Exploring the Versal AI Engines for Accelerating Stencil-based Atmospheric Advection Simulation

Nick Brown
n.brown@epcc.ed.ac.uk
EPCC at the University of Edinburgh
Edinburgh, UK

ABSTRACT

AMD Xilinx's new Versal Adaptive Compute Acceleration Platform (ACAP) is an FPGA architecture combining reconfigurable fabric with other on-chip hardened compute resources. AI engines are one of these and, by operating in a highly vectorized manner, they provide significant raw compute that is potentially beneficial for a range of workloads including HPC simulation. However, this technology is still early-on, and as yet unproven for accelerating HPC codes, with a lack of benchmarking and best practice.

This paper presents an experience report, exploring porting of the Piacsek and Williams (PW) advection scheme onto the Versal ACAP, using the chip's AI engines to accelerate the compute. A stencil-based algorithm, advection is commonplace in atmospheric modelling, including several Met Office codes who initially developed this scheme. Using this algorithm as a vehicle, we explore optimal approaches for structuring AI engine compute kernels and how best to interface the AI engines with programmable logic. Evaluating performance using a VCK5000 against non-AI engine FPGA configurations on the VCK5000 and Alveo U280, as well as a 24-core Xeon Platinum Cascade Lake CPU and Nvidia V100 GPU, we found that whilst the number of channels between the fabric and AI engines are a limitation, by leveraging the ACAP we can double performance compared to an Alveo U280.

CCS CONCEPTS

• Hardware → Emerging tools and methodologies; **• Mathematics of computing → Solvers**; **• Computer systems organization → Multicore architectures**; **Reconfigurable computing**.

KEYWORDS

Versal ACAP, AI engines, FPGAs, stencil based algorithms, VCK5000, atmospheric advection, HPC

ACM Reference Format:
Nick Brown. 2023. Exploring the Versal AI Engines for Accelerating Stencil-based Atmospheric Advection Simulation. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3543622.3573047

FPGA '23, February 12–14, 2023, Monterey, CA, USA.

1 INTRODUCTION

The Versal Adaptive Compute Acceleration Platform (ACAP) is a new type of FPGA which combines Programmable Logic (PL) with other facets including CPU-based Programmable Subsystem (PS) and AI engines [4]. These AI Engines, or AIEs and we use these two terms interchangeably throughout this paper, are of specific interest here as they are designed to accelerate highly-parallel vector operations. The Versal AI-series contains up to 400 engines running between 1 and 1.2 GHz, and each engine follows a Very Long Instruction Word (VLIW) design, capable of issuing seven instructions per cycle. AI engines are capable of undertaking 8-way vectorized single-precision floating point operations and up to 128 8 bit fixed point arithmetic operations per cycle.

The large amount of raw compute provided by the AIEs is interesting for High Performance Computing (HPC) workloads, where the ability to use the Versal's PL to tailor memory accesses bespoke to an application and the AI engines to accelerate the compute has potential. To date there have been a very limited number of preliminary AIE studies [5] [10], and-so an important outstanding question is whether these engines can be effectively leveraged for real world HPC kernels. In this work we use the atmospheric advection kernel of the Met Office NERC Cloud model (MONC) [3], which is an open source high resolution atmospheric modelling framework, as a vehicle to explore the AI engines. Following a stencil-based compute pattern, which is very common in HPC codes, in this short paper we explore how to best map this compute pattern onto the AIEs and how performance compares against other approaches. This paper is structured as follows, in Section 2 we explore the background to this work before summarising the experimental setup in Section 3. Section 4 explores structuring our AIE kernel(s) and interfacing these with the PL, before undertaking a performance comparison against other hardware in Section 5. We then conclude and discuss recommendations in Section 6.

The novel contributions of this paper are **1)** An exploration of techniques to most effectively structure AIE kernels **2)** An initial performance comparison between the AIEs and other hardware **3)** Highlighting some of the limitations of the current AIE technology that one must consider when working with the hardware.

2 BACKGROUND

2.1 The Versal AI engines

The VLIW design of Xilinx's new AI engines is such that, per cycle, each engine is capable of issuing a maximum of two loads, one store, one scalar operation, one fixed point or floating point vector operation, and two move instructions. The vector unit is of size 256 bits, and focusing on single precision floating point arithmetic in this paper, each engine is capable of undertaking up to eight single precision floating point calculations per cycle. Consequently it is important to ensure code is correctly vectorized to obtain best performance on the AIEs. Based on 400 AI engines running at 1.2GHz on the VCK5000, there is a theoretical single precision floating point performance of 3.6 TFLOPS.

AI engines are arranged in a 2D array, with engines connected to their neighbours in both dimensions. Each engine contains 16KB of program memory and 32KB of local data memory and, for the later, is able to directly access the memories of three of its neighbours providing a total of 128KB contiguous addressable data memory [8]. Furthermore, each engine has two 32 bit input streams and two 32 bit output streams which are combined with a FIFO to provide 128 bit access every four clock cycles. Lastly, AI engines connect to one of their neighbours via a cascade stream which is 384 bits wide and designed to allow arithmetic operations to be chained.

AIE code comprises two parts, kernels which will be mapped to AI engines and a graph description which connects kernels together via their streams and memories, as well as to the PL. Programmatically there are two ways in which data can enter or leave a kernel, windows and streams [9]. Windows provide a buffer, where the current data position in the window is tracked. For input windows data is consumed from this buffer by the kernel, for output windows data is written. The other approach, a stream, provides an infinite number of scalars and vectors that can be read and written by the kernel. There underlies an important difference between these two approaches, where a window of data will only progress to the next window between outer iterations of the kernel, as driven by the AIE graph, whereas streams can continually be read from and written to inside the kernel. Consequently, with windows one must frequently start and stop their kernels to refresh the window data, which is not required with streams.

Whilst the AI engines are the major focus in this paper, it is also important to highlight the general architectural improvements that Xilinx have made to the PL in their Versal series. Built on a 7nm process technology, numerous components including DDR controllers and PCIe interface have been hardened compared to previous generations [1]. Furthermore, a dedicated Network on Chip (NoC) is provided which not only connects the PL with the AIEs, but can also be used between IP blocks on the PL.

2.2 Piacsek and Williams advection kernel

Advection is the movement of values through the atmosphere due to wind and, at around 40% of the runtime, is the single longest running piece of functionality in the MONC model [3]. The code loops over three fields; U, V and W, representing wind velocity in the x, y and z dimensions respectively. This Piacsek and Williams (PW) [6] advection scheme is called each timestep of the model and calculates advection results, otherwise known as *source terms*, for each field. This advection scheme is a stencil based algorithm, of depth one, where calculating the value of a grid cell requires contributions from neighbouring values across all three dimensions.

In previous work [2] this kernel was ported to an Alveo U280 using High Level Synthesis (HLS) and leveraging the *dataflow* HLS pragma to run multiple components concurrently. The structure of this HLS kernel is illustrated in Figure 1, where the boxes are dataflow regions and arrows between these are internal HLS streams. 3D shift buffers provide a bespoke memory solution which is capable of delivering all 27-stencil values per cycle to the advection compute stages, which was found to be the optimal approach even though not all 27 neighbouring stencil values are required by the advection calculations. Given this existing structure it was our hypothesis that we could replace the advection calculation stages with streams to and from the AI engines, still leveraging the existing tailoring of memory accesses on the PL that worked well in [2], with the raw compute power of the AI engines.

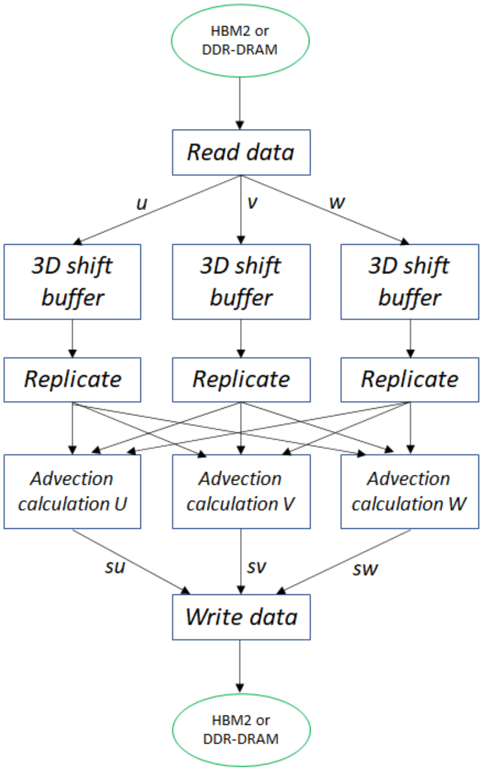

Figure 1: Dataflow design of HLS advection kernel from [2]

3 EXPERIMENTAL SETUP

In this work we are using a Xilinx VCK5000 containing a Versal VC1902 ACAP and 16GB of DDR4-DRAM. All VCK5000 runs are built using Vitis 2022.1, the PL is running at 300MHz, and the VC1902 contains 400 AI engines running at 1.2GHz. We compare against an Alveo U280 which contains 8GB of HBM2, is 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 containing a 32-core AMD EPYC 7502 processor and 256GB DRAM.

All reported results are averaged over five runs and performance results are reported as useful FLOPS, which is the number of floating point operations undertaken that contribute to the calculation's result. Our performance numbers measure device-side execution time only and do not include the time taken to copy input data to, or result data from, the host and device. This is because we are most interested in the performance of the AIEs in this work, and device-side performance therefore provides a clearer picture when comparing against other technologies that exhibit different host-device data transfer overheads.

4 AIE PORTING AND OPTIMISATION

We started by decomposing the advection stencil-based calculation into constituent operations, resulting in, for each grid cell, the code undertaking six additions, followed by six multiplications, then four subtractions and finally an addition reduction to sum these subtractions together. A floating point vector of size six is not supported by the tooling and-so we pad with an additional two empty values to make a vector of size eight. This is why we report useful, rather than total, FLOPS, as useful FLOPS ignores the processing of these empty values by only considering those floating point operations that actually contribute to the advection result.

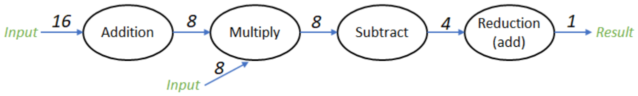

Figure 2: Illustration of AIE calculations per grid cell, with the numbers representing the number of single precision floating point numbers provided.

The structure of this kernel is illustrated in Figure 2, with the first 8-way vector addition requiring sixteen floating point numbers comprising the operands. The multiplication requires an additional eight input numbers which are multiplied by the result of the preceding addition. We packaged this as a single AIE kernel and Listing 1 provides a partial sketch of the code. In order to prepare for the vector addition, streams of four numbers are read and loaded into the appropriate locations of the *lhs* and *rhs* vectors in lines 11 to 14. These vectors are then provided as arguments to the *aie::add* method at line 16, which undertakes the vectorized addition. Multiplication, subtraction, and reductions operations are handled similarly and omitted for brevity. It can be seen at line 6 that we are looping over grid cells, and the directives at lines 7 and 8 instruct the AIE compiler to undertake software pipelining where possible, attempting to keep the VLIW slots filled as per Xilinx's best practice [8].

```
1  void cell_advection(input_stream<float> * __restrict in_A,
        input_stream<float> * __restrict in_B, output_stream<
        float> * __restrict out) {
2    aie::vector<float, 4> in_data;
3    in_data=readincr_v<4>(in_A);
4
5    int32 cells=(int32) in_data.get(0);
6    for (int i=0;i<cells;i++)
7    chess_prepare_for_pipelining
8    chess_loop_range(64,) {
9      aie::vector<float,8> lhs_nums, rhs_nums;
10
11     lhs_nums.insert(0,readincr_v<4>(in_A));
12     lhs_nums.insert(1,readincr_v<4>(in_A));
13     rhs_nums.insert(0,readincr_v<4>(in_B));
14     rhs_nums.insert(1,readincr_v<4>(in_B));
15
16     aie::vector<float,8> vadd=aie::add(lhs_nums,rhs_nums);
17     ....
18   }
```

Listing 1: Sketch of AIE advection kernel code

The AIE API provides adaptive dataflow graphs which enables parameters to be dynamically set at runtime. However this is not supported by the VCK5000 shell and as such an alternative was required for setting the number of loop iterations at line 6. This is the reason that, for lines 2 to 5 in Listing 1, four floating point numbers are read from the *in_A* stream and the first of these is extracted, cast to an integer, and used as the number of grid cells to loop over (this corresponding value has been streamed from the PL on start up). We must read the number of cells as a float because there are a maximum of two inputs and two outputs per kernel, and both inputs are required for the loading of operands.

```
1  class simpleGraph : public graph {
2  private:
3    kernel cell_advection_kernel[3];
4  public:
5    input_plio in_A[3], in_B[3];
6    output_plio out[3];
7
8    simpleGraph(){
9      cell_advection_kernel[0]=kernel::create(cell_advection);
10     ...
11     in_A[0] = input_plio::create("krnl_0_in0", plio_128_bits,
            "data/input_A.txt");
12     in_B[0] = input_plio::create("krnl_0_in1", plio_128_bits,
            "data/input_B.txt");
13     out[0] = output_plio::create("krnl_0_out1", plio_32_bits,
            "data/output_0.txt");
14     ...
15     for (int i=0;i<3;i++) {
16       connect<stream>(in_A[i].out[0],
              cell_advection_kernel[i].in[0]);
17       connect<stream>(in_B[i].out[0],
              cell_advection_kernel[i].in[1]);
18       connect<stream>(cell_advection_kernel[i].out[0], out
              [i].in[0]);
19 }}};
```

Listing 2: Sketch of AIE graph building code

This code of Listing 2 builds the high-level AIE graph, mapping kernels to individual AI engines. Three advection kernels are created, one for each field, and lines 11-13 defines the input and output ports between the PL and AIE for the first kernel (kernels two and three are omitted for brevity). These AIE ports are connected to the kernel ports in the loop at lines 15 to 19. As described in Section 2.1, physical connections between AI engines are 32 bits wide, but it can be seen for input ports we specify *plio_128_bits* at lines 11 and 12. This directs the AIE compiler that streams on the PL are 128 bits wide (of type *qdma_axis<128,0,0,0>*) and therefore data will arrive in packets of 128 bits and be unpackaged into four 32 bit stream values. The reason for this is performance, where the PL is running much slower (in our case 300MHz) compared to the AIEs (1.2 GHz) and consequently in one clock cycle the PL is providing four 32 bit numbers which the AIE will then unpack per cycle. 128 bits is the maximum width supported, and this is why in Listing 1 the eight numbers comprising either side of the calculation are read via two *readincr_v* calls of size four at lines 11-12 and 13-14.

Version	Performance *(GFLOPS)*	Compared to PL-only
PL-only (no AIEs)	14.32	-
Initial	1.99	14%
Multi-kernel	4.06	28%
Cascade stream	2.78	19%
Cascade multiplex	3.87	27%
Multi-kernel windows	0.91	6%
Chunking windows	10.32	72%
Reduction on host	16.13	113%
Double vectorization	18.48	129%

Table 1: Compute performance of different versions of AIE design compared against PL-only non-AIE implementation. All runs undertaken in single precision floating point on Xilinx VCK5000 using a problem size of 67 million grid points.

It can be seen in Listing 2 that there is a separate kernel instance created for each of the three fields, with each of these running on a separate AI engine. Whilst the calculations for each field are different, this difference lies in the specific stencil locations that are used, and the underlying arithmetic operations are the same. Consequently we are able to reuse the same kernel code, but provide different values to these from the PL side per field. The performance of this version is reported in Table 1 by the *initial* row, and it can be seen that this was significantly slower compared to instead undertaking all arithmetic operations on the PL (*PL-only (no AIEs)*).

4.1 Optimising the data transfer

The maximum 128 bit width of data between the AIEs and PL was a major reason for the poor performance of our initial version reported in Table 1. This was because, per cycle, the PL was only able to stream four single precision floating point numbers per stream to the AIE, whereas 24 were required (16 for the addition and 8 for the multiplication). The number of inputs to an AIE kernel is limited to two, therefore meaning that the PL could provide a maximum of eight values per cycle. Consequently three writes on each stream were required per grid cell and this conflict resulted in an initiation interval of three in our HLS code on the PL.

To address this we experimented with alternative kernel structures and, as illustrated by Figure 3, split the code into multiple kernels each corresponding to a specific operation. By splitting apart the addition and multiplication, so each handles four of the eight calculations, we were able to increase the overall number of streams to six (two per kernel). There is a downside, as each individual kernel is now under utilised because it is now only undertaking four vectorized operations per cycle rather than eight, but this splitting results in six, rather than two, 128 bit streams connecting the PL to AIE kernel inputs. Consequently the HLS kernel running on the PL is able to stream the entirety of a grid cell's required data each cycle, reducing the initiation interval to one.

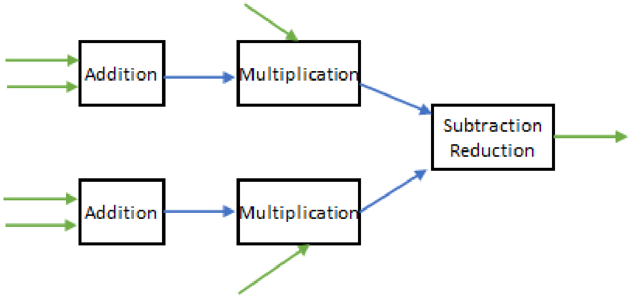

Figure 3: Multi-kernel design, with constituent operations running across AIEs and connected by streams. Blue arrows are internal streams, green arrows are external streams between the AIEs and PL.

The performance of this approach is reported by the *multi-kernel* row of Table 1, and whilst this doubled performance compared to the initial version, it was still slower than the PL-only implementation. When undertaking profiling of our multi-kernel code using Vitis analyzer, we discovered that kernels were stalling on stream reads for over 60% of the time. This is because, as described in Section 2.1, the physical streams between AI engines are 32 bits wide whereas per vectorized operation the kernel is generating 128 bits. Consequently the kernels were stalling waiting for the arrival of this data before operating upon it.

Connecting AIE kernels via cascade streams is an alternative approach and these, unlike the normal 32 bit streams, are 384 bits wide. We packed the 128 bit results into the cascade stream's *accfloat* type, and streamed the entirety of the required data in one cycle. However, the limitation with cascade streams is that they physically connect between AIE cores by travelling in a horizontal manner, and when reaching the edge of a row connecting to the core above. Consequently their connection is inflexible, with each AIE core capable of only consuming cascade stream input from a single predefined neighbour and providing cascade stream output to its other neighbour. This is a problem for our multi-kernel design illustrated in Figure 3 as the *subtraction-reduction* kernel requires inputs from two kernels, effectively requiring two cascade streams to feed into an AIE which is not possible on this architecture.

Therefore, to experiment whether cascade streams would improve performance, we adopted the design illustrated in Figure 4, where one addition kernel undertakes all eight addition operations, and a separate kernel then undertakes the multiplication, subtraction, and reduction. The performance of this configuration is reported by *cascade stream* in Table 1, and the major reason for the poor performance is that the initiation interval on the PL increased to two as streams to the addition kernel require two writes per PL cycle as all eight pairs of operands are required by the single kernel. To address this we multiplexed the cascade streaming approach, with two separate copies on the AI engines such that, on average, over two clock cycles each AIE configuration receives its data. Performance of this approach is reported by the *cascade multiplex* row in Table 1, which improved performance but was still slower than that obtained by the non-AIE PL-only approach. Incidentally we also experimented with 4-way and 8-way multiplexing but this had no measurable improvement on performance.

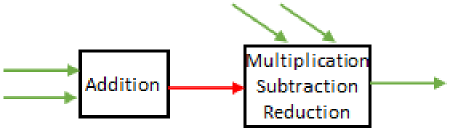

Figure 4: Cascade streaming approach, Red arrow is cascade stream, green arrows are external streams to/from the PL.

To this point we have explored connecting kernels and the PL via streams, however it is also possible to use windows which provide buffers. Importantly, an AIE can read up to 256 bits per cycle from memory compared to 32 bits from streams. Therefore we reverted to our multi-kernel design of Figure 3 and used windows instead of streams between the kernels as well as to drive input and output data between the PL. This is illustrated in Figure 5 and the performance is reported as *multi-kernel windows* in Table 1. It can be seen that the performance was extremely poor and this is because we were operating the windows on a grid cell by grid cell basis. This meant that there was no longer a pipelined loop within each kernel because between each grid cell the kernel was stopped and restarted by the AIE graph to fill and empty the windows as required by the AIE tooling.

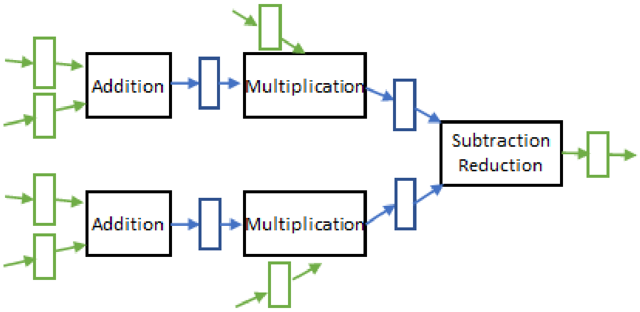

Figure 5: Multi-kernel windowing approach, coloured squares are the windows, blue connects kernels internally, and green arrows are external streams to/from the PL.

We modified our windowing approach to work in chunks, where data for a number of grid cells is buffered into the windows and these operate ping-pong fashion where one copy is filled with data from the producer (either the PL or another AIE kernel) whilst the other window copy is being consumed, with these switched between outer iterations of the AIE graph. Consequently our addition, multiplication, and subtraction-reduction AIE kernels are concurrently processing different chunks of grid cells based upon the data available, effectively operating as a pipeline. An added complication was that because AIE kernels are operating out of sync, for example the multiplication kernels are one chunk behind their corresponding addition kernels, this stalled the PL. This was because when streaming data to the AIEs, writes to the multiplication streams are blocked waiting for the window to become free, but this waiting on the PL also blocks writes to the addition streams which are required to progress the addition AIE kernel which will unlock its multiplication kernel. The solution was to implement explicit ping-pong buffering on the PL for the multiplication streams, with a dataflow region working in sizes of *chunk* which is concurrently filling a buffer with the current chunk's data and streaming out the previous chunks data to the AIE kernel.

Performance is reported by *chunking windows* in Table 1, where it can be seen that this approach has significantly increased performance on the AIEs, however it is still slightly slower than the PL-only. Based on profiling via Vitis analyzer we found that the subtraction and reduction kernel was taking around double the execution time of the other kernels and this imbalance of work was causing additional stalling. Consequently we modified the kernel to perform subtraction only and streamed back to the PL 4 floating point numbers which the PL then adds together. This is reported by the row *reduction on host* in Table 1 which outperforms the PL-only.

As described previously, in this multi-kernel approach each kernel is only working with vector sizes of four whereas the hardware is capable of undertaking eight single precision floating point operations per cycle. Working with windows, it was trivial to read two grid cells concurrently, placing the first in the lower portion of the vector and the second grid cell in the higher portion. Consequently this meant that vector operations were now running over eight operands, effectively processing two grid cells per AIE vectorized operation. This is reported by *double vectorization* in Table 1 and resulted in a performance improvement, albeit modest as we are still limited to streaming data for only one grid cell between the PL and AIEs per cycle due to the maximum of port width of 128 bits.

5 MULTIPLE HLS COMPUTE UNITS

In Section 4 we focused on a single PL HLS Compute Unit (CU). By decomposing across the advection problem's grid space, we can scale to multiple HLS CUs, all with a separate 3D part of the grid and working independently, connected to their own AIEs. Using our optimised AIE approach, which requires fifteen AIEs per HLS CU, we compared performance against other hardware options and Table 2 reports these results.

The advection kernel running on the AI engines of the VCK5000 is reported by the row *VCK5000 AIEs* in Table 2. Whilst not documented directly, there are a maximum of 78 128-bit PLIO input streaming interfaces that only become apparent during compilation as we scaled. This is because AIE tile contains eight 32-bit AI Engine to AXI4-Stream channels [7] and there are 39 tiles. Consequently, there are a total of 312 32-bit channels connecting the AIEs to the PL, or a maximum of 78 128-bit channels as each of these is built using four 32-bit links. Incidentally AIEs accessing DRAM directly, without the PL, would also encounter this limitation as the data still needs to traverse these same physical links.

With six input streams per field, and three fields per CU, this results in a maximum of four HLS CUs. We are therefore using 60 AIEs in total, and up to four CUs the performance scales well. Consequently this hardware restriction is a major limitation because, if we were able to scale to a greater number of CUs, then performance would likely increase significantly. The importance of streaming an entire grid cell per cycle between the PL and AIEs was highlighted in Section 4, and out of the two AIE kernel designs which enable this, multi-kernel and multiplexed cascade stream, the multi-kernel is preferable in this regard as it requires six input streams per field compared to eight for the multiplexed cascade stream.

The Alveo U280 was configured with six HLS CUs, which is the maximum number that can fit due to limits on the number of ports in the Alveo shell. It can be seen that performance on the Alveo U280 is similar to that obtained on the VCK5000 using AI engines, even though there are only four CUs on the VCK5000. This is especially impressive considering that the U280 contains external HBM2 memory whereas the VCK5000 only has DDR4. We are able to fit eight CUs onto the PL-only VCK5000 configuration, which does not suffer from limitations on the number of ports due to the Versal containing a NoC which HLS kernels are connected to. The *VCK5000 combined* result reports performance for a combination of the four AIE CUs with six PL-only CUs on the VCK5000, and this combined approach which leverages both the AIEs and PL for calculations delivers double the performance of the U280.

By comparison, the scheme running over the 24-core Cascade Lake Xeon Platinum CPU, which was threaded via OpenMP and compiled using GCC version 10.2 performs poorly compared with every other hardware technology. The V100 GPU version is implemented using OpenACC and version 20.9 of the Nvidia compiler, and this out-performs all other CPU and FPGA configurations, which is largely in agreement with [2]. Whilst the Versal has closed the gap with the GPU, it is unfortunate that AIE hardware restrictions ultimately limit the number of AIE CUs to four.

Description	Performance (GFLOPS)
VCK5000 AIEs *(4 CUs)*	68.73
VCK5000 PL-only *(8 CUs)*	101.78
VCK5000 combined *(4 and 6 CUs)*	145.11
Alveo U280 *(6 CUs)*	72.32
24-core Xeon Platinum CPU	23.52
V100 GPU	227.89

Table 2: Compute performance of FPGA AIE and PL-only approaches compared to 24-core Cascade Lake CPU and Nvidia V100 GPU. All runs undertaken in single precision floating point using a problem size of 67 million grid points

6 CONCLUSIONS AND RECOMMENDATIONS

In this paper we have explored porting of the PW atmospheric advection scheme to the Versal, utilising the PL for tailoring memory accesses via a 3D shift-buffer and the AIEs for undertaking computation. Representative of a much wider class of stencil-based algorithms, which are popular in HPC workloads, we found that the major challenge was being able to most effectively interface the PL and AIEs to ensure data continually flows between the two. There are several possible approaches, and we have explored how hardware and tooling limitations drive specific choices and the performance impact of these. Ultimately, we found that the most effective approach was to use windows in a ping-pong fashion, working on chunks of data within the AIE kernels which rely on software pipelined loops and fully filled 8-way vectorization. Comparing against other hardware options, we found that a major limitation in obtaining performance was in the total number of streams between the PL and AIEs, which meant we were unable to scale beyond four HLS CUs. However four CUs using AIEs on the VCK5000 performed comparatively to six CUs on the Alveo U280 with the later benefiting from HBM2. The PL-only approach on the VCK5000 delivered impressive performance against the other FPGAs and CPU, which was largely due to being able to fit eight HLS CUs onto the PL, and when combining the AIEs and PL for compute we were able to deliver a significant improvement in performance compared to other FPGA approaches and the CPU. Therefore, AIEs aside, our PL-only experiments demonstrate that the Versal is a powerful architecture and improves on the Alveo.

From a development perspective there are many advantages in using the AIEs, and this will likely make the ACAP more accessible to software developers compared to traditional FPGAs. These include the overall compilation being much quicker, the ability to undertake much of the development exploration using simulation which itself is fast, no need to rebuild the PL if the interfaces between the PL and AIEs have not changed (which means bitstream regeneration takes around a minute), and the rich profiling tooling to provide insights where bottlenecks lie in the code. However it is crucial to match the workload to the architecture, and given the bandwidth between the PL and AIEs those kernels which have a higher FLOP to byte ratio than the stencil computation described in this paper will likely suit the AIEs much better. Therefore, an important lesson from this work is to focus primarily on those kernels that will not be limited by the current generation's PL to AIE interface, and algorithms with a high FLOP to byte ratio are likely where we will see the greatest benefit from this architecture.

Considering future enhancements to the Versal, it would be beneficial if AMD Xilinx were to make the physical streams between AIEs wider than 32 bits and increase the PL to AIE stream size from 128 to 256 bits, as well as supporting a larger number of PLIO streams. Increased flexibility around vector sizes would also be useful, for instance it is not possible to have a single precision floating point vector of numerous sizes including six, which required us to pad with empty values to eight, increasing the amount of data transferred between PL and AIE. Considering the wider Vitis technology, within HLS it is not possible to create arrays of external AXI streams (e.g. of type *qdma_axis<128,0,0,0>*), this would be beneficial because interfacing with AIEs will likely require a greater number of AXI streams compared to what is currently common in HLS.

7 ACKNOWLEDGEMENTS

The authors thank the ExCALIBUR H&ES CGRA project for funding this work. We also acknowledge the ExCALIBUR H&ES FPGA testbed and AMD Xilinx HACC program for access to compute resource used in this work, the later who also kindly provided comments and technical advice. 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] Sagheer Ahmad, Sridhar Subramanian, Vamsi Boppana, Shankar Lakka, Fu-Hing Ho, Tomai Knopp, Juanjo Noguera, Gaurav Singh, and Ralph Wittig. 2019. Xilinx first 7nm device: Versal AI core (VC1902). In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, 1–28.

[2] 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.

[3] Nick Brown et al. 2015. A highly scalable Met Office NERC Cloud model. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*. University of Edinburgh, 132–137.

[4] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: VersalTM architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 84–93.

[5] David Lee, Gregory Allen, Matthew Cannon, Hunter Earnest, Paul Thelen, Nathaniel Dodds, Jeffrey McCasland, and Carol Chen. 2021. *Preliminary Results from Heavy-Ion Irradiation of the Xilinx Versal ACAP*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[6] Steve A Piacsek and Gareth P Williams. 1970. Conservation properties of convection difference schemes. *J. Comput. Phys.* 6, 3 (1970), 392–405.

[7] Xilinx. 2021. Versal ACAP AI Engine Architecture Manual (AM009). https://docs.xilinx.com/r/en-US/am009-versal-ai-engine

[8] Xilinx. 2022. AI Engine Kernel Coding Best Practices Guide (UG1079)). https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding

[9] Xilinx. 2022. Versal ACAP AI Engine Programming Environment User Guide (UG1076). https://docs.xilinx.com/r/en-US/ug1076-ai-engine-environment

[10] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. 2022. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. *arXiv preprint arXiv:2206.13734* (2022).