GPU Accelerated framework for financial nested simulations

Joris Cramwinckel

31-03-2015

Abstract

In this thesis we present a state-of-the-art approach to accelerate Monte Carlo valuations of embedded options. Due to regulations and improved risk management, nested simulations (scenarios in scenarios) are becoming increasingly important for institutional investors like: insurance companies, pension funds and housing corporations. Preferably one wishes to use a framework in which multiple related problems of nested simulations can be accelerated with GPUs. We build such a framework using advanced CUDA features from NVidias Kepler and higher architectures. CUDA streams and Hyper-Queues enable the GPU to run tasks effectively by running concurrent and overlapping CPU and GPU calculations. In addition, NVidias Multi Processing Service is used to handle offloading Monte Carlo valuations from different local processes to the GPU. Runtimes in the order of days of current implementations are reduced to the order of minutes. This broadens the horizon of the current nested simulation methodologies. Besides, the proposed framework is scales well with the number of compute nodes and GPUs per cluster.

Table of contents

1 Introduction

One of the core businesses of Ortec Finance (OF) is performing asset liability studies for financial institutions like Pensions funds, (life) insurers and housing corporations. The software solutions and methodologies used in these studies are developed and maintained internally. A common way to analyze future cash flows, is by evaluating multiple scenarios of the economy on current positions and strategies of a client. By simulating the portfolio of assets and liabilities in the future one could forecast their balance sheets and analyze the behavior at tail events. These events are important to report to regulators. Because, regulations can for instance require institutional investors to provide for a certain capital in economic tail events.

Many asset classes, such as real estate or positions on the money market are straightforward to value given an economic development, a real world scenario. However, some items in a portfolio cannot be explicitly priced. Take for instance a financial derivative, one could analytically determine the value by substituting the economic factors (e.g. current price underlying, market volatility and interest rate) in an analytical model. Nevertheless, analytical solutions to price derivatives are not always available and often not accurate enough. In these cases one must rely on other numerical methods.

Financial derivatives included on balance sheets, are more common than one might think. They can be present within common commercial financial products like: issued mortgages, life insurances and pension plans. Often these products contain agreements with the characteristics of an option. We will refer to these products as embedded options. For example, a life insurance company can promise a minimum return to their policy holders. By eliminating the downside risk, this right has the characteristics of an option for the policy holder. The valuation of embedded options on a balance sheet should be accurate since in extreme events these values can become dominant on the balance sheet [1] .

The valuation of embedded options can be done with several techniques [2][3] . In this thesis we focus on Monte Carlo (MC) valuations. Although computationally intensive, a popular and straightforward way to value complex contracts is by using MC techniques. Within a real world simulation we get simulations within a simulation. The inner simulations are performed in what we call a risk neutral measure, consequently, they are called risk neutral simulations. The concept of performing risk neutral simulations within real world simulations is what we call nested simulations.

Because of the computationally intensive nature of nested simulations, we will explore techniques to accelerate these simulations by offloading the inner simulations to GPUs. By empirical approach, we build a framework in which several nested simulation applications can be accelerated with the use of GPUs.

The runtime of current implementations of methodologies concerning nested simulations, are in the order of days. This runtime eliminates the possibility to fine-tune and evaluate the current models thoroughly.

The main question of this thesis is: Can we design a framework in which nested simulations and its applications can be accelerated to reasonable run times? Such a framework should allow model extensions to be implemented not only by GPGPU experts but mainly by researchers and consultants. In addition, the greatest common divisor between all nested simulation applications can be maintained centrally. Furthermore, faster runtimes will enable expansions to the methodologies, accuracy tweaking of calibration models and, finally, minimize the impact of nested simulations on the run time of current OF solutions.

The design of this framework will make extensive use of a set of CUDA abstractions available in the latest NVidia architectures. CUDA streams [4], Hyper-Q [5], and MPS [6] are used the efficiently offload simulations to the GPU.

We focus on three nested simulation applications: First we accelerate the calibration of risk neutral simulation models. In this application millions of Monte Carlo option valuations are to be performed. Secondly, the generation of risk neutral scenario sets is offloaded to the GPU. As this involves generating many Gigabytes of data, the challenge is to manage memory transfers properly. For both applications the GPU acceleration resulted in runtimes in the order of minutes. Finally, a technical framework is proposed to effectively run nested simulations in existing scenario analysis applications. With a mock up model of such an application we achieved almost no performance loss in incorporating a nested simulation in a scenario analysis. All this additional work can be handled by one or more GPUs.

Besides this main subject, some broader questions will be answered in order to create a clear view on GPU computing and its use by OF competitors.

The thesis is structured as follows: Firstly, an introduction and a survey are presented in which we generally discuss General Purpose GPU (GPGPU) computing. We also discuss the developments in competing companies applying HPC concepts. Moreover, the applicability of GPU computing is sketched for several OF solutions. Secondly, we describe the domain, nested simulation methodologies, in more detail. In chapter 4, we describe the GPU techniques which are used in the proposed framework. Subsequently, the nested simulation application and its framework are explained and, details on design and performance results are described. Finally, conclusions are presented and future work is proposed.

General introduction GPGPU

In this chapter we briefly explain some basic principles and developments in General Purpose GPU (GPGPU) computing. It will give the reader a helicopter view on the subject and its applicability. Practical subjects like programmability and performance concerns are discussed.

1.1 HPC computing in general

High performance computing (HPC) becomes more and more embedded in several industries. HPC is a general term for techniques which let applications run faster than they do on regular desktops or workstations. Think of a grid of regular desktops, a cluster of compute nodes or offloading parts of the calculations to other available hardware or even a combination off all. At the time of writing, the scope of HPC is still growing and gets more accessible for small companies and even individuals. For example, Amazon offers a on demand service for using different HPC instances like GPU clusters, I/O clusters or storage optimized clusters. It’s not surprising that the scope of applications using such techniques is still growing.

However, running legacy code on HPC solutions is not as easy as one might think. Since preparing legacy applications for HPC can be hard, the concepts of HPC are, at the moment of writing, only applied by early adaptors in the industry. Companies have to consider the additional investment in developing and maintaining HPC applications against any performance gain. In Finance however, lots of the concepts of HPC seem to be accepted as proven technologies. This chapter will briefly explain and describe current developments and trends in the world of HPC and GPGPU in particular.

1.2 Heterogeneous Computing

When applications are using the computational power of other hardware besides the available CPUs we call the application architecture heterogeneous. Assigning chunks of code to other hardware can benefit the total computational time. Distributing computation tasks to other hardware enables the application to run certain tasks in parallel. This is what we call task parallelism. In addition, CPUs are designed for low latency and single threaded performance. So in order to run massively parallel Single instruction Multiple Threads (SIMT) algorithms one might need different well suited hardware. GPUs for instance, are designed for high throughput and massively data parallelism. So, the benefits of heterogeneous computing lie not so heavily on the fact you are able the task paralyze your application, but to assign computations to the most suitable hardware available. In the following section we describe those different types of hardware.

1.2.1 Graphical processing units

The development of GPUs is mainly driven be the demand of the entertainment industry. Vendors like NVidia and AMD, are heavily dependent on the gaming and video processing markets. General computations on GPUs started as a niche, nowadays it’s a widely accepted concept. Vendors have launched General purpose GPUs, like NVidia Tesla line in 2006, and are in the process of optimizing the hardware for general purpose computations.

Originally, GPUs have to render each pixel of a screen for example 60 times per second. This concept results in hardware which can handle a single instruction (render a pixel) to a large amount of data (1920 x 1080 pixels for example). Since the late nineties the instruction came more complex and as of now it is possible to perform full kernels GPUs

Bottlenecks

GPU computing is certainly not the holy grail of achieving significant performance speed ups. To give a summarized view on some considerations one could encounter programming GPUs, the next sections will briefly explain the main subjects to consider:

Programmability

Over the years programming languages offered some of its performance in exchange for programmability. Close to the iron languages like Fortran and C are despite of the relatively hard programming model still the fastest common languages available.

GPUs however, cannot be instructed with common programming models, instead, they come with their own. NVidia for example, offers the CUDA (Compute Unified Device Architecture) programming model. As one can imagine, performance is a key feature of programming GPUs. So, programming them has in some extend similarities with programming instructions in C for example.

The complexity of GPU programming is not caused by the paradigms or the syntax but, programming this close to the iron. One should be aware of the hardware features of the GPU writing the instructions for. In order to fully utilize the compute capabilities of the GPU, developers creativity is often demanded.

Figure 1 shows briefly the main workflow of offloading some instruction set to the GPU.

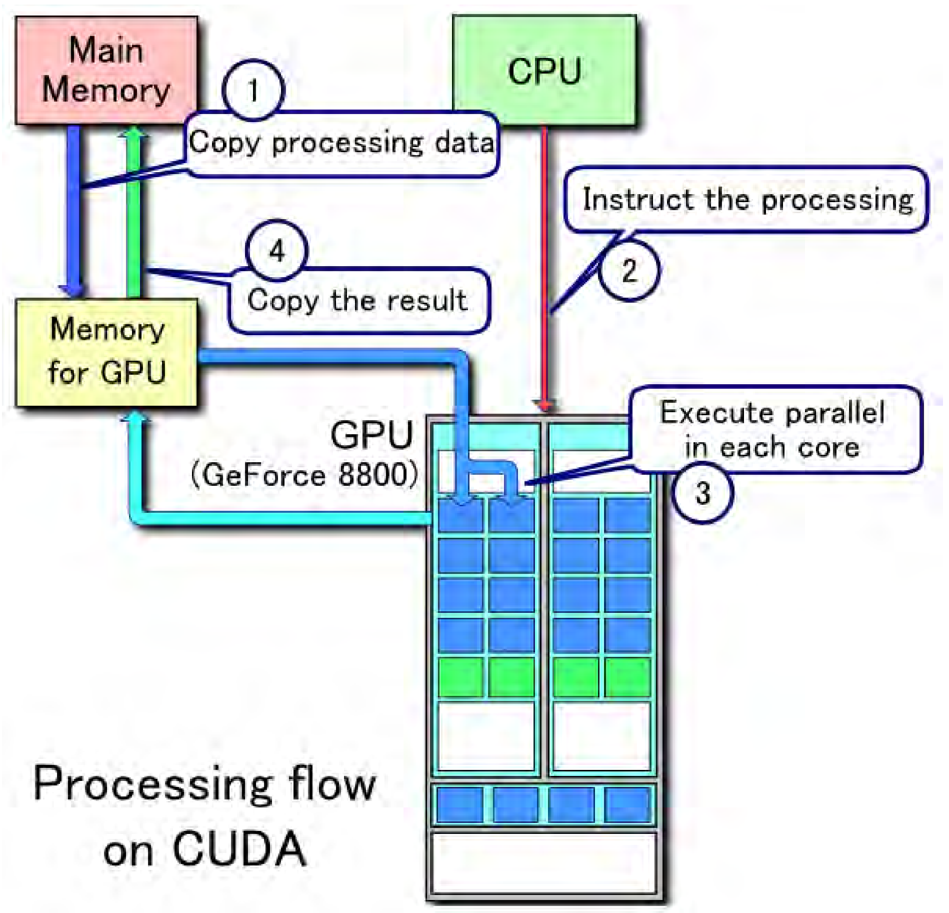

Figure 1 Workflow GPGPU programming (source: "CUDA processing flow (En)" by Tosaka - Own work.)

Memory bandwidth

Vendors tempt to quote high peak performance number in terms of Giga or even Tera FLOPS (Floating point operations). But what often is observed, is that applications are not computationally bounded, but memory bounded. In other words, for the cores to fully utilize their number crunching capabilities they have to wait for memory traffic[1]. This often requires the programmer to utilize

[1] The hardware extensively manages the workload by minimizing idle time of the processors. By replacing on hold warps with other warps. However, this still causes overhead.

advanced memory optimizations in order to gain any performance gain what so ever. The programmer can for example use (among others) the following techniques:

- Program the shared memory usage[2]
- Coalesced memory access.

We can expect that upcoming GPU generations will have an increased memory bandwidth. NVidia already announced a new generation, Pascal, which contains stacked memory. They expect to reach a memory bandwidth of around 1 Terabyte per second.

Device – Host communication

The device (GPU) memory is not the only bottleneck, communication between the host (CPU) and device runs traditionally over the PCIe bus. One can imagine that this is relatively slow. At the moment of writing there the main solutions for this bottleneck is: Overlap calculations and communication by using streams (CUDA) or queues (OpenCL). Van Werkhoven (2014) [7], proposes a model to determine on forehand which technique is most likely to perform best.

Error Correcting Code (ECC)

Regular memory contains ECC to correct for bit flips. On GPUs however, this a feature which is only supported by the latest generations. ECC is a typical GPGPU feature and is not supported by commercial gaming cards. Ignoring bit flips can result in inconsistent results or system crashes.

Single vs Double precision

GPUs naturally do not support double precision calculations. Although most GPGPUs support double precision calculations, it comes with a significant performance penalty. Figure 2 displays this penalty. We observe a quite steep curve in the GPUs peak performance. It seems that developments in this area, have still room to improve in comparison with the CPU performances.

Furthermore for most scientific applications double precision computations are crucial. In our domain we evaluated the usage of single precision in section 4.5.

[2] This is often a trade off with the hardware caching

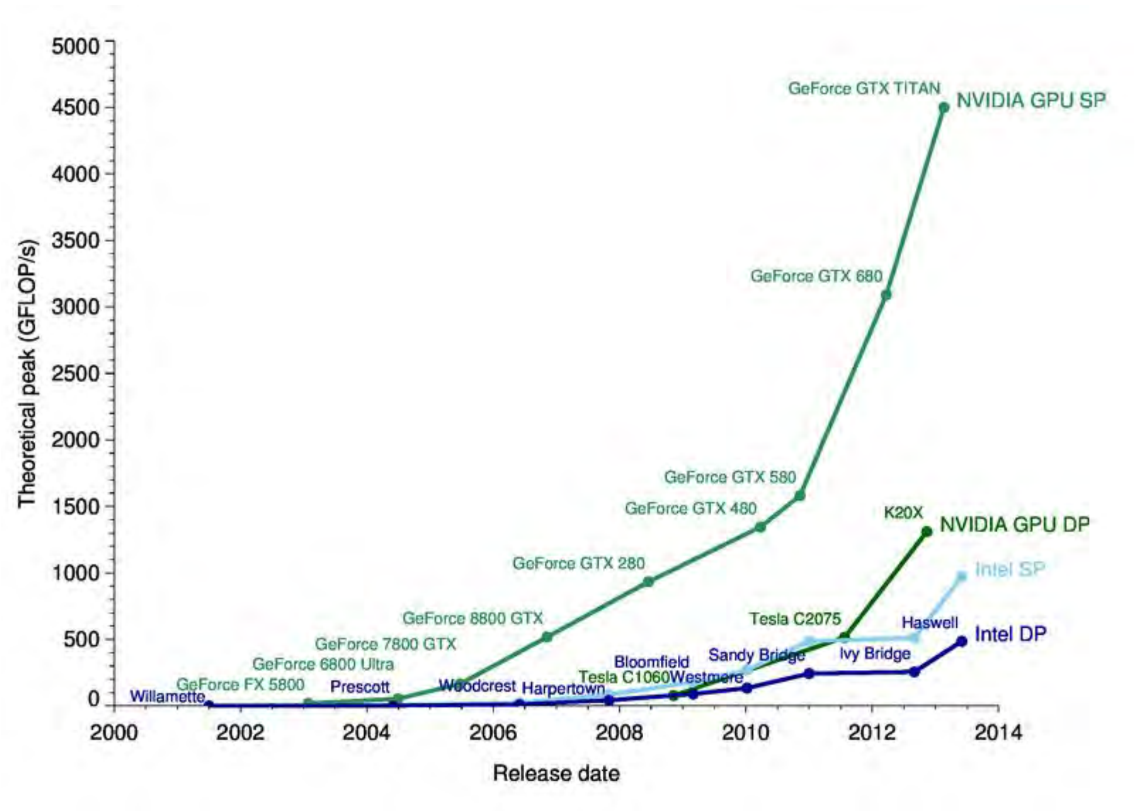

Figure 2 Performance comparison Single and double precision GPU and CPU (source: NVidia)

Speed ups

As Figure 1 shows, the theoretical speed up that can be achieved by performing calculation on GPUs instead of CPUs is roughly a factor 10. The reason why often higher speed ups are communicated is that commonly the compared CPU implementation is not fully optimized. Besides, getting the maximum performance out of a CPU is very hard or next to impossible for most programming languages.

1.2.2 Other accelerators

A many core processor does not have to be from a GPU family. In 2012 Intel released the Xeon Phi coprocessor. Where GPUs are characterized by containing 1000s of slim cores the Xeon Phi has 60 original Pentium based cores with a 4-way simultaneous multithreading. Nowadays, the Xeon Phi can be found within the first ranked super computer on top500.org, the thiane-2. Intel announced his next generation coprocessors ‘Knights landing’ to be released in 2015. Interesting is that it will not be an accelerator but a coprocessor and CPU in one[3]. This means that PCIe communication bottlenecks will be vanished.

1.2.3 HPC in the cloud

Multiple cloud computing providers offer a range of on demand HPC clusters. All in different flavors and sizes. This allows small companies or individuals to access HPC power on demand. Consequently you do not have to invest and maintain large clusters yourself. However, during the project it seemed that a ‘Amazon’-like on demand service are not commonly available for the latest generations NVidia GPUs. Nonetheless, there are some developments in on-demand GPU cloud solutions at BitBrains[4].

[3] http://www.theplatform.net/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/

[4] http://www.bitbrains.nl/

2 Survey GPGPU Ortec Finance

To give the reader a glance of GPU applications provided by companies in the domain in which OF operates, we briefly discuss several examples. A market watch on the subject is presented. Additionally, the usage of CUDA instead of other available programming models is discussed.

2.1 GPGPU in finance market watch

Confidential

2.1.1 Outsourcing HPC solutions

Confidential

2.1.2 'Minimum effort' HPC libraries/toolkits

Confidential

2.1.3 In-house HPC developing

Confidential

2.2 Many core processing needs at OF

At OF, most of the models depend on scenario analytics. Using macro-economic scenarios decision models for pension funds, real estate corporations and private wealth are exploited. Since each scenario is independent one should be able to paralyze its execution. This property matches well with the capabilities of many core processing. To investigate the need for HPC at OF we have constructed a business case which resembles most of the aspects existing models are dealing with:

- Real world scenario simulations
- Risk neutral scenario simulations
- Calibrating of models

In chapter 3 more details on the business case will be explained

2.3 Programming models

Programming GPU capable applications can be done in various ways. We discuss broadly the different methods and point out some of the interesting ones. CUDA, the programming model of NVidia, turned out to matched best with our needs.

For sake of the applications to be accelerated in this thesis, we are only interested in low level libraries. However, we point out some interesting developments in the field of more high level approaches.

2.3.1 Low abstraction lightweight GPU programming toolkits

In the world of GPUs there exists two major vendors: NVidia and AMD. CUDA (Compute Unified Device Architecture) is the programming model provided by NVidia. For this reason CUDA only supports NVidia hardware. OpenCL is CUDA's main contestant. OpenCL differs mainly in the fact it is not a vendor specific programming model. Actually, it is not only supported on GPUs but also on CPUs, intel MIC processors and FPGAs. Although, in theory possible, one should not expect that running OpenCL kernels on different hardware can be done without any interference of the developer. Often hardware specific features are used when kernels are performance optimized.

Programmability wise both NVidia and AMD provide Software Development Kits (SDK) which enables the user to profile and debug GPU kernels. However, the NVidia SDK seem to be more matured. The visual studio plugin can be of great use when taking GPU accelerated models in production.

Performance wise there is no evidence that the portability nature of OpenCL affects its performance [8]. However, CUDA comes with a set of highly optimized math libraries like CUBLAS, CUFFT and CUSP. Although OpenCL offers a few alternatives[5] , they seem not as advanced as the CUDA supported libraries.

So, programmability and performance wise CUDA has a small edge on OpenCL. While the portability nature of OpenCL can be very interesting running tasks on MIC architectures for example.

The programmability was decisive for choosing CUDA. In addition, this project focusses only on GPUs other processor types are out of scope.

2.3.2 High abstraction compiler-based approaches

Where GPU kernels are automatically generated by compilers or language runtime systems, through the use of directives, algorithm templates, and sophisticated program analysis techniques, e.g.,

- ArrayFire[6]
- OpenACC[7]
- NMath[8]

ArrayFire

Arrayfire stands out of the companies mentioned above. In addition to CUDA, it also offers OpenCL support. This enables users to access also AMD GPUs and Intel accelerators. Originally, ArrayFire offers a C/C++ library but a java and R wrappers are also publicly available. At the time of writing ArrayFire became an open source product.

OpenACC

With OpenACC one could accelerate legacy code by using pragmas, compiler directives. Under the hood its contains OpenCL. OpenACC is only compatible with C/C++ and Fortran. One can expect to accelerate simple operations on large amount of data almost effortless. However, when logic becomes more complicated on could observe significant lower speed ups.

NMath

Confidential

2.4 GPU candidate applications OF

In this section we highlight a few OF solutions and discuss whether GPU acceleration is applicable or not. If offloading calculations to the GPU is applicable one should also consider if it is worth the investment, since GPU code is much harder to produce and maintain then logic in common languages in .NET or Java.

Confidential

[5] http://streamcomputing.eu/blog/2014-01-16/OpenCL-alternatives-for-cuda-linear-algebra-libraries/
[6] http://arrayfire.com/
[7] http://www.openacc-standard.org/
[8] http://www.centerspace.net/products/nmath

2.5 Proposal commercial GPU architecture

Confidential

3 Nested simulations

In this section we briefly describe the context of the accelerated models. As already mentioned, it is of great importance insurers to value their assets and liabilities by a market value instead of a the formerly used book value. By explaining a practical case we shed some light on the concept of a embedded options. We shortly discuss why the market value of an embedded options in for instance life insurance product is hard to value consistent Within an ALM study.

While using real world simulations for ALM, inner, risk neutral simulations are used for valuation. Market conditions of RW simulation nodes, should consistently be passed to the inner RN simulations. A methodology is described to let these inner simulations inherit consistent properties of its parent simulation node.

3.1 Example: Interest rate guarantees (IRG)

A life insurance product is essentially a contract in which fixed premiums from the insured are exchanged for a payment of a fixed amount in the future. Contracts like these are often bought as an retirement reserve. One can imagine that all kinds of flavors of this product are, or were, available in the market.

IRG

A significant amount of life insurance products contain IRG, unit linked and other profit sharing contracts for example. Interest rates tended to grow over time for a large period in history. In such a scenario the policy holder is better off investing on the market instead of signing the insurers' contract. For this reason, insurers added a profit sharing clauses to their contracts. The effect on the policy holder benefits are best illustrated in Figure 3. The figure shows, that in exchange of a small margin in favorable markets the policy holder is protected against unfavorable market conditions. The orange line highlights the yearly pay off for the policy holder. The policy holders benefit has the characteristic of a put option, since the holder is protected against low or negative returns.

In [9] some examples are given for these issued guarantees in legacy products:

- 4.00% in the Netherlands and Germany
- 4.75% in Belgium
- 5.00% in Italy

Policy holders owning such policies in the current market conditions benefit quite significantly with risk-free rates near to zero

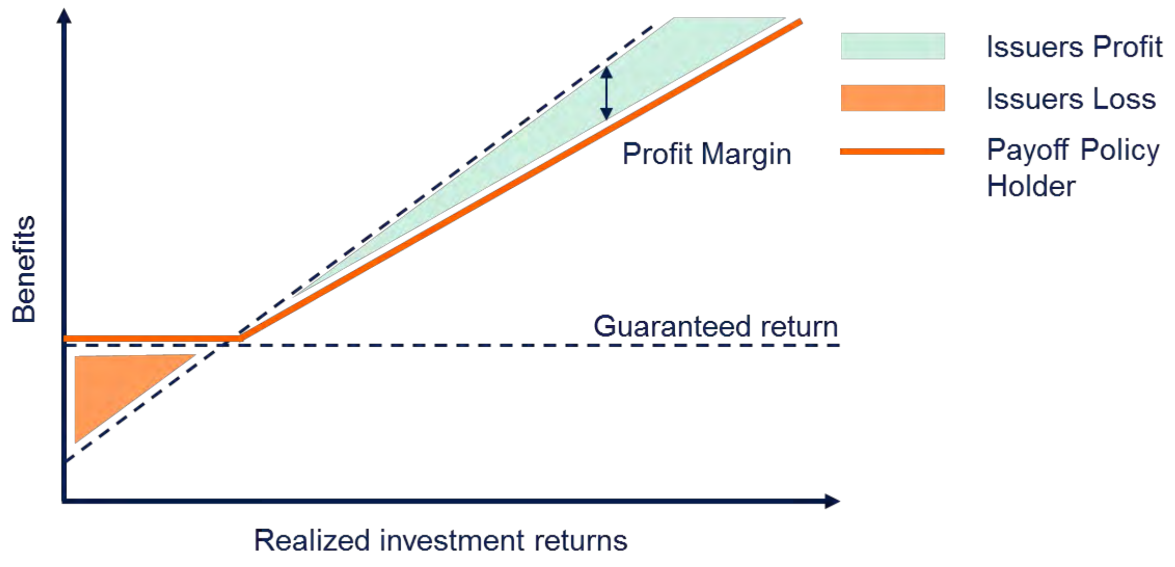

Figure 3 IRG payoff for the policy holder

In current market conditions, where interest rates are relatively low, life insurance products are not as popular anymore. Nevertheless, by the long maturing nature of this contracts, the gross amount of such products, listed as examples, on the balance sheet of an insurer today can be quite significant.

3.2 Nested simulations for insurers

Asset Liability Management (ALM)

The International Association of Insurance Supervisors [10] formulates asset and liability management as: The practice of managing a business so that decisions and actions taken with respect to assets and liabilities are coordinated. At OF such ALM studies are performed for several categories of institutional investors. By evaluating the balance sheet over multiple scenarios and time steps, one can gain insight in possible future developments. Since as previously described profit sharing contract are among others common liabilities for insurers. Within ALM it is of great importance to correctly value a portfolio of contracts like the profit sharing products we described since in certain market conditions their value can be dominant.

Valuation methods

The contract used in our example can, under assumptions, be priced by valuation of the fixed cash flows within. In Pelsser and Bouwknegt (2001) [11] a step by step explanation is presented on how such products can be priced.

So, let us assume that, for a large set of insurance contracts their fair value can be determined by a portfolio containing derivatives. How do we value such portfolio within a macroeconomic simulation like we perform in ALS studies. A possible solution is to use analytical formulas. However, most of the contracts are simply too complex to be able to mathematically define without making rigorous assumptions.

A common, but more computationally intensive method, is Monte Carlo (MC) simulation. Defined stochastic processes in financial models can be simulated and averaged. The accuracy of MC methods is a tradeoff between the number of scenarios and the computational time. Increasing the number of scenarios with a factor N commonly reduces the standard error with $\sqrt{N}$ [12]. Although there are methods available to increase the convergence, see [13].

The measure in which financial models, for option pricing, are simulated is what we call the risk neutral measure. This is due to the fact that these models are based on the assumption of ‘no-arbitrage’. This assumption states, in its simplest form, that the expected return is equal to the risk-free rate [14]. Consequently the MC simulations in our models are referred to as risk neutral (RN) simulations.

Nested simulation

Figure 4 illustrates the concept of nested simulation. The blue lines represent a macro economic scenario in which, given a policy and an investment strategy, the balance sheet of an institutional investor is evaluated. Such a scenario we further refer to as a real-world scenario. Subsequently, the simulation is called a real-world (RW) simulation. Within each point on the blue line of the figure, the model needs to value all assets and liabilities, including a possible portfolio of complex financial products. The RN simulations for valuation are represented by the orange lines. To illustrate the computational intensity of nested simulation; it is common that a RW simulation consists of 2.000 scenarios and 10 years. Consequently, 20.000 RN simulations have to be performed. For every RN simulation; 10.000 scenarios and over 10.000 periods are the be calculated. In [2] and [3] some techniques are discussed to deal with this dimensions. Nevertheless in this work we solve this computational nature by offloading the RN simulations to the GPU, and accordingly don't apply dimensions reduction techniques.

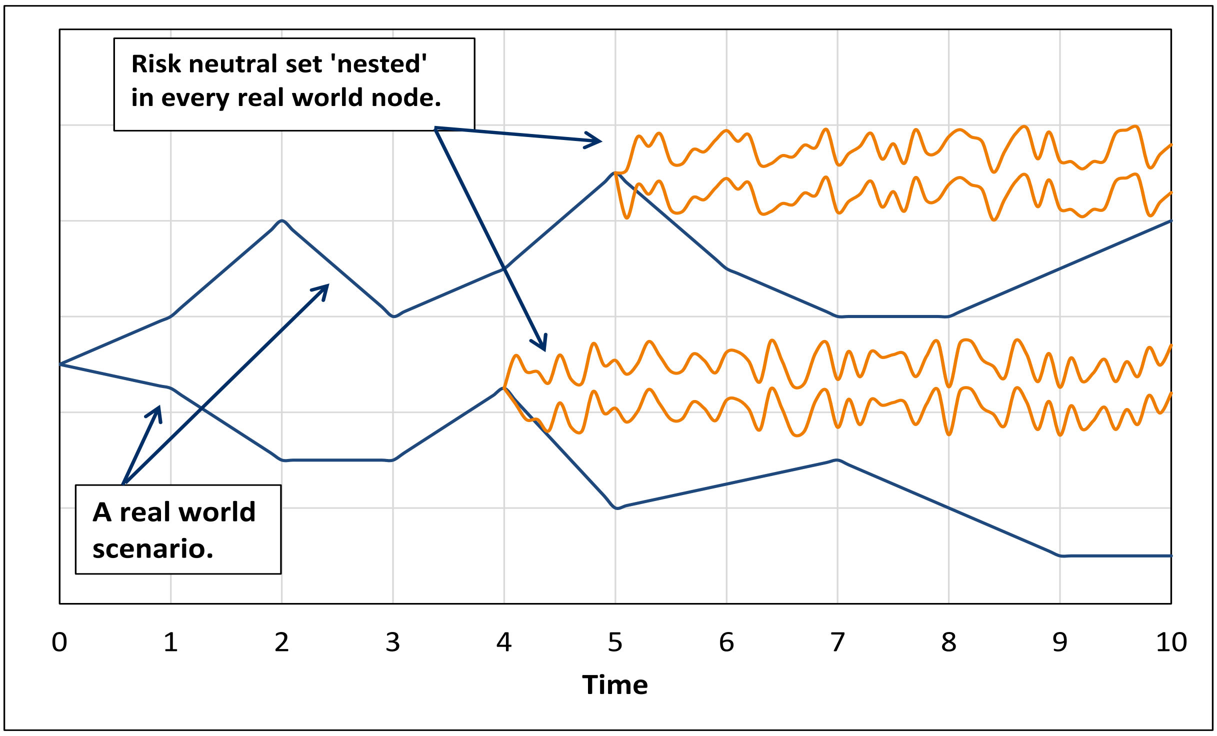

Figure 4 Example of nested simulation concept

3.3 Methodologies

In this section we present the methodologies which explain the need of the calibration of nested simulation applications. We show how consistency between the real-world and risk-neutral measures is obtained. In addition, the calibration needs are described in more detail.

Consistency

The main challenge of nested simulation is, that all 20.000 RN simulations of the previous example differ by the fact that simulated market conditions differ in each RW node. When a RN simulation is launched from a RW node, the model parameters of the RN simulation should capture all available information from the state of its parent RW simulation node, such as interest rate curves and implied volatilities. Accordingly, there is a need of a model that is able to let model parameters of a RN simulation inherit macroeconomic information of its parent RW simulation node.

By parameters models we mean, among others:

- Current levels of risk-free rates
- Volatility levels
- Levels of correlation
- Mean reversion parameters

Additionally, models can contain other specific parameters which no dot match the listed ones. The total number of parameters grows with the number of underlying assets for which financial models are calibrated. Currently, we deal with dozens of parameters per under lying asset.

By performing time series analysis techniques one would be able to model RW scenarios for each model parameter. In order to create consistency, in theory it also possible to embed macroeconomic properties in the parameter scenarios. However, implicit relations between the parameters are hard to capture. Additionally, Extending the macro economic scenario set with dozens of scenario models is expensive to develop and maintain.

Imposing structure to the model parameters

Confidential

Applications

Since this application requires tens of thousands RN simulation over the complete available history, current implementations can run for days. In our work we offloaded the calibration to the GPU as is described in paragraph 5.1. Subsequently when a historic series of x is extracted after the calibration, scenarios of x are constructed by the DSG. Having such scenarios available one is able to generate RN scenarios sets, as described in paragraph 5.2. Moreover, RN valuations can consistently be performed within a ALM model like is described in paragraph 5.3.

Equation 1 Dynamic Factor Model on Heston Euler model parameters

Confidential

4 GPU solution design

In this section we describe the requirements of the nested simulation applications and the techniques used to fulfill them.

4.1 Requirements

Usability

The nested simulation applications built in this project have a significant functionality overlap. Therefore one of the requirements is to build a general framework which is easier to maintain and expand then using separate models. In addition, the users of the models are not necessarily highly skilled in GPU computing or performance optimizations. The users should be able to expand and work with the models without having a thorough understanding of concepts like: coalesced memory access, GPU memory types, thread blocks and grids.

Performance

Besides the usability of the models, performance is another key requirement. Some of the nested simulation applications are still implemented for research purposes. Current implementations in Matlab and C++ are running in the order of days. To be able to further explore and tweak methodologies in this domain runtimes should be reduced significantly. Note that current models are not performance optimized.

4.2 Python

For prototyping purposes it is company policy to use Python. Python is an interpreted programming language, designed with the thought that code is read more often than it is written. One could expect a significant performance loss. However, Python becomes more and more accepted in the HPC industry. Implementations of Python functions are often in C or other low level languages. Even MKL[9] optimized methods are common. In addition, Python offers the ability to compile your programs in C with Cython.

Python use in HPC is associated with wrappers on libraries like MPI[10], openMP[11], CUDA, OpenCL and so forth. In our work we experienced a significant gain in programmability using Python, and in combination with BLAS, HPC and MKL optimized libraries the performance loss of using Python is not significant in the applications described here, because over 90% of the work is offloaded to either CUDA or other low level languages like C and MKL.

PyCUDA

PyCUDA[12] is a CUDA wrapper for Python. It covers the full CUDA API and also has several useful abstractions in order to increase the readability. In comparison with CUDA in C++ a significant part of the memory management (e.g. Garbage collection) is handled under the hood. In additions PyCUDA offers a few more high level abstractions. However, in our design we stayed as close as possible to original CUDA host code. Abstractions like GPU arrays, arrays where every element represents a CUDA thread, are unused. Because abstractions like these are not available when one takes the prototype in production.

PyCUDA is publicly available and mainly developed by Andreas Klöckner, who also wrote a wrapper for OpenCL, PyOpenCL[13].

[9] MKL is intel optimized Math Kernel Library in which routines for Linear algebra, engineering and several other applications are implemented.
[10] http://open-mpi.org/
[11] http://openmp.org/wp/
[12] http://mathema.tician.de/software/pycuda/
[13] http://mathema.tician.de/software/pyopencl/

4.3 CUDA Streams and kernel concurrency

When having the opportunity of building a framework from scratch, one aims to generalize the problem as much as possible. The ambition to use generic building blocks for our logic lead to exploring the CUDA stream functionality. Also the concurrency features of streams matched well with our performance needs. After briefly discussing a traditional design, we explain some of the techniques used as backbone of the GPU accelerated applications.

4.3.1 Design without streams and hyper-Q

Traditionally a developer has to put a lot of thought in designing heterogeneous applications that fully utilize the available hardware. We show by example some of the difficulties focusing on kernel concurrency and programmability.

Kernel concurrency

Within our applications, the tasks to be offloaded to the GPU have the characteristic of not fully utilizing the GPU on their own. In order to increase GPU utilization multiple computational tasks have to run concurrently. Without the concept of streams, a custom implementation could become quite complex.

One needs to fit the work of multiple computational tasks within a single kernel execution, an aggregated kernel. This routine becomes complicated when for example:

- The input for a aggregated kernel launch comes from multiple processes or is not simultaneously available.
- If, as in our applications, a sequence of tasks is to be offloaded to the GPU, performance is heavily dependent on launch dimensions, Threads per block and blocks per grid.
- If the dimensions of the problem change, the aggregated kernel execution should by adjusted by hand which is difficult to administer or dynamically which is difficult to implement.

Programmability

The host code for getting full GPU utilization would be very specific to the application and hardware. This makes such code hard to interpret, develop and maintain. In addition, one should customize device kernels to the sequence in which they are executed. Preferably one would be able to reuse certain kernels in a different setting.

4.3.2 CUDA streams and Hyper-Queues

To tackle the difficulties sketched in the previous section we developed a framework based on a set of CUDA features. The most important such feature is the usage of CUDA streams.

Streams are an abstraction of a series of tasks for the GPU. By tasks, we mean:

- Memory copies, both device to host (D2H) as host to device (H2D);
- Synchronizations;
- Computational tasks (Kernels);

The tasks in a stream are ordered FIFO. This gives a clean abstraction and therefore programmability to the problem. Within nested simulation, for example, we repeat a sequence of tasks for every scenario in every period. The stream gives us a useful abstraction of this sequence of tasks by supporting reuse and parameterization.

The programmability gain is one example of the advantages of using streams in an application. Performance wise they have to following features:

- Streams can run concurrently under certain conditions.
- Tasks for different parts of the hardware within streams can, run concurrently:
  - Computationally tasks (kernels), executed on the GPUs stream processors
  - Memory copies (D2H and H2D), executed on the GPUs copy engines
- Stream launches can be asynchronous, which provides the CPU to compute while the GPU is running concurrently

Requirements for concurrency

We can define three types of concurrency:

- CPU – GPU concurrency
- Compute and memory overlap on GPU
- Concurrent Kernel execution

CPU – GPU concurrency

Stream launches are by default asynchronous. However, memory copies are by default synchronous. Launching a stream with a regular memory copy will occupy the host thread. Nevertheless, under certain conditions streams containing memory copies can run concurrently [4]:

- Memory copies are initialized with the asynchronous API
  - The 'async' functions contain an optional stream parameter. If one leaves this blank the default stream will be used. The default stream is always synchronous and thus breaks concurrency.
- Host memory allocations in copies are pinned or page-locked.

Additionally, kernel launches should be invoked with the asynchronous API as well.

One can break the CPU – GPU concurrency by synchronizing all GPU tasks to a point from which one would for instance transfer the intermediate or final results to the CPU. Obviously the program needs to make sure that all necessary work on the device is finished before communicating the results. If synchronization between device and host is required, this can be accomplished by either several CUDA synchronize methods or by implicit synchronization. The latter is the main attribute of the difficulty of working with streams. When host commands for the GPU are invoked without a stream, the GPU assigns a default streams. The default stream differs from all other launched streams by consuming the full resources. In other words if one accidently allocates a variable through the default stream, present concurrency will be broken.

Furthermore pinning memory is not always obvious in a programming languages other than C and C++. In Python we used the Ctypes[14] package to pin memory allocations. The reason why pinned memory is crucial for CPU – GPU is obvious, when the CPU is running separate calculations, the GPU needs reserved memory space to read or write from. If this memory space is not pinned, it could be that the OS will access and manipulates the memory. The driver will check these conditions, therefore concurrency between device and host is only possible if all host memory allocations access by the device or pinned.

Compute and memory overlap

Within a stream it is possible to overlap the memory transfers with computational work by invoking the tasks with the asynchronous API. If this is crucial for the application performance, one might try to chunk the work such the overlap is optimal. Furthermore, memory copies of stream A can overlap the computations of stream B.

[14] https://pypi.python.org/pypi/ctypes?

Concurrent Kernel execution

To schedule each launched stream the driver has a Queuing system in place. If resources are available, additional work is to be executed. When a stream arrives at the device in a work queue, it is scheduled by the CUDA Work Distributor (CWD). Latest developments in NVidia cards increase the concurrency possibilities of the streams within a work queue. This is accomplished by NVidia's Hyper-Q feature.

Concurrent kernel execution is purely bounded by computational capacity and the device's architecture. The latter is at the time of writing a dominant factor. Architecture Kepler and higher support hyper-Queuing [5]. He concept of the Hyper-Q is best illustrated in Figure 5 and Figure 6. The first figure shows the scheduled work for a Fermi GPU. The yellow circles highlight the possible overlap between to tasks. As one can observe from the figure, only two tasks can run concurrent on the Fermi architecture [16].The Hyper-Q feature displayed in Figure 6, however, shows multiple concurrency dimensions. In fact, a Kepler device is capable of running 32 concurrent streams.

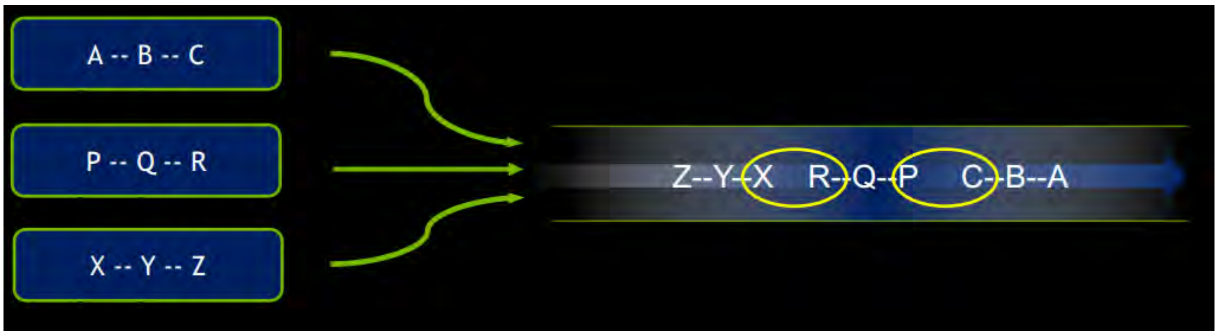

Figure 5 Task queuing on Fermi architecture (source: CUDA streams, Best practices and common pitfalls – Justin Luitjens NVidia)

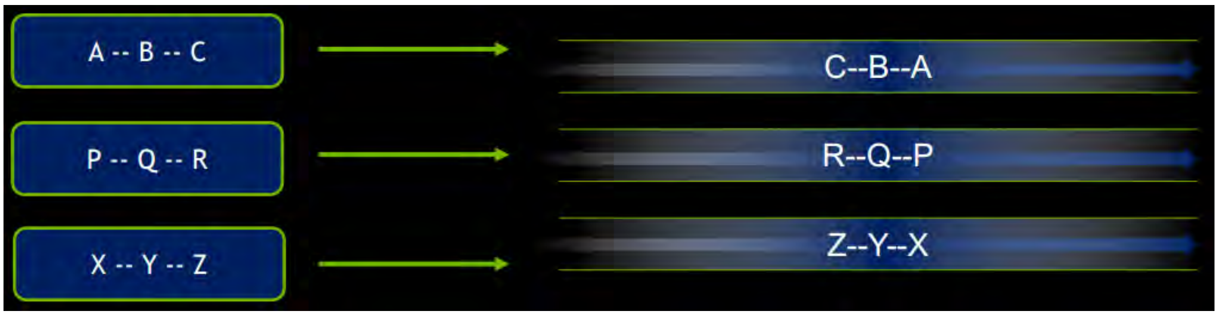

Figure 6 Task queuing (Hyper-Q) on Kepler architecture (source: CUDA streams, Best practices and common pitfalls – Justin Luitjens NVidia)

Testing on Kepler devices, we observed that reducing the amounts of blocks in the kernel launch had the effect that concurrency indeed increased. However, at a certain, point the runtime increased as well. It turned out that when launch dimensions are reduced the concurrency increases and, at some point, the duration of a single tasks increases subsequently. This observation is important in two ways: First is states that we have to care less on optimal launch dimensions. Since lowering the resources per stream will increase concurrency and keeps the GPU at least as busy. We do not need NVidia's Occupancy Calculator[15] for example. And secondly, the framework does not suffer significant overhead. When streams are large enough to utilize the complete device, increasing duration results in increasing concurrency thus, it is a zero sum game. However, the latter only holds if there are a sufficient amount of streams to be executed by the device.

In the proposed framework the hyper-Q feature will be essential since risk neutral simulations are 'small enough' to run concurrently in groups of 4 to 16 streams. Consequently, the application will perform best on Kepler or higher architectures. Figure 7 and Figure 8 illustrate a use case of Fermi versus Kepler architectures. Both figures display the concurrency of streams containing two

[15] https://devtalk.nvidia.com/default/topic/368105/cuda-occupancy-calculator-helps-pick-optimal-thread-block-size/

subsequent kernels: A large 'generatePaths' kernel followed by the kernel 'determinePayOff'. In Figure 7 we observe a two way concurrency. However, because a relative small task is running concurrent with a much larger one hardware utilization in this case is very low. On the contrary, Figure 8, displays the benefits of the hyper-Q allowing higher concurrency between streams resulting in higher hardware utilizations.

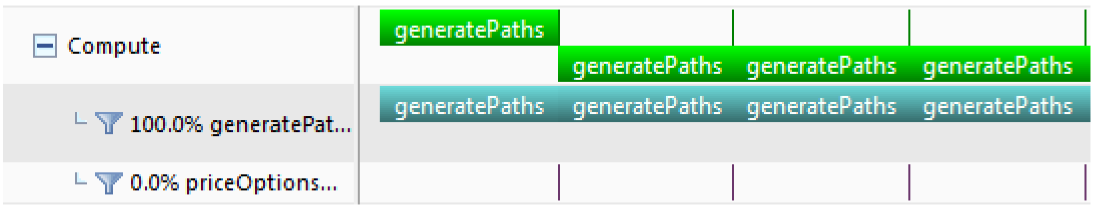

Figure 7 Stream concurrency NVidia Tesla C2050 (Fermi) – Visual profiler

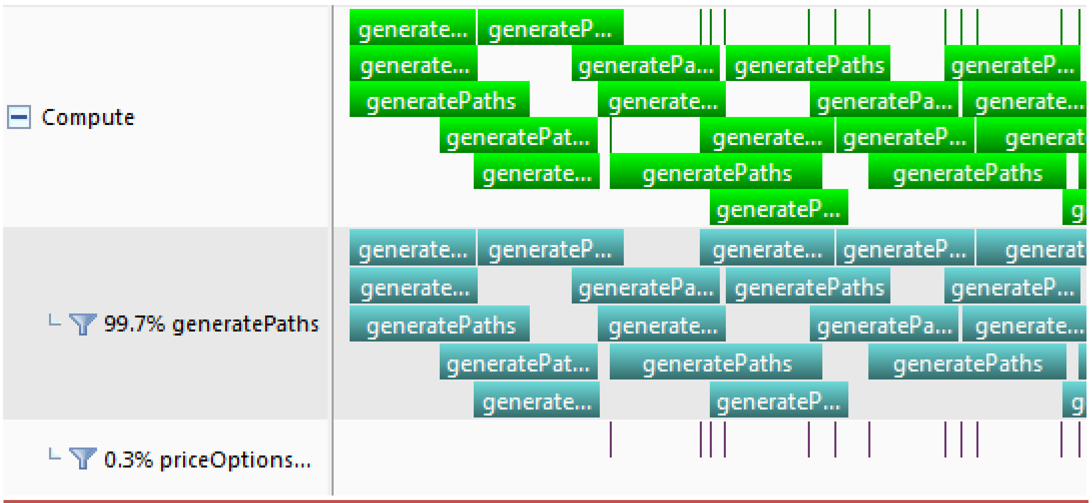

Figure 8 Stream concurrency on NVidia Tesla K20 (Kepler) - Visual profiler

4.3.3 Employment of streams and Hyper-Q

Figure 9 displays an overview on how streams are used within the proposed framework. The figure shows, concurrency between host and device and between a number of streams. The hyper-Q takes care of optimal hardware utilization by scheduling the streams concurrently. By launching streams asynchronously the GPU calculations can overlap with CPU tasks. These tasks can consist of calculations, real-world simulation steps or memory flushing to hard drive as we see later in the document. Typically, the stream launching and the pre- and post-processing tasks described in the figure run in the order of milliseconds. For nested simulations, the challenge is to perform all risk neutral simulation tasks within the timeframe of a real world simulation.

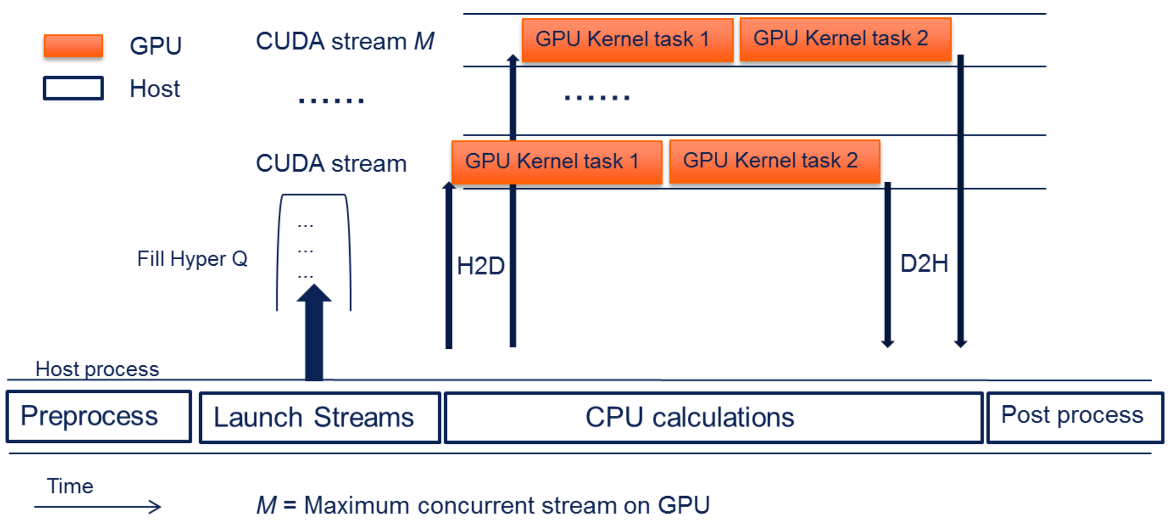

Figure 9 Basics concept of stream usage within framework

Figure 9 shows the main concept of using the hyper-Q feature in our applications. As the figure shows, the CPU prepares the workload for the GPU and launches the work in streams to the device. The device receives the streams and stacks the work in a Hyper-Q. Note here that the issue order of the streams is not necessarily the order of execution. After all streams are launched by the CPU, the local process can continue other work while the device is running the tasks in the streams concurrently. As mentioned earlier, it is important to have all local memory allocations page-locked. If this is not the case, concurrency will break.

Our applications are using the technical backbone displayed in Figure 9. The orange highlighted blocks are interchangeable and reusable between the applications. In the next chapter we will present multiple designs derived from the framework in Figure 9.

The number of streams running concurrently on the device is bounded by the available resources. Because of this one should carefully consider the block and grid dimensions of computational tasks within a stream. Running a light task on a large grid will cause the stream to consume more resources then necessary and can result in insufficient hardware utilization.

In our applications, the use of hardware resources is mainly determined by the number of scenarios used in the risk neutral measure and the number of options to price on a single scenario cloud.

4.4 Multi-processing service (MPS)

Previous work on offloading streams in a multithreaded or multiprocessing environment [17], showed significant GPU utilization in benchmark cases. In a similar way we implemented the ideas of [17] for Python processes.

In order to launch streams from different processes to a single GPU, NVidia developed a multi-processing service [6]. This software layer provides a context manager to handle work launched from different processes. MPS is exclusively available on Linux and is only provided with NVidia Tesla cards with compute capability 5 or higher. Although these restrictions limit applicability, it is a relatively cheap way to explore the concept of kernel offloading from multiple processes to a single GPU within a nested simulation. This is an important feature, since within nested simulation applications, the outer simulations can be performed on multiple cores or even multiple machines. In order to share one or more GPUs between these local or distributed processes, one needs a managing software layer. Because GPUs are connected to a single local process by something called a context.

Figure 10 and Figure 11 show the basics of MPS. If two processes both create their own contexts from which they launch computational work. The GPU scheduler needs to switch context, excluding any form of concurrency of the tasks from A and B, as Figure 10 shows. In principle, this is not on issue if the tasks from both processes are large enough to fully utilize the GPU. However, in our application we typically deal with smaller workloads. With nested simulation: real world simulations are typically run in a multi core setting. When each process of real world simulation launches streams for the GPU, the need of a service like MPS becomes clear.

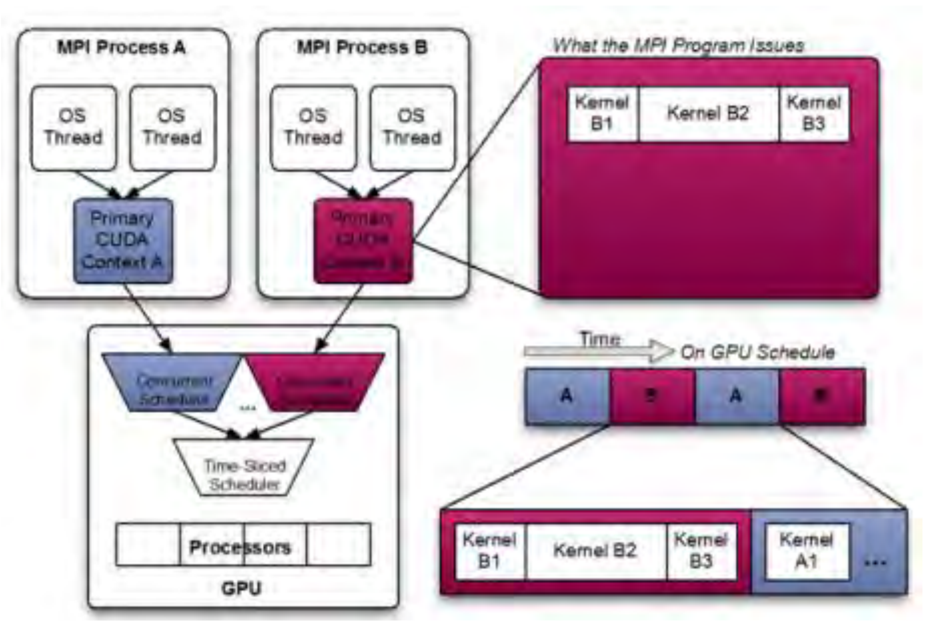

Figure 10 kernel offloading from multiple processes (source: NVidia MPS manual)

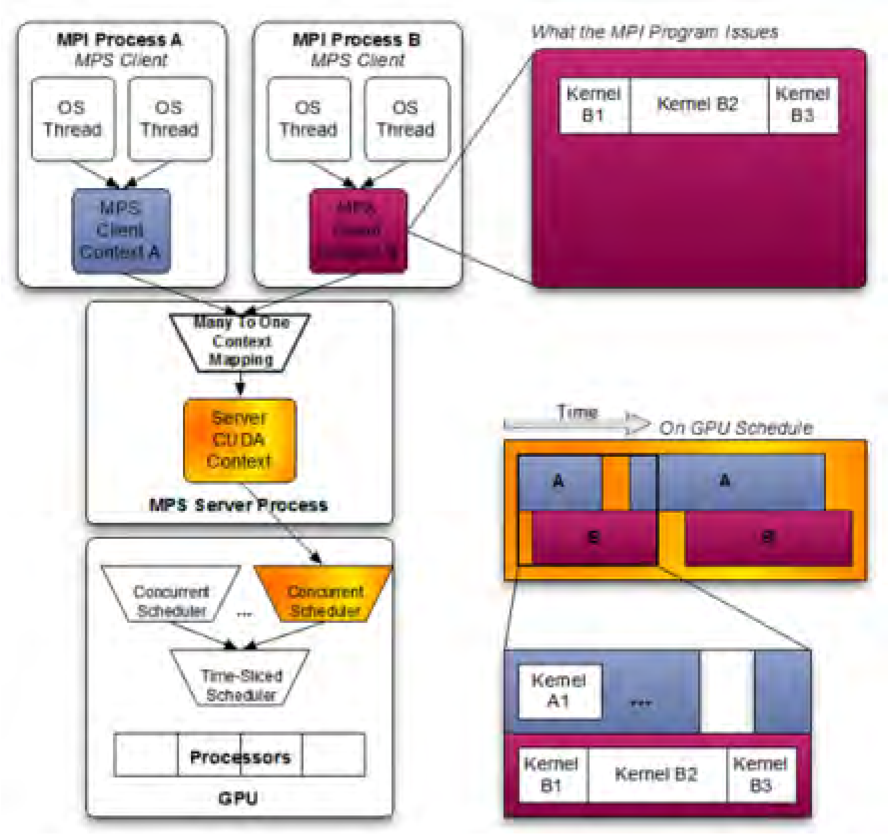

Figure 11 Kernel offloading with MPS (source: NVidia MPS manual)

Setting up MPS

For setting up the MPS a Python class, MPSManager is designed. This class manages the initialization and termination with simple methods like start_server() and stop_server(). Within this methods environmental variables are set as described in [6]

Drawbacks

One major drawback for this set up in our application is that memory is not shared between processes. The random numbers for instance, are allocated *n* times if, where *n* represents the number of processes which are associated with the MPS. The GPU virtualizes its memory in *n* partitions. For large simulation dimensions, this will result in out-of-memory errors. A way to deal with this problem is the usage of the Inter-process communication (IPC) feature of CUDA. We did not further investigate. this feature. Another, possibly more effective way is to design a service like MPS yourself. Given the OS and card limitations on the availability of MPS this might seem a good alternative for OF.

4.5 Single vs. double precision

GPUs are most effective in computational tasks that use single precision (SP). As Figure 2 already showed, one would like to keep calculations in single precision for two reasons: Most GPUs perform significantly better when using SP, and GPUs that support fast double precision (still at most half of the performance of SP) are very expensive.

To investigate the effect of using SP instead of DP in our applications we examined the two main building blocks of the kernels: Path generation and the pay-off functions.

Single precision in path generation

First, we compared the paths with the single precision paths generated by the CPU. We observed that the relative delta is less than 10^{-5} and the absolute delta is less than 10^{-8} for values in the interval [20-200]. So, even in SP, GPU results match CPU results.

If we do the same comparison with double precision CPU generated paths we observe slightly larger differences. In addition, the difference between both path sets seem to be normally distributed around zero as Figure 12 shows. But what is most important is that the statistical properties of the risk neutral scenarios are not affected by SP. Therefore we designed a test suite which can verify the arbitrage free conditions of generated paths. The tests discount the simulated values of the last period back to the initial period. If arbitrage free conditions are met by the Martingale property [14] the expected returns equal the risk free rate.

$$\mathbb{E}^{\mathbb{Q}}\left[e^{-\int_t^{t+\Delta t} r(u)du} S(t+\Delta t) \mid S(t)\right] = S(t) < \infty,$$

Equation 2 Martingale property of MC paths

Consequently , when the mean of the discounted values equals the start value of the underlying, the test passes. These tests are also called Martingale test in literature. We can conclude that our GPU generated paths satisfy the arbitrage free conditions on all implemented models. These Martingale tests are incorporated in the unit test suite of the delivered Python packages.

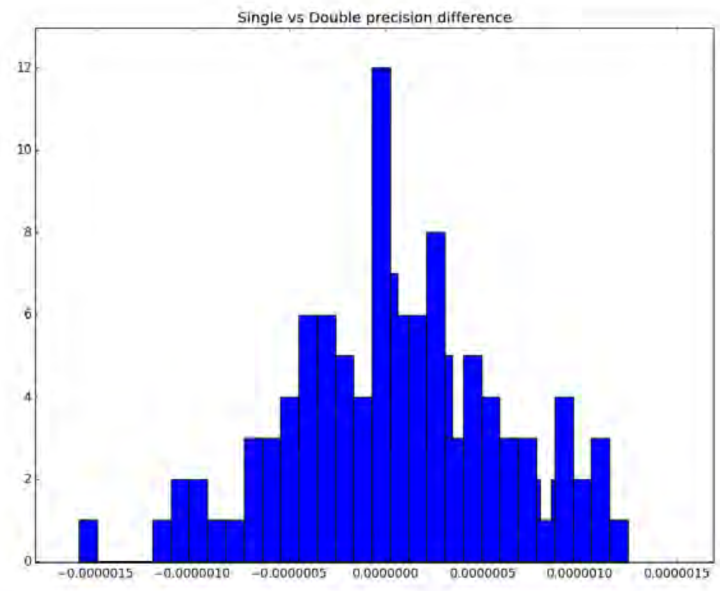

Figure 12 Histogram of the difference in the last period of sample Heston-Euler paths generated by the GPU (SP) and the CPU (DP)

Pay-off function

In the payoff function the sum of the payoff is returned. This procedure is sensitive to rounding errors caused by single precision usage. Truncation of each element before it is summed can result in an under estimation. For this reason, we used a reduce sum implementation which NVidia also used in their American option pricing example. By recursively summing items, the potential accumulated rounding error is of order $O(\log n)$ instead of order $O(n)$ when a naïve summation is performed [18].

5 Applications

During the master project, we developed two Python packages and made a mock up model to simulate the performance behavior of nested simulations in existing models. The released Python packages contain a framework to be used by the consultants and can be extended with different financial models. In the development process we started using an analytical traceable model: Black-Scholes [19] (BS). By starting with this relative simple model first, we were able to unit test most of the simulation framework. After the BS implementation, the models were extended using more advanced models like Heston [20] and Cheyette [21]. Both extensions served as a use case for the applicability of the underlying technical architecture. In the following sections we discuss the three nested simulation applications and its design. Subsequently we evaluated its performance. Finally we recap on some guidelines for using the proposed framework for model extensions.

5.1 Calibration model

In order to absorb the economic conditions from a real world scenario (the parent simulation) in the risk neutral simulation the model parameters of the risk neutral models should inherit properties of the state of its parent. As is explained in section 3.3, a Dynamic factor model (DFM) is used to incorporate time dependent and scenario dependent information in the model parameters. In the calibration process, we are interested in a constant a and b and a time depended variable x. We started to model the volatility of the BS model[19] in the DFM setting first. After covering most of the simulation logic with unit tests we extend the calibration model with a stochastic volatility model, Heston-Euler. The package is released with an example script where monthly S&P 500 implied volatilities are calibrated with the DFM. In this case, we calibrated on 106 historic months of option data.

The model consists of a few building blocks represented by Python classes, which are described further.

5.1.1 Simulators

In essence a simulator can be initialized with option data and simulation settings. A simulator contains the instruction set for the calibrator. In Figure 13 the simulator is passed to the calibration engine after de data preprocessing constructed it. Furthermore, results will be stored in the simulator object after the calibration process is finished. Currently, the abstract simulator is implemented in different flavors:

- Black-Scholes
- Heston – Euler (stochastic volatility)
- Heston – Euler - Cheyette (stochastic volatility and interest rates)

All simulators have a CPU and a GPU implementation.

The GPU simulators contain the core of the proposed technical architecture. It is this class that is reused in all nested simulation applications. Implementing an additional simulator would imply that it can be used in other nested simulation applications.

5.1.2 Evaluators

In order to evaluate the goodness of fit, each evaluation method is represented by a class. This is a clean way to pass evaluation methods into functions. Each evaluator is initialized by market data, passing the calibrated data to the evaluator it returns the goodness of fit. We implemented the following methods:

- Mean Absolute Errors (MAE)
- Sum of Squared Errors (SSE)
- Mean Relative Errors (MRE)
- R-squared
- Adjusted R-Squared
- Weighted SSE and MRE

The weighted SSE and MRE is used during test runs. This method takes in to account, for example, that an absolute error of 2.00 on a value of 600 is less significant on the goodness of fit than when this error was observed on a value of 10.00. In addition, relatively large errors on small values (like a 10% error on 0.05) are also less significant on the measure then a 10% error on 200.

5.1.3 Calibration Engines

The Calibration engine can be initialized by passing the simulator, an evaluator, and the optimization method to apply. Currently, all minimizing optimizers from Pythons SciPy [16]package are available. Most of them have efficient C implementations under the hood. Within this thesis we run all tests with the Powell method [22]. Powell's method is an unconstrained non-gradient method. The latter seems important since the objective function was minimizing an error measure based on Monte Carlo valuations. Taking derivatives of this complex function without any mathematical definition could result in false iterations caused by the standard error a MC value holds.

Calibrating implied volatility data, we use the putt call parity in order to calculate both market put and call option values. Namely, calibrating only put or call values will result in implicit heavy weights for far in the money options. By calibrating on both, prices balance this effect by heaving in the money option prices over the whole moneyness spectrum.

5.1.4 GPU details

The orange highlighted parts in Figure 13 are GPU accelerated. This cycle is called thousands of times during the calibration. Consequently, it consumes more than 90% of the runtime. All logic in the highlighted sections of Figure 13 is located in the simulator class.

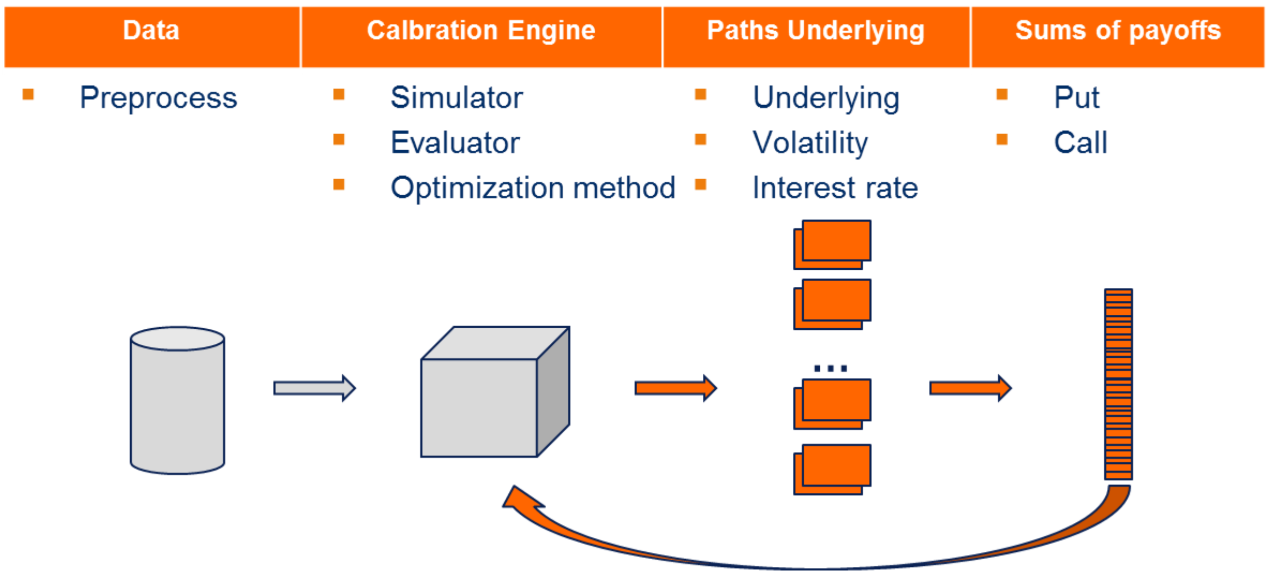

Figure 13 GPU accelerated part of calibration

Details on how the GPU offloading is done are displayed in Figure 14. This is a zoomed view on the orange highlighted proces of Figure 13. For every function evaluation, the model parameters have to be constructed first for all historic points. The CPU then launches a stream for every historic point. In our testcase have 106 months of historic quotes available. Each quote contains a volatility surface for 5 maturities and 11 different strike values. The number of streams the CPU launches is determined by the maximum amount of global memory available to temporarily store the generated paths on the device. In the preprocessing step, this maximum amount is determined and memory is allocated accordingly. If the GPU global memory is not sufficient to allocate all paths for all streams the CPU chunks the workload and memory blocks are reused.

[16] http://www.scipy.org/

Since the GPU can only run a limited number of streams concurrently, the work piles up in the hyper-Q. Subsequently, the workschedueler schedules each stream untitl the queue is empty. Meanwhile, results of finished streams are transported back to the host and evaluated.After the CPU synchonizes with the GPU to make sure every historic point is calculated, the evaluator caluculates the goodness of fit. Finally, this measure is returned as the objective function result. The optimizer generates a new set of parameters and the sequence repeats itself.

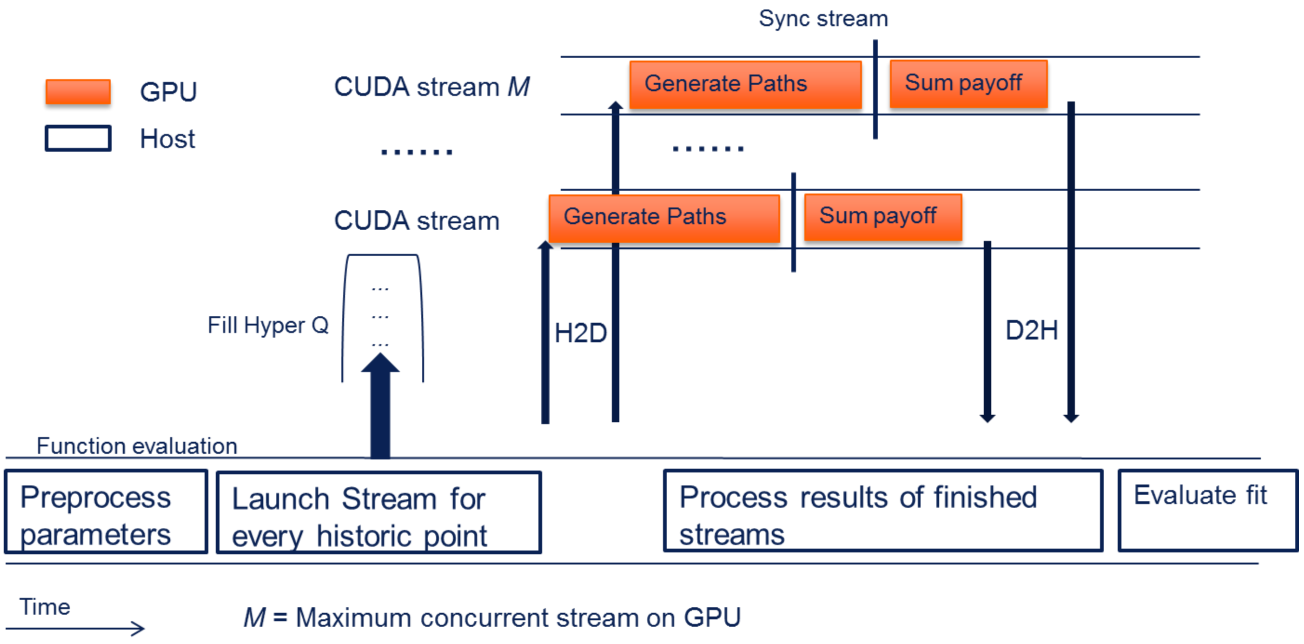

Figure 14 The usage of streams within the calibration process

For performance measures, we choose to communicate the user time speed-up of one function evaluation. Although a function evaluation undergoes CPU logic as well it is a well understood time to communicate. Very important to note is that the CPU implementation is not optimized, and runs at least for this application single threaded. However, most of the logic runs in C or MKL optimized libraries. The runtime of a single function evaluation is captured and compared in Table 1. At first sight one could argue the need of acceleration when a function already runs in the order of seconds. However, because the optimizer is calling this function tens of thousands of times the impact of any optimization will be significant.

5.1.5 Performance

Run	Path generations p/s	Options p/s	Function evaluation
CPU			
Calibration 1.000	24	1317	4.38 (s)
GPU			
Calibration 1.000	4.213	231.733	0.025 (s)
Calibration 10.000	280	15.390	0.375 (s)

Table 1 Calibration performance for a single function evaluation

We managed to reduce the runtime of the calibration process from the order of days to the order of minutes. This results are very satisfying since one could further explore and expand the methodology of the dimension reduction by the DFM and the nested simulations.

5.2 Risk Neutral Scenario generator

There are examples of clients which are interested in OF Risk Neutral (RN) scenarios only. Often such clients have fairly complex payoff functions of derivatives they want to interpreted the RN scenarios themselves. To be able to sell RN scenarios separately, one should export the paths. For the GPU, this is a new challenge. The model uses a lot of components of the calibration model, however, no pay off kernels are included and large amounts of data have to be transferred. The latter is the biggest challenge since all data transfers are done over the PCIe bus. Since RN scenario sets can vary from 64 GB to several terabytes these memory transfers a bottleneck. Accordingly, there is a great need to overlap memory transfers with computations.

For storing such large sets of data we use the HDF5[17] format. This enables us to easily control flushing memory from RAM to the hard drive. In these applications, the RN set contains thousands of scenario clouds[18]. If a set of clouds are available in RAM, the application can flush them to the RN set situated on the hard drive by a single command.

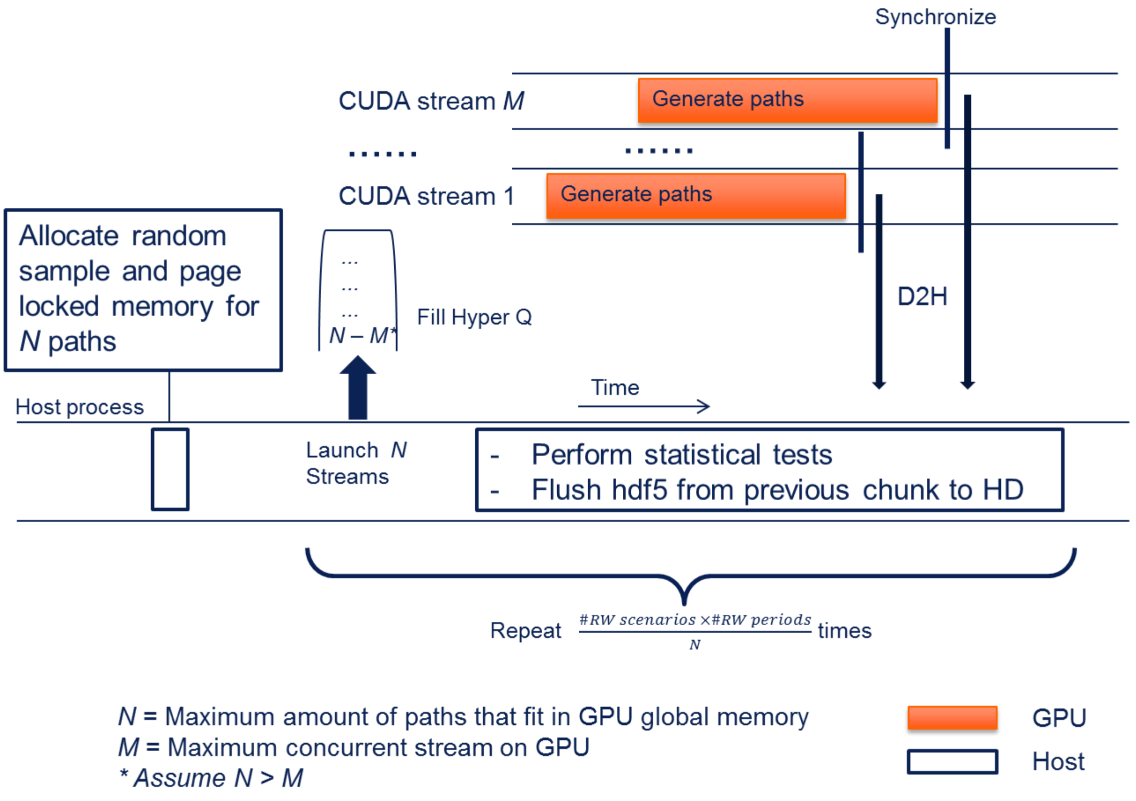
Synchronize
CUDA stream *M*
Generate paths
......
......
CUDA stream 1
Generate paths
Allocate random sample and page locked memory for *N* paths
...
...
...
*N – M**
Fill Hyper Q
D2H
Time
Host process
Launch *N* Streams
- Perform statistical tests
- Flush hdf5 from previous chunk to HD
Repeat $\frac{\#RW\ scenarios \times \#RW\ periods}{N}$ times
N = Maximum amount of paths that fit in GPU global memory
M = Maximum concurrent stream on GPU
* *Assume N > M*
GPU
Host

Figure 15 Design RN scenario export

5.2.1 GPU details

Figure 15 displays the technical framework of the application. One should notice that the design is similar to the calibration model. The basics are the same, however, the CPU is managing hard drive transfers while the GPU is generating the scenarios. As in the calibration model the maximum number of streams (*N*) to be launched by the CPU is dynamically determined on the available resources. All local allocations are page locked so CPU and GPU concurrency is ensured.

[17] http://www.hdfgroup.org/HDF5/ for more details on this format

[18] A scenario cloud is the possible evaluation of a asset class presented as a matrix with number of the scenarios as rows and the number of periods as columns, typically (10.000 by 3.600)

5.2.2 Performance

There were no implementations available yet to compare the performance of our solution. Generating a RN set of 64 Gigabytes (GB) takes approximately 20 minutes on the GPU accelerated model. Subsequently, the model generates 50 MB of scenarios per second, this is more or less the maximum writing speed of HDF5 files to the systems hard drive[19]. Accordingly, we can conclude that for this application, the hard drive writing speed seem to be is the bottleneck. On could consider other hard drive architecture, but this is not in scope of the project.

5.3 Nested Simulations

Chapter 3, already described the need of nested simulation in practice. In current models from OF, the concept of nested simulation is not yet implemented. Instead, analytical estimations based on modelled implied volatilities are used. For this reason we build a mock up model of the ALS solution from OF. In this mockup model, outer simulations are performed by a sleep statement. Since we are interested in the impact of the performance we assumed three benchmark cases. They differ in the duration of a real world simulation step per scenario per period. We assumed durations of 75, 150 and 300 milliseconds for respectively a light, medium and a heavy case.

Figure 16 displays the concept of offloading risk neutral calculations to the GPU, in comparison with a sequential version. With this concept, the goal is to perform all risk neutral tasks within the runtime of a real world period. This way no performance loss will be observed.

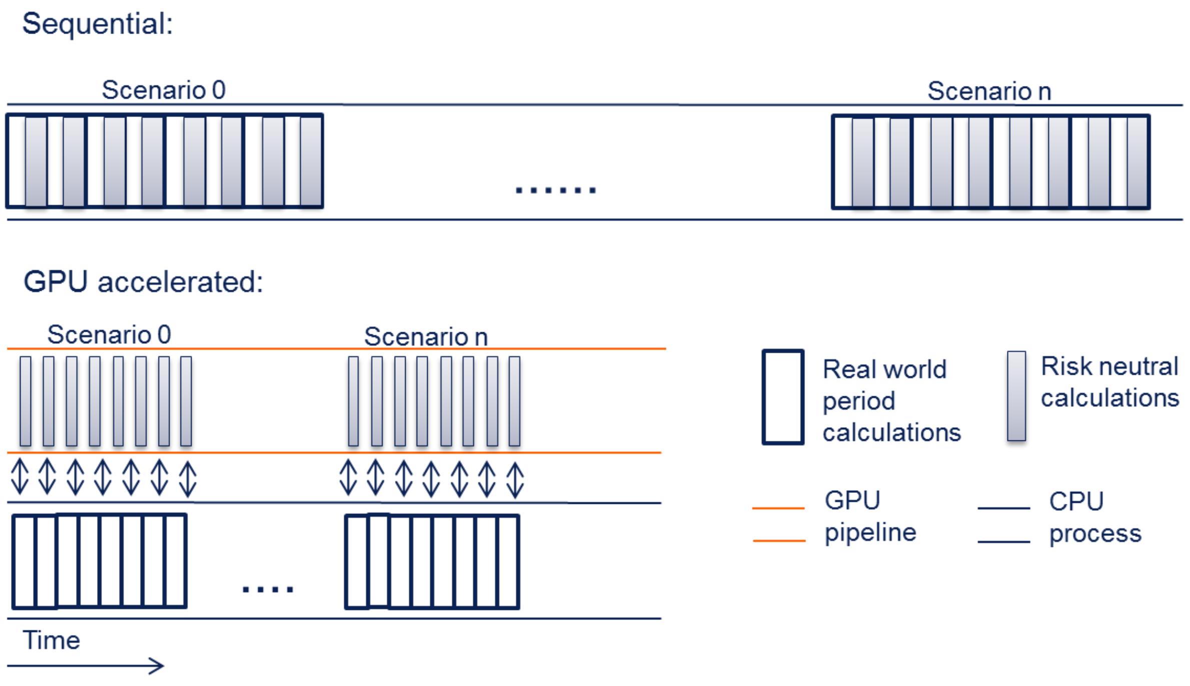

Figure 16 Task parallelism in a nested simulation

In contrast with Figure 16, the current ALS solution (section 2.4) performs real world simulation in a multicore setting. This makes it more challenging to finish the offloaded risk neutral simulation within the duration of its parent real world simulation step.

Again, we make use of the main concept of the framework previously used in the calibration and RN generation models. As Figure 17 shows, each process is launching streams for its risk neutral calculations tasks. We use the MPS functionality to manage all GPU requests from different processes. A daemon process is handling all GPU requests from its slave processes. The daemon process puts all the streams in a single hyper-Q. This results in concurrent stream execution over the different processes.

[19] https://www.hdfgroup.org/projects/DirectChunkWrite/

If resources are not sufficient, streams stack within the hyper-Q. Again, page locked memory is crucial for performance, otherwise the CPU, running the real world simulation, and the GPU cannot run concurrently.

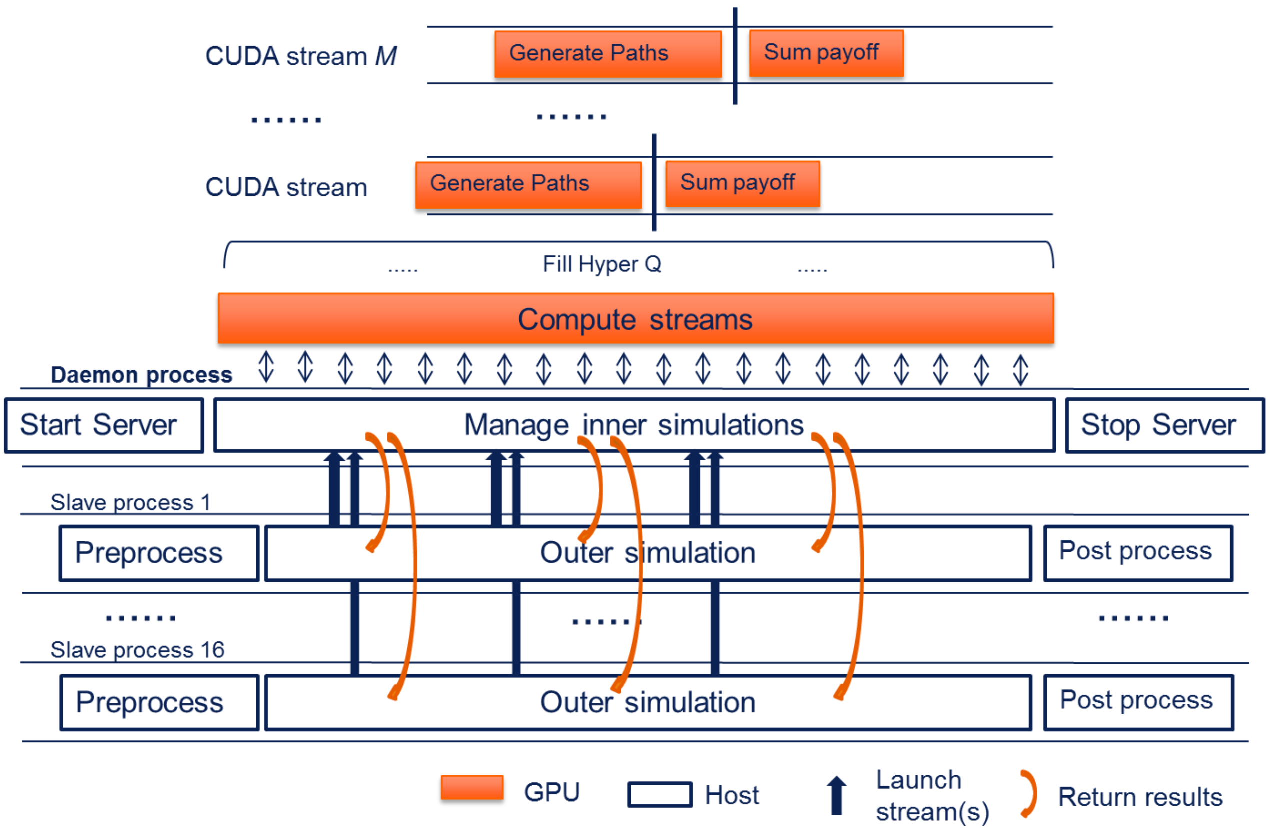

Figure 17 Diagram nested simulation with MPS

Within this framework, it is possible to not only include local but also external processes. When running the real world simulation over multiple machines, MPS will still be capable of sharing the GPU resources. Consequently, if the GPU is still underutilized by one machine, one could distribute work to additional machines. This, however, is part of future work and not in scope of the thesis. In addition, the compatibility with multiple virtual machines sharing GPUs is of interest.

On the other hand, if GPUs are being fully utilized, it is possible to add more GPUs. Running our models in a multi GPU setting is also part of future work. Because we already use streams, it is mainly a matter of distributing the streams and it allocations over multiple devices.

5.3.1 Performance

To get an idea of the performance of the proposed framework, the benchmark real world simulations are used (light, medium and heavy cases). While adding the additional work of a risk neutral simulation for each period and scenario of the real world simulation, we compare the measured performance with a theoretical performance, where we assume strong scaling, of the runtimes against the number of processors.

We run the model with the following dimensions:

- Real world simulation:
  - Number of scenarios: 2.000
  - Number of periods: 5
- Risk neutral simulation:
  - Number of scenarios: 1.024
  - Number of periods: 12.000

The theoretical reference runtimes and the measured times are displayed in Table 2. We observe that if the number of processes increases the GPU calculations become a bottleneck. Additionally when a real world simulation durations are high, The GPU is keeping up the work from up to 8 processes.

Reference runtimes (s)			
Benchmark Case	Light	Medium	Heavy
# Cores			
1	750.0	1500.0	3000.0
2	375.0	750.0	1500.0
4	187.5	375.0	750.0
8	93.8	187.5	375.0
16	46.9	93.8	187.5

Estimated runtime With MPS			
Benchmark Case	Light	Medium	Heavy
# Cores			
1	736.9	1496.9	3078.3
2	395.0	768.9	1519.4
4	221.3	390.9	763.9
8	221.4	233.6	385.4
16	199.8	205.7	236.0

Table 2 Left: Reference runtimes real world simulation mock-up model. Right: Actual runtimes with MPS

Table 3 shows the relative performance losses on the real-world simulation runtimes of the mock-up models. In addition, we propose the number of GPUs and compute nodes to be used when one aims to eliminate the performance loss.

Performance penalty Nested simulation (MPS)			
Benchmark Case	Light	Medium	Heavy
# Cores			
1	0%	0%	3%
2	5%	3%	1%
4	18%	4%	2%
8	136%	25%	3%
16	326%	119%	26%

🟩	Add Compute nodes
🟧	Add 1 GPU
🟧	Add 2 GPUs
🟧	Add 4 GPUs

Table 3 Effect of nested simulation on reference runtimes

The results in Table 3 indicate that, if resources can be scaled, the proposed architecture would in theory be able to reduce the performance impact of inner simulations to zero. Although, scalability of the architecture is not implemented yet we can conclude that the proposed architecture is most promising.

Custom MPS

Eventually, one would implemented a custom MPS-like service. By custom building such a service, the user has full control on memory allocations. Sharing memory allocations between processes running through MPS turned out to require NVidias Inter process communication (IPC) feature. However, the usage of IPC with MPS is only available from NVidia toolkit 6.0 [17] or higher and since the DAS-4 cluster[20] runs on toolkit 5.5 [18] we were not able to run such implementations. Furthermore customizing MPS eliminates the requirement to run on NVidia tesla cards only.

[20] http://www.cs.vu.nl/das4/ , This is the cluster on which most of our tests were run.

5.4 Result Validations

For the calibration models, it is of great importance that results can be validated and examined. In this section we shed some light on the result analyzing and export submodules build for the calibration model.

Singor et al [25] showed that for fitting historic S&P500 option data with the Heston model, the model parameters can be constructed by a linear combination of constants and the VIX index. In some way this is similar the DFM described in section 3.3. because it is also a linear combination of a time dependent variable. In other words: If the model is correct the decomposed vector x should show a high correlation with the VIX index. Figure 18 displays that the VIX index is indeed highly correlated with the results of the calibrated values for x. This result both validates the model and the claim stated in [25] .

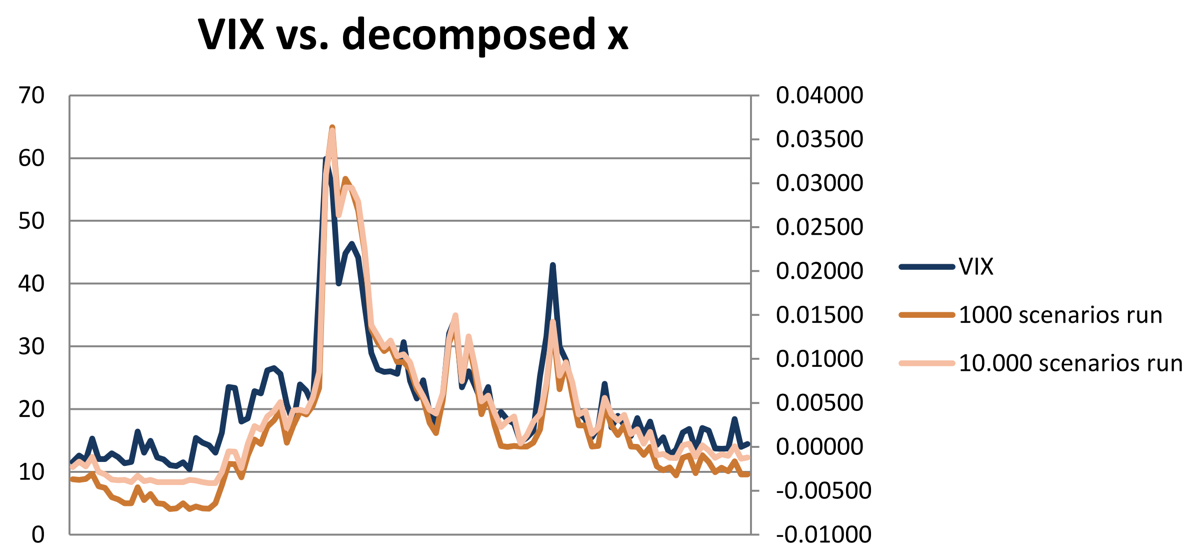

Figure 18 Historic decomposed x against the VIX index

After the calibration is done, all results are stored in a result analyzer. Since the calibrations are performed on option prices, the analyzer transforms them back to implied volatilities. Subsequently , statistics are calculated on every historic fit and on the complete set. To get a good glance of the calibrations performance, the results are exported and molded into a report. An example of such a report is available in

Appendix A: Example report.

5.5 Guidelines for model extensions

So far, we discussed three applications making use of the framework proposed in section 4.3.3. In this section we shortly recap designing guidelines for (future) model extensions. During the final stages of the project we extended the calibration model with the calibration of swaption data. This extension enabled the model to generate risk neutral interest rate scenarios consistently with the already available model for the S&P500. To implement this extensions we extended the current model in three areas:

- Adding new instrument types: Instead of plain vanilla European options one would for example be interested in more exotic one like swaptions.
- Adding underlying assets: We already discussed the S&P500 as underlying assets for plain vanilla European options. Swaptions, however, have multiple interest rates as underlying.
- Adding financial models:. By adding the swaption functionality we need a stochastic interest rate model. Multiple models are available, we choose to use the Cheyette model [21], [26] in this case.

Per area we will address on high level some guidelines for future model modifications or extensions. The guidelines are summarized by bullets.

Adding new instrument types

Confidential

Adding underlying assets

Confidential

Adding financial models

Confidential

Testing

While adding new financial models it can be of great value to export the generated paths to the CPU during the development. This way one could develop test driven until all statistic properties of the newly generated paths match the expectations.

6 Conclusions

Nested simulation applications are becoming increasingly important for institutional investors. Combining real world simulations with risk neutral simulations enables asset and liability studies for balance sheets containing complex financial products. By example, we showed that a common financial product like a life insurance can contain such complexity. Due to the intensive computationally nature of nested simulation applications CPU implementations lack to ability to explore and develop methodologies on this subject. In this thesis we proposed a GPU accelerated simulation framework in which with we implanted three nested simulation applications.

Allowing researchers to explore methodologies in a GPU accelerated matter is often accompanied by high level libraries. However, for most of the applications, these libraries are not sufficient and low level implementations on the GPU are required. For instance, former implementations of the calibration model in Matlab were not running faster using Matlabs GPU arrays. Due to low array dimensions, the GPU was not fully utilized and the memory transfer overhead became a dominant factor. Consequently, programmability often comes with a price: performance loss. In this thesis, we accelerated a set of financial applications by using a framework containing CUDA Streams, Hyper-Q and NVidias MPS. These features enabled us to build flexible simulation framework for financial nested simulations. The GPU accelerated simulation framework is applied in three nested simulation applications. The framework enables model extensions to be implemented fairly easy, as we observed during the extension of a stochastic interest rate model.

Additionally, the framework makes extensive use of features increasing kernel concurrency. We showed that running our models on GPU architectures Kepler or higher improves concurrency without observing overhead in cases where concurrency was limited.

Furthermore, the CUDA stream abstraction enabled the user to have a clean host side framework which enables the user to effectively implement model extensions without having advanced knowledge of the GPU hardware.

Each nested simulation application had its own challenges. Some creativity was needed in order to fit each application in the framework while utilizing the GPU capabilities effectively. For the Calibration model, achieving high concurrency in small scale computations was of great importance and the hyper-Q usage was crucial in achieving this concurrency. Implicit synchronizations, when one is not carefully following the concurrency guidelines for streams, was the main difficulty during the implementation.

The main challenge in the RN scenario generation was to overlap CPU an GPU computations. The Cache and flush mechanism on the CPU needed to run concurrently with the scenario generation on the GPU. By reusing pre allocated page-locked memory, we enabled such concurrency. The performance bottleneck for this application turned out to be the writing speed of the hard drive on the host.

Building the mock-up model to simulate the performance for an existing OF solution we faced multiple local processes from which calculations had to be offloaded to the GPU. Traditionally NVidia's MPS was built to share a GPU between MPI processes. We customized the MPS functionality in order to handle local Python processes instead. Herewith we achieved concurrency between stream owned by different local processes.

The performance of the accelerated models satisfied the stakeholders to a large extend. In some cases runtimes were reduced by two orders of magnitude resulting in runtimes in the order of minutes instead of days. This time reduction allows researchers to further explore and refine methodologies in the domain of nested simulations.

Finally, by proposing a scalable architecture we claim that in theory the additional work of inner simulations for existing OF solutions does not necessarily have to impact the solutions runtime. Future work will focus on applying this theory in practice. Running our applications in a scalable cloud platform is of great use. When an application is dynamically able to manage GPU resources in a virtual environment, one would be able to take the concepts described in this thesis in production. Furthermore the calibration model will be extended in order to support a broader range of asset classes.

7 Bibliography

[1] S. N. Singor, “Efficient Simulation and Valuation of Embedded Options using Monte Carlo Simulations,” 2009.

[2] S. Morrison, “Nested Simulation for Economic Capital,” Dec-2009. [Online]. Available: http://www.barrhibb.com/research_and_insights/article/nested_simulation_for_economic_capital. [Accessed: 31-Mar-2015].

[3] Singor, S. N. and Oosterlee, C.W., “Avoiding Nested Simulations.”

[4] J. Luitjens, “CUDA Streams - Best practices and pitfalls.” NVidia Corp, 2014.

[5] NVidia Corp, “Kepler GK110 Architecture Whitepaper, v1.0.” 2012.

[6] NVidia Corp, “Sharing a GPU between MPI processes: Multi-Process Service (MPS) Overview.” Oct-2013.

[7] B. van Werkhoven, “Performance models for CPU-GPU data transfers,” *14th IEEEACM Int. Symp. Clust. Cloud Grid Comput. CCGRID*.

[8] J. Fang, A. L. Varbanescu, and H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL,” in *2011 International Conference on Parallel Processing (ICPP)*, 2011, pp. 216–225.

[9] G. Finkelstein, E. McWilliam, S. Nagle, P. Beus, de, R. Leijenhorst, van, L. Maas, and J. Cui, “Guarantee and embedded options.” Jun-2003.

[10] *Standard on Asset-Liability Management*, vol. No. 13. International Association of Insurance Supervisors, 2006.

[11] P. Bouwknegt and A. Pelsser, “Market Value of Insurance Contracts with Profit Sharing,” Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 267996, Mar. 2001.

[12] A. Shapiro and T. Homem-de-Mello, “On rate of convergence of Monte Carlo approximations of stochastic programs,” *SIAM J. Optim.*, vol. 11, pp. 70–86, 2001.

[13] J. P. C. Kleijnen, A. A. N. Ridder, and R. Y. Rubinstein, *Variance Reduction Techniques in Monte Carlo Methods*. .

[14] J. C. Hull, *Options, Futures, and Other Derivatives*, 9th edition. Prentice Hall, 2014.

[15] S. N. Singor, A. Boer, and C. W. Oosterlee, “Consistent risk neutral valuation of embedded options in insurance contracts via nested simulations.” Paper is in submission.

[16] NVidia Corp, “Fermi Compute Architecture Whitepaper, v1.1.” 2009.

[17] Wende, F, Steinke, T, and Cordes, F, “Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture,” *ZIB-Rep. 14-19 June 2014*.

[18] N. J. Higham, “The Accuracy Of Floating Point Summation,” *SIAM J Sci Comput*, vol. 14, pp. 783–799, 1993.

[19] F. Black and M. Scholes, “The pricing of Options and Corporate Liabilities,” *J. Polit. Econ.*, vol. Volume 81, no. 3, pp. 637–654, Jun. 1993.

[20] Heston, S.L., “A closed-form solution for options with stochastic volatility with applications to bond and currency options.,” *Rev. Financ. Stud. 6 2*, pp. 327 – 343, 1993.

[21] I. Beyna, “The Cheyette Model Class,” in *Interest Rate Derivatives*, Springer Berlin Heidelberg, 2013, pp. 3–15.

[22] M. J. D. Powell, “An efficient method for finding the minimum of a function of several variables without calculating derivatives,” *Comput. J.*, vol. 7, no. 2, pp. 155–162, Jan. 1964.

[23] NVidia Corp, *CUDA C Programming Guide, v6.0*. 2014.

[24] NVidia Corp, *CUDA C Programming Guide, v5.5*. 2013.

[25] S. N. Singor, A. Boer, and C. W. Oosterlee, “Modeling the dynamics of equity index option implied volatilities in a real world scenario.,” *Methodol. 2014-02*, Feb. 2014.

[26] B. Hoorens, D. D. Kandhai, and M. C. G. Sterling, *“On the Cheyette short rate model with stochastic volatility.”* 2011.

8 Appendix A: Example report

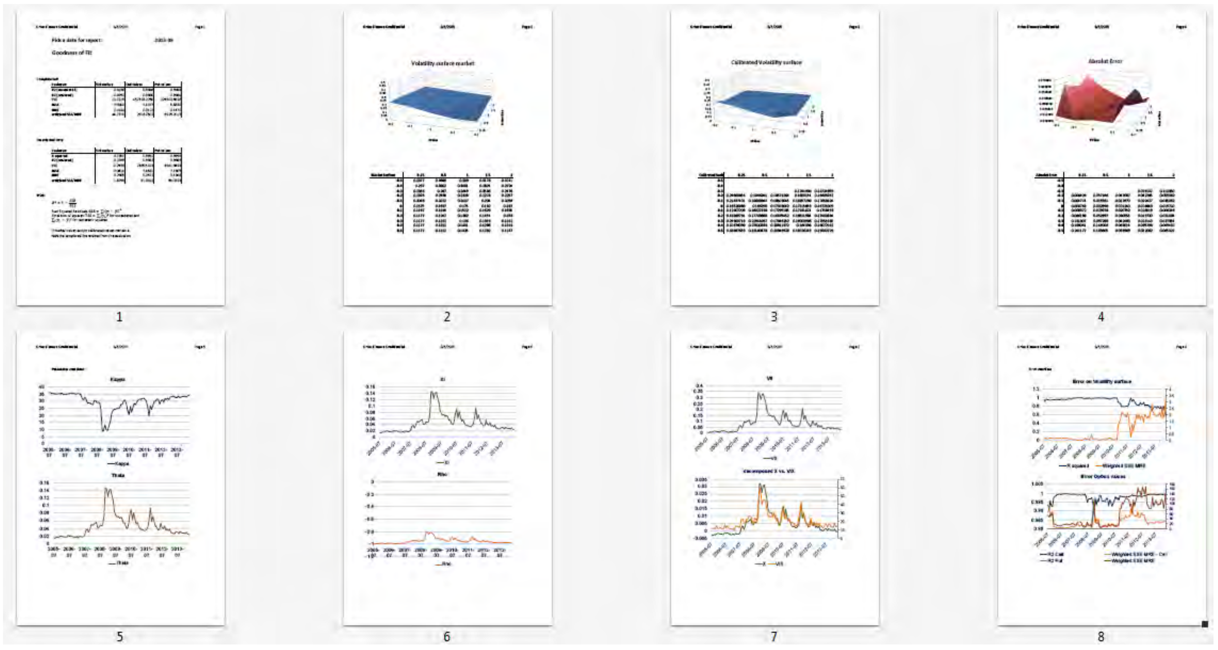

Figure 19 Overview of Calibration result report

Figure 19 shows an overview of the report the calibration model outputs. Surface plot of the market and calibrated volatility are presented for a user picked date. In addition, the error on the fit is shown in the red surface plot. The last four pages contain model parameters and goodness of fit over time.

Ortec Finance bv
Boompjes 40
3011 XB Rotterdam
The Netherlands
Tel. +31 (0)10 700 50 00
Fax +31 (0)10 700 50 01

Ortec Finance bv
Barajasweg 10
1043 CP Amsterdam
The Netherlands
Tel. +31 (0)20 700 97 00
Fax +31 (0)20 700 97 01

Ortec Finance Ltd
23 Austin Friars
London EC2N 2QP
United Kingdom
Tel. +44 (0)20 3178 3913
Fax +44 (0)20 3178 6164

Ortec Finance AG
Poststrasse 4
8808 Pfäffikon SZ
Switzerland
Tel. +41 (0)55 410 38 38
Fax +41 (0)55 410 80 36

www.ortec-finance.com

ORTEC
FINANCE