Evaluating Versal AI Engines for option price discovery in market risk analysis

Mark Klaisoongnoen  
EPCC, The University of Edinburgh  
Edinburgh, United Kingdom  
mark.klaisoongnoen@ed.ac.uk

Nick Brown  
EPCC, The University of Edinburgh  
Edinburgh, United Kingdom

Tim Dykes  
HPC/AI EMEA Research Lab,  
Hewlett Packard Enterprise  
Bristol, United Kingdom

Jessica R. Jones  
HPC/AI EMEA Research Lab,  
Hewlett Packard Enterprise  
Bristol, United Kingdom

Utz-Uwe Haus  
HPC/AI EMEA Research Lab,  
Hewlett Packard Enterprise  
Zurich, Switzerland

ABSTRACT
Whilst Field-Programmable Gate Arrays (FPGAs) have been popular in accelerating high-frequency financial workload for many years, their application in quantitative finance, the utilisation of mathematical models to analyse financial markets and securities, is less mature. Nevertheless, recent work has demonstrated the benefits that FPGAs can deliver to quantitative workloads, and in this paper, we study whether the Versal ACAP and its AI Engines (AIEs) can also deliver improved performance.

We focus specifically on the industry standard Strategic Technology Analysis Center’s (STAC™) derivatives risk analysis benchmark STAC-A2™. Porting a purely FPGA-based accelerator STAC-A2 inspired market risk (SIMR) benchmark to the Versal ACAP device by combining Programmable Logic (PL) and AIEs, we explore the development approach and techniques, before comparing performance across PL and AIEs. Ultimately, we found that our AIE approach is slower than a highly optimised existing PL-only version due to limits on both the AIE and PL that we explore and describe.

CCS CONCEPTS
- Hardware → Reconfigurable logic and FPGAs.

KEYWORDS
Option price discovery, SIMR, FPGAs, CGRAs, AI Engines, reconfigurable architectures

1 INTRODUCTION
Whilst FPGAs provide low-latency advantages for high-frequency trading, motivating developers to program them using hardware description languages (HDL) such as VHDL or Verilog, these properties are not as advantageous for quantitative finance workloads, discouraging extensive efforts in that domain. However, recent advancements both in hardware and software development ecosystems have significantly enhanced the capabilities of FPGAs increasing their usability for software developers, for instance, tools such as AMD-Xilinx’s Vitis [2] enable programming of them using C or C++, and hardened components such as AI Engines (AIEs). These advancements have consequently motivated communities to (re-)explore FPGAs for their workloads [3, 4, 12].

2 BACKGROUND
Market risk analysis involves evaluating the potential impact of price fluctuations on the financial positions held by investors and traders. Falling within the broader domain of quantitative finance, which employs mathematical models and datasets to scrutinise financial markets, these workloads are computationally intensive and demand substantial computational resources. Although the prevailing approach is to execute these models on CPUs and GPUs, there have been several activities exploring the acceleration of quantitative finance on FPGAs [6–8].

The Strategic Technology Analysis Center (STAC) serves as the orchestrating body for the industry-leading STAC Benchmark Council™. STAC members comprise some of the largest global banks, hedge funds, proprietary trading firms, exchanges, and leading technology vendors within finance. Their membership constitutes more than 400 financial institutions and over 50 technology...
vendors, with STAC being instrumental in presenting standardised financial benchmark suites for common financial workloads.

Relying on the exacting specifications of the STAC Benchmark Council for official audits, STAC members build and test their software and hardware systems for rigorous examination, optimisation, and validation against these real-world benchmarks. By contrast, in our research, we select components of the benchmarks to investigate the algorithmic, performance, and energy attributes and thus suitability of FPGAs for these workloads and are not bound to those strict auditing rules and thus much more flexible with implementation details. Consequently, the work and results presented in this paper are of a research and exploratory nature and must not be compared with official STAC reports and audit results.

2.1 Previous acceleration of the SIMR benchmark on programmable logic (PL)

In [9], the authors explored the porting of the SIMR benchmark [11] to the programmable logic of the AMD-Xilinx Alveo U280 and Intel Stratix-10 FPGAs, exploring the implementation differences between both architectures and the performance advantages of their host-device data streaming approach with both vendors. Building on a previous version, [10], Figure 1 sketches the dataflow design of the benchmark that was ported to the PL in [10] and which contains a detailed description of the algorithm and implementation. The boxes represent HLS dataflow stages and the arrows represent dataflow connections. The boxes represent HLS dataflow stages and the arrows represent dataflow connections. Each stage contains three nested loops, the outer looping over assets (the number of financial options that are being analysed), the middle looping over iterations, with data being filled or consumed per iteration of the bottom right cornered AIE in a single direction, thus imposing constraints on the mapping of kernels to AIEs.

The tooling provides several options for data communication between the kernels, which will be mapped to AIEs. These include streams, a uni-directional cascade stream and windows. Several trade-offs exist between these communication approaches, such as the AIEs are only able to read and write 32 bits per cycle from a stream, 256 bits per cycle from windows and up to 384 bits per cycle from cascade stream. Driven by the limits of the architecture, there are up to two inputs and outputs per kernel, any of which can be streams or windows. Whilst the 384-bit wide cascade stream provides higher bandwidth than a normal stream, it is uni-directional and connects the top left corner AIE on the AIE array with the bottom right cornered AIE in a single direction, thus imposing constraints on the mapping of kernels to AIEs.

Windows provide memory-mapped buffers and these work in iterations, with data being filled or consumed per iteration of the AIE graph. Whilst filling input data windows with data before the kernel starts adds some initial latency compared to streams, they can be used to provide a ping-pong buffer which fills and consumes concurrently, and data can be read from or written to windows in an arbitrary order, unlike streams. In this work, we concentrate on Single Precision (SP) Floating Point (FP) arithmetic, and each AIE is capable of operating on a vector width of eight SP FP numbers per cycle. Using AIEs to accelerate a computationally intensive CFD workload was investigated in [5] where, due to the number of inputs exhausting the streams between PL and AIEs, they were limited in scaling across the AIE array, however the algorithm explored here only requires two inputs and-so this will not be a limitation.

3 EXPERIMENTAL SETUP

For this work, we utilise an AMD-Xilinx VCK5000 featuring a Versal VC1902 ACAP with 16GB of DDR4-DRAM. All VCK5000 runs are built using Vitis 2022.1, with the PL operating at 300MHz, and the VC1902 containing 400 Al Engines running at 1.2GHz. For comparison, we use an Alveo U280 equipped with 8GB of High Bandwidth Memory (HBM2), also running at 300MHz, and Alveo kernels are built using Vitis 2021.1. Both the VCK5000 and Alveo U280 are PCIe-based cards hosted by a machine featuring a 32-core
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. Each data point is a 32-bit floating-point number.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>A</th>
<th>T</th>
<th>P</th>
<th>Elements (x10^6)</th>
<th>Datapoints (x10^6)</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>5</td>
<td>126</td>
<td>25</td>
<td>15.75</td>
<td>31.5</td>
<td>126</td>
</tr>
<tr>
<td>Small</td>
<td>10</td>
<td>126</td>
<td>25</td>
<td>31.5</td>
<td>63</td>
<td>252</td>
</tr>
<tr>
<td>Medium</td>
<td>20</td>
<td>252</td>
<td>25</td>
<td>126</td>
<td>252</td>
<td>1008</td>
</tr>
<tr>
<td>Large</td>
<td>30</td>
<td>504</td>
<td>25</td>
<td>378</td>
<td>756</td>
<td>3024</td>
</tr>
</tbody>
</table>

4 SINGLE AIE GRAPH DEVELOPMENT

In this work, we focus on the VariancePathQE stage of SIMR [9] exclusively as this is responsible for the largest portion of runtime and contains arithmetic operations that are all supported by the AIE tooling. In this section, we explore the porting and optimisation of the arithmetic operations executed by the VariancePathQE stage onto the AIEs. There are a total of 36 scalar operations that must be executed by this stage to calculate each element (i.e. for each path, of each timestep, of each asset).

During development, we enabled printf debugging (via the compiler flag –profile) and captured performance numbers in cycles by accessing the cycle counter of each AIE tile as illustrated in Listing 1. These are the cycle numbers reported in Table 2 and referred to throughout this section, however, we found that during hardware emulation the aiesimulator executes individual kernels in sequential graph order, and therefore performance cycles are useful for evaluating individual kernels for stream stalls and lock stalls but require interpretation for estimating overall graph performance.

Listing 1: Illustration of the AIE cycle counter to capture performance data

```c
void v1qe_fn(...) {
    #K cycle
    unsigned cycle_num[2];
aie::tile tile=aie::tile::current(); cycle_num[0]=tile.cy
    cles();
    ...
    cycle_num[1]=tile.cycles(); printf("start=%d,end=%d,total=%d\n",
    cycle_num[0],cycle_num[1],cycle_num[1]-cycle_num[0]);
}
```

Whilst a useful baseline, our initial naive kernel operated in scalar only and thus did not take advantage of the 8-way vectorisation for SP FP. This required leveraging the bespoke AMD Xilinx vectorisation intrinsics, such as aie::add and aie::mul, and significant restructuring of the code so that a single instruction can operate on eight floating-point elements per vector. Due to lack of support by the tooling, and availability of data, it was only possible to convert 24 of the calculations per element into vector equivalents, with 12 remaining scalar.

In Table 2, the vectorised kernel row reports performance when the code to leverage 8-way vectorisation, still in a single AIE kernel. Whilst this increases utilisation, and significantly improves performance compared to the scalar version of the code, it does add additional complexity. For instance, additional padding on the PL side will be required for cases when the number of input elements is not dividable by eight. The input padding is required as the AIE only starts processing once the first batch of eight input elements is available to the AIE entry point kernel. The reader might observe that the performance difference in Table 2 between the vectorised and scalar kernel versions is greater than eight, this is because pipelining of the loops on the AIEs was also enabled by adding the chess_prepare_for_pipelining pragma directive.

However, considering that there are 400 AIEs on the VC1902, leveraging only a single AIE does not make the best use of the resources. Consequently, we then further modified the code so that it could leverage multiple AIEs. The PL code that we are porting to the AIE comprises nine functions, and so this was the basis of our AIE design, decomposed into nine separate kernels, forming a
pipeline which drives the execution of the AIEs in a graph fashion. The objective was that, after the pipeline has been filled, overall the AIE graph running across the AIEs then yields one output result vector of eight elements per cycle. This adds an additional 20 read and write operations between the AIE kernels, and the performance for this implementation is reported as multi-AIE in Table 2, which can be observed reduces the total number of cycles further.

The interface between the PL and AIEs comprises 32-bit streams, but four of these can be bundled together and provide 128 bits on each access. This is useful due to the clock frequency mismatch between PL and AIEs, enabling four elements to be communicated by the slower PL per PL cycle. To this point, we were using a single input to the kernel (of 128 bits), and by splitting this into two inputs per core we doubled the bandwidth from reading 128 bits every four AIE cycles to 256 bits every four AIE cycles. This adds an additional 20 read operation per kernel.

To start processing the graph requires inputs v and z, and generates the output vlg.

Whilst in this section we are reporting results from running via simulation, rather than on the actual AIEs integrated with the PL, this technique still demonstrated some benefit which will be greater on the actual hardware when coupled in Section 5.

To this point, individual AIE kernels leverage more than a single vector operation, which potentially stalls the pipeline graph of kernels on the AIEs, and-so we explored placing a single operation in each kernel, resulting in 39 AIE kernels for the whole AIE graph. This is described by the row one vector operation per kernel in Table 2. As we maintain a runtime ratio\(^1\) of 1 for all kernels, every kernel is still assigned and bound to a separate AIE. For this version, the code is decomposed from its original structure in AIE kernels that contain individual arithmetic operations. It should be highlighted that the mapping of input and output ports for the graph, between kernels and for the entry point and final stage yields significant boilerplate code for the graph specification. As an example, the first two versions naïve scalar kernel and vectorised kernel in Table 2 resulted in tens of lines of code however the final one vector operation per kernel variant in Table 2 required hundreds of lines of code, adding significant complexity.

One challenge was that, as of Vitis 2022.1, the tooling supports FP division only when operating with scalars when using aie:div. Consequently, we implemented our own vector division using the inverse aie:inv followed by multiply aie:mul operations, both of which are supported for SP FP vectors. At the time of writing, there is also no support for the exponential arithmetic operation on vector types. Therefore, we implement previous stages of the variancePathQE function that builds on an intermediate result of expf, which we offload to AIE, on the programming logic of the Versal and feed the results as input to the corresponding AIE kernel.

5 COUPLING AI ENGINES WITH THE PL

Section 4 explored porting the VariancePathQE kernel of the SMIR benchmark [9] onto the AIEs, but this must be integrated with the existing parts of the code running on the PL which, as can be seen from Figure 1, require the sending of input data to VariancePathQE and integrating returned results from the AIEs with the subsequent dataflow stages LogPricePathQE, AssetPathExponential, longstaffSchwartzPathReduction and writeLongstaffSchwartzPath. A challenge is that developing for the AI Engine architecture fundamentally differs from that of the reconfigurable PL and these must be integrated in a way that the mismatched clock frequencies between the PL and AIE array can be overcome. For instance by leveraging higher bandwidth per cycle on the PL as described in Section 4. Furthermore, whilst hardware build times for the PL can be very long, these are far shorter (seconds or minutes) for the AIEs, with a re-packaging option to avoid rebuilding the entire bitstream if the AIE code alone has changed. The runs reported in Section 4 for our AIE design were using the AIE simulator, and when coupling with the PL code described in Section 5.1, some tweaks are required.

5.1 Programmable logic (PL) code

A challenge with the FPGA code is that the PL uses buffers in a nested loop as shown in Listing 2. This results in a circular dependency, which can severely limit performance on the FPGA but, due to path being the inner loop and there being a large number of paths, we can ensure that by the time a specific value is required from the cached_buf array, it has been calculated and written. Previously our approach used an HLS dependence pragma statement on line 5 to signify a false dependency for cached_buf between iterations.

```c
1  dtype cached_buf[SIZE], result;
2  asset_loop: for (unsigned int asset = 0; asset < assets; asset++) {
3    timestep_loop: for (unsigned int t=0; t < timestep; t++) {
4      path_loop: for (unsigned int path=0; path < path_size; path++) {
5        #pragma HLS dependence variable=cached_buf inter false
6        ... result = arithmetic_ops(..., cached_buf[path]);
7        cached_buf[path] = result; ...
8      }  
9    }  
10   }
```

Listing 2: Cached buffer in nested loops

However, this algorithmic pattern, where an iteration requires data from a previous iteration, is a problem on the AIEs because loops are not allowed in the AIE dataflow graph. Put simply, the AIE graph model does not support connecting the output of a graph iteration as input to the next graph iteration, and-so we must transfer data between the AIE array and PL in a loopback fashion. Consequently, to handle this we introduced a aie_loopback_adaptor HLS kernel which provides support on the PL to cache result data from the AIEs and feed this back as an input to subsequent iterations as required. This is illustrated in Figure 3, where there are two dataflow regions connected with an internal stream. The Handle return data stage reads results from the AIE array and sends them to the other stage and also to the PL kernel. The second, Loopback

---

\(^1\)The runtime ratio defines how much of the processing time of a single AI Engine core is required by the respective kernel.
function stage receives results from the previous stage (as long as it is not the first timestep, as otherwise initial values are provided by the PL kernel), stores these in on-chip BRAM or URAM, and serves the cached version required for the current path, timestep and asset. Two separate dataflow regions are required due to the interaction between the AIE array and PL kernel, as otherwise, this could stall a single region.

Figure 3: Illustration of the loopback adaptor on the PL used to support the cycling of data between AIE iterations

5.2 Single AIE graph performance

Table 3 reports the performance results of our single-PL-kernel experiments on the VCK5000. Moving the SIMR benchmark to the VCK5000 we report the performance for the version built in pure PL without any AIEs as VCK5000 PL-only. As the VCK5000 only provides relatively slower DRAM compared to the U280, it can be seen that performance on the VCK5000 is penalised where for the tiny problem size the runtime has more than doubled. This performance gap between both PL-only versions on the different FPGAs decreases when scaling the problem size up to large with the U280 still outperforming the VCK5000 by more than 1.25×.

Table 3 reports the performance of our kernels implemented across both PL and AIEs using the same naming and description as in Table 2. It can be seen that the naive scalar kernel version, implemented with streams reading and writing in 32 bits per cycle, yields the longest total execution time as this does not use the AIEs’ vectorisation capabilities, on a single AIE resulting in a more than 208× overhead compared to the VCK5000 PL-only implementation.

For the vectorised kernel version in Table 3, we report the performance implemented in 32-bit streams and alternatively with input and output windows of 128 bytes. The windows-based version with a window size of 128 bytes performs marginally better, but both versions are slower than running on the PL alone. One reason is that even with 128 bytes of window size the memory-mapped data communication mechanism does not significantly perform better than the 32-bit stream variant.

Increasing the utilisation on the AIE array, the multi-AIE version is the reading 128 bit per cycle for loads variant from Table 2 which splits the single vectorised kernel into multiple smaller kernels, each comprising multiple vector operations. Table 3 reports performance when connecting with streams or 128 byte windows, and it can be seen that using windows delivers best performance for the multi-AIE version across all problem sizes. The multi-AIE version with windows is around 2× faster than the vectorised kernel version with streams for the large problem size but still substantially slower than the PL-only version.

The one vector operation per kernel row, with 39 AIEs, provides performance that is marginally worse for the tiny problem size compared to the fastest multi-AIE version. However, when scaling the problem size this performance gap increases, with multi-AIE performing up to 1.57× better on the large problem size. Whilst we are able to increase the utilisation with a single graph instance, when profiling it was found that the many connections between these 39 AIEs increased the stall frequency. With overlapping dependencies in the graph as depicted in Figure 2, tweaking one kernel to reduce lock or memory stalls resulted in stalls in other kernels making it difficult to balance latency across the whole graph thus not fully leveraging the available compute across AIEs.

Ultimately, irrespective of the version, the AIEs fail to match the performance of running over the PL alone and there are several reasons for this. Firstly, the multi-AIE version, which is best performing, contains multiple vectorised operations per kernel and so does not provide a perfect pipeline generating results each cycle. When moving to one operation per kernel, the communication overhead between AIEs becomes dominant causing a significant number of memory stalls. Furthermore, the loopback adaptor on the PL, adds additional complexity and is a source of stalling.

6 MULTI AIE GRAPH PERFORMANCE

In the results reported in Section 5 the AIEs failed to match the performance of the PL on either the VCK5000 or Alveo U280. However, we were only using a small number of AIEs and hence the question was how might this performance be impacted when the number is increased. We therefore undertook experiments with multiple kernels, and Table 3 reports this multi-kernel performance on the PL and integrated with the AIEs across the defined problem sizes ranging from tiny to large.

In [9], we were able to fit six CUs on the Alveo U280. By contrast, on the VCK5000 the 16.3 MB of URAM on-chip memory is large enough for our large problem size to fit 16 CUs, but the 16 GB of
Table 3: Single Kernel and Multi Kernel performance across PL and AIE with total execution time in ms, averaged over five runs, Connection mode with either stream or 128-byte AIE window, stream for AIE designs fully implemented with streams, vs PL in times of VCK5000 PL-only runtime.

<table>
<thead>
<tr>
<th>Description</th>
<th>Connection mode</th>
<th>Tiny vs PL</th>
<th>Small vs PL</th>
<th>Medium vs PL</th>
<th>Large vs PL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alveo U280</td>
<td>-</td>
<td>79.29</td>
<td>132 44</td>
<td>0.68</td>
<td>474 31</td>
</tr>
<tr>
<td>VCK5000 PL-only</td>
<td>-</td>
<td>164.75</td>
<td>194 02</td>
<td>400.55</td>
<td>1718.93</td>
</tr>
<tr>
<td>naive scalar kernel</td>
<td>stream</td>
<td>34366.74</td>
<td>208 60</td>
<td>6871.15</td>
<td>354.15</td>
</tr>
<tr>
<td>vectorised kernel</td>
<td>stream</td>
<td>367.78</td>
<td>2.23</td>
<td>698.68</td>
<td>3.60</td>
</tr>
<tr>
<td>vectorised kernel</td>
<td>window</td>
<td>361.22</td>
<td>2.19</td>
<td>690 15</td>
<td>3.56</td>
</tr>
<tr>
<td>multi-AIE</td>
<td>window</td>
<td>328 18</td>
<td>1.99</td>
<td>587 32</td>
<td>3.03</td>
</tr>
<tr>
<td>one vec op per kernel</td>
<td>window</td>
<td>295.63</td>
<td>1.75</td>
<td>511 03</td>
<td>2.63</td>
</tr>
<tr>
<td>VCK5000 PL-only - 10 CU</td>
<td>-</td>
<td>298.83</td>
<td>1.81</td>
<td>555 72</td>
<td>2.86</td>
</tr>
</tbody>
</table>

Table 4: The Alveo U280 and the VCK5000 with 6 CUs, the Alveo U280 with 6 CUs, the VCK5000 version results in total execution between 2.1x and 1.7x slower for the tiny to large problem sizes respectively. Building on the host-device data transfer approach from [9] that utilises the larger off-chip memory as a streaming substrate, the VCK5000 is penalised by slower DRAM even with ten CUs compared to the same streaming approach on the Alveo U280 and its HBM with six CUs.

The multi-AIE version in Table 3 leverages six CUs on the PL, each CU being separately served with its own subgraph on the AIE. The Vitis tooling reports that the available control master interfaces on the PL are exhausted beyond six CUs and consequently, with 10 kernels on the AIEs per subgraph, and each of these bound to an exclusive, single AIE, this results in a total utilisation of 60 AIEs which is 15% of the AIE array. Therefore whilst one might attempt to overcome the performance limits of a single AIE graph discussed in Section 5.2 by scaling out, this is only possible to a limited extent. This issue is because the number of AXI ports required on the PL becoming too large, 14 are required per CU and with six CUs the tooling accommodating 84 but no more. Due to this limitation we explored coupling multiple AIE graphs to a single CU, as this addresses the need for dataflow design specifically tuned for the PL. There were four main reasons which limit performance; 1) our best performing AIE design has multiple vector operations per kernel, thus the design is unable to operate at a perfectly efficient pipeline and which generates a result each cycle, 2) when refactoring to one vector operation per kernel the design is dominated by inter-AIE communication overheads and memory stalls, 3) the loopback adaptor on the PL adds additional complexity and overhead that can stall the AIEs, 4) due to limits in the number of AXI interfaces on the PL we are unable to scale beyond 15% of the AIEs on the array, thus meaning it is not possible to ameliorate single kernel performance by scaling out.

7 CONCLUSIONS

In this paper, we focused on porting the SIMR benchmark to the Versal ACAP combining programmable logic, and specifically the VariancePathQE kernel to the AIEs. Focusing on the AIE development process, we investigated the data connections between AIEs, techniques for minimising stalling on individual AIE kernels, vectorisation of operations and the requirements to make data available fast enough to the compute. It was observed that moving to a higher utilisation on the AIE array is beneficial in reaching a lower average number of cycles per kernel, but this increases the complexity when balancing latency across kernels for graphs with competing or dependent branches.

Integration with the PL was a challenge due to the cyclical nature of the algorithm, where results from one iteration are required subsequently as input. Because cycles are disallowed by the AIE graph we developed a loopback adaptor on the PL to cache and serve intermediate results. Consequently, the uni-directional flow of the AIE graph is well-suited for applications that can be mapped onto such a graph but application requirements such as circular connections of the graph requires further engineering efforts.

When scaling from the single AIE graph to multiple graphs on the AIE array, we found that ultimately it was not possible to match the performance of the PL-only approach although this is an optimised dataflow design specifically tuned for the PL. There were four main reasons which limit performance; 1) our best performing AIE design has multiple vector operations per kernel, thus the design is unable to operate at a perfectly efficient pipeline and which generates a result each cycle, 2) when refactoring to one vector operation per kernel the design is dominated by inter-AIE communication overheads and memory stalls, 3) the loopback adaptor on the PL adds additional complexity and overhead that can stall the AIEs, 4) due to limits in the number of AXI interfaces on the PL we are unable to scale beyond 15% of the AIEs on the array, thus meaning it is not possible to ameliorate single kernel performance by scaling out.
ACKNOWLEDGMENTS

The authors express their gratitude to HPE who funded this work via their EMEA Research Lab internship programme and to STAC for granting access to the STAC-A2 benchmark and providing valuable advice and support. Additionally, we extend our acknowledgement to the ExCALIBUR H&ES FPGA and CGRA testbeds and the AMD-Xilinx HACC program for generously providing the compute resources utilised in this research.

REFERENCES