Running Financial Risk Management Applications on FPGA in the Amazon Cloud

Javier Alejandro Varela, Norbert Wehn
Microelectronic Systems Design Research Group,
University of Kaiserslautern,
67663 Kaiserslautern, Germany
{varela,wehn}@eit.uni-kl.de

Abstract

Nowadays, risk analysis and management is a core part of the daily operations in the financial industry, and strictly enforced by regulatory agencies. At the same time, large financial corporations have started migrating their operations into cloud services. Since the latter use a pay-per-use business model, there is a real need for implementations with high performance and energy efficiency, in order to reduce the overall cost of operation. And this is exactly where field programmable gate arrays (FPGAs) have a great advantage as accelerators, which enables the designer to efficiently offload the compute-intensive parts of the workload from the host central processing unit (CPU). The advantages associated to FPGAs have recently conferred them a place in cloud services, such as the Amazon Elastic Compute Cloud (EC2).

In this work, we concentrate on widely-employed portfolio risk measures, namely value-at-risk and expected shortfall, by means of compute-intensive nested Monte Carlo-based simulations. As a proof-of-concept, we have implemented and assessed the performance of these simulations on a representative complex portfolio, where the workload is efficiently distributed between CPU and FPGA on an Amazon EC2 F1 instance. Due to a lack of direct power measurement capabilities in the cloud, energy efficiency cannot be properly assessed at the moment. Nevertheless, our FPGA implementation achieves a 10x speedup on the compute intensive part of the code, compared to an optimized parallel implementation on multicore CPU, and it delivers a 3.5x speedup at application level for the given setup.
1 Introduction

A field programmable gate array (FPGA) is a programmable device that offers a large set of hardware components (such as flip-flops (FFs), lookup tables (LUTs), digital signal processors (DSPs) and Block RAMs (BRAMs)) that can be configured and interconnected to perform any given task (see e.g. [1,2]). This provides FPGAs with four main advantages:

1. the possibility of creating fully pipelined architectures,
2. (almost) infinite bit-level parallelism,
3. custom precision data-types,
4. custom memory hierarchies.

This greatly differs from the fixed hardware architecture of multicore CPUs (see e.g. [3,4]), which are general-purpose platforms with good performance per core/thread [5] (e.g. Intel’s Xeon processors). Whereas CPUs execute a software code, FPGAs offer a flexible architecture that can be efficiently adapted to the target application. Moreover, the power consumption of FPGAs is in general significantly low, which makes them an ideal accelerator when high performance-per-watt is required. These advantages have recently confered FPGAs a place in high-performance datacenters and cloud computing, such as in the Amazon Elastic Compute Cloud (EC2) (see e.g. [6,7]).

Nowadays we experience an increasing adoption of cloud computing, which is a pay-per-use service that gives the users on-demand access to a shared pool of resources [8,9]. As large financial institutions are now gradually moving their operations into the cloud (see e.g. [10,11]), we see there a great potential for FPGA-based solutions for the financial industry.

This is not a new concept: for example, Maxeler Technologies is a long-standing worldwide provider of FPGA-based systems for compute-intensive financial applications, including recent cloud-based solutions [12,14]. As a matter of fact, the vision on the use of FPGAs in high-performance computing systems and datacenters has been around for many years. This vision has been consequently followed by our Microelectronic Systems Design Group over the last decade, where our research has been focus on the application of FPGAs to different fields, including finance, in order to provide massive acceleration and high energy efficiency. In the case of financial applications, our research timeline is summarized in Fig. [1] which is the outcome of a
successful interdisciplinary research cooperation between computer engineering and financial mathematics, in the scope of the Research Training Group (RTG) 1932, funded by the German Research Foundation (DFG).

In the case of the financial industry, stricter regulations have been set in place after the 2007-2008 financial crisis. And in this regard, widely used risk measures, such as value-at-risk (VaR) and expected shortfall (ES) (also called conditional value-at-risk (cVaR)), need to be monitored and reported regularly, as part of the Basel III regulatory framework [36,37].

As the financial operations are moved into the cloud, we see two complementary needs: from the user point of view, there is a need for high-performance implementations that can minimize the overall runtime and, therefore, the total cost of execution; whereas the service provider requires an energy efficient infrastructure that can minimize the total cost of operation. As a proof-of-concept, we have implemented and assessed the Monte Carlo (MC)-based simulation of VaR and ES risk measures for a representative portfolio running on an FPGA-based F1 instance in the Amazon cloud (EC2). Since at the moment power measurements are not available in the cloud, we focus on the runtime performance (from the user point of view). To this end we employ state-of-the-art algorithmic optimizations (see [38]),
and we efficiently split the workload between the host CPU and the FPGA.

For the implementation on FPGA we employ the Xilinx SDAccel development environment, which enables the use of Open Compute Language (OpenCL) code on Xilinx FPGAs. Implementation details are given in the following sections, along with a performance comparison against an optimized code running on multicore CPU using Open Multi-Processing (OpenMP).

2 Background

Cloud computing is a relatively new computing paradigm, where users are provided with on-demand access to a shared pool of computing resources (such as computing nodes, applications, and storage) which can be configured as required by the application, with minimum management effort from the user standpoint [8]. This is a shift from the traditional approach, where large companies (e.g. in the financial industry) were forced to developed and maintain proprietary technology and large in-house hardware/software infrastructures [11]. With cloud computing, the resources are (at least in principle) available on-demand. However, the pay-per-use business model behind it makes it necessary to develop high-performance implementations that can mainly reduce the overall runtime and, therefore, the overall costs.

In this work, our goal is to present a proof-of-concept implementation of a financial risk management application running on an F1 instance (CPU+ FPGA) in the Amazon Cloud (EC2). To this end, we employ OpenCL (targeting an heterogeneous hardware configuration), the Xilinx SDAccel development framework for FPGAs, and the OpenMP application programming interface (API) for multicore CPU (used as a comparison).

The following sections cover the necessary background on OpenCL, Xilinx SDAccel, OpenMP, as well as financial risk management and option pricing.

2.1 OpenCL

The Open Compute Language (OpenCL) is an open, royalty-free standard managed by the nonprofit technology consortium Khronos Group, and it is widely used nowadays for general-purpose parallel programming. OpenCL supports both data- and task-based parallel programming models [39], and it was designed to make to code portable across all platforms that supports
In order to achieve the maximum performance in each case, optimizations are usually required based on the underlying architecture.

This framework assumes the presence of a host CPU and an accelerator (called device) connected via Peripheral Component Interconnect Express (PCIe), as shown in Fig. 2 together with the basic memory hierarchy. The host CPU handles the main application and the corresponding OpenCL setup steps (see e.g. [40]), whereas the code that runs on the device (the kernel) contains the code to be executed by each work-item (thread). A device contains compute units, which are subdivided into processing elements.

The kernels are enqueued by the host as an N-Dimensional Range (ND-Range), with a number of work-items grouped into work-groups, or as a Task, which is basically a single-threaded kernel. On the device global memory (refer to Fig. 2), the buffers are allocated by the host CPU, which are then used for all data transfers between host and device, as well as by the kernels.

OpenCL is also available for FPGAs, as detailed in the following section.

2.2 OpenCL on FPGAs: Xilinx SDAccel

In the case of FPGAs, the OpenCL code has to be compiled and then translated into a configuration file (bitstream), required for the hardware configuration. SDAccel is a vendor-specific development environment targeting Xilinx FPGAs, and it supports kernels expressed in OpenCL C, C/C++ and RTL (SystemVerilog, Verilog or VHDL) [41].

A Xilinx PCIe-based hardware device consists of a static region (with handles the PCIe connection) and a programmable region (where the kernels...
will be implemented) [41]. Since the static region needs to run continuously (to keep the PCIe connection working), the programmable region is configured using the partial reconfiguration technique (for more details, see e.g. [41]).

2.3 OpenMP

Open Multi-Processing (OpenMP) is an application programming interface (API) managed by the nonprofit technology consortium OpenMP Architecture Review Board, and it has been developed for shared-memory parallel programs targeting multiprocessor systems. OpenMP makes use of lightweight threads, and it supports the fork-join programming model [42]. These threads are basically runtime entities able to independently execute a stream of instructions, by sharing resources of the parent process (see [42]).

OpenMP provides features to be added to a sequential program (e.g. Fortran or C/C++), which describe the work allocation among the different threads, as well as the accesses to shared data. The section of code to run in parallel is basically marked with a compiler directive (pragma), and an id is automatically assigned to each thread. The main advantage of OpenMP is its ease of use and applicability to parallelize existing sequential code [42].

2.4 Financial Risk Management

Whenever a financial investment is made, it is usually diversified into a portfolio of different financial products, such as securities (e.g. stocks and bonds), foreign currency, cash (to provide a certain level of liquidity), and derivatives (which are products that derive their price from an underlying asset).

At a given horizon time, any given portfolio will be subject to a profit or a loss (P&L), exposing the original investment to some risk (see e.g. [43]). In practice, this risk exposure is typically measured by means of the value-at-risk (VaR) and expected shortfall (ES) [44], and they are enforced by regulatory agencies [36, 37]. VaR states with probability $\alpha$ that the portfolio is not expected to suffer a loss of more than $VaR$ $\$ in the next $N$ days, where $\alpha$ is usually as high as 95% or 99%, whereas ES provides an estimate of the expected loss (it is basically an average of the worst cases).

Three methods are available to compute VaR and ES (see e.g. [45]): 1) the variance-covariance method, which employs an approximation of the portfolio loss function and assumes a multivariate normal distribution of the risk
factors; 2) the historical simulation, which relies on past data on the individual returns (therefore limited to past events only), and the (nested) MC simulation of the portfolio loss function, which is considered to be the most general and versatile approach.

In mathematical terms, the MC estimator for VaR is given by

\[ \text{VaR}_\alpha(\tilde{L}_{t+\tau}) \equiv \inf_{\ell \in \mathbb{R}} \{ \tilde{F}_{t+\tau}(\ell) \geq \alpha \}, \]  

(1)

where \( \tilde{F}_{t+\tau}(\ell) \) is the empirical distribution of the simulated loss distribution \( \tilde{L}_{t+\tau} \) at time \( t + \tau \). In the case of ES, the MC estimator is given by

\[ \text{ES}_\alpha(\tilde{L}_{t+\tau}) \equiv \frac{1}{1 - \alpha} \int_0^1 \text{VaR}_\gamma(\tilde{L}_{t+\tau}) d\gamma. \]  

(2)

The drawback of the MC approach is its usually long runtime, especially if combined with the revaluation of the full portfolio \[46\]. As an example, a report presented by McKinsey & Company in 2012 showed that only 15% of the surveyed financial institutions would use this approach, with a downward tendency, whereas easier but less accurate approaches \[47\], such as historical simulation or first order approximations \[46,47\], would be preferred.

The runtime of the MC simulation is dependent on the complexity of the pricing algorithms used for the corresponding products in the portfolio. Nowadays, typical portfolios include options (a derivative contract) in order to keep the risk exposure of the given portfolio under control. An option is basically a contract that gives the holder the right, but not the obligation, to buy or sell an underlying asset at a specified price and time \[44\]. Except for very simple options, complex types usually require their own simulation, creating the nested MC simulation exemplified in Fig. 3: an external part for the underlying asset, and an internal part for the option pricing.

To generate the paths in the MC simulations, we employ the Black-Scholes (BS) model. This model has one input parameter, and it is relatively simple but very robust and widely used in practice (see e.g. \[44\]). Its discretized version under the Euler-Maruyama scheme is represented by:

\[ S_{t+1} = S_t \exp \left( (r - \frac{1}{2} \sigma^2) dt + \sigma \sqrt{dt} Z \right), \]  

(3)

where \( r \) is the risk-free interest rate (and assuming zero dividend yield), \( dt \) is the time step, and \( Z \) is a random number (RN) with standard normal distribution. This model relies on normally distributed RNs, which we generate
by means of the widely used Mersenne twister (MT) pseudo random number generator (PRNG) \cite{48} and the Box-Muller transform \cite{44,49}.

**Option Pricing Details**

In this work we make use of three widely used options: European Asian, European Barrier, and American Vanilla. European-style options are options that can only be exercised at a specific time in the future, called maturity, whereas American-style options can be exercised at any time until maturity. A Vanilla option is an option without added features regarding its payoff. On the other hand, an Asian option compares the strike price with the (in this case) arithmetic average of the prices of the underlying asset up to maturity, whereas a Barrier option is validated/invalidated conditioned on the price of the underlying asset hitting a predefined barrier. For both European options we employ a MC simulation, whereas for the American option we use in this the Cox-Ross-Rubinstein (CRR) binomial tree method (see e.g. \cite{44}).

For one-dimensional American options (with one underlying asset), the binomial tree method is computationally efficient, whereas for higher dimensions only the MC approach is tractable (see e.g. \cite{44}). FPGA-based algorithmic optimizations for the latter case are available in \cite{27,29}.

In the case of the European options, we further employ state-of-the-art bit-true algorithmic optimizations, which have been published in \cite{38}. These algorithmic optimizations are first based on the observation that in the inter-

![Figure 3: Nested MC-based simulation.](image-url)
nal part of the nested MC simulation, the sequence of RNs must be reused among the different simulations for the same (derivative) product. This optimization, which we called RNs-reuse approach, is detailed in [31]. Since the sequence of RNs is now reused, the paths can be then generated in a normalized way, and denormalized based on the final price of the corresponding external simulation. This approach is called Paths-reuse, and it is explained in detail in [38]. Besides, European-style options base their price on the stochastic information obtained from the simulation. In turn, this information can be obtained from a normalized internal simulation using the Paths-reuse approach, and demoralized based on the external simulation. This Info-reuse approach is detailed in [38], and it is employed in this work.

3 Cloud-based CPU+FPGA Implementation

The target hardware configuration is an F1 instance in the Amazon AWS EC2 cloud, which consists of an Intel Xeon processor (CPU) and a Xilinx Virtex Ultrascale+ FPGA connected via PCIe (details are given in Section 4.1). Our target application is the portfolio risk measurement under compute intensive nested MC-based simulations, which requires the following steps:

1. load the simulation parameters,
2. generate the required RN sequences,
3. compute the price of each product in the portfolio today (we assume that they are not readily available in the market),
4. run the nested MC-based simulation, either the external or the internal part based on the corresponding product in the portfolio: the outcome is a vector of simulated profit/loss (P&L) scenarios for each product,
5. aggregate the simulated scenarios (taking into account the corresponding weights in the portfolio) and substract the current value of the whole portfolio, in order to obtain the simulated profit/loss (P&L) scenarios of the whole portfolio at the selected horizon time,
6. sort the vector of simulated scenarios in ascending order: VaR is basically the value of an element in that vector (where the VaR-index is determined by the $\alpha$-quantile), and ES corresponds to the average of all the elements in that vector up to the mentioned index.
The steps 1-3 and 5-6 above can be efficiently run on the host CPU. In the case of nested MC-based simulations (step 4), the external part can be executed on multicore CPU (at least in our setup, see Section 4.2), whereas the internal part (which is the most compute-intensive part, and corresponds to the option pricing algorithms) can be efficiently parallelized and implemented on FPGA, as shown in Fig. 4. For comparison purposes, we also implement the products in the internal simulation using OpenMP for multicore CPU.

![Design overview using Xilinx SDAccel (OpenCL) for an heterogeneous CPU+FPGA configuration, and OpenMP on multicore CPU.](image)

Figure 4: Design overview using Xilinx SDAccel (OpenCL) for an heterogeneous CPU+FPGA configuration, and OpenMP on multicore CPU.

Here we employ the OpenCL API targeting an heterogeneous CPU+FPGA configuration, and to this end the Xilinx SDAccel development environment for FPGAs. The kernel code is written in C/C++ and launched as a task (refer to Sections 2.1 and 2.2), in order to have enough low-level control over the hardware architecture, and to reuse part of our legacy code written in Xilinx Vivado high-level synthesis (HLS). Nevertheless, the kernel code could also be converted to OpenCL C and launched as an NDRange.

In our setup we have three options to price in the internal part of the simulation: European Asian and European Barrier using a MC approach and Eq. (3), and American Vanilla using a CRR binomial tree method. Since both European options differ mainly on the payoff, it is possible to efficiently combine (from a simulation point-of-view) both pricing algorithms into one minimizing resources/computation: when both options use the same set of parameters, the merged kernel will yield both prices together.
In the proof-of-concept presented here, the kernels are implemented following the main parallelization and implementation guidelines in [38], but adapted for FPGAs. The innermost loops of the kernels are fully pipelined with an initiation interval of one clock cycle (II=1), and unrolled by a factor of 32. As shown in Fig. 4, there is one instance of the EuropeanAsianBarrier kernel, and in the case of the AmericanVanilla kernel the workload is equally distributed among 4 instances. Both kernels (and their respective instances) are carefully combined into a main function under a dataflow pragma, which enables them to run concurrently. Nevertheless, a similar effect can also be achieved using two independent kernels (on the device side) and (from the host side) either two separate command queues or a single out-of-order command queue. In this regard, a good set of examples are available under /home/centos/aws-fpga/SDAccel/examples/xilinx in the FPGA Developer AMI that can be obtained from the Amazon AWS Marketplace.

The kernels could also be run independently, one after the other, by reconfiguring the FPGA in each case accordingly (this was actually our first approach). However, the OpenCL cl::Program step, which is responsible of compiling at runtime from the binary file and configuring the FPGA, is currently taking \( \approx 6.8 \) seconds, far larger than the runtime of any of the kernels in our setup (this is detailed in in Section 5). This is the reason why the kernels are combined and run concurrently on the FPGA in our case.

Finally, it is necessary to mention that all vectors (buffers) include the `aligned_allocator<>` struct for proper alignment in memory, in order to avoid unnecessary `memcpy` operations at runtime (for more details, refer to file `/home/centos/aws-fpga/SDAccel/examples/xilinx/libs/xcl2/xcl2.hpp`, included in the FPGA Developer AMI indicated previously).

4 Setup

This section provides details on the hardware and simulation setup, the VaR/ES numerical results validation under the given set of parameters, and the place-and-route (P&R) resources utilization on FPGA.

4.1 Hardware Setup

*Amazon AWS EC2 Instance:* f1.2xlarge [7]. *CPU:* Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, 8 cores, 122GB DDR4, technology node: 14nm [7].
4.2 Simulation Setup

The portfolio is composed of: one stock, and three options on that stock: European Asian, European Barrier, American Vanilla 1D. Simulation model: BS under Euler-Maruyama discretization scheme (MC simulation) (see e.g. [52]) to price the stock and the European-style options, and the Cox-Ross-Rubinstein (CRR) binomial tree (BT) method (see e.g. [44]) to price the American Vanilla 1D. For the nested MC-based simulation, the following parameters are necessary: pathsMCext=32k, stepsMCext=10, pathsMCint=32k, stepsMCint=252, stepsTree=252.

All time measurements are carried out on the host side, by employing the `clock_gettime()` function from the `time.h` library, and applying the corresponding synchronization when measuring OpenCL runtime.

4.3 VaR/ES Numerical Results Validation

Value of the portfolio today = 1,367,463.63 USD, with the following initial composition: Stock-A (66.5%), European Asian option (9.8%), European Barrier option (10.5%), American Vanilla 1D (13.2%). For an $\alpha=99\%$, the 10-day risk measures (VaR, ES) = (9.8%, 10.2%), with an average P&L of 1.3% and an average profit of 7.0% (excluding losses).

Simulation parameters: Stock-A: $S_0=130.00$ USD, $\sigma=0.20$, $r=0.025$, $q=0.0$, amount=7000; European Asian: underlying=Stock-A, call option, $T=1.0$ years, $K=130.00$ USD, $\sigma=0.20$, $r=0.025$, $q=0.00$, amount=200 * 100; European Barrier: underlying=Stock-A, call-up-in option, $T=1.0$ years, $K=130.00$ USD, $S_b=162.50$ USD, $\sigma=0.20$, $r=0.025$, $q=0.00$, amount=160 * 100; American Vanilla 1D: underlying=Stock-A, put option, $T=1.0$ years, $K=130.00$ USD, $\sigma=0.20$, $r=0.025$, $q=0.00$, amount=200 * 100.

4.4 Resources Utilization

The place-and-route (P&R) resources utilization on FPGA is detailed in Table 1. The average resources distribution among the concurrent kernels is: AmericanBT1D $\approx$ 70%, EuropeanAsianBarrier $\approx$ 30% (refer to Fig. 4).
Table 1: P&R Resources utilization.

<table>
<thead>
<tr>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUT</td>
</tr>
<tr>
<td>User budget</td>
<td>879913</td>
</tr>
<tr>
<td>Kernel</td>
<td>191169</td>
</tr>
<tr>
<td></td>
<td>21.73 %</td>
</tr>
</tbody>
</table>

$^1$ Corresponds to BRAM_36Kb; UltraRAM (URAM): available 960, utilized 0%.

4.5 OpenMP Scaling

We have three kernels in the nested MC-based simulation: external part: *Stock*; internal part: *EuropeanAsianBarrier* (two options are priced in one kernel) and *AmericanVanilla*. To determine the optimal number of OpenMP threads under the given setup, we measure the runtime while scaling the number of threads. Fig. 5 shows the normalized speedup with respect to 1 thread (this makes it feasible to plot the three kernels together).

From Fig. 5, we obtain an optimal number of threads equal to 8. As a reference, the individual runtimes for (1-thread, 8-threads) are: *Stock* (7.9ms, 1.8ms); *EuropeanAsianBarrier* (30.4s, 6.4s); *AmericanVanilla* (9.7s, 2.3s).

![Figure 5: Speedup on multicore CPU while scaling the number of OpenMP threads (normalized to 1 thread), of the kernels in the nested MC-based simulation: Stock (external), EuroAsianBarrier and AmericanVanilla (internal).](image-url)
5 Results

The full runtime breakdown is detailed in Table 2, and it is graphically summarized in Fig. 6. The main comparison is made between the implementation on CPU+FPGA and the multicore CPU implementation using OpenMP (refer also to Fig. 4). The performance of the single-threaded CPU code is given here as a reference for the OpenMP version.

Table 2: Full runtime breakdown in (s).

<table>
<thead>
<tr>
<th>Description</th>
<th>CPU</th>
<th></th>
<th></th>
<th>CPU + FPGA</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single-threaded</td>
<td>openmp</td>
<td></td>
<td>full prog</td>
<td>skipped prog</td>
<td></td>
</tr>
<tr>
<td>Global Time</td>
<td>42.334</td>
<td>10.983</td>
<td>9.485</td>
<td>3.075</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– RNG</td>
<td>0.726</td>
<td>0.726</td>
<td>0.726</td>
<td>0.726</td>
<td>0.726</td>
<td>0.726</td>
</tr>
<tr>
<td>– Host (kernels)</td>
<td>40.086</td>
<td>8.733</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>– – EuroAsianBarrier</td>
<td>30.360</td>
<td>6.419</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>– – AmericanVanilla</td>
<td>9.726</td>
<td>2.315</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>– OpenCL</td>
<td>0</td>
<td>0</td>
<td>7.230</td>
<td>0.823</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– – cl::Program</td>
<td>0</td>
<td>0</td>
<td>6.814</td>
<td>0.409</td>
<td>0.209</td>
<td>0.207</td>
</tr>
<tr>
<td>– – kernel: Concurrent</td>
<td>0</td>
<td>0</td>
<td>0.207</td>
<td>0.207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– – misc. (OpenCL)</td>
<td>0</td>
<td>0</td>
<td>0.209</td>
<td>0.207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– Risk Measures</td>
<td>0.013</td>
<td>0.014</td>
<td>0.013</td>
<td>0.013</td>
<td>0.015</td>
<td>0.012</td>
</tr>
<tr>
<td>– misc. (global)</td>
<td>0.007</td>
<td>0.008</td>
<td>0.015</td>
<td>0.012</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 The OpenCL cl::Program prog(...); step automatically recognizes if the local image has been previously uploaded, and if so it skips part of the procedure and outputs the message: Successfully skipped reloading of local image.

The kernel that simulates the stock (see Section 4.2), which corresponds to the external part of the nested MC simulation, requires 7.9ms in a single-threaded code, and 1.8ms using OpenMP with 8 threads. As expected, this runtime is small compared to the one required by the internal simulations.

The Concurrent kernel mentioned in Table 2 is the one that simulates the three options concurrently on FPGA (see details in Section 3). The OpenCL runtime in Table 2, which is also graphically shown in Fig. 7, includes not only the runtime of the Concurrent kernel, but also the required initializa-
Figure 6: Runtime breakdown. The OpenCL runtime also includes the initialization overhead and the data transfers between host and device (FPGA).

...tion steps and all data transfers between host and device: only \textit{cl::Program} is indicated separately; the remaining overhead is included as miscellaneous (\textit{misc.}). In the case of the \textit{cl::Program} step (refer to Section 3), it automatically recognizes if the same local image has been uploaded in the previous call to this step, and if so it skips part of the procedure (the console will output the message: \textit{Successfully skipped reloading of local image}). The full runtime of this step is $\approx 6.8s$, which is far larger than the 207ms required by the \textit{Concurrent} kernel. Therefore, in our case we assume that this local image has been already uploaded, and that the simulation is run several times under different sets of parameters.

The kernel implementation on FPGA using Xilinx SDAccel (OpenCL) achieves approximately 10x speedup compared to the multicore CPU version using OpenMP with 8 threads (823ms vs 8.733s respectively). At application level (global runtime), this translates into a speedup of 3.5x (3.075s vs 10.983s respectively), under the assumptions given previously.

6 Conclusions

By efficiently offloading the compute-intensive part of the nested MC-based simulation from the host CPU to an FPGA, we have achieved a substantial speedup, both at kernel level (10x) and at application level (3.5x), compared
(a) With full OpenCL program step: `cl::Program prog1(ctx1, dev1, bin1);`

(b) “Successfully skipped reloading of local image” (2nd run of the application)

Figure 7: OpenCL runtime on FPGA: main breakdown.

to a multicore CPU implementation. This speedup could be further increased by carefully replicating the number of kernel instances on the FPGA, which currently occupy 45%~48% of the available BRAM and DSP resources.

To correctly evaluate the system-level energy consumption (which is the one that affects the total cost of operation), we would require power measurements in the cloud, which are currently not available at user-level. The necessary runtime measurements are already detailed in Table 2 and Fig. 6.

Although so far we have not been able to run this application on graphics processor unit (GPU) in the cloud, which is left as future work, we know that GPUs are strong competitors versus FPGAs when targeting algorithms with massive regular parallelism (like in our setup). Nevertheless, the goal of this proof-of-concept implementation was to provide a representative test case of a portfolio risk management application (here by means of value-at-risk and expected shortfall risk measures) executed on FPGA in the Amazon Cloud. In fact, one of the shortcomings preventing an earlier widespread adoption of FPGA in the cloud was the lack of proper hardware/software infrastructure, and we have shown here that this infrastructure is now available.

Acknowledgments:

We gratefully acknowledge funding from the Deutsche Forschungsgemeinschaft (DFG) within the research training group 1932 ”Stochastic Models for Innovations in the Engineering Sciences”, and the Center of Mathematical and Computational Modeling CM² of the University of Kaiserslautern. The authors alone are responsible for the content of this whitepaper.
References


Proceedings of the 24th IEEE International Conference of Field Programmable Logic and Applications (FPL), pages 1–8, Sept. 2014.


