Low-power option Greeks: Efficiency-driven market risk analysis using FPGAs

Low-power option Greeks: Efficiency-driven market risk analysis using FPGAs

Mark Klaisoongnoen
Mark.Klaisoongnoen@ed.ac.uk
EPCC at the University of Edinburgh
Edinburgh, UK

Nick Brown
EPCC at the University of Edinburgh
Edinburgh, UK

Oliver Thomson Brown
EPCC at the University of Edinburgh
Edinburgh, UK

ABSTRACT

Quantitative finance is the use of mathematical models to analyse financial markets and securities. Typically requiring significant amounts of computation, an important question is the role that novel architectures can play in accelerating these models. In this paper we explore the acceleration of the industry standard Securities Technology Analysis Center's (STAC) derivatives risk analysis benchmark STAC-A2™ by porting the Heston stochastic volatility model and Longstaff and Schwartz path reduction onto a Xilinx Alveo U280 FPGA with a focus on efficiency-driven computing.

Describing in detail the steps undertaken to optimise the algorithm for the FPGA, we then leverage the flexibility provided by the reconfigurable architecture to explore choices around numerical precision and representation. Insights gained are then exploited in our final performance and energy measurements, where for the efficiency improvement metric we achieve between an 8 times and 185 times improvement on the FPGA compared to two 24-core Intel Xeon Platinum CPUs. The result of this work is not only a show-case for the market risk analysis workload on FPGAs, but furthermore a set of efficiency driven techniques and lessons learnt that can be applied to quantitative finance and computational workloads on reconfigurable architectures more generally.

KEYWORDS

Market risk analysis, efficiency-driven computing, STAC-A2, FPGAs, reconfigurable architectures, option Greeks

1 INTRODUCTION

Market risk analysis involves determining the impact of price movements on financial positions held by investors or traders. Sitting under the broader field of quantitative finance, the use of mathematical models and datasets to analyse financial markets, such workloads are heavy users of computational resource. Whilst running these models on CPUs is currently dominant, there have been some successes with exploring the acceleration of quantitative finance using FPGAs [8] [11] [9]. However to date the majority use of FPGAs in the financial world has been in high-frequency trading.

One of the blockers to FPGAs gaining traction in quantitative finance is the historically significant time investment required in programming reconfigurable architectures and need for detailed hardware-level knowledge on behalf of developers. Whilst the benefits of FPGAs to high-frequency trading have been sufficiently obvious to make programming FPGAs in hardware description languages (HDL) such as VHDL or Verilog worthwhile, the benefits have been less forthcoming for quantitative workloads to warrant such efforts. Nevertheless in recent years FPGAs have become far more capable both in terms of hardware and software development ecosystem, and with tool chains such as Xilinx' Vitis [5], one can now program FPGAs by writing code in C or C++. Consequently the increased programmability of these devices means that programming an FPGA is now much more a question of software development rather than hardware design, and this has been a major enabler for numerous communities to recently explore FPGAs for their workloads [7] [22] [6] more in-depth.

Quantitative finance is one of these communities interested in the potential performance and energy advantages of FPGAs, and in this paper we explore porting models comprising a major component of the STAC-A2 market risk analysis benchmark to an Alveo U280 FPGA. The paper is structured as follows; in Section 2 we briefly survey related activities and describe the context of this work, before in Section 3 detailing the experimental setup used throughout this paper and report baseline performance and energy of our benchmark kernel of interest on the CPU across numerous problem sizes. Section 4 then describes the porting and optimisation of the code from the Von Neumann based CPU algorithm to a dataflow representation optimised for the FPGA, before exploring the performance and energy impact of changing numerical representation and precision. In Section 5 we report multi-kernel performance and energy usage based upon applying the optimisation techniques and appropriate numerical choices that were highlighted in the previous section. This paper then concludes in Section 6 before describing further work. The result of this work is not only a comprehensive efficiency-driven exploration of major components of STAC-A2 on the Alveo FPGA, but furthermore lessons that can be applied more widely to high performance numerical modelling on FPGAs.

2 BACKGROUND AND RELATED WORK

The ability for FPGAs to provide low latency handling of data has meant that they have been successfully applied to high-frequency trading for a number of years [14]. Traditionally such high-frequency codes were written in HDL [1], however FPGAs are yet to gain ubiquity in quantitative finance for accelerating financial computational models, and a major reason is that their benefits have been less clear. In recent years vendors such as Xilinx and Intel have invested significantly in new generations of more capable hardware and substantially improved their software development ecosystems. Xilinx's Vitis toolchain [20] is an example, where using High Level Synthesis (HLS) programmers can write code for FPGAs in C or C++. This technology significantly improves productivity, opening up the programming of FPGAs to a much wider community, and significantly reducing the barrier to entry. Consequently these advantages makes the use of FPGAs more realistic for computational workloads such as quantitative finance, enabling software developers to port their codes more easily.

Nevertheless HLS is not a silver bullet, and whilst this technology has made the physical act of programming FPGAs much easier, one must still select appropriate kernels that will suit execution on FPGAs [4] and recast their Von Neumann style CPU algorithms into a dataflow style [12] to obtain best performance. Whilst there have been some successes in accelerating quantitative finance on FPGAs [8] [11] [9], and Xilinx have recently provided support in their open source Vitis Library [17] for numerous quantitative finance primitives, there is still much exploration to be undertaken especially with the objective of efficiency-driven computing looking to optimise both performance and energy efficiency. Furthermore, more development is needed of the underlying algorithmic techniques to inform software developers how best to port their codes to FPGAs.

2.1 Securities Technology Analysis Center

The Securities Technology Analysis Center (STAC) acts as a forum for some of the world's largest global banks, hedge funds and hardware companies in the area of finance. With membership comprising over 400 financial institutions and more than 50 technology vendors, STAC provide industry standard financial benchmarks suites representing common workloads. Based on STAC's benchmark specifications, members can test, optimise and validate their technology against these world-leading benchmarks to compare their software codes and hardware infrastructure against a common market baseline. When undertaking such audits STAC members must comply with strict rules, and while this is beneficial for a fair comparison, in this research we are using the benchmarks differently as we are not looking to undertake any official audits and results should not be compared to audited results. Instead, we use selected benchmarks as drivers to explore algorithmic, performance, and energy properties of FPGAs, consequently meaning that we are able to leverage components of the benchmarks in a more experimental manner.

2.2 STAC-A2: Market risk analysis

The STAC-A2 benchmark [16] focuses on real-world market risk analysis [13] which is an important, ongoing task for investors, trading firms and regulatory authorities. Financial models are deployed to analyse the impact of price movements in the market on financial positions held by investors. Understanding the risk carried by individual or combined positions is crucial for such organisations, and provides insights how to adapt trading strategies into more risk tolerant or risk averse positions. The quality of market risk management is not only driven by demand from investors to track changing market conditions, but also due to increased requirements by regulatory authorities. With expanding numbers of financial positions in a portfolio and increasing market volatility, the complexity and workload of risk analysis has risen substantially in recent years and requires model computations that yield insights for trading desks within acceptable time frames.

Market risk analysis relies on analysing financial derivatives which derive their value from an underlying asset, such as a stock, where an asset's price movements will change the value of the derivative. For each asset, risk analysis relies on understanding sensitivities to market changes which is known as Greeks. Computing these risk sensitivity Greeks involves a high computational workload based on numerical models and many financial firms manage dedicated data centres which are, in-part, apportioned to this workload. Whilst computational performance is one essential aspect of effective risk analysis, energy efficiency is also important because of the dedicated infrastructure in-place and increased frequency of generation and usage of derived risk information in general.

It is therefore worthwhile exploring opportunities for efficiency-driven market risk analysis by leveraging reconfigurable architectures, benefiting from their innate energy efficiency, and the industry standard STAC-A2 benchmark. The benchmark itself involves path generation for each asset using the Andersen Quadratic Exponential (QE) method [2] which undertakes time-discretization and Monte Carlo simulation of the Heston stochastic volatility model [10] before pricing the option using Longstaff and Schwartz [15] for early option exercise. Previously Xilinx developed a proprietary implementation of this benchmark on their Alveo U250 FPGAs which, when running over eight U250s, obtained a 1.48 times speed up compared to the CPU [19] in an official STAC audit.

As an official STAC audit Xilinx had to comply with a strict series of guidelines that ensure results are highly trustworthy, but these govern what can and can not be changed in the code, and therefore limit flexibility. By contrast we are not undertaking an official audit, but instead using the benchmark as a vehicle to better understand the use of reconfigurable architectures for this workload and develop appropriate dataflow techniques. Consequently we have more choice around which parts we offload and are able to undertake more extensive code level changes.

When profiling STAC-A2 on the CPU we found that over 97% of the runtime for the reference implementation was spent on the Heston stochastic volatility model and path reduction in Longstaff and Schwartz. However only around 50% of the CPU cycles were completing useful work in these parts of the code, with approximately 20% of cycles stalled due to memory bottlenecks and the rest stalled due to other core-bound issues. Consequently an important question is whether, by exploiting the ability of reconfigurable architectures to tailor the electronics to the code, it is possible to ameliorate these CPU issues and at the same time benefit from the typically greater energy efficiency of the FPGA [3].

2.2.1 General code structure. The components of the STAC-A2 benchmark we are focusing on in this work operate over paths, which can be thought of as the accuracy of the sensitivities that are being computed. For each path the benchmark works across assets and timesteps, the former representing distinct derivatives that sensitivities are being computed for and the later denotes time with one timestep per trading day. Each kernel of the code typically loops through in this order, paths as the outer, assets as the middle and timesteps as the inner loops.

Each asset has an associated Heston model configuration and this is used as input along with two double precision numbers for each path, asset, and timestep to calculate the variance and log price for each path and follow Andersen's QE method [2]. Subsequently the exponential of the result for each path of every asset of every timestep is computed. Results from these calculations are then used an an input to the Longstaff and Schwartz model. The Longstaff and Schwartz model comprises two parts, a reduction calculation

and then a quadratic curve fit. We only undertake the reduction on the FPGA because this accounts for the majority of the runtime in that model and code required as part of quadratic curve fit is more verbose and less suited to reconfigurable architectures. All computations in the reference implementation are undertaken, by default, using double precision floating-point arithmetic, and in total there are 307 floating-point arithmetic operations required for each element (every path of every asset of every timestep).

3 EXPERIMENT AND BENCHMARK SETUP

Table 1 defines the five classes of problem size used throughout this work in evaluating the CPU and FPGA benchmark implementations. It should be stressed that these problem sizes do not represent an official STAC audit configuration, but instead have been selected in this research to provide a wide range of data sizes under test. For these problem sizes we vary the number of timesteps, ranging from 6 months for the tiny problem size to 5 years for the huge problem size, and assets being studied. As described in Section 2.2, each element comprises two double precision numbers hence in Table 1 the number of data points is double the number of elements. All experiments undertaken report results from executing the Heston stochastic volatility model and path reduction in Longstaff and Schwartz which are our areas of focus in this work.

Table 1: Problem Sizes with defined number of assets (A), time steps (T) and paths (P). The number of elements is the product of $A * T * P$. Each element requires two data points (Dpoints). Each data point is a 64-bit floating-point number.

Problem size	A	T	P	Elements ($\times 10^6$)	Dpoints ($\times 10^6$)	Size (MB)
Tiny (T)	5	126	25 k	15.75	31.5	252
Small (S)	10	126	25 k	31.5	63	504
Medium (M)	20	252	25 k	126	252	2016
Large (L)	30	504	25 k	378	756	6048
Huge (H)	50	1260	25 k	1575	3150	25200

All CPU runs are undertaken by threading, using OpenMP, across two 24-core Xeon Platinum (Cascade Lake) 8260M CPUs which are fitted into a single node of our test system and energy measured via RAPL. All CPU runs are executed across all 48 physical cores as this was found to be the optimal CPU configuration. For the FPGA runs we use a Xilinx Alveo U280, running at the default clock frequency of 300MHz, which contains an FPGA chip with 1.08 million LUTs, 4.5MB of on-chip BRAM, 30MB of on-chip UltraRAM, and 9024 DSP slices. This PCIe card also contains 8GB of High Bandwidth Memory (HBM2) and 32GB of DDR DRAM on the board. The FPGA card is hosted in a system with a 26-core Xeon Platinum (Skylake) 8170 CPU and energy metrics gathered via the Xilinx Runtime Library (XRT). All bitstreams are built using the Xilinx Vitis framework version 2021.2 which at the time of writing is the latest version. All reported results are averaged over five runs and total FPGA runtime and energy usage includes measurements of the kernel, data transfer and any required data reordering on the host.

3.1 CPU performance and energy baseline

Table 2 reports performance and energy usage of the STAC-A2 Heston stochastic volatility model and Longstaff and Schwartz path reduction running over the two 24-core Xeon Platinum CPUs across the problem sizes described in Table 1 [1]. Specifically it is these performance and energy values that we are aiming to improve upon by porting to the FPGA in this work.

Table 2: CPU baseline performance on two 24-core Intel Xeon Platinum 8260M CPUs across all 48 physical cores.

	Single precision		Double precision	
Problem size	Runtime (ms)	Energy (J)	Runtime (ms)	Energy (J)
Tiny (T)	372.15	93.72	369.07	94.35
Small (S)	629.18	151.50	638.67	153.85
Medium (M)	1545.94	397.48	1551.90	393.34
Large (L)	4574.41	1277.14	4558.11	1266.59
Huge (H)	15825.42	5011.16	15561.09	4900.16

4 FPGA PORTING AND OPTIMISATION TECHNIQUES

4.1 Algorithmic optimisations

As described in Section 2.2, in this work we focus on porting the STAC-A2 Heston model and Longstaff and Schwartz path reduction functionality onto an Alveo U280 FPGA. Table 3 reports performance, card power (average power drawn by FPGA card only), and total energy (energy used by FPGA card and host for data manipulation) for different versions of a single FPGA kernel implementing these models for the tiny benchmark size and against the two 24-core CPUs for comparison. In this subsection we focus mainly on performance, with the objective being that by reducing the runtime this will then also help to reduce the total energy used. The total runtime, which includes data transfer to and from the FPGA, and kernel-only runtime are reported, with *initial FPGA* being the first version of the kernel on the FPGA. At around 62 times slower than the CPU, this first version left plenty of opportunity for optimisation, largely because it was still Von Neumann based, and additionally it also drew the largest average card power of all the FPGA versions, although the power differences are fairly minimal but this combination resulted in significant energy usage.

To optimise the kernel for FPGAs we first refactored top-down, moving the code into a dataflow style. This is illustrated in Figure 1 where the algorithm was decomposed into constituent components each of which is a separate function called from within an HLS *DATAFLOW* region. These functions are running concurrently and connected by HLS streams, meaning that data is capable of continually streaming from one to the next with progress made each cycle. External memory accesses were also batched into width of 512-bits as per best-practice [18]. The performance of adopting this

[1] The experiments conducted have not been designed to comply with official STAC benchmarking rules and regulations. Therefore the experimental results that we present are of a research nature and are not representative of official STAC audits.

Table 3: Single FPGA kernel performance of tiny problem size for different versions and against Xeon Platinum CPUs

Description	Total Runtime (ms)	Kernel Runtime (ms)	Card Power (W)	Total Energy (J)
Two 24-core CPUs	369.07	-	-	94.35
Initial FPGA	22882.23	22829.36	30.58	699.73
Dataflow enabled	9307.98	9267.43	29.87	278.03
Loop interchange	236.39	179.83	29.34	9.12
Double buffering	115.35	72.32	29.67	5.51

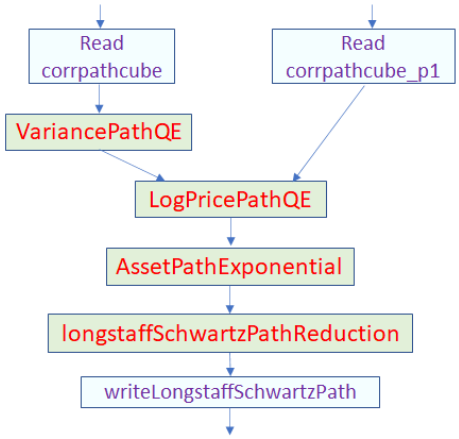

Figure 1: Illustration of dataflow design where stages are running concurrently connected via HLS streams

dataflow approach is reported by *dataflow enabled* in Table 3, and whilst it can be seen this significantly improved performance, it was still 25 times slower than the CPU.

The reason for this performance shortfall was that, whilst we had optimised top-down by splitting the kernel into concurrently running dataflow regions, within each region the code was frequently stalling. Put simply, we were failing to *keep the data flowing* because we had not yet also optimised bottom-up at the individual loop level. Most egregious was the fact that numerous spatial dependencies were limiting effective pipelining of loops and this is illustrated in Listing 1 for the *logPricePathQE* function. It can be seen that there are three loops with the inner, *timesteps* loop, calling the *Y1QE* function for each iteration and results written to the *i+1* element of the *asspath* array. However an input to the *Y1QE* function is the resulting value calculated at the previous timestep loop iteration, *asspath[i]*, and consequently the call to *Y1QE* in one iteration depends upon results calculated at the previous iteration. This was problematic because *Y1QE* undertakes 37 double precision floating-point operations which in total requires 457 cycles, and all these cycles must have completed before the next inner loop iteration can start to be processed.

```
1  void logPricePathQE(unsigned int timesteps, ..., double*
       asspath) {
2    for (unsigned int j=0; j<paths; j++) {
3      for (unsigned int k=0; k<assets; k++) {
4        asspath[0] = .....;
5        for (unsigned int i=0; i<timesteps; i++) {
6  #pragma HLS PIPELINE II=1
7          double W = ....;
8          asspath[i+1] = Y1QE(..., asspath[i], W);
9  }}}}
```

Listing 1: Illustration of spatial dependency issue

To optimise performance one should aim for the loop level initiation interval, the number of cycles between processing one iteration and the next, to be one [18], where a new loop iteration is processed each cycle. Due to the spatial dependency a new iteration could only be processed every 457 cycles and the reason was that, based on the mathematics of the algorithm, there is a dependency between the processing of one timestep and the next. We therefore needed to ensure that there would be at-least 457 cycles between processing subsequent timesteps, and loop interchange was undertaken to achieve this, moving the outer loop over *paths* to be the inner loop.

This loop interchange is sketched in Listing 2, where the results for each path's *Y1QE* calculation is cached in *cached_asspath* and whilst the calculations involving one timestep still depend on the previous timestep for that path, by undertaking this reordering there are *paths* cycles between each subsequent timestep iteration (i.e. each middle loop iteration). As long as the number of paths is greater than 457 then there is no longer a spatial dependency. The HLS dependence pragma at line 3 in Listing 2 is required because the number of paths is a runtime parameter and consequently HLS can not guarantee at synthesis time that this is large enough.

Listing 2 sketches this loop reordering for the *logPricePathQE* dataflow region, and this was also needed for the *variancePathQE* region for similar reasons. Facilitating this reordering required changing the data layout, as illustrated in Figure 2, however doing so resulted in data streamed from *AssetPathExponential* in Figure 1 in the wrong orientation for the subsequent *longstaffSchwartzPathReduction* calculation. This Longstaff Schwartz function undertakes a reduction across assets, calculating the maximum value for each path and timestep held by any asset. In the *longstaffSchwartzPathReduction* function we therefore create a local on-chip buffer of size paths by timesteps, which acts as a local cache to store the current maximum value for each asset and whose values are read and updated as data streams in. However, this caching approach limits the number of paths and timesteps to the amount of on-chip memory, for instance the tiny problem size would require around 25MB of on-chip memory and the huge problem size around 250MB. These sizes are beyond the on-chip BRAM memory available on the Alveo U280 and consequently we decompose paths into batches, within each batch looping over the assets, timesteps and number of paths in that batch before moving onto the next batch.

```
1  void logPricePathQE(unsigned int timesteps, ..., double*
       asspath) {
2    double cached_asspath[MAX_PATHS];
3  #pragma HLS dependence variable=cached_asspath inter
       false
4    for (unsigned int k=0; k<assets; k++) {
5      for (unsigned int i=0; i<timesteps; i++) {
```

```
6           for (unsigned int path=0; path<paths; path++) {
7   #pragma HLS PIPELINE II=1
8               double W = ....;
9               asspath=Y1QE(..., cached_asspath[path], W);
10              cached_asspath[path]=asspath;
11  }}}}
```

Listing 2: Reordering data and loops to fix spatial dependency

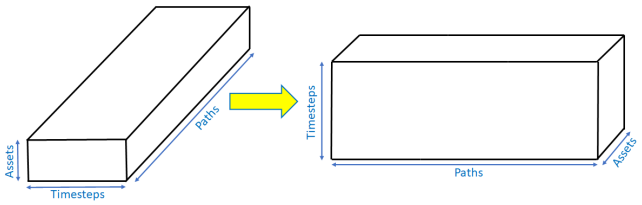

Figure 2: Change to data layout required for loop interchange

This decomposition of paths into batches is illustrated in Figure 3, where it can be seen that the *paths* dimension has been split in this illustration into three batches. Each batch is processed completely before the next is started, and as long as the number of paths in each batch is greater than 457, the depth of the pipeline in *Y1QE*, then calculations can still be effectively pipelined. The on-chip memory required for caching in the *longstaffSchwartzPathReduction* calculation is still fairly large, around 5MB for path batches of size 500 paths and 1260 timesteps, and therefore we place this in the Alveo's UltraRAM rather than smaller BRAM.

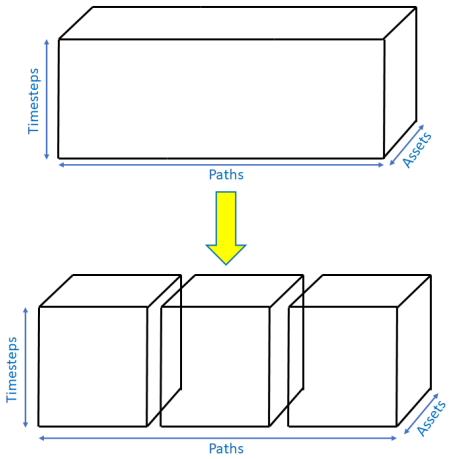

Figure 3: Illustration of data decomposition along paths dimension into batches that are then processed consecutively

Ultimately the objective had been to optimise the loops bottom-up, but doing so required a reorganisation of the data which then had a knock on effect of requiring caching and batching of paths to fit the memory of the architecture. The performance of our kernel on the Alveo U280 at this point is reported by *loop interchange* in Table 3, where we are working in batches of 500 paths per batch, and hence 50 batches, and it can be observed that the FPGA kernel is now outperforming the two 24-core Xeon Platinum CPUs for the first time. It can also be seen that the difference between the total and kernel execution times is greater than with previous versions, this is because now total runtime also includes the time required to reorder input data on the host before transfer, and reordering of result data after kernel execution because of the revised data layout depicted in Figure 2. Likewise energy for these reordering activities is also included as part of total energy in Table 3.

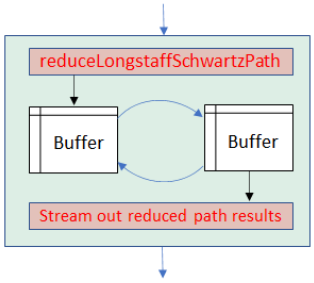

Figure 4: Illustration of double buffering approach over batches of path for longstaffSchwartzPathReduction

However the *longstaffSchwartzPathReduction* function was filling the maximum values over assets in each batch of paths and only streaming out resulting maximum values from the cache once complete. This stalled the dataflow because, whilst filling was occurring, then no streaming was being performed and vice-versa. Consequently we adopted a double buffering approach, where the first buffer is filled for the current batch of paths and results from processing the previous batch are concurrently streamed out from the second buffer. This is illustrated in Figure 4, where *buffer* is a ping-pong buffer that is switched between the two dataflow regions between each batch of paths. This enables the reduction calculation and streaming of output data to run concurrently from the second batch of paths onwards and Table 3 reports this optimisation as *double buffering*. It can be seen that the overall execution time (including data transfer and data reordering on the host) is now 3.2 times less than the two 24-core Xeon Platinum CPUs, and the kernel runtime alone (ignoring data transfer and data reordering) is 5.1 times less than the CPUs. It is noteworthy that our optimised kernel execution time on the FPGA is around 320 times faster than the initial, non-optimised, FPGA kernel execution time and this demonstrates the importance of applying such dataflow techniques to the algorithm. Interestingly these optimisations did not increase the power draw, and this combined with the significantly reduced runtime has resulted in approximately a 140 times reduction in energy draw between the initial and the optimised FPGA versions, and requires 17 times less energy than the two CPUs.

4.2 Numeric optimisation

The STAC-A2 benchmark specification requires double precision floating-point arithmetic which is the numerical representation used until this point. However the ability to tailor execution on the

Table 4: Percentage deviation (lower is better) for each data type running on the FPGA from the reference implementation running in double precision floating-point on the CPU. The deviation is independent from the number of assets or paths.

timesteps	floating-point			ap_fixed							
	half	float	double	<8,3>	<8,4>	<16,6>	<16,8>	<32,12>	<32,16>	<64,12>	<64,24>
126	0.06%	< 0.00%	< 0.00%	326.80%	394.44%	112.44%	28.32%	1.39%	0.38%	100.00%	100.00%
252	0.38%	< 0.00%	< 0.00%	39.96%	373.43%	1887.67%	8514.76%	1.39%	0.83%	100.00%	100.00%
504	1.01%	< 0.00%	< 0.00%	27.78%	433.62%	2055.00%	6173.76%	1.44%	3.96%	100.00%	100.00%
1260	11.14%	< 0.00%	< 0.00%	38.17%	404.17%	286.26%	98.56%	1.75%	16.88%	100.00%	100.00%
2520	14.65%	< 0.00%	< 0.00%	25.43%	405.35%	1246.68%	98.29%	80.08%	11.84%	100.00%	100.00%

Table 5: Resource utilisation on the U280 for a single FPGA kernel with chosen data types implemented on FPGA in percentage of the available resources. These are all built with 500 paths per batch and a maximum number of 1260 timesteps. Asterisk (*) marks fixed-point precision data types.

dtype	LUT	LUT Mem	REG	BRAM	URAM	DSP
half	1.99%	0.62%	1.49%	1.32%	16.04%	2.59%
float	3.53%	0.88%	2.18%	1.60%	16.04%	3.53%
double	10.11%	2.35%	5.77%	2.92%	16.04%	8.88%
<8,3>*	2.96%	0.53%	1.79%	1.43%	16.04%	1.42%
<8,4>*	2.81%	0.50%	1.73%	1.43%	16.04%	1.43%
<16,6>*	7.02%	0.93%	3.71%	2.10%	16.04%	3.06%
<16,8>*	7.02%	0.93%	3.71%	2.10%	16.04%	3.06%
<32,12>*	11.17%	0.97%	5.21%	1.88%	16.04%	2.71%
<32,16>*	21.46%	2.34%	9.95%	3.31%	16.04%	7.84%
<64,12>*	1.70%	0.19%	0.69%	0.50%	16.04%	0.71%
<64,24>*	1.69%	0.19%	0.68%	0.50%	16.04%	0.71%

FPGA means provides more flexibility than on the CPU, where Xilinx's Vitis HLS supports double, single, and half precision floating-point data types as well as arbitrary precision fixed-point. Consequently it is instructive to explore the properties of performance, power draw, energy efficiency, accuracy, and resource utilisation for these alternative numerical precision and representations.

As the host CPU only supports double and single precision floating-point natively, data is transferred from host to device in either one of these formats (double precision for 64-bit fixed or floating-point, and single precision for all others) and then typecast to the respective data type implemented on the FPGA if required. Results are subsequently typecast back to to appropriate host floating-point precision before transferred from the device. Consequently computation on the FPGA is undertaken in our chosen precision and floating or fixed-point, but data transfer always occurs in double or single precision floating-point.

Table 4 presents an accuracy comparison when using these different data representations on the FPGA. Deviations can proliferate and hence we explore how the accuracy changes as the number of timesteps progresses. Table 4 reports percentage accuracy against results obtained with the reference implementation running in double precision floating-point arithmetic on the CPU. It can be seen that on the FPGA floating-point double, single, and half precision, along with fixed-point single precision yield the lowest deviations whereas other configurations diverge significantly.

Whilst the accuracy results reported in Table 4 indicate that some configurations would be unsuited for this benchmark, it is still instructive to explore their resource utilisation properties along with energy and performance. The FPGA's resource utilisation of these different configurations for a single FPGA kernel is reported in Table 5, where irrespective of the numerical representation the amount of UltraRAM (URAM) consumed is constant. This is because of the mapping imposed by the HLS tooling to fit the memory access pattern across the constituent URAM banks. Generally, double precision floating-point has around the highest general resource utilisation. Single and half precision floating-point are rather better, apart from the URAM, and there isn't a clear pattern of resource utilisation across the different fixed-point representations.

Table 6 reports the performance and energy usage for a single FPGA kernel with different numerical representations of each benchmark problem size. It should be highlighted that as we are focused on single kernel at this stage we only include the tiny, small, and medium problem sizes due to the fact that large and huge must run multi-kernel. This is because the maximum size of an individual buffer (per kernel) that can be transferred via XRT onto the Alveo is 4GB [21]. It can be seen that moving to fixed-point arithmetic or reduced precision is not a silver bullet, and in-fact the Xilinx HLS tooling is able to efficiently exploit the Alveo U280 for floating-point computation. In general, optimal performance is obtained when using floating-point representation and it is interesting that at the kernel execution level double precision tends to slightly outperform single and half precision. Furthermore it is difficult to predict the performance of the fixed-point configurations, for instance with 32-bit fixed-point there is a performance difference if one chooses 12 or 16 bits to represent the numbers above the decimal point.

The average power draw in Watts is also interesting, where it can be seen that single precision floating-point arithmetic tends to draw more power than half precision for the small and medium benchmarks most likely due to the more complex core. Moreover, in comparison to fixed-point arithmetic, floating-point is competitive in terms of power draw, with the power draw difficult to predict for fixed-point arithmetic, with no real clear pattern between configurations. It can be seen that the main factor impacting performance and energy efficiency is whether the host is working with double or single precision numbers, as this significantly impacts data reordering and transfer times.

Table 6: Single kernel performance and power details for different numerical precision and representation across benchmark problem sizes. Asterisk (*) marks fixed-point precision data types. [1] Input data is reordered on CPU before transfer to FPGA and result data is reordered on CPU after transfer from FPGA to CPU as described in Section 4.1.

Problem size	Dtype on CPU	Dtype on FPGA	Overall (ms)	Reorder[1] (ms)	Xfer on (ms)	Execute (ms)	Xfer off (ms)	Reorder[1] (ms)	$\overline{Card}$ (W)	$\overline{CPU}$[1] (W)	$Total$ (J)
Tiny (T)		half	105.97			78.25			29.06		4.58
		<16,6>*	104.38			76.66			28.38		4.47
		<16,8>*	107.54			79.82			28.66		4.59
	float	float	102.85	8.54	15.70	75.13	1.66	1.82	29.01	173.75	4.48
		<32,12>*	105.96			78.24			28.76		4.55
		<32,16>*	107.59			79.87			29.99		4.72
		double	115.35			72.32			29.67		5.51
	double	<64,12>*	119.66	11.72	28.02	76.63	1.62	1.67	28.67	185.21	5.53
		<64,24>*	118.18			75.15			28.58		5.47
Small (S)		half	194.02			143.11			28.62		8.82
		<16,6>*	191.18			140.27			28.60		8.73
		<16,8>*	197.04			146.13			28.38		8.86
	float	float	188.44	17.11	29.45	137.53	2.07	2.27	28.97	197.11	8.72
		<32,12>*	194.05			143.14			28.86		8.86
		<32,16>*	197.09			146.18			29.33		9.03
		double	211.94			132.35			29.50		10.12
	double	<64,12>*	219.87	20.61	54.58	140.28	2.10	2.30	23.03	198.17	10.04
		<64,24>*	217.17			137.58			28.23		10.00
Medium (M)		half	747.25			543.50			28.57		36.00
		<16,6>*	736.40			532.65			28.64		35.74
		<16,8>*	758.64			554.89			28.35		36.18
	float	float	725.92	81.57	116.39	522.17	3.80	1.99	28.77	203.93	35.52
		<32,12>*	747.27			543.52			28.84		36.18
		<32,16>*	758.57			554.82			29.45		36.92
		double	838.83			502.55			29.50		45.90
	double	<64,12>*	868.92	117.80	211.83	532.64	3.80	2.85	28.04	204.81	45.59
		<64,24>*	858.54			522.26			28.15		45.38

5 PERFORMANCE AND ENERGY PROFILE

Building on the work reported in Section 4, we replicated the number of kernels on the FPGA such that a subset of batches of paths is processed by each kernel concurrently. The insights gained by experimentation with numerical representation in Section 4.2 mean that we focused on double, single, and half precision floating-point arithmetic for our final overall experiments in this section.

The resource utilisation reported in Table 5 illustrated that irrespective of the precision in use, the kernels require around 16% of the FPGA's UltraRAM which limits the overall number of kernels to six. For single and half precision the UltraRAM utilisation constrains the number of kernels and therefore for problem sizes up to and including *large* we build our kernels with 504 maximium timesteps only. This enables us to fit ten kernels onto the FPGA for such problem sizes, and this limit of ten is imposed by Vitis only making 32 ports available and each of our kernels requiring three ports. 1260 maximum timesteps is required for the huge problem size and consequently we are limited to six kernels for half and single precision. For double precision other resource constraints limit the number of kernels to six regardless of problem size. Figure 5, where the vertical axis is in log scale, reports the performance (in runtime) obtained by our FPGA kernel against the two 24-core Xeon Platinum CPUs for different problem sizes of the benchmark and floating-point precisions. It can be seen that irrespective of double or single, the CPU's performance is significantly worse than that obtained by the multiple FPGA kernels for all configurations, with single and half precision on the FPGA consistently fastest. There are two reasons why half and single are faster than double on the FPGA, firstly because of the difference in the number of kernels up to the *large* problem size, and secondly because with double precision data transfer between the host and device is 64-bit, whereas with single and half it is 32-bit. Consequently there is twice the amount of data being reordered and transferred for double which results in additional overhead. For the huge problem size FPGA performance drops, not only because of the reduced number of kernels for single and half precision, but further more we must use the slower DDR-DRAM rather than HBM2 to fit the data.

Figure 6 reports the overall energy usage, in Joules, of our experiments, where the vertical axis is log scale. It can be seen that the two 24-core Xeon Platinum CPUs require the most energy which

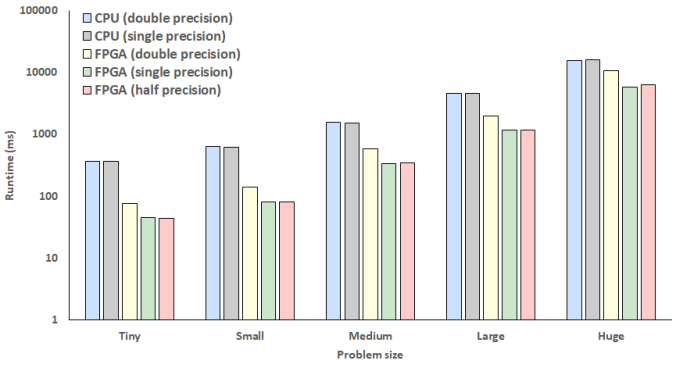

Figure 5: Runtime performance for FPGA kernels and CPU

is from a combination of poorest performance and highest power draw. Single and half precision requires the least overall energy, and double precision on the FPGA is considerably more energy efficient than the CPU but worse than single and half precision. The reason for the increased energy requirement of double precision on the FPGA is a combination of the lower performance and increased power draw on both the FPGA and host data reordering. Energy for the FPGA increases significantly for the huge problem size because we are using DDR-DRAM in that configuration.

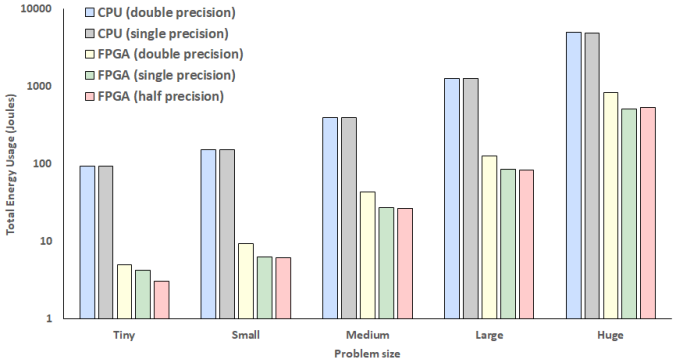

Figure 6: Total energy usage for FPGA kernels and CPU

6 CONCLUSIONS AND FURTHER WORK

In this paper we have explored the role of FPGAs in delivering efficiency-driven computing for market risk analysis via the STAC-A2 Heston and Longstaff and Schwartz models on an Alveo U280 FPGA. Describing the algorithmic level dataflow optimisations that resulted in over 320 times increase in performance on the FPGA between the initial Von Neumann kernel and optimised dataflow algorithm, we then explored the role of different numerical representations and precision with the observation that floating-point arithmetic is highly competitive against fixed-point using the latest Xilinx Vitis toolchain and Alveo FPGA family for performance, power draw, energy efficiency, and resource utilisation. The major performance advantage at the single kernel level in moving to reduced precision was in reducing the overhead of data reordering on the host and data transfer via PCIe between the host and device. When moving to multiple FPGA kernels the smaller amount of resources required for single and half precision floating-point meant that more kernels could fit onto the chip, increasing concurrency.

Based upon insights gained from these explorations we then undertook detailed performance and energy comparisons for our benchmark kernel across different problem sizes, demonstrating that the FPGA is able to undertake the computation at significantly higher performance compared to the two 24-core Xeon Platinum Cascade Lake CPUs, and this combination of much higher performance and significantly reduced power draw has meant that the over all energy usage is very much less on the FPGA compared with the CPU, especially when the data fits into HBM2.

An important area of further work will be to explore streaming data to and from the FPGA rather than bulk copying all data to the device before execution begins. Currently data reordering and transfer accounts for up to a third of the runtime reported in Section 5, and a streaming approach would enable smaller chunks of data to be transferred before beginning kernel execution and to initiate transfers when a chunk has completed reordering on the host. However the Alveo U280 shell only supports AXI4 interface rather than AXIS which will complicate such an approach. Furthermore, we also plan to target the AI engines of Xilinx's next generation Versal architecture, where the chip contains up to 400 of these engines and each is a (single precision) floating-point or arbitrary precision fixed-point vectorised accelerator. Leveraging the AI engines could compliment the insights reported in Table 6, for instance as 32-bit fixed-point delivers acceptable accuracy it would be interesting if the AI engines resulted in increased performance and reduced energy usage.

We conclude that the result of this work not only demonstrates the clear benefit of leveraging an FPGA for efficiency-driven market risk analysis quantitative finance workloads, but furthermore the optimisation techniques described and lessons learnt from numerical experiments are applicable more widely to computational models on FPGAs.

ACKNOWLEDGMENTS

The authors would like to thank STAC for access to the STAC-A2 benchmark and for their advice and assistance. We also acknowledge the ExCALIBUR H&ES FPGA testbed and Xilinx XACC program for access to compute resource used in this work.

REFERENCES

[1] Irene Aldridge. 2013. *High-frequency trading: a practical guide to algorithmic strategies and trading systems.* Vol. 604. John Wiley & Sons.
[2] Leif BG Andersen. 2007. Efficient simulation of the Heston stochastic volatility model. *Available at SSRN 946405* (2007).
[3] Brahim Betkaoui, David B Thomas, and Wayne Luk. 2010. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *2010 International Conference on Field-Programmable Technology.* IEEE, 94–101.
[4] Nick Brown. 2020. Exploring the acceleration of Nekbone on reconfigurable architectures. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC).* IEEE, 19–28.
[5] Nick Brown. 2020. Weighing Up the New Kid on the Block: Impressions of using Vitis for HPC Software Development. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL).* 335–340.
[6] Nick Brown. 2021. Accelerating advection for atmospheric modelling on Xilinx and Intel FPGAs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 767–774.

[7] Nick Brown. 2021. Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 9–20.

[8] Nick Brown, Mark Klaisoongnoen, and Oliver Thomson Brown. 2021. Optimisation of an FPGA Credit Default Swap engine by embracing dataflow techniques. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 775–778.

[9] Dionysios Diamantopoulos, Raphael Polig, Burkhard Ringlein, Mitra Purandare, Beat Weiss, Christoph Hagleitner, Mark Lantz, and François Abel. 2021. Acceleration-as-a-μService: A Cloud-native Monte-Carlo Option Pricing Engine on CPUs, GPUs and Disaggregated FPGAs. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 726–729.

[10] Steven L. Heston. 2015. A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. *The Review of Financial Studies* 6, 2 (04 2015), 327–343.

[11] Gordon Inggs, Shane Fleming, David B Thomas, and Wayne Luk. 2015. Is High Level Synthesis Ready for Business? An Option Pricing Case Study. In *FPGA Based Accelerators for Financial Applications*. Springer, 97–115.

[12] Dirk Koch et al. 2016. FPGA Versus Software Programming: Why, When, and How? In *FPGAs for Software Programmers*. Springer, 1–21.

[13] P. Lankford, L. Ericson, and A. Nikolaev. 2012. End-User Driven Technology Benchmarks Based on Market-Risk Workloads. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 1171–1175.

[14] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High frequency trading acceleration using FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 317–322.

[15] Francis A. Longstaff and Eduardo S. Schwartz. 2015. Valuing American Options by Simulation: A Simple Least-Squares Approach. *The Review of Financial Studies* 14, 1 (06 2015), 113–147.

[16] STAC Research. 2021. *STAC-A2 Central*. Retrieved March 28, 2022 from https://stacresearch.com/a2

[17] Xilinx. 2019. *Vitis Accelerated Libraries*. https://github.com/Xilinx/Vitis_Libraries

[18] Xilinx. 2021. *Optimization Techniques in Vitis HLS*. Retrieved March 28, 2021 from https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Optimization-Techniques-in-Vitis-HLS

[19] Xilinx. 2021. STAC Report: STAC-A2 (derivatives risk) on 8 x Alveo U250 FPGA cards in a BOXX GX8-M. *STAC research audits* (2021).

[20] Xilinx. 2021. *Vitis Unified Software Platform Documentation*. Retrieved August 28, 2021 from https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/index.html

[21] Xilinx. 2022. *Application Acceleration Development*. Retrieved March 28, 2021 from https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Buffer-Creation-and-Data-Transfer

[22] Chen Yang, Tong Geng, Tianqi Wang, Rushi Patel, Qingqing Xiong, Ahmed Sanaullah, Chunshu Wu, Jiayi Sheng, Charles Lin, Vipin Sachdeva, et al. 2019. Fully integrated FPGA molecular dynamics simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–31.

Table A1: Overall multi-kernel performance and energy efficiency

Problem size	dtype	Runtime (ms)		Energy (J)		Performance (s^{-1})		Eff_B	Eff_A	Eff_A / Eff_B
		CPU	FPGA	CPU	FPGA	CPU	FPGA	CPU	FPGA	Improvement
Tiny	float	372.15	46.11	93.72	4.26	2.687089	21.687270	0.02867145	5.09090835	177.56
	double	369.07	75.97	94.35	4.99	2.709513	13.163091	0.02871768	2.63789393	91.86
Small	float	629.18	82.01	151.50	6.27	1.589370	12.193635	0.01049089	1.94475836	185.38
	double	638.67	141.39	153.85	9.33	1.565754	7.072636	0.01017715	0.75805316	74.49
Medium	float	1545.94	341.82	397.48	27.22	0.646856	2.925516	0.00162739	0.10747672	66.04
	double	1551.90	592.79	393.34	43.89	0.644371	1.686938	0.00163820	0.03843559	23.46
Large	float	4574.41	1178.14	1277.14	84.46	0.218607	0.848796	0.00017117	0.01004968	58.71
	double	4558.11	1971.64	1266.59	125.41	0.219389	0.507192	0.00017321	0.00404427	23.35
Huge	float	15825.42	5838.23	5011.16	510.49	0.063189	0.171285	0.00001261	0.00033553	26.61
	double	15561.09	10644.86	4900.16	835.14	0.064263	0.093942	0.00001311	0.00011249	8.58

A PERFORMANCE EVALUATION DOCUMENT

Table A1 provides data for questions 8 to 14. This table includes the overall evaluation of multi-kernel performance and energy efficiency for the baseline (CPU) and the accelerated (FPGA) system.

A.1 Target system

For the FPGA runs reported in this paper we use a Xilinx Alveo U280, running at the default clock frequency of 300MHz . The FPGA card is hosted in a system with a 26-core Intel Xeon Platinum (Skylake) 8170 CPU and energy metrics gathered via XRT. All bitstreams are built using the Xilinx Vitis framework version 2021.2 which at the time of writing is the latest version.

A.2 Baseline system

We compare the STAC-A2 benchmark specific hardware configured on FPGA against the optimised CPU version. All CPU runs are undertaken by threading using OpenMP across two 24-core Xeon Platinum (Cascade Lake) 8260M CPUs at 2.40GHz which are fitted into a single node of our test system and energy measured via RAPL. All CPU runs are executed across all 48 physical cores as this was found to be the optimal single-node CPU configuration.

A.3 Target benchmark

Major components of the STAC-A2 derivatives risk analysis benchmark which comprises market risk analysis workloads including monte carlo methods, the Heston stochastic volatility model, the Andersen Quadratic Exponential (QE) method and the Longstaff and Schwartz model for option pricing in quantitative finance.

A.4 Performance metric

Market risk analysis is a critical workload for trading floors and our chosen problem sizes reflect real-world workloads as defined in Table 1. These problem sizes are used throughout this work in evaluating the CPU and FPGA benchmark implementations. While these workloads are executed multiple times on a daily basis, trading floors typically require results from computations within tight time frames. Therefore, for performance, we report the runtime (in ms) of these workloads for the chosen real-world problem sizes.

A.5 Energy measurement metric

Energy consumption in Joules is measured for the whole FPGA card, and energy reported for the FPGA also includes energy required for data reordering on the host and transfer to/from the host. For the CPU baseline, we measure the CPU energy consumption in Joules for all 48 physical cores across two CPUs.

A.6 Experimental procedure to measure performance

All experiments undertaken report runtime from executing the Heston stochastic volatility model and path reduction in Longstaff and Schwartz. FPGA runtime combines execution, data transfer, and data reordering time. FPGA execution and data transfer time is gathered using OpenCL profiling information via *getProfilingInfo* on the corresponding OpenCL events, which returns time at nanosecond resolution. Timing on the CPU is gathered using the *clock_gettime* call which returns time in nanosecond resolution.

A.7 Experimental procedure to measure power or energy consumption

We run the five classes of problem sizes. For each problem size on the baseline CPU, we measured the energy consumption across all 48 CPU cores via RAPL. For the FPGA, we measured the average power draw over the FPGA runtime via XRT. All reported results are averaged over five runs and total FPGA runtime and energy usage includes measurements of the kernel, data transfer and any required data reordering on the host.

A.8 Other required metrics

See Table A1 which reports the following metrics (questions 8 to 14) across our benchmark sizes and selected floating-point data types:

- **Performance result (target system)**
- **Power or energy result (target system)**
- **Efficiency of target system (Eff_A)**
- **Performance result (baseline)**
- **Power or energy result (baseline)**
- **Efficiency of baseline system (Eff_B)**
- **Efficiency improvement (Eff_A / Eff_B)**