THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Fast and energy-efficient derivatives risk analysis: Streaming option Greeks on Xilinx and Intel FPGAs

Citation for published version:
Klaisoongnoen, M, Brown, N & Brown, O 2022, Fast and energy-efficient derivatives risk analysis: Streaming option Greeks on Xilinx and Intel FPGAs. in *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC).* Institute of Electrical and Electronics Engineers, pp. 18-27, Eighth International Workshop on Heterogeneous High-performance Reconfigurable Computing, Dallas, Texas, United States, 14/11/22. https://doi.org/10.1109/H2RC56700.2022.00008

Digital Object Identifier (DOI):
10.1109/H2RC56700.2022.00008

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. Oct. 2024

Fast and energy-efficient derivatives risk analysis: Streaming option Greeks on Xilinx and Intel FPGAs

Mark Klaisoongnoen
EPCC
The University of Edinburgh
The Bayes Centre, Edinburgh, UK
Mark.Klaisoongnoen@ed.ac.uk

Nick Brown
EPCC
The University of Edinburgh
The Bayes Centre, Edinburgh, UK

Oliver Thomson Brown
EPCC
The University of Edinburgh
The Bayes Centre, Edinburgh, UK

***Abstract*—Whilst FPGAs have enjoyed success in accelerating high-frequency financial workloads for some time, their use for quantitative finance, which is the use of mathematical models to analyse financial markets and securities, has been far more limited to-date. Currently, CPUs are the most common architecture for such workloads, and an important question is whether FPGAs can ameliorate some of the bottlenecks encountered on those architectures. In this paper we extend our previous work accelerating the industry standard Securities Technology Analysis Center's (STAC®) derivatives risk analysis benchmark STAC-A2™, by first porting this from our previous Xilinx implementation to an Intel Stratix-10 FPGA, exploring the challenges encountered when moving from one FPGA architecture to another and suitability of techniques. We then present a host-data-streaming approach that ultimately outperforms our previous version on a Xilinx Alveo U280 FPGA by up to 4.6 times and requiring 9 times less energy at the largest problem size, while outperforming the CPU and GPU versions by up to 8.2 and 5.2 times respectively. The result of this work is a significant enhancement in FPGA performance against the previous version for this industry standard benchmark running on both Xilinx and Intel FPGAs, and furthermore an exploration of optimisation and porting techniques that can be applied to other HPC workloads.**

***Index Terms*—Derivatives market risk analysis, STAC-A2, FPGAs, reconfigurable architectures, option Greeks**

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, which is the use of mathematical models and datasets to analyse financial markets, such workloads are heavy users of computational resources. Whilst running these models on CPUs and GPUs is currently the dominant approach, there have been some successes with exploring the acceleration of quantitative finance using FPGAs [1] [2] [3]. However, to date, the majority of use of FPGAs in the financial world has been in the high-frequency trading sector.

Quantitative finance is one of these communities interested in the potential performance and energy advantages of FPGAs, and in previous work [4], we explored the potential of Xilinx FPGAs, specifically the Xilinx Alveo U280, in providing efficiency-driven computing. This considered both the performance and energy efficiency of a solution and in that work, compared against two 24-core Intel Xeon Cascade Lake CPUs, we were able to demonstrate an overall efficiency improvement, which combines metrics of performance and energy usage, of over 91 and 177 times for double and single precision respectively on the FPGA. This was for the smallest benchmark problem size, however, and the improvement decreased to 8 times (for double precision) with the huge problem size. Whilst an eight times improvement over the CPU is still respectable, the major reason for this reduction was that whilst the runtime was much lower for the FPGA kernel than that of the CPU, a significant portion, over 50% for the huge problem size, was spent in data reordering on the host and transfer between the host and device. Consequently, it was our belief that, by undertaking further work on the FPGA, we could improve on this and make the FPGA even more beneficial for this workload. Furthermore, in [4] we only compared FPGA performance against the CPU, and an important open question was how the FPGA would also compare against a GPU implementation.

In this paper we explore an overlapping technique that enables streaming between the host and device, providing concurrent FPGA execution and data reordering and transfer. We apply this to both the existing Xilinx implementation and also to a new Intel Stratix-10 version, enabling us to explore the suitability of dataflow techniques that were successful on the Xilinx to the Stratix-10. This paper is structured as follows, in Section I we explore the background to this work, introducing the STAC-A2 benchmark in full before describing the existing Xilinx FPGA implementation and limitations. This is followed by Section II which describes the experimental setup used in subsequent sections before we explore the port of our FPGA kernel to the Intel Stratix-10 in Section III. We then describe our data streaming approach which supports the overlapping of data reordering, transfer, and STAC-A2 benchmark kernel[1]execution in Section IV, before scaling up the number of FPGA kernels and undertaking a comparison of benchmark performance in Section V on the Intel Xeon Platinum CPU, V100 GPU, Xilinx Alveo U280, and Intel Statix-10. This paper is then concluded by Section VI which also describes further work.

[1]The STAC-A2 benchmark kernel refers to our implementation of logic defined in the STAC-A2 benchmark specification. Therefore, any reported results must not be compared to official STAC audit results.

©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/H2RC56700.2022.00008

I. 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 [5]. 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 compared to high-frequency trading. Recent advances made by vendors, including new generations of more capable hardware and substantially improved software development ecosystems, mean that the use of FPGAs is now more realistic for computational workloads such as quantitative finance. The result is that software developers can more easily port their codes and gain both performance and energy usage benefits. However, a key challenge still lies in the optimal algorithmic structure of kernels running on the FPGA and therefore explorations of porting real-world codes and how one achieves performance is useful for the community.

The Securities Technology Analysis Center (STAC) coordinates the STAC Benchmark Council™ which includes market participants such as global banks, hedge funds proprietary trading firms, exchanges, and leading technology vendors. With membership comprising over 400 financial institutions and more than 50 technology vendors, STAC provides industry standard financial benchmark suites representing common workloads. Based on the STAC Benchmark Council's 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 components of the 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.

A. STAC-A2: Derivatives risk analysis

The STAC-A2 benchmark [6] focuses on real-world derivatives risk analysis [7] which is an important, ongoing task for banks, investors, trading firms and regulators. Financial models are used 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 on how to adapt trading strategies into more risk tolerant or risk averse positions.

Derivatives 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 are 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.

The STAC-A2 benchmark involves path generation for each asset using the Andersen Quadratic Exponential (QE) method [8] which undertakes time-discretization and Monte Carlo simulation of the Heston stochastic volatility model [9] before pricing the option using Longstaff and Schwartz [10] for early option exercise. Previously Xilinx developed a proprietary implementation of this benchmark, different than our implementation presented in this paper, on their Alveo U250 FPGAs which, when running over eight U250s, obtained a 1.48 times speedup compared to the CPU [11] in an official STAC audit. For this audit, Xilinx had to comply with a strict series of guidelines that ensure results are trustworthy and comparable to previous stacks under test, but these govern what can and can not be changed in the code. By contrast, as we are not undertaking an official audit, we have more choice around which parts we offload and are able to undertake more extensive code-level changes.

When profiling STAC-A2 workloads 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* [12]. However, only around 50% of the CPU cycles were completing useful work in these parts of the code, with the rest stalled due to memory or other core-bound issues. Consequently, an important question is whether, *by exploiting the tailoring offered by the FPGA, it is possible to ameliorate these CPU issues and at the same time benefit from the typically greater energy efficiency of the FPGA [13]*. The components of the STAC-A2 benchmark we are focusing on operate over paths, which can be thought of as the accuracy of the sensitivities 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 latter 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 [8]. The exponential of the result for each path of every asset of every timestep is computed, and results from these calculations are then used as input to the Longstaff and Schwartz model. 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).

B. Existing FPGA kernel structure

Our existing FPGA kernel was written in C++ and used High-Level Synthesis (HLS) to transform this into the resulting Register Transfer Level (RTL). The kernel is structured around

six dataflow regions running concurrently and streaming data between them as illustrated in Figure 1.

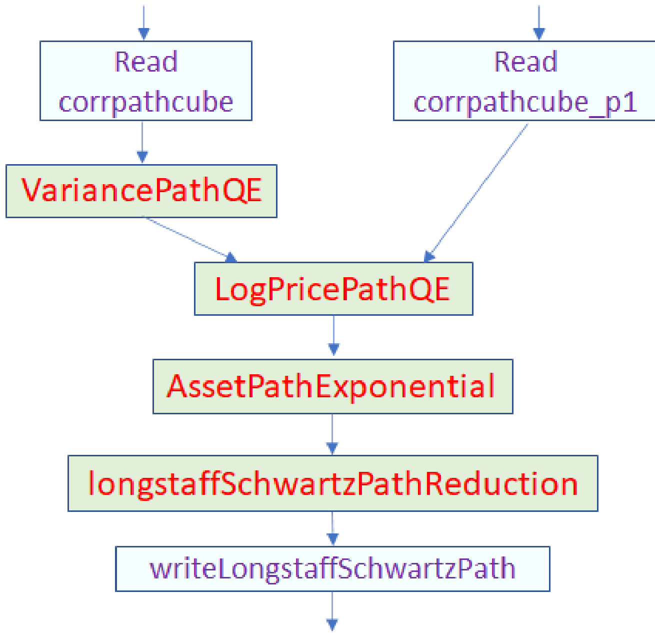

Fig. 1. Illustration of dataflow design where stages are running concurrently connected via HLS streams

As described in [4], a major initial challenge was that there were spatial dependencies between iterations of the inner timestep loop, which was because results from processing one timestep are required as an input to the next. There were up to 457 cycles required for processing a timestep, which resulted in a very high initiation interval in a number of places, and in one case meant that the loop could not be pipelined at all. To address this we reordered the operations, and by undertaking loop interchange moving the loop over paths to be the inner loop, were able to avoid this spatial dependency and thus pipeline the loops successfully.

However, whilst this solved the spatial dependency issue, it resulted in two additional challenges. Firstly, the *longstaffSchwartzPathReduction* dataflow stage, which undertakes a reduction across assets, calculates the maximum value for each path and timestep held by any asset. This requires data in a different orientation to that now streamed out by the proceeding, *AssetPathExponential* stage. To solve this, in the *longstaffSchwartzPathReduction* function, we created a local on-chip buffer of size s by *timesteps*. This acts as a local cache to store the current maximum value for each asset, whose values are read and updated as data streams in. However, this caching approach limited the number of paths and timesteps to the amount of on-chip memory, for instance even the smallest problem size would require around 25MB of on-chip memory. Consequently, we decomposed the paths into batches, within each batch looping over the assets, timesteps and number of paths in that batch before moving on to the next batch. Once the reduced values have been calculated for each batch and results streamed out, the memory can be reinitialised and overwritten. Our approach is illustrated in Figure 2, where we adopted double buffering with the first buffer filled for the current batch of paths and results from processing the previous batch being concurrently streamed out from the second buffer. In Figure 2 *buffer* is a ping-pong buffer that is switched between the two dataflow regions between each batch of paths, thus enabling the reduction calculation and streaming of output data to run concurrently from the second batch of paths onwards.

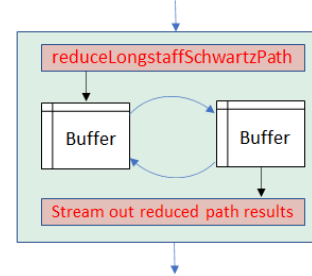

Fig. 2. Illustration of double buffering approach over batches of path for longstaffSchwartzPathReduction

The other challenge associated with our loop interchange was that the ordering of data must be changed, and therefore data reordering on inputs was required before FPGA computation could begin, and the reordering of results. This reordering was undertaken on the host CPU, both for input data before transfer to the device and on result data streaming from the device. However, especially when it came to double precision workloads, this was costly and ended up accounting for over 60% of the runtime for the largest problem sizes.

An illustration of the benchmark execution timeline is provided in Figure 3, where each activity ran consecutively. Effectively we had shifted the bottleneck from being a spatial dependency on the FPGA to data reordering on the CPU, and whilst the overhead of data reordering on the CPU was considerably less than the spatial dependency on the FPGA, this was still costly. Nevertheless, even with such an overhead the FPGA still demonstrated performance benefits over the CPU, at best 8 times and at worst 1.5 times faster.

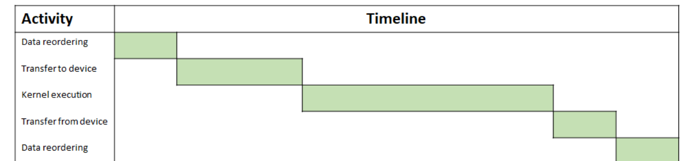

Fig. 3. Illustration of timeline for existing, no overlapping approach

II. Experiment and Benchmark Setup

Table I 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 real-world 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 I-A, each element comprises two double precision numbers hence in Table I 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 I
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. EACH DATA POINT IS A 64-BIT FLOATING-POINT NUMBER.

Problem size	**A**	**T**	**P**	Elements ($\times 10^6$)	Datapoints ($\times 10^6$)	Size (MB)
Tiny	5	126	25 k	15.75	31.5	252
Small	10	126	25 k	31.5	63	504
Medium	20	252	25 k	126	252	2016
Large	30	504	25 k	378	756	6048
Huge	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 at 2.40GHz which are fitted into a single node of our test system and energy measured via RAPL [14]. All CPU runs are executed across all 48 physical cores as this was found to be the optimal single-node CPU configuration. For the FPGA runs reported in this paper 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. We also use a Bittware 520N-MX which contains an Intel Stratix-10 MX2100 FPGA, with 702720 ALMs, 29.9MB of on-chip memory, and 3960 DSP blocks. This FPGA also contains 16GB of HBM2 external memory. The clock on the Intel Stratix-10 is more dynamic, achieving 320MHz with one kernel but this then decreases to 254MHz as we scale.

The FPGA cards are hosted in a system with a 32-core AMD EPYC 7502 CPU with 256GB DRAM and energy metrics gathered via the Xilinx Runtime Library (XRT) and AOCL respectively. All bitstreams are built for the U280 using Xilinx's Vitis framework version 2021.2 which at the time of writing is the latest compatible version for the U280, and Quartus Prime Pro 20.4 for the Stratix-10. All reported results are averaged over ten runs and total FPGA run-time and energy usage include measurements of the kernel, data transfer and any required data reordering on the host. Configurations and environments are described in Appendix A.

III. PORTING TO THE INTEL STRATIX-10

We adopted the same dataflow algorithmic approach for the Intel Stratix-10 as we had developed in [4] for the Xilinx Alveo U280 and illustrated in Figure 1. Using the Intel Quartus Prime Pro tooling, each dataflow region was a separate OpenCL kernel and connected to other kernels via Intel's channel extension. Whilst this was less convenient from the perspective of development and driving the kernels, for instance, each individual kernel had to be separately started and controlled from the host, it enabled data to flow throughout the design and with each loop resulted in an initiation interval of one.

The implementation of double buffering was the largest challenge during this port, and following Intel's best practice guide [15] our first approach involved two separate OpenCL kernels connected via the external HBM2 memory acting as the buffer. Using a channel between the kernels to provide synchronisation, and the explicit *mem_fence* call to ensure that memory and channel writing were scheduled correctly, we were able to implement the double buffer, albeit using a more manual approach than with Xilinx. However, because of the connection to external memory, this was around three times slower than the Xilinx kernel when we benchmarked it.

Instead, as illustrated in Listing 1, we found that adopting an approach more similar to that undertaken with the Xilinx HLS kernel provided improved performance. A single array is defined in on-chip memory at line four and, working in batches of paths at line 6, this is then provided to the *fillbuffer* function for that batch at line eight which undertakes the reduction for the current batch of paths across assets and filling the *data_store* buffer. The *servebuffer* function at line nine will then stream out reduced values for the previous batch of paths. Whilst this approach to double buffering follows more closely that suited by the Xilinx tooling, this technique was not clear from the Intel best practice documentation but was effective, and reduced the overall kernel runtime by around two times.

```
1  __kernel
2  void longstaffSchwartzPathReduction_1(unsigned int
        assets, unsigned int paths, unsigned int timesteps,
        PathGroup pg) {
3
4    DEVICE_DATATYPE data_store[MAX_PATHS][
          MAX_TIMESTEPS];
5
6    for (unsigned int outer_path_group = 0;
          outer_path_group < pg.number_groups;
          outer_path_group++) {
7      int path_size=outer_path_group == pg.
            number_groups - 1 ? pg.
            last_group_num_paths : pg.paths_per_group;
8      fillbuffer(assets, path_size, timesteps, data_store);
9      servebuffer(path_size, timesteps, data_store);
10   }
11 }
```

Listing 1. Illustration of code to drive double buffering on Intel kernel

Another challenge encountered was that it is only possible to direct each kernel argument to a single bank of HBM2 memory with the Intel tooling. The Intel Stratix-10 MX2100 on the Bittware 520N-MX card contains thirty-two 512MB banks of

memory, but limiting the location of inputs or results into one of these meant that external data was effectively limited to 512MB. Hence we were able to run the tiny and small benchmarks only in this configuration due to the data sizes involved. Xilinx addresses this in their tooling by allowing a port to connect across multiple HBM2 banks driven by the Vitis configuration options, but this is not supported with the Intel tooling and hence a more manual approach was required.

To work around this limitation we defined multiple kernel arguments, each connected to different external HBM2 banks. Working round-robin between these arguments, which was especially convenient when we moved to the streaming approach described in Section IV as this enabled us to cycle in chunks, we were able to handle data sizes greater than 512MB. Whilst this supported the processing of up to and including the large problem size, the Bittware 520N-MX card we are using is not currently fitted with any external DDR-DRAM and so we are limited to the Intel Stratix-10 MX2100's 16GB of HBM2 which precludes the execution of the huge problem size.

TABLE II
KERNEL RUNTIME PERFORMANCE (EXCLUDING THE OVERHEAD OF DATA TRANSFER OR REORDERING) FOR A SINGLE STAC-A2 BENCHMARK RUNNING DOUBLE PRECISION ON THE XILINX ALVEO U280 AND THE INTEL STRATIX-10. COMPARED TO THE REFERENCE IMPLEMENTATION RUNNING OVER TWO 24-CORE INTEL CASCADE LAKE XEON PLATINUM CPUS.

Problem size	**Xilinx U280** (ms)	**Intel Stratix-10** (ms)	**Intel Xeon CPU** (ms)
Tiny	101.96	105.24	369.07
Small	189.03	192.17	638.67
Medium	718.18	754.22	1551.90
Large	1498.71	1582.81	4558.11

A comparison of the kernel runtime only (excluding the overhead of data reordering on the host and transfer between the host and device) of our Intel Stratix-10 implementation against other technologies is illustrated in Table II [2]. The FPGAs kernels are non-streaming here, and we focus on kernel runtime only as it is instructive to explore the performance differences between Xilinx and Intel technologies given a similar algorithmic design and implementation. It can be seen that the performance delivered by both tools is similar, with the Xilinx tooling and hardware slightly faster than the Intel counterpart, but both significantly out-performing the reference benchmark running threaded across two 24-core Xeon Platinum CPUs.

IV. DATA STREAMING APPROACH

As described in Section I-B, our existing approach undertook data reordering, transfer, and benchmark kernel execution sequentially thus resulting in up to a 60% overhead where the FPGA was idle waiting for the host. It was our hypothesis that we could reduce this by overlapping data reordering and transfer on the host with kernel execution on the FPGA. Similar approaches have been used successfully for GPU workloads [16] [17], and this would mean that for the majority of the time both the FPGA would be busy computing and the host busy reordering and transferring data. However, both FPGAs cards considered in this paper, the Xilinx Alveo U280 and Bittware 520N-MX, only support DMA memory transfers rather than streaming. There is some provision in the toolchains for host-to-device streaming. For instance, the Xilinx U250 has a legacy streaming shell although it is not particularly convenient to interact with the host, and there is some general support in Quartus Prime Pro for host-side pipes [18] for the Intel Stratix-10. However, neither the U280 shell nor Bittware 520N-MX BSP provide host-to-device dataflow pipes.

Consequently, whilst data streaming was a natural way of viewing this overlapping from the benchmark kernel's perspective, we had to develop a host-to-device data transfer approach that would provide the illusion of data streaming whilst being compatible with the underlying DMA mechanisms. The high-level architectural view of our approach is illustrated in Figure 4, where the STAC-A2 benchmark HLS kernel, containing the design illustrated in Figure 1, was modified to remove the external memory reading and writing and to replace these AXI4 data interfaces with AXIS interfaces on the Xilinx kernel and OpenCL channels on the Intel kernel. This provides streaming to the *VariancePathQE* and *LogPricePathQE* dataflow stages for input data and from the *longstaffSchewartzPathReduction* stage for results. This was the only change needed at the STAC-A2 benchmark HLS kernel level, from its perspective now operating in a streaming fashion.

The STAC-A2 benchmark kernel's input data AXI stream is connected to an input streaming kernel, and the AXI stream for output data to a result streaming kernel. Both these streaming kernels are also written in HLS and interact with the HBM2 external memory, reading and writing data. Consequently, inputs are streamed into the benchmark kernel and results, when available, are streamed out. These streaming kernels are in fact fairly simplistic, either reading data in widths of 512 bits and streaming constituent elements to the benchmark kernel or receiving streamed-in results and writing these to memory in widths of 512 bits.

The STAC-A2 benchmark kernel is run once from the host, looping over the entire domain. By contrast, the streaming kernels are executed many times from the host and loop over a small chunk of paths within a single execution. For the streaming of input data, the host also works in these chunks and will first reorder data associated with a chunk into a new memory location and then transfer this reordered data into the FPGA's HBM2 via PCIe using the *enqueueMigrateMemObjects* OpenCL call. OpenCL events initiate the execution of the input streaming kernel once this chunk's data transfer has been completed. The opposite happens for result data, after the result streaming kernel has written a chunk's results to HBM2 and then marshalled via OpenCL events, once the result streaming HLS kernel has completed data transfer occurs for the resulting data from HBM2 to the host. An OpenCL

[2]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 III
XILINX SINGLE-KERNEL PERFORMANCE AND POWER DETAILS ACROSS BENCHMARK PROBLEM SIZES FOR OUR PREVIOUS [4] AND STREAMING APPROACH. DATA REORDERING, TRANSFER, AND BENCHMARK EXECUTION TIME ARE INCLUDED AND USING A CHUNK SIZE OF 1000 PATHS AND 8 CPU THREADS FOR DATA REORDERING.

Problem size	**dtype**	**Runtime (ms)**						**Energy (J)**					
		Previous			***Streaming***			***Previous***			***Streaming***		
		Overall	***FPGA***	***Host overhead***	***Overall***	***FPGA***	***Host overhead***	***Overall***	***FPGA***	***Host***	***Overall***	***FPGA***	***Host***
Tiny	float	102.85	92.49	10.36	79.29	72.29	7.00	4.48	2.68	1.80	3.04	2.08	0.96
	double	115.35	101.96	13.39	88.88	78.53	10.35	5.51	3.03	2.48	3.81	2.37	1.44
Small	float	188.43	169.05	19.38	132.44	125.54	6.90	8.72	4.90	3.82	4.61	3.62	0.99
	double	211.94	189.03	22.91	150.43	140.24	10.19	10.12	5.58	4.54	5.70	4.25	1.45
Medium	float	725.92	642.36	83.56	474.31	464.79	9.52	35.52	18.48	17.04	14.70	13.29	1.41
	double	838.83	718.18	120.65	518.11	502.72	15.39	45.90	21.19	24.71	17.53	15.33	2.21
Large	float	1684.61	1186.51	498.10	1070.13	1056.84	13.29	97.98	45.87	52.11	40.62	38.67	1.95
	double	2127.89	1498.71	629.18	1364.72	1345.76	18.96	124.38	52.93	71.45	46.17	43.47	2.69

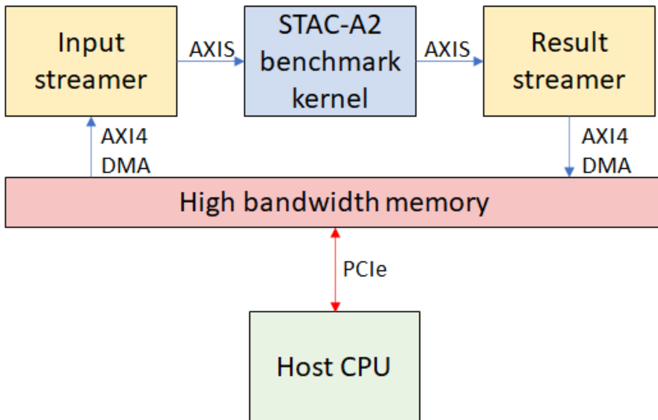

Fig. 4. Illustration of streaming architecture approach

callback is used to execute the host-side data reordering of results when the chunk's data transfer to the host completes. In both the streaming kernels we select the *ap_ctrl_chain* control protocol which enables a kernel to start processing the next kernel operation before completing the current one [19]. This provides an additional input signal *ap_continue* to the kernel and applies back pressure, improving performance.

In this manner, data reordering, transfer, and streaming kernel execution are queued up by the host using OpenCL to marshal control. Whilst the STAC-A2 benchmark kernel has remained largely unchanged and executed once over the entire domain, as illustrated by Figure 5, with this chunking approach for the majority of the time input data is continually being streamed in and results streamed out. Throughout the time that the benchmark kernel is running, data reordering and transfers for subsequent chunks are in progress on the host. There is still a small amount of overhead on the host, handling the first chunk of input data which must be transferred before the compute kernel can begin and the last chunk of results after they are generated, but compared to the timeline with no overlapping in Figure 3, this overhead is much less.

Input and result data reordering on the host is parallelised using threading via OpenMP. Whilst this is not an ideal use of threading, as there is a danger of false sharing negatively impacting performance, empirically we found that eight OpenMP threads are the optimal setting for performance. This choice over the number of threads effectively provides a throttle, enabling us to try and ensure that the FPGA kernels are not stalling waiting for data to be available, but that there are still enough host threads available for the concurrent reordering of input and result data.

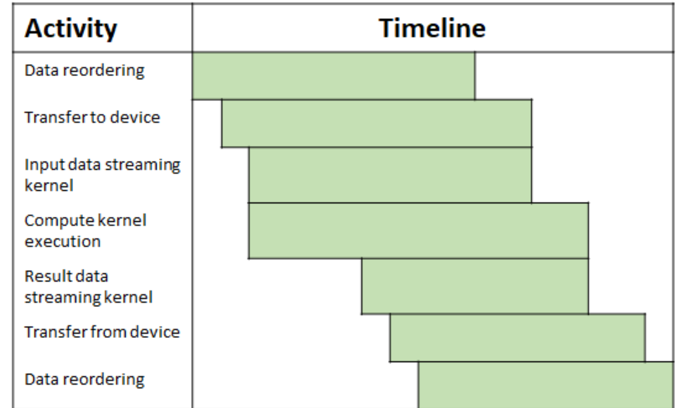

Fig. 5. Illustration of timeline for data streaming approach which enables overlapping of data reordering on the host, data transfer between the host and device, and benchmark execution on the FPGA.

In our streaming approach, the HBM2 DRAM is being used as a substrate, to provide a temporary storage location between the host and benchmark kernel and in this manner, we are able to implement data streaming requiring a limited number of changes to the STAC-A2 benchmark kernel whilst only having access to a non-streaming shell. The chunk size that we are operating on is set by the host, and therefore it is trivial to experiment with different-sized chunks without requiring a lengthy rebuild of the bitstream. It should be noted that Section I-B described a batch of paths being operated upon to decouple the on-chip memory used for double buffering with the overall problem size that could be handled. The chunking of paths between the host and FPGA is entirely separate from

TABLE IV
OVERALL MULTI-KERNEL RUNTIME AND ENERGY ACROSS BENCHMARK PROBLEM SIZES FOR SINGLE AND DOUBLE PRECISION. USING SIX FPGA COMPUTE UNITS ON THE XILINX U280 AND FOUR ON THE INTEL STRATIX-10. WITH A CHUNK SIZE OF 1000 PATHS AND 8 CPU THREADS FOR DATA REORDERING.

Problem size	dtype	Runtime (ms)					Energy (J)				
		Xeon CPU	*V100 GPU*	*FPGA Xilinx previous*	*FPGA Xilinx streaming*	*FPGA Intel streaming*	*Xeon CPU*	*V100 GPU*	*FPGA Xilinx previous*	*FPGA Xilinx streaming*	*FPGA Intel streaming*
Tiny	float	372.15	1119.88	46.11	77.81	93.65	93.72	62.43	4.26	2.55	3.71
	double	369.07	1148.14	75.97	67.33	79.13	94.35	64.12	4.99	2.51	4.25
Small	float	629.18	1195.73	82.01	98.46	102.99	151.50	67.12	6.27	3.20	5.21
	double	638.67	1258.58	141.39	97.64	110.55	153.85	70.57	9.33	3.76	6.35
Medium	float	1545.94	1656.99	341.82	221.71	254.30	397.48	100.71	27.22	7.40	14.56
	double	1551.90	1932.30	592.79	286.90	301.52	393.34	123.88	43.89	10.92	18.73
Large	float	4574.41	2850.95	1178.14	544.48	618.79	1277.14	187.99	84.46	18.46	42.58
	double	4558.11	3718.64	1971.64	714.96	772.63	1266.59	246.14	125.41	27.15	61.03
Huge	float	15825.42	-	5838.23	1928.11	-	5011.16	-	510.49	54.68	-
	double	15561.09	-	10644.86	2301.29	-	4900.16	-	835.14	92.89	-

these batches of paths, and these two choices can be set very different to suit optimal performance as required.

Table III reports performance and energy usage details for a single STAC-A2 benchmark kernel when run on the Xilinx Alveo U280. *Previous* reports results from the non-streaming approach described in [4], and *streaming* the approach described in this paper. For performance numbers we report the overall benchmark runtime, the time that the FPGA is active, and the host-side overhead time (when the host is active but the FPGA is idle and waiting). It can be seen that the overall time is consistently less with the streaming compared to previous approaches, with the host-side overhead significantly reduced compared to the previous approach as we increase the benchmark, and thus the data, size.

The energy usage reported in Table III surprised us because, as the same amount of overall work is being undertaken, we had assumed that energy usage would be fairly similar between the two approaches. However, it can be seen that the streaming approach consumes considerably less energy than the previous, non-streaming, approach. Interestingly this comes largely from the host-side, and that is because we are only using eight threads for data reordering with the streaming approach compared with 48 threads with the previous approach. All those 48 threads were beneficial to the previous approach because the data reordering needed to be completed as quickly as possible before the execution could begin. However with the streaming approach as long as the benchmark kernel on the FPGA is kept fed with data, then as described this can be used more as a throttle to lower energy consumption and to give better control of the overall energy consumption.

Energy usage is also less on the FPGA with the streaming approach, and this is due to the reduced FPGA execution time. The FPGA is idle for more of the time in the previous approach, but the amount of power drawn by an idle but configured FPGA is not much less than when actively computing. Consequently, the idle FPGA is wasting significantly more energy than when we are able to keep it busier for longer.

V. EFFICIENCY-DRIVEN EVALUATION

We then scaled up the number of STAC-A2 benchmark kernels that would fit on the FPGA, decomposing chunks of paths across these kernels. Each benchmark kernel is paired with streaming kernels which are also replicated, thus a *compute unit* represents a single benchmark kernel, input and result streamer. We were able to fit six compute units onto the Xilinx Alveo U280 but only four onto the Intel Stratix-10 due to exhausting Logic Array Blocks (LABs) beyond that point.

Table IV reports the performance and energy usage of the multi-compute unit STAC-A2 benchmark running on Xilinx Alveo U280 and Intel Stratix-10 FPGAs, along with CPU and GPU performance numbers for comparison. The CPU numbers represent running across both 24-core Intel Cascade Lake CPUs in the node, which was found to be optimal for performance, and it can be seen that regardless the FPGAs outperform the CPU considerably.

Running over a Nvidia V100, the GPU results reported in Table IV surprised us as we had hypothesised that the GPU would be far more competitive. Whilst the GPU is not the focus of this paper, we should note that we spent considerable time optimising the kernel for the GPU in order to ensure a fair comparison. For example, undertaking loop interchange and unrolling to enable enough vectorisation to take advantage of the architecture. Such optimisations did have a significant relative performance improvement on the GPU, but in absolute terms, the performance obtained is poor compared with the CPU until the large problem size and regardless can not match that of the FPGAs. There are no huge problem size runs on the GPU as the card only has 16GB of memory.

There are two reasons why the GPU performance falls short of that delivered by the FPGA. Firstly we have used *OpenACC* to develop the kernel and whilst this is a commonly used and accessible choice, so was perfectly sensible to select here, it would be potentially possible to obtain better performance by porting into *CUDA* directly. Secondly, when profiling on the GPU via *nvprof* we found that the GPU was stalled on external

TABLE V
OVERALL MULTI-KERNEL PERFORMANCE, ENERGY EFFICIENCY, AND OVERALL EFFICIENCY IMPROVEMENT COMPARED WITH THE CPU BASELINE.

Problem size	dtype	Performance (s^{-1})					Efficiency (s^{-1}/J)					Improvement ($Efficiency_{CPU}/Efficiency_{ACCELERATOR}$)			
		Xeon CPU	*V100 GPU*	*FPGA Xilinx previous*	*FPGA Xilinx streaming*	*FPGA Intel streaming*	*Xeon CPU*	*V100 GPU*	*FPGA Xilinx previous*	*FPGA Xilinx streaming*	*FPGA Intel streaming*	*V100 GPU*	*FPGA Xilinx previous*	*FPGA Xilinx streaming*	*FPGA Intel streaming*
Tiny	float	2.69	0.89	21.69	12.85	10.68	0.028671	0.014303	5.090908	5.039929	2.878182	0.5	177.6	175.8	100.4
	double	2.71	0.87	13.16	14.85	12.64	0.028718	0.013583	2.637894	5.917219	2.973513	0.5	91.9	206.0	103.5
Small	float	1.59	0.84	12.19	10.16	9.71	0.010491	0.012460	1.944758	3.173878	1.863662	1.2	185.4	302.5	177.6
	double	1.57	0.79	7.07	10.24	9.05	0.010177	0.011259	0.758053	2.723858	1.424517	1.1	74.5	267.6	140.0
Medium	float	0.65	0.60	2.93	4.51	3.93	0.001627	0.005992	0.107477	0.609513	0.270080	3.7	66.0	374.5	166.0
	double	0.64	0.52	1.69	3.49	3.32	0.001638	0.004178	0.038436	0.319188	0.177070	2.6	23.5	194.8	108.1
Large	float	0.22	0.35	0.85	1.84	1.62	0.000171	0.001866	0.010050	0.099492	0.037953	10.9	58.7	581.2	221.7
	double	0.22	0.27	0.51	1.40	1.29	0.000173	0.001093	0.004044	0.051517	0.021207	6.3	23.3	297.4	122.4
Huge	float	0.06	-	0.17	0.52	-	0.000013	-	0.000336	0.009485	-	-	26.6	752.2	-
	double	0.06	-	0.09	0.43	-	0.000013	-	0.000112	0.004678	-	-	8.6	356.7	-

memory access for a large proportion of the time, for the tiny problem size and single floating-point precision more than 68% and for double floating-point precision more than 79% is spent on data transfer, which is similar to the profile found on the CPU. Therefore, whilst the GPU has a very significant amount of raw compute performance available via the CUDA cores, it can not keep these fed with data sufficiently well to make the best use of them. By comparison, our ability to tailor the memory access to the specific application in question means that the data can keep flowing and keep the compute busy for a greater proportion of the time.

When comparing the multi-kernel performance of our previous Xilinx and new streaming Xilinx kernels in Table IV it can be seen that our previous Xilinx version for the tiny and small problem sizes in single precision out-performs our streaming approach. When compared against single-kernel performance reported in Table III it can be seen that multiple kernels only start to become beneficial for the streaming approach when we reach the medium problem size. This is because, before that point, the data sizes involved are too small to make effective use of our chunking and overlapping approach. The performance of our streaming approach on the Intel Stratix-10 follows a similar pattern to that on the Xilinx Alveo U280, however, is slower due to four rather than six compute units being able to fit. There are no results for the huge problem size on the Intel FPGA due to the Bittware 520N-MX card only containing the Intel Stratix-10 MX2100's 16GB of HBM2 and that not being large enough to hold the memory needed.

The energy usage follows a similar pattern to the performance in Table IV, where the large power draw of the two 24-core Intel Xeon Cascade Lake CPUs and the V100 GPU, combined with the poorer performance compared to the FPGA means that the FPGA requires considerably less energy regardless. When comparing the previous and streaming approaches on the FPGA, it can be seen that the streaming approach consumes considerably less energy, especially for larger problem sizes and this corresponds to the single kernel behaviour observed in Table 4. The Intel FPGA streaming approach uses more energy than the Xilinx streaming approach and this is due to two factors, firstly the runtime is slightly larger, and secondly, the Intel Stratix-10 draws considerably more power than the Xilinx Alveo U280 FPGA.

A. *Efficiency driven metrics*

In [4] we used an efficiency-driven metric to compare against the CPU, and in Table V we expand those numbers to also consider the V100 GPU and streaming approach on both the Xilinx and Intel FPGAs. Summarised in Table V across the STAC-A2 problem sizes, where the improvement is relative to the efficiency on the two, 24-core, Xeon Platinum Cascade Lake CPUs. It can be seen that the overarching issue that we were looking to solve, namely that the efficiency improvement for the previous FPGA approach decreased as one moved to larger problem sizes has been addressed here with both the Xilinx and Intel streaming approaches maintaining their improvement relative to the CPU. The higher energy usage and slightly lower performance of the Intel Stratix-10 compared to the Xilinx Alveo U280 means that in these experiments the U280 demonstrates the greatest improvement over the CPU.

VI. CONCLUSIONS

In this paper, we have explored further activities porting modified STAC-A2 benchmarks to FPGAs. Extending our previous study, we first ported the benchmark to the Intel Stratix-10 FPGA and, using Quartus Prime Pro, investigated how the algorithmic structure that was successful with Xilinx's Vitis could be applied to this other family of FPGAs. The most challenging aspect was the ability to handle the double buffering approach and we found that instead of following the Intel best practice guide it was preferable to adopt an approach similar to that promoted by the Xilinx tooling.

Secondly, we have explored a data streaming approach to enable the overlapping of host and FPGA activities. Working around the limitations of the Xilinx and Bittware card shells not supporting host-to-device streaming, we leverage the HBM2 memory on both technologies and developed adaptor streaming kernels. Delivering up to 4.6 times better performance and requiring 9 times less energy than the previous Xilinx FPGA kernel, this overlapping host and FPGA activities has been highly effective, especially for larger data sets. We also compared against an Nvidia V100 GPU, discovering that whilst we had undertaken numerous optimisations to suit that

architecture the ability to tailor memory accesses on the FPGA provides important performance advantages.

A next step will be to generalise the streaming approach described here. There is considerable boilerplate required, for instance the HLS streaming kernels and host-side management code, and these could be auto-generated by the programmer providing an abstract definition of their data streaming, along with pre and post-processing steps.

Acknowledgement

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 AMD Xilinx HACC program for access to compute resource used in this work. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

References

[1] N. Brown, M. Klaisoongnoen, and O. T. Brown, "Optimisation of an FPGA Credit Default Swap engine by embracing dataflow techniques," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 775–778.

[2] G. Inggs, S. Fleming, D. B. Thomas, and W. Luk, "Is high level synthesis ready for business? an option pricing case study," in *FPGA Based Accelerators for Financial Applications*. Springer, 2015, pp. 97–115.

[3] D. Diamantopoulos, R. Polig, B. Ringlein, M. Purandare, B. Weiss, C. Hagleitner, M. Lantz, and F. Abel, "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, 2021, pp. 726–729.

[4] M. Klaisoongnoen, N. Brown, and O. T. Brown, "Low-power option greeks: Efficiency-driven market risk analysis using fpgas," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2022, pp. 95–101.

[5] C. Leber, B. Geib, and H. Litz, "High frequency trading acceleration using fpgas," in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 317–322.

[6] STAC Research. (2021, Mar.) STAC-A2 Central. [Online]. Available: https://STACresearch.com/a2

[7] P. Lankford, L. Ericson, and A. Nikolaev, "End-user driven technology benchmarks based on market-risk workloads," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 1171–1175.

[8] L. B. Andersen, "Efficient simulation of the heston stochastic volatility model," *Available at SSRN 946405*, 2007.

[9] S. L. Heston, "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options," *The Review of Financial Studies*, vol. 6, no. 2, pp. 327–343, 04 2015.

[10] F. A. Longstaff and E. S. Schwartz, "Valuing American Options by Simulation: A Simple Least-Squares Approach," *The Review of Financial Studies*, vol. 14, no. 1, pp. 113–147, 06 2015.

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

[12] M. Klaisoongnoen, N. Brown, and O. Brown, "I feel the need for speed: Exploiting latest generation fpgas in providing new capabilities for high frequency trading," in *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, ser. HEART '21. New York, NY, USA: Association for Computing Machinery, 2021.

[13] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 94–101.

[14] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*.

[15] Intel. (2020) Intel FPGA SDK for OpenCL Pro Edition: Programming Guide. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683846/20-4/introduction.html

[16] E. Papenhausen and K. Mueller, "Rapid rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction," in *2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (2013 NSS/MIC)*, 2013, pp. 1–2.

[17] N. Brown, A. Lepper, M. Weiland, A. Hill, B. Shipway, and C. Maynard, "A Directive Based Hybrid Met Office NERC Cloud Model," in *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, ser. WACCPD '15. New York, NY, USA: Association for Computing Machinery, 2015.

[18] Intel. (2020) OpenCL Host Pipe Design Example. [Online]. Available: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/host-pipe.html

[19] N. Brown and D. Dolman, "it's all about data movement: Optimising fpga data access to boost performance," in *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*.

Appendix A
Artifact Description Appendix

A. Description

1) Check-list (artifact meta information):

- **Program:** C++
- **Compilation:** GCC version 10.2 with -O3, Vitis version 2021.2, Quartus Prime Pro version 20.4. On the GPU we used the Nvidia Compiler version 22.2-0.
- **Data set:** Runs were based on the tiny, small, medium, large, and huge problem sizes which are described in this paper with a standard execution of the benchmark provided by the official STAC-A2 reference implementation.
- **Run-time environment:** A variety of machines were used for comparison, all running Linux. For the Alveo U280 the environment at the time of writing was used (XRT 202110.2.11.634, xdma and xdma-dev 201920.3 deployment and development target platforms). For the Intel Stratix-10 we used version 1.8.1 of the BittWare 520N-MX Gen3x16 BSP.
- **Hardware:** We used a Xilinx Alveo U280 and BittWare 520N-MX for the FPGA runs, these are hosted in systems with a 32-core AMD EPYC 7502 CPU with 256GB of RAM. For CPU comparison runs we ran on two Xeon Platinum Cascade Lake (8260M) processors with 192GB RAM. For GPU runs we ran on the Cirrus tier-2 UK HPC machine, which provides a NVIDIA Tesla V100-SXM2-16GB (Volta) GPU, hosted by two Intel Xeon Gold Cascade Lake (6248) CPUs and 384 GB RAM.
- **Binary:** Xilinx Vitis or Quartus Prime Pro is required to synthesise the kernel and generate the bitstream.
- **Execution:** We built and executed all executables on Linux.
- **Output:** We measured all performance using OpenCL's performance mechanism and also undertook secondary timing using the *gettimeofday* call with microsecond resolution to ensure consistency. All results were checked against the CPU reference running for correctness.
- **Publicly available?:** No, STAC-A2 is commercially sensitive and only available to STAC members.

2) Hardware dependencies: Any machine running Linux with appropriate Alveo U280 or BittWare 520N-MX FPGA PCIe card installed

3) Software dependencies: GCC version 10.2 (although other versions are also compatible), the support libraries installed for the board and the Vitis or Quartus Prime Pro platform.

4) Datasets: Runs were based on the STAC-A2 benchmark using the problem sizes described in this paper.

B. Installation

We synthesised our kernel using Vitis HLS via the *v++* command or Quartus Prime Pro via *aoc*. The host codes are written in OpenCL (the appropriate libraries ship with Vitis and Quartus Prime Pro) and launching our bitstream simply involved executing the host code, which via the appropriate OpenCL calls programmed the device as appropriate along with the size of problem in assets, timesteps, and paths.

C. Experiment workflow

1) Develop the appropriate HLS/OpenCL kernel
2) If on Xilinx then use Vitis HLS to synthesise this and generating corresponding *.xo* files
3) If on Xilinx use Vitis HLS in linking mode to generate the bitstream *.xclbin* file
4) If on Intel Stratix-10 then use Quartus Prime Pro to build kernel which will also undertake synthesis, placement, and routing to generate the bitstream.
5) Compile the host OpenCL code using GCC
6) Execute the host code, which will launch the bitstream

D. Evaluation and expected result

We compared our results against the existing STAC-A2 CPU reference implementation. On the CPU this ran across both 24-core 8260M Cascade Lake Xeon Platinum CPUs in the node and was threaded via OpenMP. The reference implementation is not threaded, and therefore we had to apply this previously when comparing performance in [4]. For the V100 GPU runs we developed and optimised a GPU version of the benchmark for this paper. All calculated values have been checked at the element level to ensure that they are producing consistent results, and all performance and power results reported in this paper are averaged over ten runs.

E. Experiment customization

It is, of course, possible to experiment with the problem sizes and use these to represent different amounts of data and computation. It is also possible to change the path chunk size (number of paths transfered per chunk between host and FPGA), along with number of OpenMP threads for data reordering on the host, and path batch size (for double buffering on the FPGA).

APPENDIX B
ARTIFACT EVALUATION

A. Results Analysis Discussion

In the host code we use OpenCL's profiling capability which provides microsecond resolution timings for event (kernel execution) starting and ending. We also implemented manual timing via the *gettimeofday* call, to provide a second timing comparison point and ensure what OpenCL reported was correct (both approaches to timing matched very closely.) All calculated values were checked for consistency between the FPGA and CPU versions to ensure that they are calculating the same quantities and we were undertaking a fair experiment. For all experiments runtimes were averaged over at-least ten runs, and power consumption figures were reported by XRT for the Xilinx FPGA, AOCL for the Intel FPGA, RAPL for the CPU, and *nvidia-smi* on the GPU.