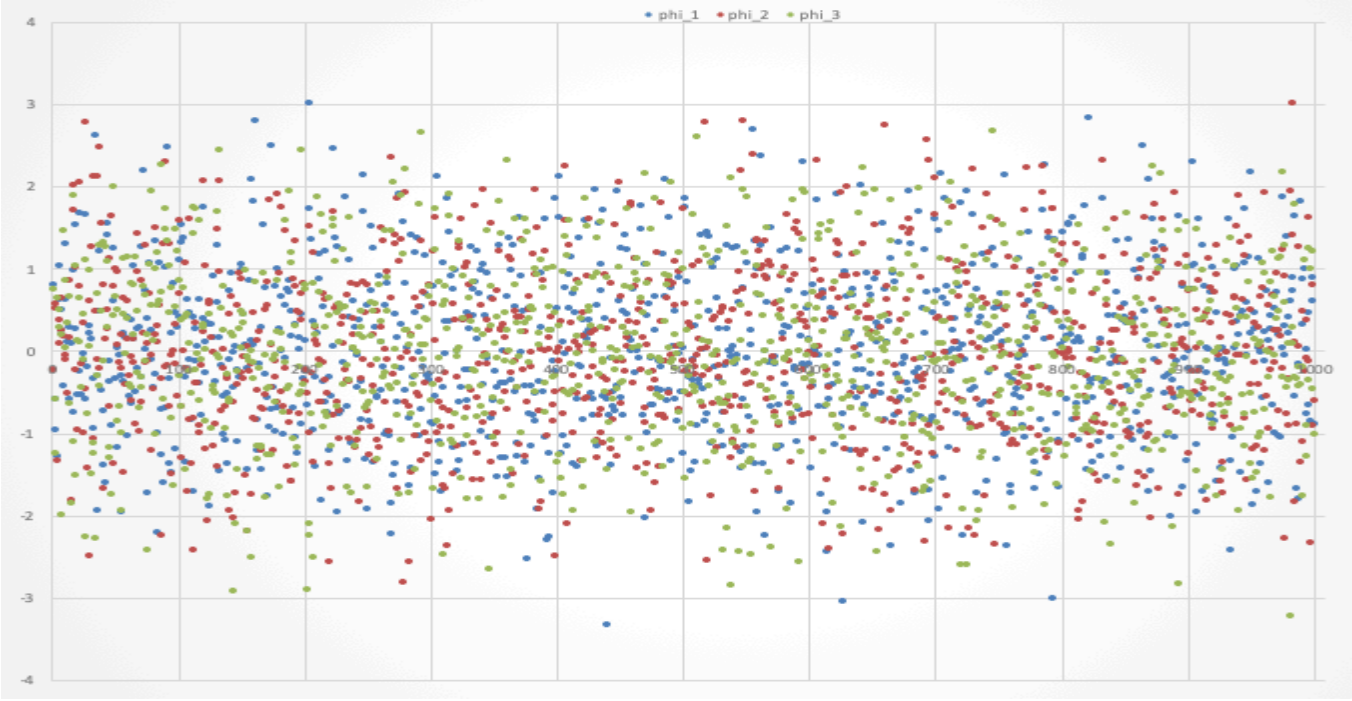

Figure 16. Independent Random Number Variates (0, 1.0)

```
/**
 * * RNG init Kernel
 */

#ifdef RNG_HOST_API
void initRNG2_kernel(float* rngNrmVar, const unsigned int seed, int rnd_count)
{
    const float mean = 0.0;  const float stddev = 1.0;
    curandGenerator_t generator;
    CURAND_CALL( curandCreateGenerator(&generator, CURAND_RNG_PSEUDO_DEFAULT) );
    CURAND_CALL( curandSetPseudoRandomGeneratorSeed(generator, 1234ULL) );
    CURAND_CALL( curandGenerateNormal(generator, rngNrmVar, rnd_count, mean, stddev) );
    CUDA_RT_CALL( cudaDeviceSynchronize() );
    CURAND_CALL(curandDestroyGenerator(generator));
}
```

Figure 17. Using the NVIDIA Host API for pseudo random generation.

As discussed, 3 normal distributed random variates are generated per simulation timestep. Where rnd_count represents the total number of pseudo random numbers to generate. Given _simN (total number of exposure profiles) equals to 50000, pathN as the result of dividing dtau/dt equals to 2500 which represents the number of paths required to simulate the Musiela SDE with 3 factors (actual number of pseudo random presents in the diffusion term of the SDE f(t+dt) [4.3.2.1] a total number of 375000000 floating RNG values are generated.

The pseudo random algorithm used by the application is the one that comes with the value CURAND_RNG_PSEUDO_DEFAULT (Figure 30). Usually, financial quant libraries use other pseudo random generators of better quality as Mersenne Twister or MRG32k3a [JAECK] but for simplicity the default pseudo random generator CURAND_RNG_PSEUDO_XORWOW provided by NVIDIA Developer Toolkit was used for this case. The function curandGenerateNormal is used to generate a normal distributed value with variance 1.0 and mean 0.0.

The cuRAND API allows to create and store the RNG values in device memory. That way accessing this information although in global memory is more efficient that moving data from RAM to Device Global memory. Figure 30 shows how the cuRAND API is used by the application developed for this study.

It has been observed a very low occupancy of their GPU device code as it is required to access an array of random numbers from global memory. The ideal is to find a balance between performance and quality of random numbers, which is problem dependent. The most important part of cuRAND-based projects is to understand the randomness requirements.

Appendix B
Simulation Grid of a Gaussian HJM Model

Tenor ->	*0.00*	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0	6.5
0	0.046138361	0.04525117	0.04291581	0.04283311	0.04349772	0.04405379	0.04443952	0.0447085	0.04490347	0.04505662	0.04518447	0.04529405	0.04538615	0.04545834
0.01	0.045555314	0.04471086	0.04248503	0.04247484	0.04318819	0.0437854	0.04420726	0.04450741	0.04472867	0.04490317	0.04504755	0.04516895	0.04526839	0.04534371
0.02	0.047563978	0.04667493	0.04447394	0.04445641	0.04514348	0.04571125	0.04610292	0.0463721	0.04656155	0.04670328	0.04681388	0.04690055	0.04696442	0.04700346
0.03	0.047139727	0.04620074	0.04402551	0.04400968	0.0446872	0.04524924	0.04564094	0.04591541	0.04611492	0.04627086	0.04639938	0.04650727	0.04659529	0.04666115
0.04	0.048305127	0.04725424	0.04504388	0.04497212	0.04558572	0.0460906	0.0464337	0.04666759	0.04683396	0.04696359	0.04707206	0.04716571	0.04724491	0.047307
0.05	0.04848508	0.04732648	0.04508296	0.04495724	0.04550896	0.0459585	0.04625461	0.0464493	0.04658371	0.04668798	0.04677717	0.04685718	0.04692796	0.04698655
0.06	0.048990117	0.0477934	0.04558005	0.04545975	0.04600511	0.04645091	0.04674775	0.0469472	0.04709	0.04720575	0.04730915	0.04740574	0.04749523	0.04757442
0.07	0.047754776	0.04652911	0.0443518	0.04424165	0.04478441	0.04522901	0.04552793	0.04573241	0.0458828	0.04600827	0.04612318	0.04623283	0.04633671	0.04643149
0.08	0.047705764	0.04647289	0.04434399	0.04424948	0.0447886	0.04522461	0.04551222	0.04570267	0.04583634	0.04594236	0.04603515	0.04612011	0.04619695	0.04626255
0.09	0.047734971	0.04651355	0.04445037	0.0443894	0.04494336	0.04538956	0.0456848	0.04588033	0.04601647	0.04612224	0.04621209	0.04629148	0.04636026	0.04641549
0.1	0.049436203	0.04820439	0.04618027	0.04612362	0.04666087	0.04708329	0.04734999	0.0475124	0.04761102	0.04767498	0.04771893	0.04774865	0.04776429	0.04776335
0.11	0.048577488	0.04735137	0.04538957	0.04536907	0.04592949	0.04637535	0.0466675	0.04685701	0.04698406	0.04707736	0.04715127	0.04721133	0.04725753	0.04728725
0.12	0.04985496	0.04849503	0.04645595	0.04633399	0.04678212	0.04711804	0.04730443	0.0473922	0.04742133	0.04742029	0.04740329	0.04737581	0.04733782	0.04728674
0.13	0.049935655	0.04855399	0.04654786	0.04643454	0.04688015	0.04721517	0.0474039	0.04749689	0.04753375	0.04754253	0.04753711	0.04752273	0.04749917	0.0474637
0.14	0.050743468	0.04927401	0.04723787	0.04707504	0.04746466	0.04774958	0.04789555	0.04795263	0.04795991	0.04794485	0.04792091	0.04789291	0.04786028	0.04782005
0.15	0.050244991	0.04877891	0.04678986	0.04664288	0.04703064	0.04730873	0.04744483	0.04748904	0.04748048	0.04744658	0.04740088	0.04734835	0.04728864	0.04721902
0.16	0.051188361	0.04955727	0.04746484	0.04720213	0.04747456	0.04764942	0.04769553	0.04766222	0.04758786	0.04749907	0.0474087	0.04732106	0.04723523	0.04714796
0.17	0.050301248	0.04884663	0.04695688	0.04685214	0.04725182	0.04753603	0.04767597	0.04772151	0.04771156	0.04767324	0.04761997	0.04755674	0.04748336	0.04739733
0.18	0.050103253	0.04867409	0.04684613	0.0467699	0.04717843	0.04746404	0.04759988	0.04763601	0.04761148	0.04755352	0.04747579	0.04738355	0.04727694	0.04715388
0.19	0.050242127	0.04867469	0.04676761	0.04660238	0.04692456	0.0471374	0.04721495	0.04720636	0.04714977	0.04707148	0.04698428	0.04689269	0.0467961	0.04669179
0.2	0.050121135	0.04853282	0.04664339	0.04646782	0.04676503	0.0469499	0.04699815	0.04695894	0.04687035	0.04675861	0.04663654	0.04650875	0.04637479	0.04623214
0.21	0.049623375	0.04797591	0.04607571	0.04587229	0.04613733	0.04629656	0.04632684	0.04627674	0.04618374	0.04607344	0.04595815	0.04584201	0.04572416	0.04560175
0.22	0.048749862	0.04715813	0.04534684	0.04520221	0.0455092	0.0457048	0.04576751	0.04574589	0.04567741	0.04558763	0.04548889	0.04538545	0.04527664	0.04515981
0.23	0.049438043	0.04772519	0.0458426	0.04561399	0.0458369	0.04595868	0.04595878	0.04588505	0.04577426	0.04565123	0.04552767	0.04540727	0.04528884	0.04516928
0.24	0.050978831	0.04911057	0.0471187	0.04676613	0.04686315	0.04686748	0.04675965	0.04658704	0.04638589	0.04618048	0.0459821	0.04579406	0.04561485	0.04544112
0.25	0.05050829	0.04875437	0.04690108	0.04665271	0.04683291	0.04691027	0.04686707	0.04675086	0.04659808	0.04643314	0.04626758	0.04610501	0.04594432	0.04578256
0.26	0.050871142	0.04903593	0.04714057	0.04683161	0.04694544	0.04696078	0.04686107	0.04669351	0.04649413	0.04628693	0.04608316	0.04588617	0.04569466	0.04550556
0.27	0.050387737	0.04867208	0.04691745	0.04671464	0.04691334	0.04700281	0.04696823	0.04685698	0.04670533	0.04653747	0.0463649	0.04619135	0.04601592	0.045836
0.28	0.052168376	0.05027784	0.04839245	0.0480478	0.04810601	0.04806587	0.04791342	0.04769546	0.04744759	0.0471934	0.04694383	0.04670215	0.04646702	0.04623546
0.29	0.05235855	0.0503147	0.04832001	0.04785613	0.04779744	0.04765203	0.04740665	0.04710742	0.04678924	0.04647494	0.04617486	0.04589167	0.04562356	0.04536708
0.3	0.051338957	0.04928663	0.04731872	0.04686445	0.04681057	0.04667446	0.04644376	0.04616407	0.04586975	0.04558309	0.04531396	0.04506467	0.04483305	0.04461537
0.31	0.051047745	0.04901823	0.04709929	0.04666943	0.0466281	0.04650219	0.0462806	0.0460088	0.04572101	0.04543936	0.04517367	0.04492626	0.04469501	0.04447633
0.32	0.051499465	0.0495151	0.0476585	0.04725909	0.0472294	0.04710603	0.04687956	0.04659571	0.04628896	0.04598171	0.04568413	0.04539894	0.0451245	0.04485774
0.33	0.051452633	0.04946062	0.04762265	0.04721984	0.04717727	0.04704064	0.04680167	0.0465059	0.04618759	0.04586894	0.04556	0.04526343	0.0449776	0.04469949
0.34	0.050411487	0.04846662	0.04670009	0.04634591	0.04634155	0.04624166	0.04603902	0.04577894	0.04549545	0.04521048	0.04493395	0.04466841	0.04441222	0.04416236
0.35	0.051129401	0.04917902	0.04742379	0.04705223	0.04701463	0.04687486	0.0466272	0.04631719	0.04597908	0.04563501	0.04529519	0.04496256	0.04463588	0.04431262
0.36	0.051223097	0.04934841	0.04768688	0.04738249	0.04739798	0.04730603	0.04710215	0.04683186	0.04652933	0.04621669	0.04590418	0.04559485	0.04528763	0.04498022
0.37	0.051139711	0.0492935	0.04768546	0.04741581	0.04745933	0.047397	0.04722517	0.04698896	0.04672217	0.04644648	0.04617187	0.04590112	0.04563298	0.045365
0.38	0.049915161	0.04817356	0.04668332	0.04650248	0.04661827	0.04662029	0.04650618	0.04632117	0.04609912	0.0458618	0.04561934	0.04537478	0.04512715	0.04487434
0.39	0.048556843	0.04689114	0.04549093	0.04537308	0.04553716	0.04558121	0.04550409	0.04535105	0.045156	0.04494074	0.04471553	0.04448358	0.04424418	0.0439955
0.4	0.049633115	0.04794023	0.04652915	0.0463767	0.04649469	0.04648946	0.0463611	0.0461549	0.04590479	0.04563265	0.04534885	0.04505682	0.04475608	0.04444513
0.41	0.049031569	0.04726009	0.04579782	0.04558199	0.04563604	0.04557388	0.0453962	0.0451478	0.04486208	0.04456036	0.0442526	0.04394186	0.04362734	0.04330728
0.42	0.04938786	0.04757446	0.04609619	0.04585172	0.04587596	0.04579026	0.04559587	0.04533705	0.04504664	0.04474541	0.04444289	0.04414171	0.04384075	0.04353795
0.43	0.050529039	0.04866845	0.04716247	0.0468725	0.0468452	0.04670955	0.04646765	0.04616355	0.04582985	0.04548712	0.04514474	0.04480529	0.04446763	0.04412973
0.44	0.049897793	0.04800546	0.04649405	0.04618825	0.04614593	0.04600308	0.04576223	0.04546675	0.04514856	0.04482757	0.04451257	0.04420564	0.04390517	0.04360874
0.45	0.049423079	0.04751602	0.04600668	0.04568445	0.04561878	0.04545303	0.04519056	0.04487454	0.04453665	0.04419662	0.04386313	0.04353818	0.04322019	0.04290676
0.46	0.049174785	0.04711827	0.04549074	0.04504659	0.04486617	0.04459946	0.04425002	0.0438602	0.04346089	0.04307097	0.04269841	0.04234455	0.0420072	0.04168341
0.47	0.048958298	0.04699665	0.04547093	0.04510237	0.04498303	0.04477025	0.0444687	0.04412078	0.04375744	0.04339766	0.04304957	0.04271474	0.04239126	0.04207652
0.48	0.047372287	0.04547698	0.04403009	0.04372029	0.04365096	0.0434864	0.04323203	0.04293004	0.04261115	0.04229412	0.04198695	0.04169116	0.04140483	0.04112536
0.49	0.046449855	0.04458784	0.04318329	0.04289339	0.04283297	0.04267337	0.04242098	0.04211799	0.04179508	0.04147102	0.04115391	0.0408454	0.0405438	0.04024675
0.5	0.046356785	0.04436464	0.04286199	0.04247438	0.04232695	0.04209695	0.04179068	0.04144926	0.04110236	0.04076771	0.04045248	0.04015743	0.03988004	0.03961718
0.51	0.046742583	0.0446421	0.04305627	0.04258012	0.04234936	0.04204731	0.04168047	0.04128926	0.04090258	0.04053747	0.04020044	0.03989172	0.03960827	0.03934648
0.52	0.047459291	0.0453597	0.04378803	0.04331009	0.04307247	0.04276491	0.04239452	0.04200144	0.04161426	0.04124975	0.04091425	0.04060784	0.0403274	0.04006931
0.53	0.04848804	0.04633603	0.04472828	0.04420206	0.04391467	0.04356202	0.04315171	0.04272347	0.0423055	0.04191419	0.04155556	0.04122947	0.04093262	0.04066124
0.54	0.048092233	0.04587477	0.04421964	0.04363622	0.04329233	0.04288984	0.04243672	0.0419722	0.04152397	0.04110791	0.04072968	0.04038877	0.04008161	0.03980418
0.55	0.048904222	0.04652715	0.04474274	0.04403271	0.04357474	0.04307601	0.04254429	0.04201776	0.04152303	0.04107498	0.04067831	0.04033162	0.0400305	0.03977014
0.56	0.048783992	0.04631674	0.04445949	0.04366767	0.04312929	0.0425573	0.04195982	0.04137453	0.04082759	0.04033339	0.03989625	0.03951448	0.03918339	0.03889795
0.57	0.048285214	0.04582043	0.04397813	0.04318823	0.04264907	0.04207934	0.04148757	0.04091107	0.04037556	0.03989508	0.03947364	0.03910933	0.03879726	0.03853229
0.58	0.048420629	0.04592805	0.04406958	0.0432504	0.04267918	0.0420803	0.04146278	0.04086359	0.04030812	0.03981012	0.03937337	0.03899579	0.03867242	0.03839801
0.59	0.049579766	0.04701986	0.04510919	0.04422876	0.04359732	0.04294467	0.04228017	0.04164029	0.04104993	0.04052239	0.04006109	0.03966363	0.03932477	0.03903908
0.6	0.049692642	0.0472813	0.04551096	0.04474004	0.04419815	0.04362136	0.04302019	0.04243154	0.04188073	0.0413815	0.04093779	0.0405478	0.04020694	0.03991043
0.61	0.047352736	0.04497554	0.04325094	0.04251451	0.04200569	0.04146578	0.04090554	0.04036135	0.03985801	0.03940879	0.03901724	0.0386812	0.0383958	0.038156
0.62	0.047830424	0.0454982	0.04382046	0.04311105	0.0426197	0.04209329	0.04154332	0.04100615	0.04050658	0.04005786	0.03966367	0.03932199	0.03902815	0.03877738
0.63	0.04732474	0.04504323	0.04341441	0.04273126	0.04225382	0.04173467	0.04118619	0.04064487	0.04013568	0.03967206	0.03925794	0.03889166	0.0385689	0.03828531
0.64	0.047117493	0.04483819	0.04321548	0.04252182	0.04202724	0.04148962	0.04092196	0.04036068	0.03983061	0.03934515	0.03890822	0.03851825	0.03817105	0.03786241
0.65	0.046437004	0.04422649	0.04267338	0.04203025	0.04157707	0.04107698	0.04054362	0.04001332	0.03951086	0.03904959	0.03863351	0.03826112	0.03792841	0.03763137
0.66	0.04769875	0.04544933	0.04386254	0.04317159	0.0426662	0.0421146	0.04153098	0.04095151	0.04040081	0.03989215	0.03942947	0.03901133	0.03863377	0.0382929
0.67	0.048466945	0.04625374	0.04470885	0.04404652	0.04356596	0.04304023	0.04248385	0.04193267	0.041411	0.0409318	0.04049882	0.04011045	0.03976266	0.03945148
0.68	0.048082818	0.04599949	0.04457985	0.04402014	0.04362935	0.04318567	0.04270433	0.04222125	0.0417608	0.04133607	0.04095097	0.04060415	0.04029186	0.04001047
0.69	0.04825091	0.04609631	0.04461706	0.04399138	0.04353735	0.04303764	0.0425076	0.04198265	0.04148664	0.04103218	0.04062282	0.04025684	0.03993024	0.0396391
0.7	0.049068217	0.0470132	0.04562886	0.04507656	0.04468397	0.04423871	0.04375687	0.04327394	0.04281384	0.04238932	0.04200411	0.04165678	0.0413436	0.04106104
0.71	0.04969579	0.047644	0.04626721	0.04571007	0.04530941	0.04485739	0.04437043	0.04388375	0.04342102	0.04299474	0.04260852	0.04226081	0.04194786	0.0416661
0.72	0.050507491	0.0485197	0.04719957	0.04667607	0.04629565	0.04585543	0.0453727	0.04488291	0.04440999	0.04396676	0.04355719	0.04318017	0.04283244	0.04251094
0.73	0.049869875	0.04792514	0.04665451	0.04617106	0.04582998	0.04543254	0.04499612	0.04455567	0.04413465	0.04374541	0.04339156	0.04307169	0.04278225	0.04251996
0.74	0.050859348	0.04896292	0.04773165	0.04726403	0.04692474	0.04651992	0.04606788	0.04560392	0.04515182	0.04472433	0.0443255	0.04395446	0.04360821	0.04328408
0.75	0.050824938	0.04889518	0.04762992	0.04711319	0.04671874	0.04625721	0.04574745	0.04522476	0.04471293	0.04422477	0.04376446	0.04333129	0.04292252	0.04253571
0.76	0.051507931	0.04953249	0.04822591	0.047658	0.0472112	0.04670025	0.04614431	0.04557839	0.04502601	0.04449977	0.04400368	0.04353696	0.04309682	0.04268082
0.77	0.053521557	0.0514251	0.05001034	0.04933293	0.04878357	0.04818069	0.04754335	0.04690595	0.04629137	0.04571163	0.04517027	0.04466604	0.04419575	0.04375658
0.78	0.055035863	0.052862	0.05137091	0.05060514	0.04996407	0.04927034	0.04854347	0.04781778	0.0471161	0.04645044	0.04582439	0.04523683	0.04468473	0.04416547
0.79	0.055860993	0.05365647	0.05213967	0.05134012	0.05066554	0.04994234	0.0491902	0.04844306	0.04772338	0.04704284	0.0464048	0.04580793	0.04524906	0.04472546
0.8	0.054406847	0.05236381	0.05099626	0.05032177	0.04975689	0.04913236	0.04846859	0.04779976	0.04714857	0.04652698	0.04593872	0.04538287	0.04485674	0.04435806
0.81	0.054151826	0.05206903	0.05066857	0.0499557	0.04935501	0.04870077	0.04801341	0.0473266	0.04666255	0.04603281	0.04544071	0.04488506	0.04436288	0.04387169
0.82	0.052446659	0.05046471	0.04916548	0.04854247	0.04802829	0.04746076	0.04686043	0.04626056	0.04568298	0.04513891	0.04463143	0.04415912	0.04371887	0.04330804
0.83	0.053131296	0.0511819	0.04991331	0.0493079	0.04880644	0.0482505	0.04766091	0.0470708	0.04650186	0.04596518	0.04546383	0.04499643	0.04455993	0.04415179
0.84	0.053217105	0.05130191	0.05006652	0.04948248	0.0489986	0.04846002	0.04788784	0.04731494	0.0467628	0.04624237	0.04575661	0.0453041	0.0448818	0.04448722
0.85	0.053836206	0.05196414	0.05076683	0.05020557	0.0497373	0.04921088	0.04864785	0.04808106	0.04753198	0.04701158	0.04652294	0.04606481	0.04563436	0.0452293

Tenor ->	7.0	7.5	8.0	8.5	9.0	9.5	10.0	10.5	11.0	11.5	12.0	12.5	13.0	13.5	14.0
0	0.0455078	0.04553419	0.04554187	0.04553424	0.04551313	0.04547758	0.04542292	0.04534448	0.04523777	0.04509786	0.04492559	0.04472353	0.04449451	0.0442428	0.04397318
0.01	0.04539241	0.04541451	0.04541464	0.04539644	0.04536195	0.04531044	0.04523748	0.0451387	0.04500989	0.04484649	0.04464968	0.04442229	0.04416746	0.04388977	0.04359425
0.02	0.04701542	0.04700056	0.04696367	0.04690851	0.04683725	0.04674924	0.04664023	0.04650602	0.04634256	0.04614558	0.04591646	0.0456582	0.04537415	0.04506906	0.04474815
0.03	0.04670236	0.04671898	0.0467155	0.04669538	0.04666043	0.04660966	0.04653854	0.04644259	0.04631747	0.04615875	0.04596756	0.04574664	0.04549907	0.04522935	0.0449424
0.04	0.0473492	0.04737133	0.0473775	0.04737079	0.04735264	0.04732162	0.04727288	0.04720158	0.04710306	0.04697265	0.04681117	0.04662102	0.046405	0.04616724	0.04591233
0.05	0.04702985	0.04705746	0.04707313	0.04707957	0.04707783	0.04706608	0.04703913	0.04699184	0.0469192	0.04681631	0.04668371	0.04652348	0.04633809	0.04613138	0.0459076
0.06	0.04764008	0.04769166	0.04773269	0.04776565	0.0477913	0.04780757	0.04780903	0.04779036	0.04774632	0.04767194	0.04756757	0.0474351	0.04727681	0.04709635	0.04689777
0.07	0.04651385	0.04658317	0.04664282	0.04669511	0.0467406	0.04677699	0.04679875	0.04680038	0.04677654	0.04672221	0.04663763	0.04652455	0.04638517	0.04622298	0.04604189
0.08	0.04631389	0.0463507	0.04637654	0.04639393	0.04640364	0.04640353	0.04638831	0.04635274	0.04629172	0.0462006	0.04607988	0.04593157	0.04575812	0.04556331	0.04535127
0.09	0.04645442	0.04647701	0.04648702	0.04648711	0.04647817	0.04645821	0.04642213	0.04636488	0.04628156	0.04616781	0.04602435	0.04585341	0.04565766	0.04544105	0.04520794
0.1	0.04774351	0.04770522	0.0476526	0.0475887	0.04751474	0.04742906	0.04732696	0.04720382	0.04705513	0.04687703	0.0466707	0.04643876	0.04618431	0.04591172	0.04562573
0.11	0.04729811	0.0472905	0.04726843	0.04723476	0.04719057	0.047134	0.04706025	0.04696456	0.04684235	0.04668974	0.04650782	0.04629909	0.04606657	0.04581453	0.04554759
0.12	0.04722028	0.04713895	0.04704677	0.04694661	0.04683952	0.04672364	0.0465942	0.04644648	0.04627596	0.0460789	0.04585646	0.04561119	0.04534619	0.04506576	0.04477459
0.13	0.04741396	0.04735039	0.04727682	0.04719597	0.04710868	0.0470129	0.0469037	0.04677626	0.04662591	0.04644888	0.04624622	0.04602035	0.04577426	0.04551211	0.04523847
0.14	0.0477696	0.04770915	0.04764225	0.04757127	0.04749672	0.0474162	0.04732452	0.04721655	0.04708737	0.04693303	0.04675431	0.04655338	0.04633294	0.04609692	0.04584955
0.15	0.0471372	0.04704371	0.04694234	0.04683568	0.04672443	0.0466064	0.04647666	0.04633033	0.04616279	0.04597041	0.04575429	0.04551685	0.04526108	0.04499117	0.04471163
0.16	0.04705648	0.04696088	0.04686435	0.04676895	0.04667477	0.04657905	0.04647632	0.0463612	0.04622854	0.04607429	0.04589903	0.04570468	0.04549373	0.04526984	0.04503699
0.17	0.04729672	0.04718244	0.04705844	0.04692748	0.0467904	0.04664512	0.04648695	0.04631129	0.04611377	0.04589122	0.045645	0.04537782	0.04509297	0.04479487	0.04448828
0.18	0.04701287	0.04685526	0.04668539	0.04650635	0.04631931	0.04612254	0.04591173	0.04568268	0.04543143	0.04515529	0.04485605	0.04453682	0.04420128	0.04385427	0.04350095
0.19	0.04657764	0.04645438	0.04632565	0.04619384	0.04605937	0.04591981	0.04577017	0.04560558	0.04542145	0.04521449	0.04498586	0.044738	0.04447397	0.04419792	0.04391433
0.2	0.04607893	0.04591614	0.04574759	0.04557583	0.04540141	0.04522204	0.04503295	0.04482946	0.04460719	0.04436317	0.04409875	0.04381661	0.04352	0.0432133	0.04290119
0.21	0.04547258	0.04533737	0.04519952	0.04506119	0.04492252	0.04478081	0.04463093	0.04446785	0.04428688	0.04408475	0.0438625	0.04362246	0.04336754	0.04310179	0.04282951
0.22	0.04503307	0.04489737	0.04475632	0.04461226	0.04446548	0.04431345	0.04415126	0.0439741	0.04377753	0.04355859	0.04331856	0.04306	0.04278606	0.04250101	0.04220937
0.23	0.04504627	0.04492037	0.04479467	0.044671	0.04454914	0.04442602	0.04429627	0.04415463	0.0439962	0.04381765	0.04361981	0.04340478	0.04317528	0.0429351	0.04268831
0.24	0.04527033	0.04510281	0.04494136	0.04478748	0.04464063	0.04449742	0.04435221	0.04419947	0.04403406	0.04385247	0.04365527	0.04344431	0.04322207	0.04299205	0.04275806
0.25	0.04561767	0.04545042	0.045284	0.04512028	0.04495907	0.04479735	0.04462989	0.04445157	0.04425771	0.04404527	0.04381528	0.04356999	0.04331233	0.04304621	0.04277586
0.26	0.0453167	0.04512876	0.04494475	0.04476636	0.04459318	0.04442201	0.04424745	0.04406427	0.04386766	0.04365452	0.04342576	0.04318349	0.04293051	0.0426706	0.04240784
0.27	0.04564991	0.04545884	0.04526619	0.04507407	0.04488246	0.04468853	0.04448738	0.04427419	0.04404465	0.04379618	0.04353016	0.04324916	0.04295645	0.04265627	0.04235312
0.28	0.0460055	0.04577794	0.04555582	0.04534078	0.04513238	0.04492737	0.04472046	0.04450646	0.04428072	0.04404034	0.04378636	0.04352098	0.0432471	0.04296858	0.04268956
0.29	0.04511981	0.04488217	0.04465665	0.04444441	0.04424445	0.04405303	0.04386437	0.04367283	0.04347331	0.04326254	0.04304108	0.0428107	0.04257382	0.04233386	0.04209447
0.3	0.044409	0.04421407	0.04403277	0.0438659	0.04371212	0.04356734	0.0434255	0.04328067	0.04312749	0.04296247	0.0427859	0.04259927	0.04240473	0.0422054	0.04200463
0.31	0.04426772	0.04406945	0.04388378	0.04371157	0.04355151	0.04339959	0.04324983	0.04309641	0.04293411	0.04275962	0.04257335	0.04237689	0.04217253	0.04196348	0.0417532
0.32	0.04459671	0.04434223	0.04409704	0.04386244	0.04363759	0.04341892	0.04320096	0.04297843	0.04274664	0.04250284	0.04224799	0.0419842	0.04171427	0.04144192	0.04117112
0.33	0.04442725	0.04416176	0.04390576	0.04366055	0.04342527	0.04319635	0.04296836	0.04273605	0.04249481	0.04224202	0.04197868	0.04170695	0.0414297	0.04115068	0.0408739
0.34	0.04391703	0.04367715	0.04344543	0.04322311	0.04300929	0.04280034	0.04259084	0.04237556	0.04214991	0.04191136	0.04166092	0.04140077	0.04113378	0.04086372	0.0405946
0.35	0.04399149	0.0436739	0.04336296	0.04306035	0.04276555	0.04247535	0.04218479	0.04188911	0.04158422	0.0412681	0.04094226	0.04060933	0.04027267	0.0399365	0.03960529
0.36	0.04467156	0.04436329	0.0440587	0.0437596	0.04346562	0.04317372	0.04287913	0.0425773	0.04226438	0.04193862	0.04160173	0.04125659	0.04090674	0.04055662	0.04021088
0.37	0.04509602	0.04482754	0.04456267	0.04430301	0.04404797	0.04379432	0.04353713	0.0432717	0.04299407	0.04270238	0.04239823	0.04208434	0.0417641	0.04144181	0.04112196
0.38	0.04461555	0.04435265	0.04408901	0.04382652	0.04356485	0.04330104	0.04303049	0.04274884	0.04245248	0.04213995	0.04181318	0.0414752	0.04112979	0.04078153	0.04043524
0.39	0.04373707	0.04347105	0.04320105	0.04292919	0.04265534	0.04237679	0.04208922	0.04178854	0.04147145	0.04113684	0.04078693	0.04042504	0.04005522	0.03968234	0.03931148
0.4	0.04412383	0.04379462	0.04346139	0.04312647	0.04278996	0.04244942	0.04210079	0.04174029	0.04136494	0.04097397	0.04056992	0.04015639	0.03973774	0.03931912	0.0389059
0.41	0.04298128	0.04265155	0.04232161	0.04199349	0.04166694	0.04133918	0.04100587	0.04066296	0.04030723	0.03993767	0.03955656	0.03916723	0.03877375	0.038381	0.03799405
0.42	0.04323266	0.04292677	0.04262348	0.04232444	0.04202904	0.04173416	0.04143514	0.04112762	0.04080812	0.0404754	0.04013141	0.0397792	0.03942255	0.03906601	0.03871435
0.43	0.04379101	0.04345339	0.04312005	0.0427926	0.04247038	0.04215025	0.04182759	0.04149805	0.04115821	0.04080689	0.0404461	0.0400789	0.03970909	0.03934125	0.03898015
0.44	0.04331535	0.04302652	0.04274496	0.04247179	0.04220587	0.04194359	0.04167988	0.04140999	0.04113009	0.04083865	0.04053724	0.04022852	0.03991585	0.03960339	0.03929547
0.45	0.04259698	0.04229241	0.04199575	0.04170808	0.04142825	0.04115263	0.04087618	0.0405942	0.04030293	0.0400009	0.03968975	0.03937216	0.03905155	0.03873209	0.03841812
0.46	0.04137173	0.04107313	0.04078971	0.04052192	0.04026796	0.04002359	0.0397832	0.03954149	0.03929419	0.0390393	0.03877789	0.03851207	0.03824469	0.03797933	0.03771977
0.47	0.0417694	0.04147123	0.04118435	0.04090947	0.04064505	0.04038712	0.04013038	0.03986986	0.03960163	0.03932407	0.03903856	0.03874753	0.03845416	0.03816234	0.03787613
0.48	0.04085172	0.04058524	0.04032823	0.04008139	0.03984311	0.03960941	0.039375	0.03913495	0.03888538	0.03862473	0.03835442	0.0380769	0.03779537	0.03751373	0.03723606
0.49	0.03995347	0.03966555	0.03938549	0.03911413	0.03885007	0.03858951	0.03832739	0.03805902	0.03778079	0.03749142	0.03719257	0.03688694	0.03657795	0.03626975	0.03596662
0.5	0.0393673	0.03913121	0.03891055	0.03870532	0.03851324	0.0383297	0.03814883	0.03796514	0.03777428	0.0375742	0.03736579	0.03715095	0.03693231	0.0367132	0.03649712
0.51	0.03910436	0.03888221	0.03868116	0.03850066	0.0383379	0.03818771	0.03804376	0.03790009	0.03775186	0.03759662	0.03743477	0.03726772	0.03709763	0.03692733	0.03675983
0.52	0.03983156	0.0396144	0.03941886	0.0392443	0.03908777	0.03894405	0.03880674	0.03866984	0.03852851	0.03838029	0.03822554	0.03806563	0.03790267	0.03773945	0.03757889
0.53	0.04041318	0.04018856	0.03998819	0.03981121	0.03965446	0.03951253	0.03937883	0.03924725	0.0391128	0.03897292	0.03882782	0.03867872	0.03852757	0.03837697	0.03822971
0.54	0.0395541	0.03933122	0.03913605	0.0389674	0.0388218	0.03869353	0.03857575	0.03846208	0.03834732	0.03822869	0.03810613	0.03798063	0.03785385	0.03772815	0.03760603
0.55	0.03954738	0.03936126	0.03921144	0.03909583	0.03901012	0.03894773	0.038901	0.03886277	0.03882705	0.0387903	0.03875166	0.03871129	0.03867007	0.03862953	0.03859134
0.56	0.0386548	0.03845272	0.03829107	0.03816751	0.03807738	0.03801384	0.03796901	0.03793547	0.03790705	0.03787999	0.03785322	0.03782666	0.03780094	0.03777734	0.03775729
0.57	0.03831091	0.03813174	0.03799394	0.03789494	0.03782988	0.03779173	0.03777244	0.03776443	0.03776144	0.03775957	0.03775761	0.03775532	0.03775318	0.03775231	0.03775395
0.58	0.03816901	0.03798396	0.03784188	0.03774004	0.03767346	0.03763497	0.03761643	0.03761022	0.03761	0.03761184	0.03761445	0.0376175	0.03762139	0.03762715	0.03763593
0.59	0.03880277	0.03861414	0.03847192	0.03837309	0.03831237	0.03828232	0.03827456	0.03828123	0.0382958	0.03831414	0.03833473	0.038357	0.03838112	0.03840786	0.03843813
0.6	0.03965518	0.03944015	0.03926467	0.03912633	0.03902044	0.0389402	0.03887788	0.0388263	0.03877962	0.03873441	0.03868982	0.03864594	0.03860361	0.03856425	0.0385294
0.61	0.03795848	0.0378019	0.0376853	0.03760591	0.03755874	0.03753666	0.0375317	0.0375364	0.03754469	0.03755292	0.03755997	0.03756567	0.03757058	0.03757584	0.03758271
0.62	0.0385666	0.03839469	0.03826085	0.0381625	0.03809481	0.03805085	0.03802286	0.03800365	0.03798739	0.0379707	0.03795269	0.03793343	0.0379137	0.03789485	0.03787838
0.63	0.03803823	0.03782693	0.03765097	0.03750808	0.03739378	0.03730152	0.03722392	0.03715419	0.03708696	0.03701928	0.03695064	0.03688153	0.03681312	0.03674715	0.0366855
0.64	0.03758985	0.03735277	0.0371508	0.03698179	0.03684131	0.03672294	0.03661944	0.03652418	0.03643195	0.03633998	0.03624793	0.03615642	0.03606675	0.03598082	0.03590061
0.65	0.0373677	0.03713697	0.03693894	0.03677155	0.03663053	0.03650957	0.03640162	0.03630023	0.03620041	0.03609961	0.03599765	0.03589533	0.03579415	0.03569613	0.03560346
0.66	0.03798653	0.03771433	0.03747609	0.0372698	0.03709121	0.03693408	0.03679146	0.03665701	0.03652587	0.03639561	0.03626616	0.03613842	0.03601397	0.03589493	0.03578355
0.67	0.03917467	0.03893178	0.03872249	0.03854463	0.03839382	0.03826372	0.03814727	0.03803809	0.03793125	0.03782429	0.03771706	0.03761037	0.03750572	0.03740515	0.0373108
0.68	0.03975804	0.03953445	0.03933961	0.03917161	0.03902632	0.03889767	0.03877893	0.03866403	0.03854838	0.03842987	0.03830865	0.03818586	0.03806329	0.03794329	0.03782828
0.69	0.03938128	0.03915633	0.03896386	0.03880163	0.03866518	0.03854816	0.03844355	0.03834503	0.03824779	0.03814946	0.03804996	0.03795013	0.03785151	0.03775615	0.03766621
0.7	0.04080725	0.04058212	0.04038551	0.04021546	0.04006777	0.03993638	0.0398146	0.03969645	0.03957747	0.03945566	0.03933125	0.03920541	0.03908001	0.0389574	0.03884005
0.71	0.0414137	0.0411905	0.04099627	0.04082895	0.04068428	0.04055614	0.0404378	0.04032328	0.04020812	0.04009032	0.0399701	0.03984859	0.03972763	0.03960953	0.03949672
0.72	0.04221436	0.04194298	0.04169707	0.04147497	0.04127287	0.04108512	0.0409055	0.04072852	0.04055027	0.04036927	0.04018624	0.04000281	0.03982131	0.03964454	0.03947542
0.73	0.04228326	0.04207219	0.04188667	0.04172477	0.04158236	0.0414535	0.04133173	0.04121132	0.04108815	0.04096051	0.04082888	0.04069463	0.04055984	0.04042704	0.04029889
0.74	0.0429811	0.04269985	0.04244081	0.04220253	0.04198143	0.04177212	0.04156871	0.04136608	0.0411607	0.04095149	0.04073949	0.04052668	0.04031569	0.04010963	0.03991171
0.75	0.04217017	0.0418267	0.04150595	0.04120668	0.04092548	0.04065719	0.04039615	0.04013751	0.03987802	0.03961687	0.03935534	0.03909566	0.03884069	0.03859378	0.03835836
0.76	0.04228821	0.04191978	0.04157607	0.04125575	0.04095533	0.04066959	0.04039285	0.04012023	0.03984849	0.03957681	0.03930645	0.0390396	0.03877909	0.03852822	0.03829038
0.77	0.04334744	0.04296868	0.04262041	0.04230085	0.04200608	0.04173045	0.04146791	0.04121319	0.04096272	0.0407153	0.04047183	0.04023409	0.04000452	0.03978604	0.03958161
0.78	0.04367813	0.04322322	0.04280098	0.04240973	0.04204568	0.04170334	0.04137682	0.04106106	0.04075269	0.04045071	0.0401562	0.03987112	0.03959808	0.03934015	0.03910046
0.79	0.04423608	0.04378127	0.04336107	0.04297363	0.04261495	0.0422794	0.04196095	0.04165441	0.04135632	0.04106557	0.04078311	0.04051075	0.04025098	0.04000672	0.03978094
0.8	0.04388628	0.04344221	0.04302628	0.04263707	0.04227101	0.04192289	0.04158718	0.04125918	0.04093592	0.0406168	0.04030324	0.03999753	0.03970262	0.03942189	0.03915877
0.81	0.04341068	0.04298038	0.04258093	0.04221058	0.04186546	0.04154011	0.04122871	0.04092636	0.04062985	0.04033837	0.04005307	0.03977601	0.03950987	0.03925775	0.03902285
0.82	0.04292569	0.04257216	0.0422474	0.04194944	0.04167421	0.04141607	0.04116907	0.04092816	0.04069002	0.04045371	0.04022024	0.0399915	0.03977002	0.03955875	0.0393607
0.83	0.04377115	0.04341843	0.04309358	0.04279467	0.04251766	0.04225697	0.04200673	0.04176199	0.04151955	0.04127857	0.04104015	0.04080626	0.0405795	0.0403629	0.04015952
0.84	0.04411951	0.04377907	0.04346584	0.04317781	0.04291093	0.04265961	0.042418	0.04218119	0.04194602	0.04171169	0.04147933	0.04125091	0.04102906	0.04081678	0.04061717
0.85	0.04484903	0.04449412	0.04416467	0.04385885	0.04357274	0.04330097	0.04303788	0.04277882	0.04252087	0.04226346	0.04200794	0.0417565	0.04151196	0.04127754	0.04105653

Appendix C
Numerical Simulation Driver

```
409     int blockSize = payOff.dtau / dt;
410
411     std::cout << "intermediate simulations " << blockSize << "number of paths" << pathN << std::endl;
412
413     for (int path = 0; path < pathN; path += blockSize)
414     {
415         // open mp parallel region
416         for (int s = 0; s < exposuresCount; s++)
417         {
418             // internal openmp vectorization
419             __generatePaths_kernel_cpu(
420                 &numeraires[s * _TIMEPOINTS],
421                 _TIMEPOINTS,
422                 drift,
423                 volatilities,
424                 &rngNrmVar[s * pathN * 3],
425                 spot_rates,
426                 &simulated_rates[s * _TIMEPOINTS],
427                 &simulated_rates0[s * _TIMEPOINTS],
428                 &accum_rates[s * _TIMEPOINTS],
429                 pathN,
430                 path
431             ); //dt = 0.01, dtau = 0.5
432         }
433
434         // update simulated rates (swap pointers)
435         std::swap(simulated_rates, simulated_rates0);
436     }
437
438     t_end = std::chrono::high_resolution_clock::now();
439     elapsed_time_ms = std::chrono::duration<double, std::milli>(t_end - t_start).count();
440     std::cout << "total time taken to run all " << pathN * exposuresCount << " HJM MC simulation " << elapsed_time_ms << "(ms)" << std::endl;
```

Appendix D
Naïve Parallelized Numerical Simulation Algorithm

```
720  /*
721  * Risk Factor Generation naive acceleration
722  */
723
724  void riskFactorSim(
725      int gridSize,
726      int blockSize,
727      int numberOfPoints,
728      double2* numeraires,
729      double* rngNrmVar,
730      double* simulated_rates,
731      double* simulated_rates0,
732      double* accum_rates,
733      const int pathN,
734      double dtau = 0.5,
735      double dt = 0.01)
736  {
737
738      for (int path = 0; path < pathN; path++)
739      {
740          __generatePaths_kernel <<< gridSize, blockSize >>> (
741              numberOfPoints,
742              numeraires,
743              rngNrmVar,
744              simulated_rates,
745              simulated_rates0,
746              accum_rates,
747              pathN,
748              path,
749              dtau,
750              dt
751          );
752
753          CUDA_RT_CALL(cudaDeviceSynchronize());
754
755          // update simulated rates (swap pointers)
756          std::swap(simulated_rates, simulated_rates0);
757      }
758
759  }
```

Appendix E
Accelerated Numerical Simulation Algorithm

```
590  /*
591  * Risk Factor Generation block simulation  with Shared Memory
592  */
593
594  void riskFactorSim4(
595      int gridSize,
596      int blockSize,
597      int numberOfPoints,
598      double2* numeraires,
599      double* rngNrmVar,
600      double* simulated_rates,
601      double* simulated_rates0,
602      double* accum_rates,
603      const int pathN,
604      double dtau = 0.5,
605      double dt = 0.01)
606  {
607      int simBlockSize = dtau / dt;
608
609      for (int path = 0; path < pathN; path += simBlockSize)
610      {
611          __generatePaths_kernel4 <<< gridSize, blockSize >>> (
612              numberOfPoints,
613              numeraires,
614              rngNrmVar,
615              simulated_rates,
616              simulated_rates0,
617              accum_rates,
618              pathN,
619              path,
620              dtau,
621              dt
622          );
623
624  #ifdef CUDA_SYNC
625          CUDA_RT_CALL(cudaDeviceSynchronize());
626  #endif
627
628          // update simulated rates (swap pointers)
629          std::swap(simulated_rates, simulated_rates0);
630      }
631  }
```

Appendix F
Accelerated Vanilla Interest Rate Swap Mark to Market

```
__global__
void _exposure_calc_kernel(real* exposure, real2* numeraires, real notional, real K, int simN, real dtau = 0.5f)
{
    __shared__ real cash_flows[TIMEPOINTS];
    real discount_factor;
    real forward_rate;
    real libor;
    real cash_flow;
    real sum = 0.0;
    real m = (1.0 / dtau);

    int globaltid = blockIdx.x * TIMEPOINTS + threadIdx.x;

    // calculate and load the cash flow in shared memory
    if (threadIdx.x < TIMEPOINTS) {
        forward_rate = numeraires[globaltid].x;

#ifdef SINGLE_PRECISION
        libor = m * (expf(forward_rate / m) - 1.0);
#else
        libor = m * (exp(forward_rate / m) - 1.0);
#endif
        discount_factor = numeraires[globaltid].y;
        cash_flow = discount_factor * notional * d_accrual[threadIdx.x] * (libor - K);
        cash_flows[threadIdx.x] = cash_flow;

#ifdef EXPOSURE_PROFILES_DEBUG
        printf("Block %d Thread %d Forward Rate %f libor %f Discount %f CashFlow %f \n", blockIdx.x, threadIdx.x, forward_rate, libor, discount_factor, cash_flow);
#endif
    }
    __syncthreads();

#ifdef EXPOSURE_PROFILES_DEBUG2
    if (threadIdx.x == 0) {
        for (int t = 0; t < TIMEPOINTS; t++) {
            printf("t - indext %d CashFlow %f \n", t, cash_flows[t]);
        }
    }
#endif

    // calculate the exposure profile
    if ( threadIdx.x < TIMEPOINTS )
    {
        for (int t = threadIdx.x + 1; t < TIMEPOINTS; t++) {
            sum += cash_flows[t];
        }
        sum = (sum > 0.0) ? sum : 0.0;
        exposure[globaltid] = sum;

#ifdef EXPOSURE_PROFILES_DEBUG
        printf("Block %d Thread %d Exposure %f \n", blockIdx.x, threadIdx.x, sum);
#endif
    }
    __syncthreads();
}
```

Appendix G
Simulation kernel naïve implementation

```
361  template <typename real, typename real2>
362  __global__
363  void __generatePaths_kernel(
364      int numberOfPoints,
365      real2* numeraires,
366      real* rngNrmVar,
367      real* simulated_rates,
368      real* simulated_rates0,
369      real* accum_rates,
370      const int pathN, int path,
371      real dtau = 0.5, double dt = 0.01)
372      {
373          // calculated rate
374          real rate;
375          real sum_rate;
376
377  #ifdef RNG_HOST_API
378          real phi0;
379          real phi1;
380          real phi2;
381  #endif
382
383          // Simulation Parameters
384          int stride = dtau / dt; //
385          real sqrt_dt = sqrtf(dt);
386
387          int t = threadIdx.x % TIMEPOINTS;
388          int gindex = blockIdx.x * numberOfPoints + threadIdx.x;
389
390          // Evolve the whole curve from 0 to T ( 1:1 mapping t with threadIdx.x)
391          if ((threadIdx.x < numberOfPoints) && (gindex < gridDim.x * numberOfPoints))
392          {
393              if (path == 0) {
394                  rate = d_spot_rates[t];
395              }
396              else {
397                  // Calculate dF term in Musiela Parametrization SDE
398                  real dF = 0;
399
400                  if (t == (TIMEPOINTS - 1)) {
401                      dF = simulated_rates[gindex] - simulated_rates[gindex - 1];
402                  }
403                  else {
404                      dF = simulated_rates[gindex + 1] - simulated_rates[gindex];
405                  }
406
407                  // Normal random variates
408  #ifdef RNG_HOST_API
409                  real* rngNrms = (real*)rngNrmVar;
410                  int rndIdx = blockIdx.x * pathN * VOL_DIM + path * VOL_DIM;
411                  phi0 = rngNrms[rndIdx];
412                  phi1 = rngNrms[rndIdx + 1];
413                  phi2 = rngNrms[rndIdx + 2];
414  #else
415                  if (threadIdx.x == 0) {
416                      curandStateMRG32k3a* state = (curandStateMRG32k3a*)rngNrmVar;
417                      curandStateMRG32k3a localState = state[blockIdx.x];
418                      phi0 = curand_uniform(&localState);
419                      phi1 = curand_uniform(&localState);
420                      phi2 = curand_uniform(&localState);
```

```
421                     state[blockIdx.x] = localState;
422                 }
423                 __syncthreads();
424 #endif
425 
426                 // simulate the sde
427                 rate = __musiela_sde2(
428                     d_drifts[t],
429                     d_volatilities[t],
430                     d_volatilities[TIMEPOINTS + t],
431                     d_volatilities[TIMEPOINTS * 2 + t],
432                     phi0,
433                     phi1,
434                     phi2,
435                     sqrt_dt,
436                     dF,
437                     simulated_rates[gindex],
438                     dtau,
439                     dt
440                 );
441             }
442 
443 #ifdef HJM_PATH_SIMULATION_DEBUG
444             printf("Path %d Block %d Thread %d index %d Forward Rate %f phi0 %f phi1 %f phi2 %f \n",
445             path, blockIdx.x, threadIdx.x, gindex, rate, phi0, phi1, phi2);
446 #endif
447             // accumulate rate for discount calculation
448             sum_rate = accum_rates[gindex];
449             sum_rate += rate;
450             accum_rates[gindex] = sum_rate;
451 
452             // store the simulated rate
453             simulated_rates0[gindex] = rate; //
454 
455             // update numeraire based on simulation block
456             if (path % stride == 0) {
457                 if (t == (path / stride)) {
458                     numeraires[gindex].x = rate;
459                     numeraires[gindex].y = __expf(-sum_rate * dt);
460 #ifdef HJM_NUMERAIRE_DEBUG
461                     printf("Path %d Block %d Thread %d index %d Forward Rate %f Discount %f\n",
462                     path, blockIdx.x, threadIdx.x, gindex, rate, __expf(-sum_rate * dt));
463 #endif
464                 }
465             }
466         }
467     }
468 
469 /**
470 * Shared Memory & Global Access Memory optimizations & block simulation
471 */
```

Appendix H
Simulation kernel implementation with shared memory

```
473  template <typename real, typename real2>
474  __global__
475          void __generatePaths_kernel4(
476              int numberOfPoints,
477              real2* numeraires,
478              real* rngNrmVar,
479              real* simulated_rates,
480              real* simulated_rates0,
481              real* accum_rates,
482              const int pathN,
483              int path,
484              real dtau = 0.5, real dt = 0.01)
485      {
486          // calculated rate
487          real rate;
488          real sum_rate = 0;
489          real phi0;
490          real phi1;
491          real phi2;
492
493          __shared__ real _ssimulated_rates[BLOCK_SIZE];
494
495          // Simulation Parameters
496          int stride = dtau / dt; //
497          real sqrt_dt = sqrtf(dt);
498
499          //int t = threadIdx.x % TIMEPOINTS;
500          int t = threadIdx.x;
501          int gindex = blockIdx.x * numberOfPoints + threadIdx.x;
502
503          // load the accumulated rate for a given timepoint
504          sum_rate = accum_rates[gindex];
505
506          // load the latest simulated rate from global memory
507          if ((threadIdx.x < numberOfPoints) && (gindex < gridDim.x * numberOfPoints)) {
508              if ( path > 0) {
509                  _ssimulated_rates[threadIdx.x] = simulated_rates[gindex];
510              }
511          }
512          __syncthreads();
513
514          //
515          for (int s = 0; s < stride; s++)
516          {
517              if ((threadIdx.x < numberOfPoints) && (gindex < gridDim.x * numberOfPoints))
518              {
519                  if (path == 0) {
520                      rate = d_spot_rates[t];
521                  }
522                  else {
523                      // Calculate dF term in Musiela Parametrization SDE
524                      real dF = 0;
525
526                      if (t == (TIMEPOINTS - 1)) {
527                          dF = _ssimulated_rates[threadIdx.x] - _ssimulated_rates[threadIdx.x - 1];
528                      }
529                      else {
530                          dF = _ssimulated_rates[threadIdx.x + 1] - _ssimulated_rates[threadIdx.x];
531                      }
532
533                      // Normal random variates broadcast if access same memory location in shared memory
```

```
                real* rngNrms = (real*)rngNrmVar;
                int rndIdx = blockIdx.x * pathN * VOL_DIM + (path + s)* VOL_DIM;
                phi0 = rngNrms[rndIdx];
                phi1 = rngNrms[rndIdx + 1];
                phi2 = rngNrms[rndIdx + 2];

                // simulate the sde
                rate = __musiela_sde2(
                    d_drifts[t],
                    d_volatilities[t],
                    d_volatilities[TIMEPOINTS + t],
                    d_volatilities[TIMEPOINTS * 2 + t],
                    phi0,
                    phi1,
                    phi2,
                    sqrt_dt,
                    dF,
                    _ssimulated_rates[threadIdx.x],
                    dtau,
                    dt
                );
            }

            // accumulate rate for discount calculation
            sum_rate += rate;

        }

        __syncthreads();

        if ((threadIdx.x < numberOfPoints) && (gindex < gridDim.x * numberOfPoints))
        {
            _ssimulated_rates[threadIdx.x] = rate;
        }

        __syncthreads();

    }

    // update the rates and the rate summation for the next simulation block
    if ((threadIdx.x < numberOfPoints) && (gindex < gridDim.x * numberOfPoints))
    {
        simulated_rates0[gindex] = rate;
        accum_rates[gindex] = sum_rate;
    }

    // update numeraire based on simulation block
    if ( t == (path + stride) / stride ) {
        numeraires[gindex].x = rate;  // forward rate
#ifdef double_ACC
      numeraires[gindex].y = expf(-sum_rate * dt);
#else
      numeraires[gindex].y = exp(-sum_rate * dt);
#endif
        }
    }
```

Appendix I
Multi-gpu implementation

```
#ifdef MULTI_GPU_SIMULATION
#pragma omp parallel num_threads(num_gpus)
    {
        int gpuDevice = omp_get_thread_num();
#endif
        cudaSetDevice(gpuDevice);

        // Kernel Execution Parameters
        int N = 1;
        int blockSize = N * BLOCK_SIZE;
        int numberOfPoints = N * TIMEPOINTS;
        int gridSize = scenarios_gpus / N;

        nvtxRangePush("Simulation");

        TIMED_RT_CALL(
            riskFactorSim4(
                gridSize,
                blockSize,
                numberOfPoints,
                d_numeraire[gpuDevice],
                rngNrmVar[gpuDevice],
                simulated_rates[gpuDevice],
                simulated_rates0[gpuDevice],
                accum_rates[gpuDevice],
                pathN,
                payOff.dtau,
                dt
            ),
            "Execution Time Partial HJM MC simulation"
        );

        // TRACE main
        nvtxRangePop();

        nvtxRangePush("Pricing");
        // Exposure Profile Calculation  TODO (d_exposures + gpuDevice * TIMEPOINTS)
        // Apply Scan algorithm here
        TIMED_RT_CALL(
            exposureCalculation(scenarios_gpus, BLOCK_SIZE, d_exposures[gpuDevice], d_numeraire[gpuDevice], payOff.notional, payOff.K, scenarios_gpus),
            "exposure calculation"
        );
        nvtxRangePop();

        //Replace all this block by a column reduction of the matrix
        // Partial Expected Exposure Calculation and scattered across gpus
        nvtxRangePushA("Exposure");

        TIMED_RT_CALL(
            __expectedexposure_calc_kernel(partial_exposure[gpuDevice], d_exposures[gpuDevice], d_x[gpuDevice], d_y[gpuDevice], cublas_handle[gpuDevice], scenarios_gpus),
            "partial expected exposure profile"
        );
        nvtxRangePop();

#ifdef MULTI_GPU_SIMULATION
    }
#endif
```

Appendix J
System Configuration

Software Platform

OS	Windows 10.0.1.19041 x86_64
NVIDIA SDK	NVIDIA SDK 10.0
Libraries	Intel MKL, cuBLAS, cuRAND

CPU

Description	Intel® Xeon® W-2123 CPU @ 3.60GHz
Number of Cores	8
Memory	64GB

GPU

Description	GeForce RTX 2070 SUPER
Number of GPU	4
GPU Chip	TU104
SM Count	40
Device Memory Size	8GB
Device Memory Theoretical B/W	427.31 GiB/s
Clock Rate	1.80GHz

Visual Studio C++ Compiler Options

Host C/C++	/GS /GL /W3 /Zc:wchar_t /I"./" /I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2\include" /I"../../common/inc" /I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2\include" /I"C:\Program Files (x86)\IntelSWTools\compilers_and_libraries\windows\mkl\include" /Zi /Gm- /O2 /Fd"x64/Release/vc142.pdb" /Zc:inline /fp:precise /D "WIN32" /D "_MBCS" /errorReport:prompt /WX- /Zc:forScope /Gd /MT /FC /Fa"x64/Release/" /EHsc /nologo /Fo"x64/Release/" /Fp"x64/Release/CVA_IRS.pch" /diagnostics:column
CUDA C/C++	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2\bin\nvcc.exe" -gencode=arch=compute_75,code=\"sm_75,compute_75\" --use-local-env -ccbin "C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.27.29110\bin\HostX86\x64" -x cu -I./ -I../../common/inc -I./ -I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2\include" -I../../common/inc -I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2\include" -I"C:\Program Files (x86)\IntelSWTools\compilers_and_libraries\windows\mkl\include" -I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2\include" --keep-dir x64\Release -maxrregcount=0 --machine 64 --compile -cudart static -Xcompiler "/openmp /wd 4819" -lineinfo -lnvToolsExt -DWIN32 -DWIN32 -D_MBCS -D_MBCS -Xcompiler "/EHsc /W3 /nologo /O2 /Fdx64/Release/vc142.pdb /FS /Zi /MT " -o x64/Release/simulation_GPU.cu.obj "C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\7_CUDALibraries\MC_SingleAsianOptionP\src\simulation_gpu.cu"

Appendix K
Full Code Implementation

The code implementation used for this work is publicly available at https://github.com/mahsanchez/cva_hjm_cuda