RISK ANALYTICS IN TIME

HIGH PERFOMANCE IN COMPUTATIONAL FINANCE BY UTILIZING NVIDIA GPUS

UnRisk
risk analytics in real time

Dr. Andreas Binder is CEO of MathConsult GmbH and head of its computational finance group. They develop advanced numerical solutions for pricing and risk management of financial instruments. For more information: www.unrisk.com

Dr. Oliver Tennert is Director Technology Management & HPC Solutions at transtec AG, one of the leading European providers of HPC solutions. For more information: www.transtec.de

TABLE OF CONTENTS

Table of Contents 2
1 Introduction 2
2 Model Problems 2
3 GPU Hardware Architecture and the CUDA Development Environment........ 4
3.1 What is GPU Computing? 4
3.2 CUDA Parallel Architecture and Programming Model........ 4
3.3 CUDA: The Developer's View........ 5
4 Results Achieved by the UnRisk GPU Extension........ 6
4.1 Calibration 6
4.2 Monte Carlo Simulation 7
5 Conclusion 8
6 References........ 8
7 Acknowledgement 9

1 INTRODUCTION

The analysis and risk management of financial instruments requires fast, robust and reliable computational methods for their valuation, which has to be repeated at least on a daily basis due to changing market conditions. For trading purposes, almost real-time computing requirements should be fulfilled.

The computational challenges in fulfilling these requirements are enormous. With today's modern graphics processing units (GPUs), cost-efficient high performance devices are available, which, in combination with specific software solutions, bring supercomputing power to the desktop.

The plan for this paper is as follows: In section 2, we present two model problems which are very well suited for the massively parallel GPU architecture, one in the calibration of model parameters for a stochastic volatility model, one in Monte Carlo simulation of exotic options. In section 3, we describe the main flavours of the NVIDIA Tesla architecture, its performance benchmarks and present the key points for making GPU code efficient. Finally, in section 4, we present results for the model problems and compare computing times to the CPU reference implementation.

2 MODEL PROBLEMS

When observing quoted market data of European options on equity (or on equity indices) with the same expiry date, one realises that the implied Black volatility for different strike prices typically is not constant but exhibits a "smile" or a "skewness" behavior. There at least two possible reasons: From the insurance side, risk-averse investors are prepared to pay a higher option premium to insure their portfolio value against significant drops, thus implying a higher volatility for lower strike prices. From the phenomenological side, one observes "volatility clustering" in the spot market: There are time intervals with higher realised volatility and intervals

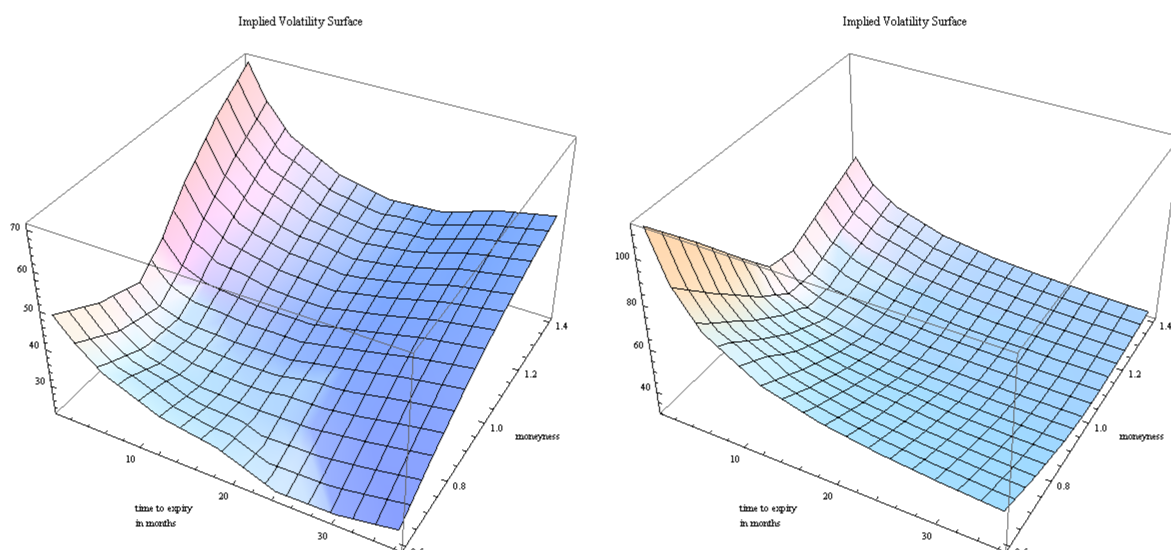
with lower volatility. This leads to more pronounced fat tails compared to the log-normal distribution of Black Scholes.

Stochastic volatility and other volatility models like the Variance Gamma model, the Normal Inverse Gaussian model or the Bates model are able to recover some of the features described above. Here we mention the Heston model:

$$dS_t = r(t)S_t dt + \sqrt{v_t} S_t dW_t^1$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^2$$
$$dW_t^1 dW_t^2 = \rho dt$$

with S_t standing for the market price of the equity at time *t, v* being the variance process of the return , which is assumed to be mean-reverting (reversion speed κ, equilibrium level θ and a volatility of variance σ).

Depending on the parameters ρ, κ σ, θ and the iniital variance v0, the implied volatility surface of a Heston model may exhibit quite different shapes, e.g.



Two volatility surfaces obtained by different Heston parameter sets.

If we want to valuate an exotic option like a barrier option or a lookback option under such a Heston model , we first have to identify the model parameters from market prices of basic derivative instruments, typically vanilla call / put options with different strike prices and different maturities. For liquid underlyings like the most important equity indices (S&P 500, FTSE, DAX, EuroStoxx, Nikkei), several hundred of such options are quoted almost continuously.

The calibration process then is an optimization routine which, starting from one or many initial guesses for the model parameters, iteratively updates these parameters in order to achieve a fit as good as possible.

Some remarks:
The Feller condition $2\kappa\theta > \sigma^2$ guarantees that the variance process in the Heston model remains strictly positive with probability 1. Requiring the Feller condition to be fulfilled, makes the set of admissible parameters non-convex and the optimisation problem much more difficult than without the Feller condition.

- The error surface (in the 5d parameter space) exhibits many local extrema. Therefore, it makes sense to think of techniques of global optimization.

- Under these aspects it is not surprising that the parameter identification problem for the Heston model may require the valuation of more than a million vanilla options, with each option valuation meaning the solution of a partial differential equation with one time and two state variables.
- In order to achieve good performance results, we therefore have to think of implementations which combine advanced numerical schemes for the solution of these partial differential equations and their realization on possibly parallel hardware

Having calibrated the model parameters, we are in the position to valuate exotic options on the basis of these parameters. The method of choice for a wide range of structured financial instruments is Monte Carlo simulation. Again, this can be implemented very efficiently on massively parallel hardware architectures. As a specific example, we want to valuate a lookback option: During the lifetime of the option, the closing prices of the underlying (here the FTSE 100 index) are observed, and the payoff at maturity is max (maximum of closing prices – strike price, 0). For the modelling of the price development of the underlying, a Heston model as above should be used.

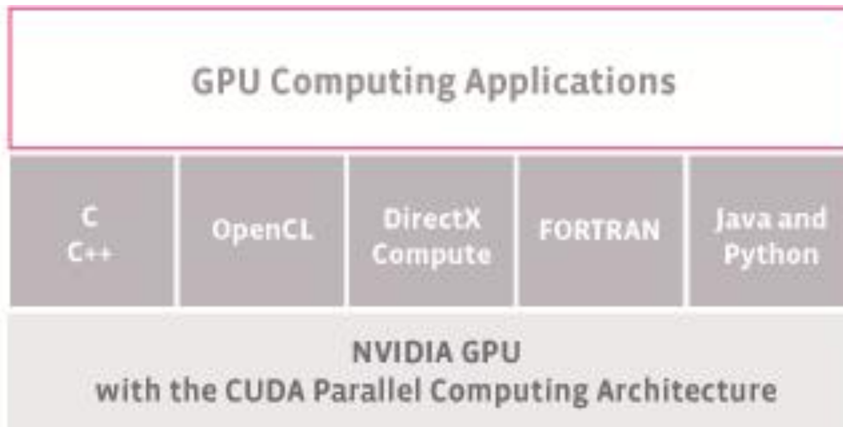
3 GPU HARDWARE ARCHITECTURE AND THE CUDA DEVELOPMENT ENVIRONMENT

3.1 What is GPU Computing?

GPU computing is the use of a GPU (graphics processing unit) to do general purpose scientific and engineering computing. The model for GPU computing is to use a CPU and GPU together in a heterogeneous computing model. The sequential part of the application runs on the CPU and the computationally intensive part runs on the GPU. From the user's perspective, the application just runs faster because it is using the high-performance of the GPU to boost performance. The application developer has to modify the application to take the compute-intensive kernels and map them to the GPU. The rest of the application remains on the CPU. Mapping a function to the GPU involves rewriting the function to expose the parallelism in the function and adding "C" keywords to move data to and from the GPU. GPU computing is enabled by the massively parallel architecture of NVIDIA's GPUs called the CUDA architecture. The CUDA architecture consists of 100s of processor cores that operate together to crunch through the data set in the application.

3.2 CUDA Parallel Architecture and Programming Model

The CUDA parallel hardware architecture is accompanied by the CUDA parallel programming model that provides a set of abstractions that enable expressing fine-grained and coarse-grain data and task parallelism. The programmer can choose to express the parallelism in high-level languages such as C, C++, Fortran or driver APIs such as OpenCL and DirectX-11 Compute. The CUDA parallel programming model guides programmers to partition the problem into coarse sub-problems that can be solved independently in parallel. Fine-grain parallelism in the sub-problems is then expressed such that each sub-problem can be solved cooperatively in parallel. The CUDA GPU architecture and the corresponding CUDA parallel computing model are now widely deployed with 100s of applications and nearly a 1000 published research papers.



The CUDA parallel architecture.

3.3 CUDA: The Developer's View

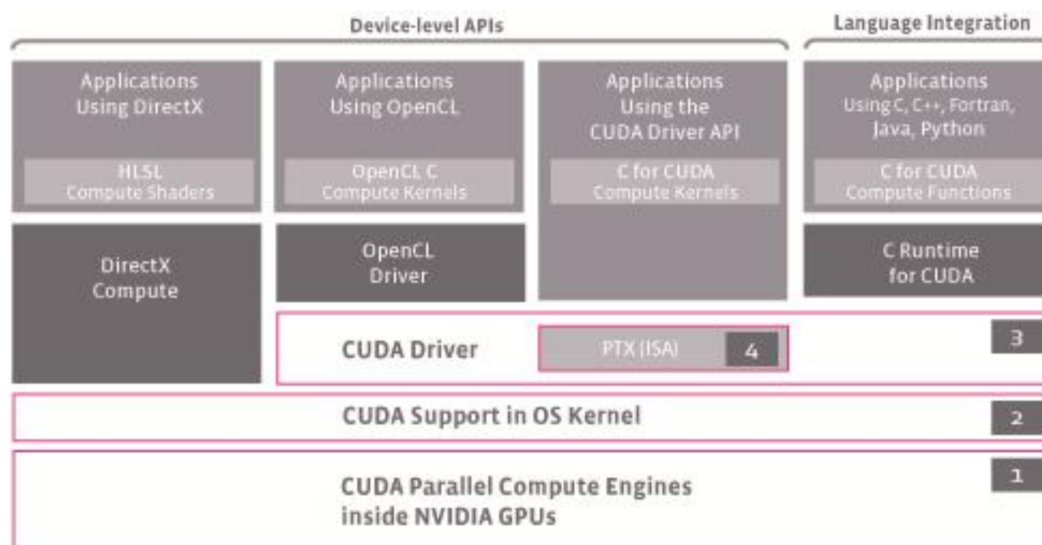
The CUDA package includes three important components: the CUDA Driver API (also known as "Low-Level API"), the CUDA toolkit (the actual development environment including runtime libraries) and a Software Development Kit (CUDA SDK) with code examples.

The CUDA toolkit is in principle a C development environment and includes the actual compiler (nvcc), an update of the PathScale C compiler, optimized FFT and BLAS libraries as well as a visual profiler (cudaprof), a gdb-based debugger (cudagdb), shared libraries for the runtime environment for CUDA programs (the "Runtime API") and last but not least, comprehensive documentation including a developer's manual.

The CUDA Developer SDK includes examples with source codes for matrix calculation, pseudo random number generators, image convolution, wavelet calculations and a lot more besides.

The CUDA Software Development Environment supports two different programming interfaces:

- a device-level programming interface, in which the application uses DirectX Compute, OpenCL or the CUDA Driver API directly to confi gure the GPU, launch compute kernels, and read back results
- a language integration programming interface, in which an application uses the C Runtime for CUDA and developers use a small set of extensions to indicate which compute functions should be performed on the GPU instead of the CPU



The CUDA architecture: (1): parallel compute engines inside NVIDIA GPUs; (2): OS kernel-level support for hardware initialization and configuration; (3): user-mode driver providing a device-level API for developers; (4): PTX instruction set architecture (ISA) for parallel computing kernels and functions.

When using the device-level programming interface, developers write compute kernels in separate files using the kernel language supported by their API of choice. DirectX Compute kernels (aka “compute shaders”) are written in HLSL. OpenCL kernels are written in a C-like language called “OpenCL C”. The CUDA Driver API accepts kernels written in C or PTX assembler. When using the language integration programming interface, developers write compute functions in C and the C Runtime for CUDA automatically handles setting up the GPU and executing the compute functions. This programming interface enables developers to take advantage of native support for high-level languages such as C, C++, Fortran, Java, Python, and more, reducing code complexity and development costs through type integration and code integration.

4 RESULTS ACHIEVED BY THE UNRISK GPU EXTENSION

4.1 Calibration

We start with the problem to calibrate the Heston parameters in a least-squares sense from market data of vanilla options on the FTSE100 index. The market data set in this example was based on market quotes for 304 options with different strikes and maturities up to 21 months. For the calculation of the option prices from the Heston parameters we used a parallelised version of the Fourier cosine method as presented in [Fang, Oosterlee] . It turns out that this parallel option valuation in the Cuda environment is most efficient when the summation index of the Fourier series is constant and not chosen adaptively. Roughly speaking, the GPU cores can obtain their peak performance if they all have to do the same numerical operations and if no conditional branches disturb the monotonicity of their work.

Remembering the fact that the error surface may exhibit many local minima, for each market data set, we determined the optimal set of Heston parameters by applying global optimization techniques; in the specific example we used differential evolution [Brabazon, O’Neill]. As expected, global optimization techniques are slow, and the computation time for obtaining the reference solution was more than 8 hours on a single CPU.

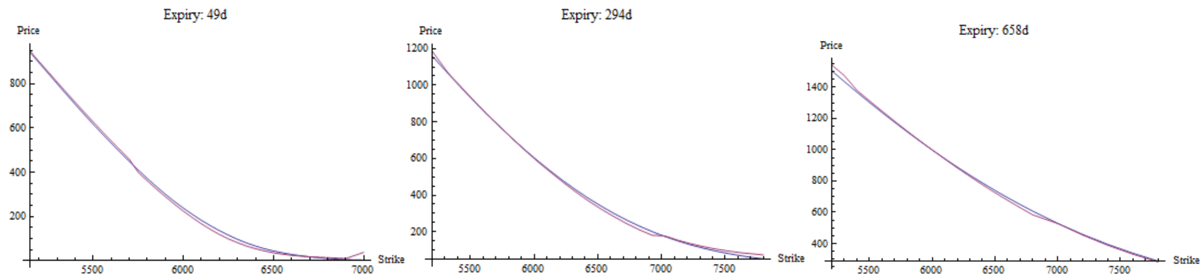
It turns out that the optimal residual would be achieved by a parameter set violating the Feller condition. If we try to apply gradient-based (local) minimisation algorithms and force the Feller condition to be fulfilled by projection, the local minimum depends strongly on the initial parameter guess for the optimisation routine. Hence, in order to avoid the drawbacks of both approaches, our approach takes a set of initial parameter points which cover the set of admissible points reasonably, and then starts local optimisation routines. A further improvement can be achieved by carrying out an extra optimisation on the (4D) boundary manifold of the Feller condition.

For the FTSE100 data sets, this hybrid optimisation method typically needs between 1 and 1.5 million option valuations. The following table shows the performance achieved for different multithreading levels:

MULTITHREADING	ELAPSED TIME (SECONDS)	SPEEDUP OVER 1XCPU
1 CPU thread	780.9	1.00
2 CPU threads	440.5	1.77
4 CPU threads	275.8	2.83
8 CPU threads	204.0	3.83
16 CPU threads	139.9	5.58
1x GPU (C2050)	8.5	92.23

FTSE 100Heston calibration on the following hardware 2 x Xeon E5520 @ 2.27 GHz, 4 Cores, 24GB RAM, 1x Tesla C2050.

The following figures show the quality of the fit of the Heston prices achieved with the optimal parameter set to the market prices.

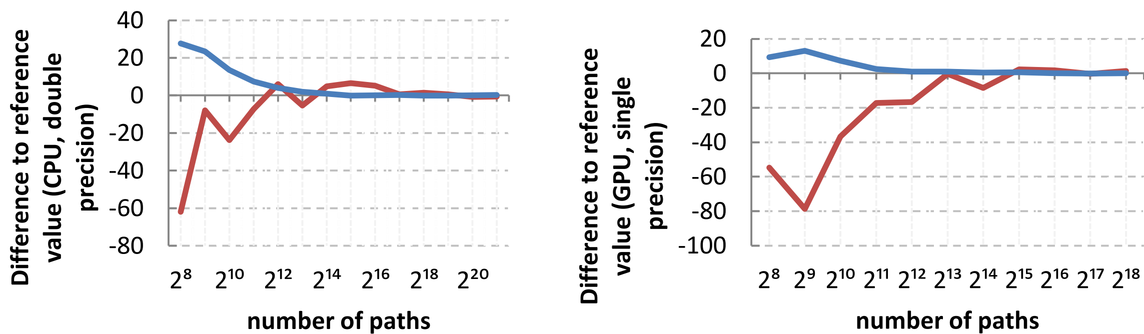


Quality of the Heston fit for different expiries: Heston call prices are blue, market prices are red.

4.2 Monte Carlo Simulation

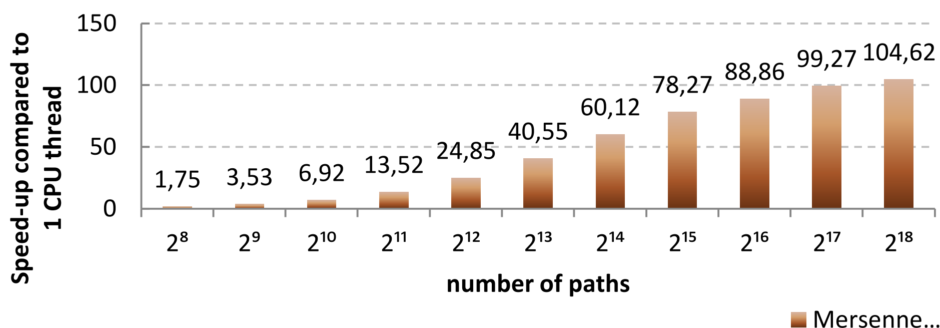
For the valuation of the lookback option as described in section 2, we implemented two variations:
(1) plain Monte Carlo, using the Mersenne twister random number generator and Euler stepping,
(2) quasi Monte Carlo based on Sobol points in combination with Brownian bridges.

The spot price of the FTSE 100 was (as of June 1st, 2008) 6053.5, the strike price for the lookback option 6000. The reference value for the lookback option (expiry: 1 year) was 1043.03 (obtained by the mean value of the finest Monte Carlo and the finest Quasi Monte Carlo simulation). We obtained the following convergence behavior.



Convergence of (Quasi) Monte Carlo methods for the lookback option: Mersenne Twister (red), Sobol points (blue).

As expected, the low discrepancy Sobol points method converge significantly faster, both on the CPU and on the GPU. For the speed-up of the Mersenne twister on the GPU in relation to the CPU, we get:



To obtain a speed-up in the order of 100, it is essential that the problem size is not too small. Here, the rightmost bar stands for 2^{18} =262144 paths (each consisting of 365 days). The elapsed GPU time was 78 milliseconds.

As the convergence properties of the low discrepancy Sobol points are better than those of plain Monte Carlo, for this type of problem, Sobol points are preferable. With 8192 paths, the accuracy is excellent, and the computing time was 7 milliseconds. Detailed results are contained in the following table:

1xGPU	Mersenne Twister				Sobol			
Paths	Value	Time (ms)	Speedup	Diff	Value	Time	Speedup	Diff
2^8	1097.81	4.60	1.75	-54.78	1033.62	5.15	2.14	9.41
2^9	1121.66	4.61	3.53	-78.63	1030.01	5.22	3.89	13.03
2^{10}	1079.73	4.70	6.92	-36.70	1035.77	5.76	6.75	7.26
2^{11}	1060.26	4.83	13.52	-17.23	1040.66	5.56	13.54	2.37
2^{12}	1059.70	5.23	24.85	-16.67	1042.14	5.82	25.77	0.89
2^{13}	1043.39	6.36	40.55	-0.35	1042.07	6.99	42.36	0.96
2^{14}	1051.61	8.58	60.12	-8.58	1042.60	10.84	54.63	0.44
2^{15}	1040.75	13.19	78.27	2.28	1042.38	16.52	71.36	0.65
2^{16}	1041.39	23.20	88.86	1.64	1043.00	27.94	84.23	0.04
2^{17}	1043.41	41.53	99.27	-0.38	1043.18	52.12	90.25	-0.14
2^{18}	1041.67	78.77	104.62	1.36	1043.08	96.41	97.47	-0.05

5 CONCLUSION

Modern high performance GPUs like the NVIDIA C20xx series provide, in combination with the CUDA development platform, powerful engines for high performance tasks in computational finance. The UnRisk GPU extension obtained speedups in the order of 100 compared to the single CPU thread for several model problems in calibration and valuation of stochastic volatility problems.

6 REFERENCES

H. Albrecher, A. Binder, Ph. Mayer: Einführung in die Finanzmathematik, 2009, Birkhäuser
Aichinger, Binder, Fürst, Kletzmayr: A Fast and Stable Heston Model Calibration on the GPU, to appear
A.Brabazon, M.O'Neill: Biologically Inspired Algorithms for Financial Modelling, 2006, Springer.
F. Fang, K. Oosterlee: A novel pricing method for European options based on Fourier-cosine series expansions. SIAM J. Sci. Comput. 31: 826-848 (2008).
Glasserman: Monte Carlo Methods in Financial Engineering (Stochastic Modelling and Applied Probability), Springer, 2003.
R. Storn, On the Usage of Differential Evolution for Function Optimization" NAFIPS 1996, Berkeley, pp. 519 - 523.
K. Madsen, H.B. Nielsen, O. Tingleff, Methods for Non-Linear Least Squares Problems, 2nd edition, IMM (2004).
NVIDIA Fermi Architecture Whitepaper:
http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
UnRisk: http://www.unrisk.com

7 ACKNOWLEDGEMENT

Part of this work was supported within the research program „Industrial Competence Centers“ by the Upper Austrian and by the Austrian Federal Government. We thank Michael Aichinger (RICAM), Johannes Fürst, Christian Kletzmayr (both MathConsult) for valuable discussions.