Local Volatility FX Basket Option on CPU and GPU

Jacques du Toit[1] and Isabel Ehrlich[2]

Abstract

We study a basket option written on 10 FX rates driven by a 10 factor local volatility model. We price the option using Monte Carlo simulation and develop high performance implementations of the algorithm on a top end Intel CPU and an NVIDIA GPU. We obtain the following performance figures:

	Price	Runtime(ms)	Speedup
CPU	0.5892381192	7,557.72	1.0x
GPU	0.5892391192	698.28	10.83x

The width of the 98% confidence interval is 0.05% of the option price. We confirm that the algorithm runs stably and accurately in single precision, and this gives a 2x performance improvement on both platforms. Lastly, we experiment with GPU texture memory and reduce the GPU runtime to 153ms. This gives a price with an error of 2.60e-7 relative to the double precision price.

1 Introduction

Financial markets have seen an increasing number of new derivatives and options available to trade. As these options are ever more complex, closed form solutions for pricing are often not readily available and we must rely on approximations to obtain a fair price. One way in which we can accurately price options without a closed form solution is by Monte Carlo simulation, however this is often not the preferred method of pricing due to the fact that it is computationally demanding (and hence time-consuming). The advent of massively parallel computer hardware (multicore CPUs with AVX, GPUs, Intel Xeon Phi also known as Intel Many Integrated Core[3] or MIC) has created renewed interest in Monte Carlo methods, since these methods are ideally suited to the hardware. For many financial products, Monte Carlo methods can now give accurate prices fast enough to be useful to a financial institution's activities. This article will examine a traditionally more demanding product, namely an FX basket option driven by a multi-factor local volatility model. The aim is to evaluate the performance of this product on two computing platforms: a top end Intel CPU and an NVIDIA GPU. In particular, we study the impact of mixed precision programming both on speed and accuracy. Many practitioners still do most of their calculations in double precision, even for Monte Carlo simulation. Our aim is to quantify the performance benefits of mixed precision, as well as the impact on stability and accuracy.

2 Basket options and analytic approximations

Basket options provide a perfect example of options that do not have a readily available closed form solution. In addition, the available approximations have several major shortcomings. A basket option is a multidimensional derivative whose payoff is dependent on

[1]Numerical Algorithms Group
[2]Imperial College London
[3]Xeon Phi is the product name for Intel's many integrated core (MIC) technology

the value of a weighted sum of N assets (called the *underlyings*). This weighted sum means that the basket's distribution is unknown. There have been a number of attempts in the literature to find approximations to the price of a basket option. Broadly speaking, these can be split into two groups: models that assume constant volatility, and models that try to remain consistent with the volatility smile. Models in the first group typically try to approximate the distribution of the basket with a more analytically tractable distribution. Levy [11] suggests that the distribution of an arithmetic average is well-approximated by the lognormal distribution when the underlying price process follows geometric Brownian motion. Other approximations include the reciprocal gamma distribution suggested by Milevsky and Posner [12] or a member of the Johnson family of distributions. These approximations are frequently not appropriate for pricing basket options since they fail to capture and replicate the observed market smile. In addition, it has been shown that the prices obtained from these approximations are not reliable: the approximations are based on matching the basket's true distribution to that of a more tractable distribution and therefore fail to capture the effects of correlation between the underlying assets. This is a major shortcoming since the effect of correlation is what makes the basket option valuable. In a perfectly correlated market, pricing the basket is tantamount to pricing N call options. However in all other cases the price of the basket will be less than the sum of N call options while still providing protection against falls in asset values. Finally, these approximations require the weights to be strictly positive. This is not an issue for Monte Carlo simulation.

The second group consists of models which try to capture the observed market smile. Examples of these include Brigo and Mercurio's [2] lognormal-mixture model, Huynh's [8] Edgeworth series expansion model and Xu and Zheng's [13] local volatility jump diffusion model, amongst others. Many of the approximations arising from these models fail to be of use when dealing with a large number of assets. This is mainly due to the amount of computation required to build and calibrate the models. For example the lognormal-mixture model typically requires a large number of densities which rapidly (exponentially) increases as the number of underlyings increases, thus making the model too expensive. Once again all these approximations require that the weights remain positive.

Separate from the approximation literature is a rapidly growing collection of more general models of asset dynamics which try to capture the smile (and sometimes the smile dynamics). The best known of these are probably the Heston model (with term structure), the local volatility model and the stochastic local volatility models. These are general models of asset behaviour and are typically too complex for extracting approximation formulae. To obtain option prices these models have to be tackled numerically, either by (quasi) Monte Carlo simulation, by PDE techniques, or by other methods (e.g. semi-analytic formulae and transformation identities). Traditionally the cost of these numerical methods has meant that such general asset models have not been used for pricing basket options. Our performance results suggest that Monte Carlo may now be a feasible pricing method for baskets driven by these models.

3 Local volatility FX basket option

The model we will focus on is the local volatility model. This is a flexible model which is widely used in practice, and can yield arbitrage free call option prices when paired with arbitrage-free interpolation methods. As such we follow the work of Dupire [4] with a few modifications to remain consistent with the conventions in the FX market.

A basket call option consists of a weighted sum of N correlated assets and has a price given by

$$C = e^{-r_d T}\mathbb{E}\left(B_T - K\right)^+ \tag{1}$$

where r_d is the domestic risk free interest rate, $K > 0$ is the strike price, and B_T is given by

$$B_T = \sum_{i=1}^{N} w^{(i)} S_T^{(i)} \,. \tag{2}$$

Here $S^{(i)}$ denotes the value of i^{th} underlying asset (FX rate) for $i = 1, \ldots, N$ and the $w^{(i)}$s are a set of weights with $\sum_{i=1}^{N} w^{(i)} = 1$. In the local volatility model, the evolution of each underlying asset is governed by the stochastic differential equation (SDE)

$$\frac{dS_t^{(i)}}{S_t^{(i)}} = \big(r_d - r_f^{(i)}\big)dt + \sigma^{(i)}\big(S_t^{(i)}, t\big)dW_t^{(i)} \tag{3}$$

where $r_f^{(i)}$ is the foreign risk free interest rate for the i^{th} currency pair, $(\mathbf{W}_t)_{t\geq 0}$ is a correlated N-dimensional Brownian motion with $\big\langle W^{(i)}, W^{(j)} \big\rangle_t = \rho^{(i,j)} t$ and $\rho^{(i,j)}$ denotes the correlation coefficient for $i, j = 1, \ldots, N$.

The function $\sigma^{(i)}$ in (3) above is called the *local volatility* of the i^{th} asset, and is typically unknown. Since $\sigma^{(i)}$ depends on both time and the current asset level, one might expect that the local volatility model could capture the observed market smile, and indeed it can be shown that this is true. Following Dupire [4], let $C_{BS}(S, K, T, r_f, r_d, \theta)$ denote the Black-Scholes price of a call option with maturity T and strike K written on an underlying of price S so that

$$C_{BS} \equiv C_{BS}(S, K, T, r_f, r_d, \theta) = Se^{-r_f T} N(d_1) - Ke^{-r_d T} N(d_2) \tag{4}$$

where θ denotes the volatility, $d_{1,2}$ are given by

$$d_{1,2} = \frac{\ln(S/K) + (r_d - r_f \pm \frac{1}{2}\theta^2)T}{\theta\sqrt{T}} \tag{5}$$

and $N(x)$ denotes the cumulative Normal distribution function evaluated at $x \in \mathbb{R}$. Then it can be shown (suppressing the superscript (i)) that the local volatility $\sigma(K, T)$ in (3) is given by the Dupire formula

$$\sigma^2(K, T) = \frac{\theta^2 + 2T\theta\theta_T + 2\big(r_T^d - r_T^f\big)KT\theta\theta_K}{\big(1 + Kd_1T\theta_K\big)^2 + K^2T\theta\big(\theta_{KK} - d_1T\theta_K^2\big)}. \tag{6}$$

In the context of (6) θ is called the Black-Scholes *implied volatility* and is defined as the unique value yielding equality in (4) for some market observables C_{BS}, S, K, T, r_f, r_d given and fixed. The implied volatility θ is thus viewed as an *implicit function* of K and T (as well as the other variables), and the Dupire formula assumes that $\theta \equiv \theta(K, T) \in C^{2,1}$. One therefore takes the quotes C_{BS} observed in the market for different strikes K and maturities (tenors) T and inverts (4) to find the corresponding implied volatilities $\theta(K, T)$. This function is referred to as the *volatility surface* or *the smile.*

Note that (6) implies there are infinitely many implied volatility quotes. However the market only trades a handful of call options at different strikes and maturities. The task is therefore to create a $C^{2,1}$ function from this fairly small set of discrete quotes which can be used in (6). Doing this in an arbitrage-free manner is by no means trivial especially when working in the FX market. This has been a topic of much research over the last decade (see e.g. Andreasen and Huge [1], Kahalé [9], Fengler [6] and Glaser and Heider [7] among many others). A common approach taken by market practitioners is to ignore the finer points of arbitrage free interpolation and simply to use cubic splines. This is the approach that we have adopted and although simple, from a computational point of view we believe the method has enough complexity to show how the hardware platforms would perform with a more sophisticated interpolation method.

3.1 Monte Carlo scheme

To solve the SDE (3) we use an Euler-Maruyama scheme on the log process. We use n_T time steps each of length Δt so that $n_T\Delta t = T$, the maturity of the basket option. With $t_\tau = \tau\Delta t$ for $\tau = 1, \ldots, n_T$ the discretised version of (3) then becomes

$$\log\left(\frac{S^{(i)}_{t_{\tau+1}}}{S^{(i)}_0}\right) - \log\left(\frac{S^{(i)}_{t_\tau}}{S^{(i)}_0}\right) = \left(r_d - r^{(i)}_f - \tfrac{1}{2}\sigma^{(i)}(S^{(i)}_{t_\tau}, t_\tau)^2\right)\Delta t + \sigma^{(i)}(S^{(i)}_{t_\tau}, t_\tau)\sqrt{\Delta t}\,\mathbf{d_i}\mathbf{Z}_\tau \quad (7)$$

where $\mathbf{Z}_\tau$ is an $N \times 1$ column vector of standard Normal random numbers and $\mathbf{d_i}$ is the i^{th} row of the Cholesky factorisation of the $N \times N$ correlation matrix, in other words $\mathbf{d_i}\mathbf{d_j}^T = \rho^{(i,j)}$ for all $i, j = 1, \ldots, N$.

The local volatility $\sigma^{(i)}(S^{(i)}_{t_\tau}, t_\tau)$ is computed by (6) which in turn requires the implied volatility (and its derivatives) at the point $(S^{(i)}_{t_\tau}, t_\tau)$ for all $\tau = 1, \ldots, n_T$. This calculation must therefore be performed for each sample path at each time step, and comprises the bulk of computation.

3.2 Constructing the implied volatility surface

For the sake of completeness, we briefly describe how we construct the implied volatility surface, or more precisely, the sets of interpolating splines which represent the surface. This is part of the setup process which precedes the Monte Carlo simulation, and these calculations are not reflected in our runtimes.

As mentioned, the local volatility functions $\sigma^{(i)}$ must be calculated from the implied volatility surface observed in the market. Typically the FX market provides quotes for five "deltas" per tenor, and quotes are given in "delta space" not strike space. The FX market has its own unique conventions and price quoting methods: for example there are four different delta types. For further details on how to determine which delta is in use and how to convert from delta space to strike space, please refer to Clark [3].

Once we have transformed the market quotes from delta space to strike space we can use them to build a smooth implied volatility surface. For this we use cubic splines. Recall that the market implied volatility surface consist of a collection of tenors, and at each tenor there are 5 different implied volatility quotes given at 5 different strikes. The first step is to fit a cubic spline in the strike direction through the 5 quotes at each tenor.

Each of the tenors is now treated as follows. Fix a particular tenor. We fit 5 piecewise monotonic Hermite polynomials from time 0 to time T, with each polynomial passing through one of the quotes at this tenor. Since each tenor's quotes are at different strikes, this requires interpolating values with the previously fitted cubic splines. For all the Monte Carlo time steps $\tau\Delta t$ that lie between this tenor and the previous tenor, we use the Hermite polynomials to interpolate implied volatility values at each of the 5 strikes. Finally for each of these $\tau\Delta t$, cubic splines are fitted in the strike direction through the 5 implied volatility values just computed. The slopes at the end points of these splines are projected to form linear extrapolation functions to cover the left and right (in the strike direction) tails of the implied volatility surface.

Repeating this procedure for each tenor yields a sequence of cubic splines in the strike direction, one at each Monte Carlo time step $\tau\Delta t$ for $\tau = 1, \ldots, n_T$, which can be used to compute θ, θ_K and θ_{KK} in (6). Essentially the same procedure can be used to compute θ_T. Note that exact spline derivatives are computed – finite difference approximations are not used. As we use real market data (which is not guaranteed to be arbitrage free) together with rather ad-hoc interpolation and extrapolation techniques, we do encounter negative local volatilities which implies the existence of arbitrage in our model. This can be corrected by using an arbitrage-free interpolation method, e.g. Fengler [6].

4 Implementing the model

The main computational burden consists of evaluating the Dupire formula (6) for each sample path at each time step. For this we need the implied volatility θ as well as several of its derivatives, which are computed from two sets of cubic splines (one for $\theta, \theta_K, \theta_{KK}$ and another for θ_T). Each Monte Carlo time step therefore has associated with it 2 cubic splines for each of the N underlyings.

From the point of view of model implementation, the main difference between FX markets and equity markets is the relative paucity of market quotes in the former: as mentioned, there are only 5 quotes per tenor, whereas liquid equity products can easily have 15 or more quotes per tenor. This means that the splines used for equity products are much bigger than the splines used for FX products. For example, in our FX model all the spline data for a single underlying only requires about 100KB storage. Many of the splines can therefore fit into L1 cache (or shared memory on a GPU). An equity model would require much more storage, which in turn has cache implications.

One implementation option is to process a sample path through all time steps before moving on to the next sample path, since all the splines essentially fit in cache. However this approach will not work (especially on a GPU) when dealing with equity assets. For this reason we chose to bring the splines for a particular Monte Carlo time step into cache and then advance all the sample paths to the next time step. This results in much more traffic back and forth to main memory, but since there are many floating point calculations this traffic should effectively be hidden.

Note from (7) that it is possible to deal with each asset individually when doing the Euler-Maruyama time stepping. It is only when computing the payoff (2) that all the N assets are required at the same time.

4.1 GPU implementation

The GPU code is written in CUDA. Normal random numbers are generated in GPU memory using the MRG32k3a random number generator from the NAG Numerical Routines for GPUs. The ordering is chosen to match the order of the standard serial algorithm[4] of the generator (see L'Ecuyer [5]) as this greatly eases debugging. The remainder of the calculation is split into two kernels: one to advance all the sample paths from time 0 to time T, and the other to compute the payoff. All the cubic splines and the input data are copied to GPU memory. In the sample path kernel each thread block deals with only one asset (but there are multiple thread blocks per asset). At the start of the kernel the corresponding row $\mathbf{d_i}$ of the Cholesky factorisation of the correlation matrix (see (7) above) is brought into shared memory[5]. At each time step, the corresponding cubic spline is brought into shared memory and is used to advance all the sample paths to the next time step.

4.2 CPU implementation

The CPU code is written in C and we took considerable trouble to ensure that we used the Advanced Vector Extension (AVX) vector units on the CPU as much as possible. We wrote a fully vectorised, fully parallelised MRG32k3a implementation which returns numbers in the same order as the standard serial MRG32k3a algorithm. For the path calculation we tried two different approaches. The first is essentially the same as the

[4]The NAG GPU generator can maintain the original serial order, or return numbers in a permuted order aimed at peak performance. The permuted order is faster. For more information see the NAG Numerical Routines for GPUs documentation.

[5]We did investigate using CuBLAS (dtrmm) to correlate the Normal random numbers, but this turned out to be slower than correlating the random numbers inside the sample paths kernel

GPU implementation: all the sample paths are advanced from one time step to the next. Through a combination of elemental functions (see the Intel Compiler Documentation [10]) and manual loop unrolling it is possible to vectorise the loop over all sample paths. The kernel therefore has an outer loop over all time steps, with a fully vectorised and parallelised inner loop over sample paths. The second approach swaps the two loops: the outer loop is over all sample paths, with the inner loop over all time steps. The idea here is to vectorise the outer loop, however we were unable to persuade the compiler to do this even though we tried several different options[6]. In the end, the first approach was faster.

We note that due to the manual loop unrolling, the CPU algorithm is hard-coded to a basket with 10 underlyings (the problem size we chose to study) and is therefore not a general purpose code. The GPU code handles any number of underlyings. In addition, we note that vectorising the sample path loop reduced the double precision runtime by 43%, even though each vector unit can process 4 doubles at once.

5 Results

We started off with a pure double precision code and gradually replaced various parts of the calculations with single precision to see the effect on accuracy and speed. Most modern processors will handle single precision calculations at least twice as fast as double precision calculations. Indeed, NVIDIA's Kepler series of compute cards have three times higher single precision performance than double precision. Many finance practitioners admit to doing all their calculations in double precision but don't seem to have a clear idea why. This is true even of Monte Carlo codes. Given that Monte Carlo integration is an inherently random procedure, which gives little more than a point estimate and a confidence interval for the true answer, it seems questionable that the extra accuracy of double precision is adding anything meaningful to the answers. The concern among practitioners seems to be about numerical stability, and whether mixed precision algorithms offer enough of a speedup to warrant the effort. These are questions we set out to explore.

5.1 Test problem

We considered a 1 year basket option on 10 currency pairs: EURUSD, AUDUSD, GBPUSD, USDCAD, USDJPY, USDBRL, USDINR, USDMYR, USDRUB and USDZAR all with equal weighting. The spot rates were taken from market data as of Sept 2012 and the correlation matrix was estimated. The interest rates, both foreign and domestic, and the strike price were taken to be zero in the interest of simplicity.

In the Monte Carlo simulation we used 150,000 sample paths, each with 360 time steps. The simulation requires 540 million random numbers, which is 4.12GB in double precision.

5.2 Hardware and compilers

The various computing platforms we used are as follows:

1. CPU: Intel Xeon E5-2670, an 8 core processor with 20MB L3 cache running at a base frequency of 2.6GHz but able to ramp up to 3.3GHz under computational loads. The processor supports AVX instructions and has 256bit vector units (4 doubles). The system has 130GB memory and hyperthreading is disabled. We used the Intel compiler `icc` version 12.1.0 with flags

```
-O3 -xHost -openmp -g -restrict
```

[6]Elemental functions, manually unrolling various loops, restructuring if statements and various compiler pragmas.

2. GPU: NVIDIA K20Xm with 6GB RAM. The GPU clock rate is 700MHz, ECC is off and the memory bandwidth is around 200GB/s. We used gcc version 4.4.6 and CUDA toolkit 5.0 with `nvcc` version 5.0.0.2.1221 and flags

   ```
   -O2 -Xptxas -v --generate-code arch=compute_35,code=sm_35
   ```

   Note: timings for GPU include all overheads and memory transfers.[7]

5.3 Base double precision results and scaling figures

Table 1 gives the double precision results for both platforms. The width of the confidence

	Price	C.I. Width	Runtime(ms)	Speedup	RNG Runtime(ms)
CPU	0.5892381192	3.195e-4	7,557.72	1.0x	1,144.85
GPU	0.5892391192	3.195e-4	698.28	10.83x	125.14

Table 1: *Double precision results showing price, width of 98% confidence interval, overall runtime, speedup relative to CPU, and runtime for the random number generator only.*

interval is about 0.05% of the price. For completeness, we give scaling figures for the CPU in Table 2. The CPU runtime roughly halves each time we double the number of threads,

CPU Threads	1	2	4	8
Runtime(ms)	57,354.13	30,319.64	14,953.17	7,557.72

Table 2: *Scaling figures for the CPU.*

indicating very good strong scaling.

5.4 Mixed precision

Calculations performed in single precision (i.e. using IEEE 32bit floating point numbers) have an accuracy of around 1e-7. In particular, if one performs the same calculation in single and double precision, one would expect the relative error between the two results to be around 1e-7. If the error is significantly worse than this, it may suggest that the calculation is unstable in single precision, or that there are significant accumulations of round off errors which double precision helps to alleviate.

To analyse the impact of mixed precision programming we focus initially on the GPU code since the new Kepler series GPUs from NVIDIA have three times higher single precision performance than double precision. We successively changed various parts of the algorithm from double to single precision to see the effect on accuracy and performance. The results are given in Table 3. The changes are as follows:

Dp The original double precision code.

1. Use single precision Normal random numbers. The numbers are cast to doubles as soon as they are read in by the path calculation kernel. The speedup is therefore due to the faster random number generator only. Note that the relative error is already on a par with what one would expect from a single precision algorithm, and the error is not substantially different from the overall errors for steps Sp and 4.

[7]Costs of initialising the CUDA runtime and creating contexts are excluded.

Step	Price	Incremental Relative Error	Overall Relative Error	Runtime(ms)	Speedup
Dp	0.5892391192		0.0	698.21	1.00x
1	0.5892389949	2.11e-7	2.11e-7	637.60	1.10x
2	0.5892389969	3.39e-9	2.08e-7	425.41	1.64x
3	0.5892389787	3.09e-8	2.38e-7	402.70	1.73x
Sp	0.5892388787	1.70e-7	4.08e-7	347.32	2.01x
4	0.5892392722	6.68e-7	2.60e-7	153.82	4.54x

Table 3: *Mixed precision modifications of the GPU code along with relative error of each step vs. the previous step, relative error of each step vs. double precision code, runtimes, and speedups vs double precision.*

2. In addition to the previous step, do the implied volatility spline calculations in single precision. The spline calculations comprise the bulk of the computation, and switching these to single precision gives an appreciable reduction in runtime.

3. In addition to the previous steps, evaluate the Dupire formula (6) in single precision. Perhaps surprisingly, this does not give a big reduction in runtime.

Sp The entire simulation (random numbers, path calculation kernel and payoff kernel) is done in single precision.

4. The local volatility surfaces for each asset are pre-computed on a grid with 361 time points and 400 space points, and these pre-computed surfaces are sampled in the path calculation kernel by using layered textures with bilinear interpolation. We discuss this more fully in section 5.5 below.

Moving the entire calculation from double to single precision gives a speedup of 2.01x which is substantial, but is quite a bit less than the theoretical speedup of 3x. It is not clear why the achieved figure is so different from the theoretical.

In terms of accuracy, the single precision code has a relative error (compared to the double precision answer) of 4e-7, which suggests that the algorithm is stable in single precision and that there are no significant accumulations of round off errors.

The results of running the entire calculation in single precision on the CPU are given in Table 4. The more than 2x speedup for the CPU code is no doubt due to more data

	Price	Relative Error	Runtime(ms)	Speedup	RNG Runtime(ms)
CPU (sp)	0.5892389162	3.45e-7	3,312.53	2.28x	957.92

Table 4: *Results for full single precision code on CPU showing the price, relative error vs. double precision result, overall runtime, speedup vs double precision runtime, and runtime of the random number generator only.*

fitting in the various levels of cache. The single precision option prices differ across both platforms due to the different orders in which values are added up. This is to be expected with single precision code on different parallel architectures.

5.5 GPU texture memory

In light of the performance figures of the final step in Table 3 we briefly describe our experiments with GPU texture memory. A *texture* is a one, two or three dimensional array which sits in GPU memory and which is accessed through the hardware texture reference units. Textures are normally used in graphics applications such as computer games, however they suit the local volatility model extremely well.

The hardware texture units allow a GPU program to “sample” a texture at random points: the texture units take an input coordinate, transform it to a coordinate in the texture, find the nearest elements in the texture to the coordinate, and then perform linear interpolation between the nearest elements to approximate the value of the texture at the input coordinate. For example if one sets up a two dimensional local volatility surface as a texture, then a GPU program could sample the local volatility surface at each point on a stock price path and the texture units would perform bilinear interpolation between the four nearest local volatility elements in order to approximate the local volatility at the current stock price.

Textures have seen some limited use in computational finance. Practitioners usually cite two concerns: firstly textures only work with single precision data, and secondly the linear interpolation performed by the hardware units is low precision. The texture units on NVIDIA GPUs use 8 bit fixed precision to represent the fractional part of the (transformed) input coordinate[8]. The fractional part therefore has a resolution of $2^{-8} \approx 0.004$ so that for example two (transformed) input coordinates of 10.001 and 10.003 would both be interpolated to the same value.

To determine the possible benefits of texture memory we evaluated the local volatility surface for each asset on a grid with 361 time points and 400 space points. These calculations were done in double precision and the results cast to single precision. These values were then copied to a layered texture[9] so that the volatility surface for each asset could be sampled using the hardware texture units. This resulted in the impressive speedup reported in the final step in Table 3. Note that the accuracy of this approach is in line with a full single precision calculation.

We varied the size of the grid both in space and time but observed no change in runtime. Decreasing the size of the grid merely led to worse accuracy. On the other hand increasing the size of the grid beyond 361x400 seemed to give no improvement in accuracy. In general practitioners who wish to use textures should prefer larger grids over smaller ones, although how large is large enough obviously depends on the problem.

6 Acknowledgments

The authors would like to thank NVIDIA for providing access to their PSG cluster.

References

[1] Andreasen, J., Huge, B. (2011) Volatility interpolation. *Risk Magazine*, March 76-79.

[2] Brigo, D., Mercurio, F. (2000) Displaced and Mixture Diffusions for Analytically-Tractable Smile Models. *Mathematical Finance-Bachelier Congress 2000*, Springer : Berlin.

[3] Clark, I. (2011) Foreign Exchange Option Pricing. *John Wiley & Sons Ltd* , Chichester.

[4] Dupire, B. (1994) Pricing with a Smile. *Risk* , 7 (1) 18-20.

[8] There is a 9th bit which ensures a fraction of 1 can be represented exactly.

[9] A layered texture roughly speaking is an array of textures.

[5] L'Écuyer, P. (1999) Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47 (1)159-164.

[6] Fengler, M. (2009) Arbitrage free smoothing of the implied volatility surface. *Quantitative Finance*, 9 (4) 417-428.

[7] Glaser, J., Heider, P. (2012) Arbitrage free approximation of call price surfaces and input data risk. *Quantitative Finance*, 12 (1) 61-73.

[8] Huynh, C. (1994) Back to Baskets. *Risk* , 7 (5), 59-61.

[9] Kahalé, N. (2004) An arbitrage free interpolation of volatilities. *Risk*, 17 102-106.

[10] Intel Corporation. Intel C++ Compiler XE 13.1 User and Reference Guide. *Available online on the Intel website.*

[11] Levy, E. (1992). Pricing European Average Rate Currency Options. *Journal of International Money and Finance*, 11 (5), 474-491.

[12] Milevsky, M., Posner, S. (1998). A Closed-form Approximation for Valuing Basket Options. *The Journal of Derivatives*, Summer 1998, 54-61.

[13] Xu, G., Zheng, H. (2010) Basket Options Valuation for a Local Volatility Jump-Diffusion Model with Asymptotic Expansion Method. *Insurance : Mathematics and Economics* , 47, 415-422.