Exploring the acceleration of the Met Office NERC Cloud model using FPGAs

Exploring the acceleration of the Met Office NERC Cloud model using FPGAs

Nick Brown

EPCC, The University of Edinburgh, Bayes Centre, Edinburgh, UK
n.brown@epcc.ed.ac.uk

Abstract. The use of Field Programmable Gate Arrays (FPGAs) to accelerate computational kernels has the potential to be great benefit to scientific codes and the HPC community in general. With the recent developments in FPGA programming technology, the ability to port kernels is becoming far more accessible. However, to gain reasonable performance from this technology it is not enough to simple transfer a code onto the FPGA, instead the algorithm must be rethought and recast in a data-flow style to suit the target architecture. In this paper we describe the porting, via HLS, of one of the most computationally intensive kernels of the Met Office NERC Cloud model (MONC), an atmospheric model used by climate and weather researchers, onto an FPGA. We describe in detail the steps taken to adapt the algorithm to make it suitable for the architecture and the impact this has on kernel performance. Using a PCIe mounted FPGA with on-board DRAM, we consider the integration on this kernel within a larger infrastructure and explore the performance characteristics of our approach in contrast to Intel CPUs that are popular in modern HPC machines, over problem sizes involving very large grids. The result of this work is an experience report detailing the challenges faced and lessons learnt in porting this complex computational kernel to FPGAs, as well as exploring the role that FPGAs can play and their fundamental limits in accelerating traditional HPC workloads.

Keywords: FPGAs · High Level Synthesis · MONC · HPC acceleration

1 Introduction

The Met Office NERC Cloud model (MONC) [1] is an open source high resolution modelling framework that employs large eddy simulation to study the physics of turbulent flows and further develop and test physical parametrisations and assumptions used in numerical weather and climate prediction. A major atmospheric model used by UK weather and climate communities, MONC replaces an existing model called the Large Eddy Model (LEM) [2] which was an instrumental tool, used by scientists, since the 1980s for activities such as development and testing of the Met Office Unified Model (UM) boundary layer scheme [3], convection scheme [4] and cloud microphysics [5]. In order to further the state of the art, scientists wish to model at a greater resolution and/or near real time which requires large amounts of computational resource. The use of modern HPC machines is crucial, however the problems are so challenging that any opportunity to accelerate the model is important. Whilst MONC has traditionally been run across thousands of Intel CPU cores in modern supercomputers [1], a key question is what sort of architecture is optimal going forwards, and what changes are required to the code?

The idea of converting an algorithm into a form that can program a chip directly, and then executing this at the electronics level, has the potential for significant performance and energy efficiency advantages in contrast to execution on general purpose CPUs. However, the production of Application Specific Integrated Circuits (ASICs) is hugely expensive, and so a middle ground of Field Programmable Gate Arrays (FPGAs) tends to be a good choice. This technology provide very many configurable logic blocks sitting in a sea of configurable interconnect, and tooling developed by vendors supports programmers converting their algorithms down to a level which can configure these fundamental components. With the addition of other facets on the chip, such as fast block RAM (BRAM), Digital Signal Processing (DSP) slices, and high bandwidth connections off chip, FPGAs are hugely versatile. It's a very exciting time for this technology because, whilst they have a long heritage in embedded systems and signal processing, more recently there has been significant interest in using them more widely, such as the deployment of FPGAs in Amazon's F1 cloud computing environment

However, the use of FPGAs in scientific computing has, until now, been more limited. There are a number of reasons for this, not least the significant difficulty in programming them. But recent advanced in high level programming tools means that this technology is now more accessible for HPC application developers. However it isn't enough to simply copy some code over to the FPGA tooling and *hit go*, a programmer must to change the entire way in which they approach their algorithms, moving to a data-flow style [6], in order to achieve anywhere near good performance.

In this paper we describe work done porting the computationally intensive advection kernel of the MONC atmospheric model to FPGAs. We compare this against the performance one can expect from more traditional Intel based CPU systems, and explore the very many options and pitfalls one must traverse in order to obtain good performance of codes on FPGAs. In short, the contributions of this paper are

- Exploration of the steps requires to port a computationally intensive kernel onto FPGAs using HLS. We will show that it is not enough to simply copy the code over, but instead the whole approach needs to be rethought and recast.
- An experience report of using FPGAs to solve a computationally intensive kernel on large grids. We run experiments up to grid cells of 257 million grid points, each point requiring over fifty double precision operations.
- A detailed performance comparison, for this application, of the performance characteristics of our FPGA accelerated kernel in comparison to running on

Intel CPUs commonly found in HPC machines. We are looking to answer the question, is it worth fitting Intel based systems with FPGAs?

This paper is structured as follows, in Section 2 we describe the general background, introducing the MONC model in more detail, the FPGA hardware we are using in this work and describe the approach we have adopted in terms of programming the FPGA. In Section 3 we describe the development of our FPGA kernel, in HLS, and explore the different steps that were required to obtain reasonable performance from this code. In Section 4 we explore the block design adopted to integrate our kernel with the wider infrastructure supporting it. A performance comparison of our FPGA solution against Intel CPU products commonly found in HPC machines is explored in Section 5, before we draw conclusions and discuss further work in Section 6.

2 Background

2.1 Met Office NERC atmospheric model

The Met Office NERC Cloud model (MONC) has been developed in Fortran 2003 and, like many LES models, proceeds in timesteps, gradually increasing the simulation time on each iteration until reaching a predefined termination time. The model works on prognostic fields, u, v and w for wind in the X, Y and Z dimensions, and a number of other fields which we do not considered in this paper. Figure 1 illustrates the high level structure of a timestep, where each piece of functionality executes sequentially, one after another. Initially, all prognostic fields are halo swapped between neighbouring processes and then the sub-grid functionality determines model parameterisations. Next, the dynamics group, often referred to as the dynamical core, performs Computational Fluid Dynamics (CFD) in order to solve modified Navier-Stokes equations, which is followed by the pressure solver, solving the Poisson equation. The timestep then concludes with some miscellaneous functionality such as checking for model termination. Over 70% of the runtime is spent in the dynamical core and, in particular, the advection scheme. Advection calculates movement of values through the atmosphere due to wind, and at around 50% of the overall runtime, is the single longest running piece of functionality. A number of different advection schemes are provided, and these require all the model's prognostic fields in order to complete their computation.

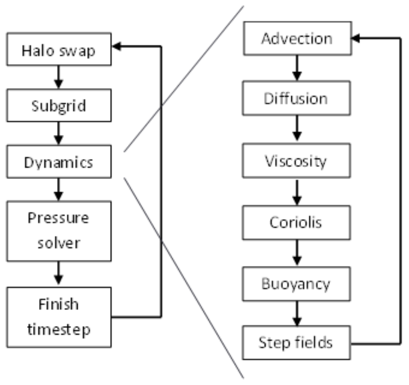

Fig. 1: *High level structure of a single MONC timestep*

In this work we concentrate on the Piacsek and Williams [7] advection scheme which accounts for around 40% of the model runtime. Listing 1.1 illustrates the

MONC Fortran PW advection for advecting the u variable with this scheme. Although this kernel also advects the v and w fields inside the same loop, details for those fields are omitted from Listing 1.1 for brevity as the calculations involved are very similar to those for advecting u. It can be seen that the kernel is composed of three loops, each representing a dimension of our 3D space, and the inner loop, k, in dimension Z loops up a single column. Starting at the second element of the column, this calculates contributions to the *source term* of the flow field, *su*, based upon values held in u, v and w. Later on in the timestep, after the dynamical core has run, the source terms are then integrated into the flow fields. This advection kernel is a stencil based code, of depth one, accessing values of the three flow fields across all the three dimensions. All calculations are performed in double precision.

```
do x=1, x_size
  do j=1, y_size
    do k=2, z_size
      su(k, y, x) = tcx * (u(k,y,x-1) * (u(k,y,x) + u(k,y,x-1)) - u(k,y,x+1) * (u(k,
            y,x) + u(k,y,x+1)))

      su(k, y, x) = su(k, y, x) + tcy * (u(k,y-1,x) * (v(k,y-1,x) + v(k,y-1,x+1))
            - u(k,y+1,x) * (v(k,y,x) + v(k,y,x+1)))

      if (k .lt. z_size) then
        su(k, y, x) = su(k, y, x) + tzc1(k) * u(k-1,y,x) * (w(k-1,y,x) + w(k-1,y,x
              +1)) - tzc2(k) * u(k+1,y,x) * (w(k,y,x) + w(k,y,x+1))
      else
        su(k, y, x) = su(k, y, x) + tzc1(k) * u(k-1,y,x) * (w(k-1,y,x) + w(k-1,y,x
              +1))
      end if
    end do
  end do
end do
```

Listing 1.1: *Illustration of the PW advection scheme for the u field only*

2.2 Hardware setup

For the work described in this paper we are using an ADM8K5 PCI Express card, manufactured by Alpha Data, which mounts a Xilinx Kintex Ultrascale KU115-2 FPGA. This FPGA contains 663,360 LUTs, 5520 DSP48E slices and 4320 BRAM 18K blocks. The card also contains two banks of 8GB DDR4-2400 SDRAM, external to the FPGA, and a number of other interfaces which are not relevant to this work. Because the FPGA used for this work is part of the Xilinx product family, it is their general ecosystem, including tooling, that we use in this work. However, we believe that the lessons learnt apply more generally to product families of FPGAs from other vendors too.

This PCIe card is plugged into an Intel Xeon system, which contains two Sandybridge CPUs, each with four physical cores running at 2.40GHz, and 32GB

RAM (16GB per NUMA region). Our approach is to run MONC on the CPU and study the benefit of offloading the PW advection scheme onto the PCIe mounted FPGA. Not only does this involve performing the double precision calculations for all three fields illustrated in Listing 1.1, but also transferring the necessary flow field data onto, and resulting source terms back from, the card. Whilst some FPGAs such as the Zynq use a more embedded style, where typically ARM cores are combined with FPGA fabric on the same chip, we believe this PCIe setup is more interesting in the field of HPC. There are a number of reasons for this, firstly because a powerful Xeon CPU can be used on the host side, secondly because a large amount of memory can be placed close to the FPGA on the PCIe card to handle the processing of large problems, and thirdly because this is a very common accelerator architecture already adopted in HPC GPU systems.

2.3 FPGA programming techniques and our approach

The traditional approach to programming FPGAs has been to develop codes in a Hardware Description Language (HDL) such as VHDL or Verilog. However, this is a very time consuming process [8] which requires significant expertise and experience. As such, higher level programming tools have been developed to assist, and High Level Synthesis (HLS) is amongst the most prevalent of these. A kernel, written in C, C++ or System C, is automatically translated, via HLS, into the underlying HDL. Driven by pragma style hints provided by the programmer, this substantially speeds up development time and allows for application developers to take advantage of the knowledge and experience of the FPGA vendor. An example of this is in floating point operations, where HLS will automatically include pre-built floating point cores to perform operations, instead of the HDL developer having to develop their own solution.

HLS can be used as a building block of further programming abstractions, and recently the use of OpenCL for programming FPGAs has become popular [9]. Decorating their code via OpenCL abstractions, a tool chain such as Xilinx's SDAccel converts this into a form understandable by HLS, then uses HLS to generate the appropriate HDL and integrates this into a wider design based upon a board specific support package. An alternative approach which, in contrast to OpenCL, requires a bit more work on behalf of the programmer, is the use of the high-level productivity design methodology [10]. In this technique, the FPGA is configured using a block design approach, where existing IP blocks are imported and connected together by the programmer. This emphasises the reuse of existing IP and the idea of a *shell*, providing general foundational functionality that the programmer's kernel, via an IP block generated by HLS, can be dropped into and easily integrated. By separating the shell from the kernel, the general shell infrastructure can be reused for many different applications, and updating the functionality of the kernel, which is quite common during development, often just requires re-importing the IP block into the board design. This approach also eases testing, as the kernel and shell can be validated separately.

In this work we followed the high-level productivity design methodology, where one explicitly writes a C kernel for HLS, generates the HDL and export this as an IP block. This block is then integrated with a shell, providing the general infrastructure. There were two reasons for adopting this approach, firstly because it gave us more control over the configuration of our design, and we think that some of the lessons learnt for HPC codes could then potentially feed into higher level abstractions, such as those provided by OpenCL. Secondly, we chose this approach because SDAccel, the implementation of OpenCL for Xilinx FPGAs, is an extra commercial product that requires further licencing.

There have been a number of previous activities investigating the role that FPGAs can play in accelerating HPC codes. One such example is [11], where the authors investigated using the high-level productivity design methodology to accelerate solving of the Helmholtz equation. They offloaded the matrix-vector updates requires as part of this solver onto a Zynq Ultrascale, however the performance they observed was around half of that when the code was run on a twelve core Broadwell CPU. Crucially, in our work, we are focused on accelerating a much more complicated kernel. In [11] the author's matrix-vector kernel involved looping over two double precision floating point operations, whereas in comparison the kernel we are offloading to the FPGA comprises of fifty three double precision floating point operations, twenty one double precision additions or subtractions, and thirty two double precision multiplications. We are also running on much larger grid sizes, and whereas in [11] the authors were limited to a maximum data size of 17MB due to keeping within the BRAM on the Zynq, in the work detailed in this paper we consider grid sizes resulting in 6.44GB of prognostic field data (and a further 6.44GB for the field source terms), necessitating the use of external SDRAM on the PCIe card.

3 Developing the PW advection HLS kernel

Kernel description	**Runtime (milliseconds)**	**LUT usage**	**DSP48E usage**	**BRAM-18K usage**
Reference on CPU	676.4	NA	NA	NA
Initial port	51498	9743	85	0
Pipeline directive on inner loop	14130	11356	58	64
Local BRAM for column data	3213.2	27598	267	130
Local BRAM batches columns in Y	1513.2	37474	393	453
Extract all variables	1301.6	38393	469	312
Burst mode on port	1097.2	40913	469	324
Re-order X and Y loops	621.3	41151	469	324
Replace memcpy with explicit loops	568.1	40638	466	324
Tune double precision cores and clock to 310Mhz	514.9	27601	406	324

Fig. 2: *Runtime of the PW advection kernel alone for different steps taken when porting it to HLS for a problem size of x=512,y=512,z=64 (16.7 million grid cells)*

Figure 2 illustrates the performance of our HLS PW advection kernel over numerous steps that we applied one after another, for an experiment of x=512, y=512, z=64 (16.7 million grid cells). We use this table to drive our discussions in this section about the different steps required to optimise an HLS kernel and, for reference, the top entry of Figure 2, *Reference on CPU*, illustrates the kernel's runtime over a single Sandybridge CPU core on the host. Our focus in this section is the HLS kernel alone, and as such we ignore the transferring of data which must occur to the PCIe before the kernel runs and the copying back of data once the kernel has completed. This transferring is considered in more detail in the next section.

As a first step we ported the Fortran PW advection kernel for advecting the u, v and w flow fields, (see Listing 1.1) into C and the initial port of this onto the FPGA involved little more than just copying the C into the HLS tool and applying the correct directives to determine the type of external port interfaces. This is illustrated in Listing 1.2, for the double precision array u, representing wind in the X dimension, and source term su which containing advected values calculated by this kernel for the X dimension. There are also arrays for v, sv, w and sw which are omitted from Listing 1.2 due to brevity. In this section we refer to these six arrays as the *kernel's external data arrays*, and these use the AXI4 protocol. The *offset=slave* specifier bundles them into a single port and instructs HLS that these point to different addresses in the control bus port. Other scalar variables such as the size of the data in each dimension, *size_x* and *size_y* in Listing 1.2, are also provided via the AXI4-Lite control bus port. This control bus port is exposed to the host as a memory block, which the host can then write to, and HLS provides the explicit location in this block of the different variables. This initial version was trivial, both from a code and FPGA utilisation perspective, requiring less than 10,000 LUTs, a handful of DSP48E slices and no BRAM, however at 51498 milliseconds (51 seconds) it was very slow.

```
int pw_advection(double * u, double * su, ..., int size_x, int size_y, ...) {
    #pragma HLS INTERFACE m_axi port=u offset=slave
    #pragma HLS INTERFACE m_axi port=su offset=slave

    #pragma HLS INTERFACE s_axilite port=size_x bundle=CTRL_BUS
    #pragma HLS INTERFACE s_axilite port=size_y bundle=CTRL_BUS
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
    .....
}
```

Listing 1.2: *Skeleton of HLS main function, illustrating external data interfaces*

We next added the HLS pipeline directive, *#pragma HLS PIPELINE II=1*, to our inner loop, working up a single column. This instructs HLS to pipeline the processing of the inner loop and *II* is the initiation interval latency, which instructs HLS how often to add a new element of data to the pipeline, in this case every cycle if possible. HLS takes the inner loop, containing our 53 double precision operations, and breaks this up into individual pipeline stages which can run concurrently. These are then fed with data from the outer loops and, once the pipeline is filled, each stage is running, at the same time, on a different element of data before passing the result to the next stage. From Figure 2 it can be seen that this significantly decreased the runtime of the kernel, by around five times, but this was still around twenty time slower than running on the CPU.

The bottleneck at this stage was that the data port, used to implement the kernel's external data arrays, that we read from extensively in the calculation of a single column, does not support more than one access per clock cycle. Hence we were instructing HLS to pipeline the inner loop, so that functionality within it runs concurrently on different elements of data. But crucially HLS realised that there would be numerous conflicts on the data port if it fully pipelined the calculations, and as such very severely limited the design of the pipeline. To address this, we created a number of local arrays to hold all the data required for working with a single column, and this is illustrated in Listing 1.3. We actually created twenty two arrays, for the six kernel external data arrays, three input flow field and three output source term arrays, which included columns in other X and Y dimension indexes. Threes of these local arrays, *u_vals*, *u_xp1_vals* (holding a column of data of the X+1 column) and *u_vals2* are illustrated in Listing 1.3 and these are set to a static size (*MAX_VERTICAL_SIZE*) as array sizes can not be dynamically sized in HLS. The arrays are filled with the data required for a column via the memory copies at lines 5 to 7 (note *u(i,j,0)* is a preprocessor directive that expands out to index the appropriate 3D location in the array), before executing the calculations needed on that column. The idea was that all accesses on the port are before the calculations start and therefore there are no conflicts during the pipelined inner loop. We use on-chip BRAM to store these array, and HLS can make these either single ported or dual ported, meaning that it can either be accessed once or twice independently in a clock cycle. The challenge here is that, for a specific column, there are very many accesses to each array within the inner loop, for instance there are nine accesses to the *u_vals* array which holds the current column's data for the *u* external data array.

```
 1  double u_vals[MAX_VERTICAL_SIZE], u_xp1_vals[MAX_VERTICAL_SIZE], u_vals2
        [MAX_VERTICAL_SIZE], ....;
 2
 3  for (unsigned int i=start_x;i<end_x;i++) {
 4      for (unsigned int j=start_y;j<end_y;j++) {
 5          memcpy(u_vals, &u(i,j,0), sizeof(double) * size_z);
 6          memcpy(u_xp1_vals, &u(i+1,j,0), sizeof(double) * size_z);
 7          memcpy(u_vals2, &u(i,j,0), sizeof(double) * size_z);
 8          ....
 9          for (unsigned int k=1;k<size_z;k++) {
10          #pragma HLS PIPELINE II=1
11              .....
12          }
13      }
14  }
```

Listing 1.3: *Using local BRAM to store data for a single column data*

To address this we duplicated these same arrays, for instance *u_vals2* which holds the same data as *u_vals*. Whilst another way around this in HLS is to use partitioning, effectively splitting the array up across multiple BRAM controllers, due to the dynamic size of the inner loop, we would have been forced to partition the arrays into single elements, and this resulted in worse utilisation and performance. In comparison, duplicating the BRAM array worked well.

It can be seen from Figure 2 that this local copying of data decreased the runtime by over four times. However the major disadvantage of the approach in Listing 1.3 is that the outer loops of j in the Y dimension and i in the X dimension are no longer continually feeding data into the pipelined inner loop. Instead, the inner loop runs in a pipelined fashion just for a single column, in this case of maximum 64 data elements, and then must drain and stop, before memory copies into local arrays are performed for the next column. Bearing in mind the pipeline of this inner loop is, as reported by HLS, 71 cycles deep, with an initiation interval, the best HLS can provide, of 2 cycles and assuming a column size of 64 elements, for each column the pipeline will run for 199 cycles but for only 57 of these cycles (28%) is the pipeline full utilised, the rest of the time it is either filling or draining.

To address this we extended our local BRAM arrays to hold data for multiple columns in the Y dimension, extending each array from *MAX_VERTICAL_SIZE* to *MAX_VERTICAL_SIZE * Y_BATCH_SIZE*. In this situation the middle loop, j, working in the Y dimension runs in batches, of size *Y_BATCH_SIZE*. For each batch it will copy the data for *Y_BATCH_SIZE* columns, and then process each of these columns. The major benefit of this approach is that our pipeline, working up the column in the inner loop, is now fed by *Y_BATCH_SIZE* columns rather than one single column. Additionally, at this point, HLS reported that it had been able to reduce the initiation interval down from two to one, effectively doubling the performance of the inner loop pipeline. Assuming a *Y_BATCH_SIZE* of 64 and that the column size is still 64, the pipeline now runs for 4167 cycles, 97% of which the pipeline is fully filled. This represents a significant increase in utilisation, and ultimately performance, because the pipeline is able to process, and hence generate a result, every clock cycle for 97% of the time it is running. As per Figure 2, this over halved the kernel execution time at the cost of increasing the BRAM usage by over three times.

At this point the individual lines of code for our inner loop kernel, containing the fifty three double precision floating point operations, were still laid out similarly to Listing 1.1, where the calculations for a specific value of the source term were in one line. We had trusted HLS to extract out the individual variable accesses, and structure these appropriately, but we found that actually HLS does a fairly poor job of identifying which variables are shared and hence can be reused between calculations. As such we significantly restructured the code, splitting up calculations into their individual components of reading data into temporary variables and then using that single variable whenever the value is required in the inner loop. This is the *Extract all variables* entry of Figure 2 and had two impacts. Firstly, it reduced the pipeline depth from 71 cycles deep to 65, and hence provided a modest increase in performance, but also it reduced the number of reads on our local arrays and so we were able to remove a number of duplicate local arrays which reduced the overall BRAM usage by around 30%.

When issuing memory copies, for instance in lines 5 to 7 of Listing 1.3, the port must be read which accesses data from external SDRAM, and the same is true in the other direction when writing data. In the default mode, ports will tend to issue an access for every individual element, but instead it is possible to decorate the kernel's external data array variable definitions (e.g. *u* and *su*) with pragmas to instruct HLS to issue bursts of data, retrieving *n* elements in one go. This is important in our situation because we are never just reading one element at a time, but instead the data for *Y_BATCH_SIZE* columns. The *HLS INTERFACE* pragma, as illustrated in Listing 1.2, was modified for our kernel's external data arrays (e.g. *u* and *su*) with the addition of *num_read_outstanding=8 num_write_outstanding=8 max_read_burst_length=256 max_write_burst_length=256*. This directs HLS to read and write in bursts of size *256*, the maximum size, and supports holding up to eight of these bursts at any one time. The *latency* modifier advises HLS to issue the access before it is needed, in this example around 60 cycles beforehand. HLS uses BRAM to store these bursts and, as can be seen in Figure 2, resulted in a modest increase in BRAM usage but also a reasonable decrease in execution time.

At this point we are working in batches of columns and as our middle, *i*, loop running over the Y dimension, reaches the limit of one batch it stops and retrieves data from memory for the next batch. Crucially this happens for every iteration in the outer loop, *i*, over the X dimension and as the code progresses from one level in X to the next, then all batches in Y are run again. The problem with this is that there are fifteen memory copies required for every batch and this involves significant amounts of time accessing the DRAM. It is possible to address this by moving the outer loop, *i*, over the X dimension, inside the *y* middle loop which is running over a single batch of columns. This means that memory accesses themselves in the X dimension are effectively pipelined too, and is illustrated by Listing 1.4. For brevity we just show a subset of the variables and local arrays, but it is enough to demonstrate the approach. It can be seen that the loop ordering has changed, such that the outer loop is now looping *m* times, once per batch of *MAX_Y_SIZE* columns at line 1. The start of a batch requires data to be copied into the *up1_vals* variable, representing the column plus one in the X dimension. Then, as the loop progresses through levels in the X dimension, for each next level, *u_vals* is populated with data from *up1_vals* and only the plus one level in the X dimension, i.e. *up1_vals* needs to go to SDRAM memory to retrieve the *i+1* column in X. All other copies, for instance *u_vals* at line 5, are accessing chip local BRAM which is much faster than going off chip to the SDRAM. This significantly reduces the number of off chip data accesses to DRAM that need to be performed and almost halves the runtime of the kernel.

```
1 for (unsigned int m=start_y;m<end_y;m+=MAX_Y_SIZE) {
2     memcpy(up1_vals, &u(start_x,m,0), sizeof(double) * MAX_VERTICAL_SIZE*
          MAX_Y_SIZE);
3     ....
4     for (unsigned int i=start_x;i<end_x;i++) {
5         memcpy(u_vals, up1_vals, sizeof(double) * MAX_VERTICAL_SIZE*
              MAX_Y_SIZE);
```

```
        memcpy(up1_vals, &u(i+1,m,0), sizeof(double) * MAX_VERTICAL_SIZE*
            MAX_Y_SIZE);
        ....
        for (unsigned int j=0;j<MAX_Y_SIZE;j++) {
            ....
        }
    }
}
```

Listing 1.4: *Reordering the X and Y loops to pipeline memory access in the X dimension*

Until this point we have relied on the use of the *memcpy* function to copy data from one location to another. However bearing in mind there are multiple copies of local column data arrays due to BRAM port limits, e.g. *u_vals* and *u_vals2*, issuing a separate memcpy for each of these when we loop into the next X dimension is quite slow because HLS will not execute these memory copies concurrently. Instead, replacing the *memcpy* calls with explicit loops, where each index location is read from the source array and then written to each of the target arrays was faster. In fact, more generally we found that replacing all the *memcpy* calls with an explicitly pipelined loop that performed the copying in user code, was beneficial. This is represented as the *Replace memcpy with explicit loops* entry of Figure 2 and it can be seen that not only did we obtain a modest increase in performance, but it also decreased our LUT utilisation slightly.

The default clock on the ADM8K5 board is 250Mhz, and so a period of 4ns was used initially, with HLS estimating a clock period of 3.75ns due to limits in double precision multiplication. However, via configuring the HLS floating point kernels it was possible to tune them. Using *#pragma HLS RESOURCE variable=a core=DMul_maxdsp latency=14*, HLS was instructed to use the *DMul_maxdsp* double precision floating point core (leveraging DSP slices as much as possible for the double precision multiplication) with a latency of 14 cycles for all multiplications involving the variable *a*. This latency is the core's pipeline depth and, by increasing it, it is possible to reduce the minimum clock period. We applied this directive to all variables that are involved in double precision multiplication, and found that the best clock period we could get from the double precision multiplication core was 2.75ns. Whilst the latency value can go all the way up to twenty, above fourteen made no difference to the period. As such we were able to reduce our clock period to 3.2 (there is a 12.5% clock uncertainty), meaning we could run our kernel at 310Mhz instead of 250Mhz. The pipeline depth has increased from 65 to 72, but due to the increase in clock frequency, the overall latency for data to progress through the pipeline has gone from 2.6e-7 seconds to 2.3e-7 seconds, so there is an increase in overall performance.

From Figure 2 it can be seen that the LUT and DSP48E utilisation dropped very significantly with this last configuration. This was because we also instructed HLS to use the full DSP core when it came to double precision addition and subtraction. It is the use of this core that reduced the LUT usage by around a quarter, and also, ironically, slightly reduced the number of DSP48E slices too.

As a result of the steps applied in this section, we have reduced the runtime of our HLS kernel by over 100 times, from being 75 times slower than running over a single Sandybridge CPU core on the host, to being around a quarter faster, however as noted at the start of the section this is just the kernel execution time and ignores the DMA transfer time needed to get data on and off the board before and after the the kernel runs.

4 Putting it all together, the block design

Once developed, we then need to integrate our PW advection kernel with general infrastructure to connect our kernel to the PCIe interface and on card SDRAM. The general workflow is that field data is transferred from the host to the on card SDRAM via DMA and kernels are then run, reading data from the SDRAM and writing source term results to SDRAM, and once complete results are transferred back from the SDRAM to the host via DMA. Figure 3 illustrates the block design of our system, and in this design we are using four PW advection kernels, towards the top centre of Figure 3 with the HLS logos. The IP block on the bottom left is the PCIe interface, providing four independent DMA channels that can be used for communication and a direct slave interface that can also be used for communication. The two big blocks on the far right are memory controllers for the on card SDRAM, each controller responsible for one of the two banks of on card 8GB memory. We connect the first two PCIe interface DMA channels to the first memory controller, and the other two DMA channels to the second memory controller. In between the PCIe interface and their corresponding SDRAM memory controller, these connections pass through infrastructure which, for instance, converts the clock from the PCIe clock domain to the SDRAM memory controller clock domain. In this design, to avoid bottlenecks, the two banks of 8GB memory are entirely separate and it is not possible for an IP block connected to one bank to access memory of the other bank.

Figure 4 provides a more detailed view of the integration of our PW advection kernel IP blocks. For purposes of illustration, we have slightly moved the appropriate IP blocks around, when compared to Figure 3, so the topical ones are in the same image. On the bottom left of Figure 4, the *ADM PCIe* block is the PCIe interface and the four DMA channel ports on the right of this IP block, along with the direct slave port can be clearly seen. The clocking wizard, top left, converts the 250Mhz reference clock up to 310Mhz for the PW advection IP blocks as described in Section 3. The direct slave interface is connected to the kernel's control port, and by writing or reading the appropriate bit we can manage the kernels such as starting or tracking progress. This connection goes, via an AXI4 clock converter IP block, to the slave interface of the left most AXI interconnect. This interconnect splits the direct slave data according to its address, the appropriate data then routed to its corresponding PW advection kernel. These addresses are defined in the block design address editor.

The main data port, *m_axi_gmem*, of the PW advection kernel is on the right of the PW advection IP block and it is through this port that the kernel's

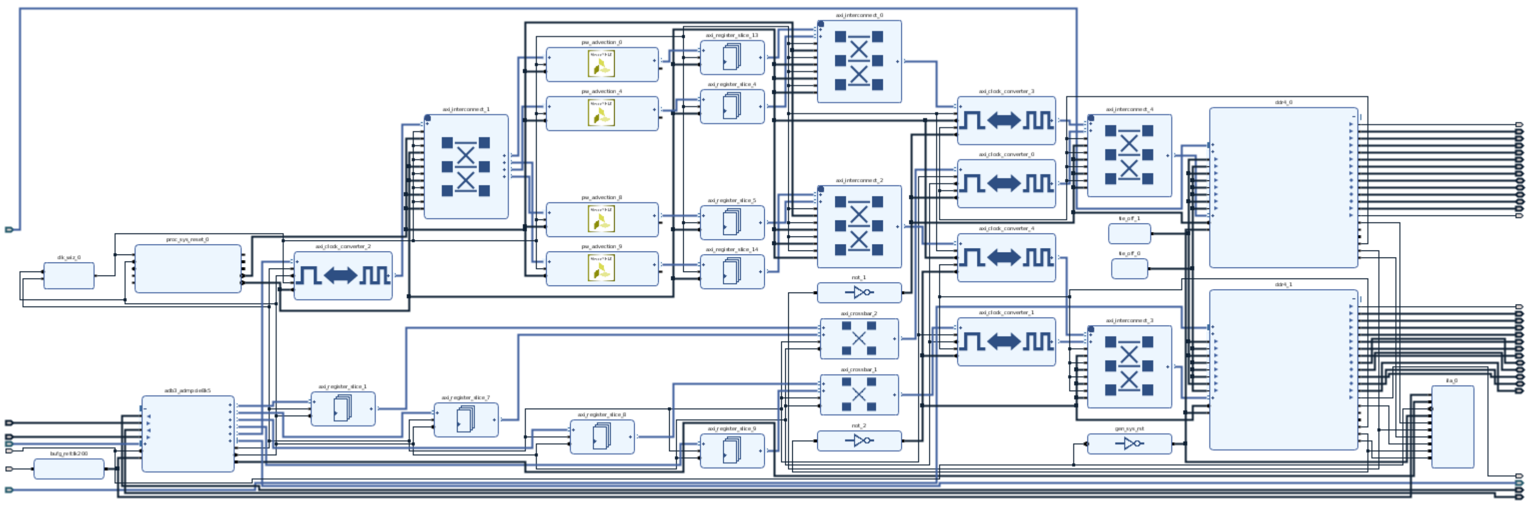

Fig. 3: *The MONC PW advection board design, containing four of our PW advection HLS kernels and other general infrastructure to support this*

external data arrays such as u and su are routed. This AXI4 data port connects, via an AXI register slice, to an AXI interconnect on the right of Figure 4. Our PW advection kernels are split into two groups, one group connecting to the first SDRAM memory controller (one bank of 8GB RAM) and the second group connecting to the second DDR SDRAM memory controller (the second bank of 8GB RAM). This is the reason for the two AXI interconnects on the right, one for each group of kernels and we do it this way with the aim of reducing congestion. For purposes of illustration, in Figures 3 and 4 we limited ourselves to just four PW advection IP blocks. However our design is scalable and adding additional PW advection kernels just requires reconfiguration of the appropriate AXI interconnects to add more ports and assigning an address to the new IP block's control bus port.

When it came to the shell and overall kernel integration, we experimented with a number of different designs to understand which would provide the best performance. The main driver here is the speed to access the SDRAM and avoid memory accesses becoming a bottleneck. We found this design, where we keep the two banks of 8GB SDRAM entirely separate and connect each to two different DMA channels, to provide the best performance. A summary of these investigations is illustrated in Figure 5, which describes the transfer time, via DMA, to copy 1.6GB of data from the host to the DRAM on the PCIe card. The first row of Figure 5 is the design that we have described in this section. The second row, *one memory controller only*, is when all four DMA channels are used but we only copy the data into one of the memory controllers, and it can be seen that this increases the transfer time by around a fifth. The third entry is where there are two memory controllers, each serviced by two DMA channels, but these are connected together in one large memory space so any memory access can see all the 16GB SDRAM. This is slightly slower than our split approach, because all memory accesses go through a single AXI interconnect, but

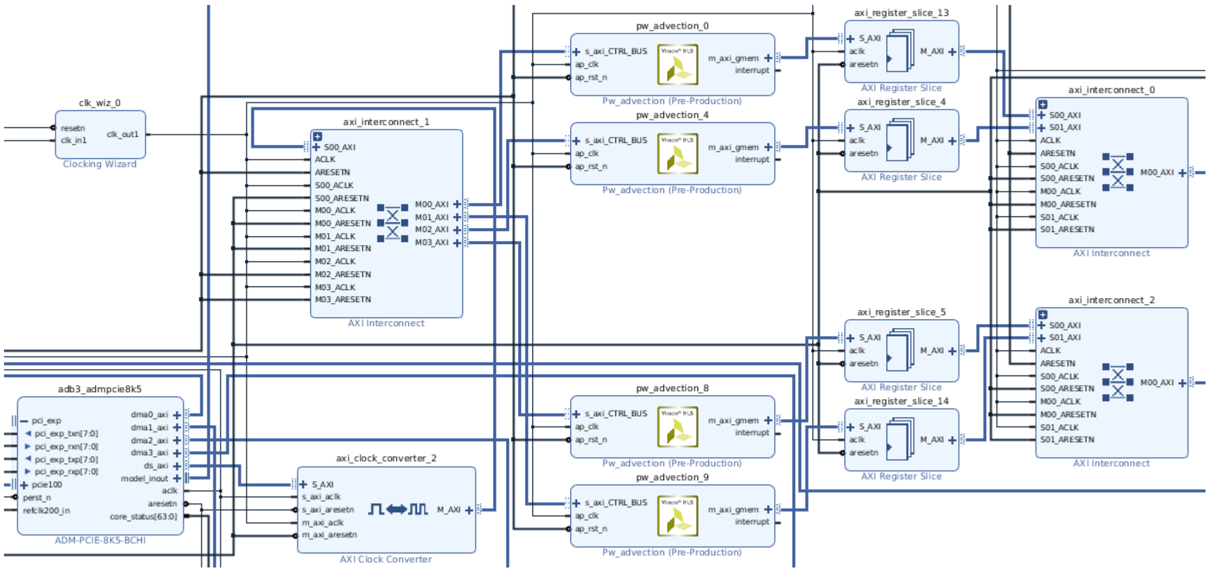

Fig. 4: *Integration of the PW advection HLS IP blocks in our board design*

there isn't much in it. The last entry of the table, is where the two banks of memory is kept separate, but we only drive these via one single DMA channel. This adds around a third to the overall DMA transfer time because transfers on the same channel need to queue up and so being able to spread them out across multiple channels and transfer concurrently is optimal.

Description	**DMA transfer time (milliseconds)**
Design described here	232
One memory controller only	280
Two memory controllers connected	239
One DMA channel per memory controller	342

Fig. 5: *DMA transfer time for different configurations with a data size of 1.6GB*

5 Performance comparison

We built the block design described in Section 4 with twelve PW advection kernels as described in Section 3. In order to fit within the limits of the Kintex FPGA we instructed HLS to use the medium DSP core for double precision multiplication. This resulted in an overall design utilisation of 78.5% of the Kintex's LUTs, 84.2% BRAM-18k blocks and 89% of the chip's DSP48E slices. It took around fifteen hours of CPU time to build the entire design, most of which was spent in the place and route phases.

Once built, we compared the performance of our PW advection FPGA design against a C version of the same PW advection algorithm, threaded via OpenMP across the cores of the CPU. For all runs the host code was compiled with GCC version 4.8 at optimisation level 3 and the results reported are averaged across fifty timesteps. Figure 6 illustrates a performance comparison of our FPGA kernel against the CPU only code running on Sandybridge, Ivybridge and Broadwell CPUs for a standard MONC stratus cloud test case of size x=1012, y=1024, z=64 (67 million grid cells). For each technology there are two runtime numbers, in milliseconds. The first, *optimal performance*, illustrates the best performance we can get on the CPU by threading over all the physical cores (4 in the case of Sandybridge, 12 in the case of Ivybridge, 18 in the case of Broadwell, and 12 in the case of our FPGA design.) We also report a four core number, which includes only running over four physical cores, or PW advection kernels in the case of our FPGA design, as this is the limit of the Sandybridge CPU and it allows more direct comparison.

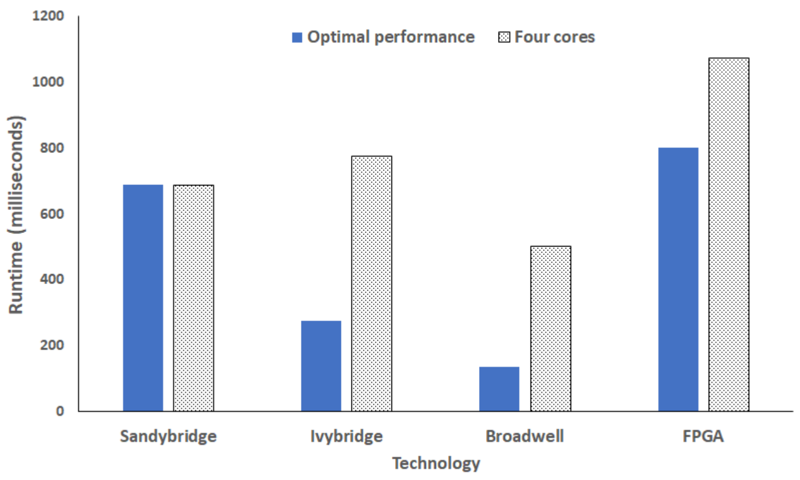

Fig. 6: *Performance comparison of x=1012, y=1024, z=64 (67 million grid points) with a standard status cloud test-case*

It can be seen from Figure 6 that our optimal FPGA version performs slower in comparison to all CPU products tested and this trend continues when we consider four core performance. This might seem strange seeing that, in Section 3, our HLS kernel is faster than running on a single core of Sandybridge and actually is comparable to running on a single core of Broadwell. However, crucially there we were just concerned with the kernel execution time and ignored the DMA transfer time and the results reported in Figure 6 contain both these aspects.

To understand this further, Figure 7 illustrates the same experiment setup as we scale the number of PW advection kernels from one to twelve. For each configuration we report the total runtime and then break it down to the total time required for DMA transfer of data both to and from the card, and the kernel execution time. In all cases we distribute the cores as evenly as possible across groups, for instance two cores uses 1 core from each group, four cores uses two cores from each group. This experiment represents a grid size of 67 million points, and three fields, each point of which is double precision, hence a total of 3.32GB is being transferred. It can be seen that, for small numbers of advection kernels, the kernel runtime is by far the most significant. However this scales well and as we reach four kernels and beyond the DMA transfer time becomes dominant and 70% of the total time with twelve kernels is in DMA transfer.

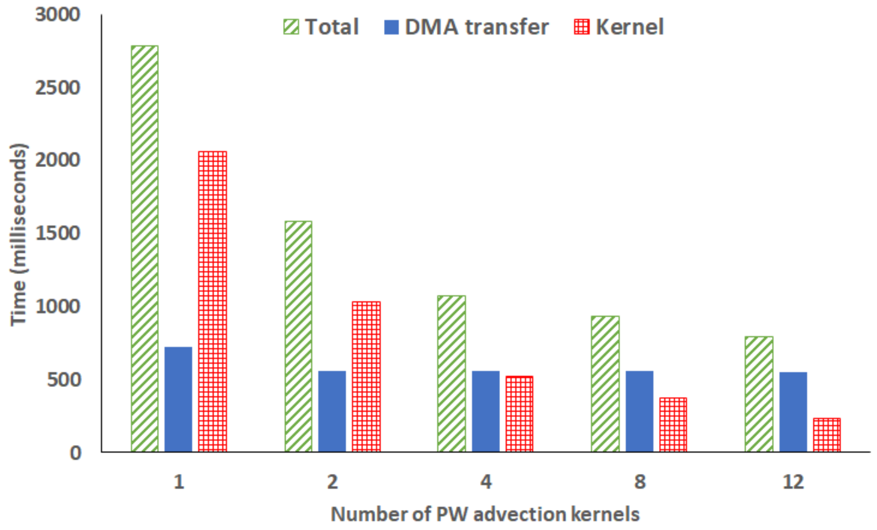

Fig. 7: *Runtime of different numbers of PW advection kernel broken down into constituent parts when scaling number of kernels, x=1012, y=1024, z=64 (67 million grid cells) with a standard status cloud test-case*

Figure 8 illustrates how the time, in milliseconds, changes as we scale the number of grid cells, note that this is a log scale. We report three numbers for our FPGA approach (12 PW advection kernels), the total time, the execution time of the kernel and the total DMA transfer time. For comparison we also illustrate the runtime of the code on 18 physical cores of Broadwell and 12 physical cores of Broadwell (as we have 12 PW advection kernels). Whilst our PW advection FPGA version is slower than both Broadwell configurations, the FPGA HLS kernel itself is faster at 1 million grid cells, competitive with both at 4 million grid cells and competitive with the 12 Broadwell cores until 16 million grid cells. In terms of FLOPs, at 268 million grid cells our HLS kernel is providing 14.36 GFLOP/s (in comparison to 12 cores of Broadwell at 17.75 GFLOPs/), however when one includes the DMA transfer time this drops down to 4.2 GFLOP/s and so illustrates the very significant impact that DMA transfer time has on our results. The limit with some other investigations such as [11], is that they focus on the embedded CPU-FPGA Zynq chip, and limit their system size very severely to the BRAM on that chip. As such they don't encounter this transfer time overhead, but this is crucially important to bear in mind for processing realistic problems that are of interest to scientists.

6 Conclusions and further work

In this paper we have described our approach in porting the PW advection kernel of the MONC atmospheric model onto FPGAs. Using HLS, we explored in

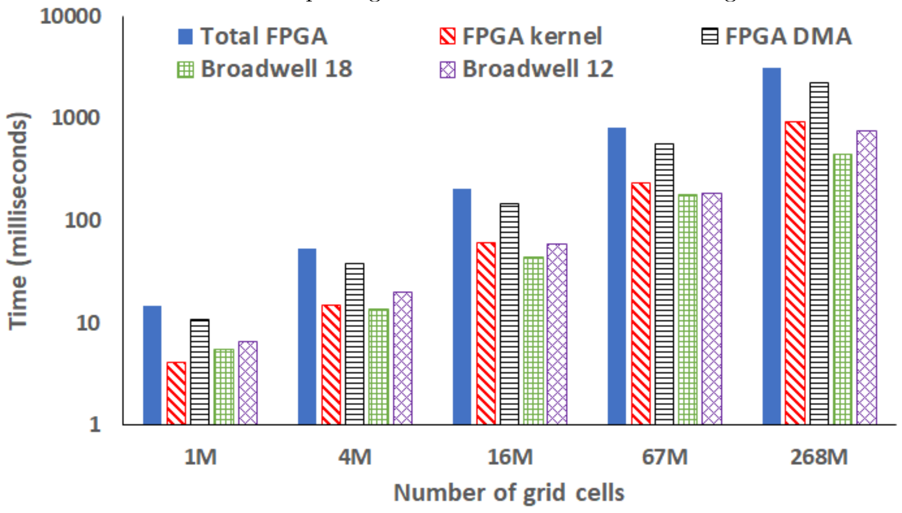

Fig. 8: *Runtime of our FPGA PW advection code (12 kernels) vs Broadwell as we scale the grid size with a standard stratus cloud test-case*

detail the different steps required in porting the kernel and how best to structure the board design. We have shown that it is crucial that HPC application developers rethink and recast their code to suit the data-flow pattern, and have demonstrated a 100 times performance difference between code that does not do this and the same functionality, albeit where the code looks very different, tuned for the FPGA. This re-enforces the point that, whilst it is fairly simple for an HPC applications developer to import some C code into HLS and synthesis this, significant work and experimentation is required to get reasonable performance.

When considering only the kernel execution time, our HLS kernel outperformed a single core of Sandbridge and performs comparable with a single Broadwell core. But when including the DMA transfer time we found that this is a very severe limitation of our FPGA design in contrast to the performance one can gain from the same advection kernel threaded over the cores of Intel CPUs commonly found in HPC machines.

When it comes to further work, it is this DMA transfer time that needs to be tackled. At the largest problem size of 268 million grid cells explored in this paper, a total of 12.88GB needs to be transferred which takes 2.2 seconds and represents a transfer rate of 5.85 GB/s, which is reasonable based on the specifications of this PCIe card. One idea is to use single rather than double precision, which would effectively halve the amount of data that needs to be transferred, although the DMA transfer time at 134 million grid points is still substantially slower than Broadwell execution time at 268 million grid points. Another idea is to chunk up the DMA transfer, starting the appropriate PW advection kernel as soon as the applicable chunk has arrived rather than all the data, this could be driven by a host thread or extra logic in the block design.

There is also further exploration that can be done based on our HLS kernel and one aspect would be to look at how best to further increase the clock speed. The 2.8ns period of the double precision multiply currently limits us to 310Mhz, but replacing double precision with single or fixed point could enable higher clock frequencies and single precision would also halve the amount of data transferred from SDRAM to the kernels. This last point is important, because we believe that SDRAM access time is now the major source of overhead in our HLS kernel.

Therefore, we conclude that, whilst FPGAs are an interesting and generally speaking viable technology for accelerating HPC kernels, there is still work to be done to obtain performance competitive to modern day Intel CPUs. When running large system sizes on PCIe mounted FPGAs, the cost of transferring data to and from the card can be very significant and severely limits performance. This is important to bear in mind and, going forwards, if we can address these limits then we feel that FPGAs will become a much more competitive approach.

7 Acknowledgements

The authors would like to thank Alpha Data for the donation of the ADM8K5 PCIe card used throughout the experiments of work.

References

1. Brown, N., et al., 2015, April. A highly scalable Met Office NERC Cloud model. In Proceedings of the 3rd International Conference on Exascale Applications and Software (pp. 132-137). University of Edinburgh.
2. Brown, A.R., et al., 1997, Large-eddy simulation on a parallel computer. In Proceedings Turbulence and diffusion (240)
3. Lock, A.P., 1998. The parametrization of entrainment in cloudy boundary layers. Quarterly Journal of the Royal Meteorological Society, 124(552), pp.2729-2753.
4. Petch, J.C. and Gray, M.E.B., 2001. Sensitivity studies using a cloudresolving model simulation of the tropical west Pacific. Quarterly Journal of the Royal Meteorological Society, 127(577), pp.2287-2306.
5. Hill, A.A., et al., 2014. Mixedphase clouds in a turbulent environment. Part 1: Largeeddy simulation experiments. Quarterly Journal of the Royal Meteorological Society, 140(680), pp.855-869.
6. Ma, Y., et al., 2017, February. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 45-54). ACM.
7. Piacsek, S.A. and Williams, G.P., 1970. Conservation properties of convection difference schemes. Journal of Computational Physics, 6(3), pp.392-405.
8. Maxfield, C., 2004. The design warrior's guide to FPGAs: devices, tools and flows. Elsevier.
9. Muslim, F.B., et al., 2017. Efficient FPGA implementation of OpenCL high-performance computing applications via HLS. IEEE Access, 5, pp.2747-2762.
10. Xilinx, 2018, UltraFast High-Level Productivity Design Methodology Guide [Online]. [11 April 2019]. Available from: https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf
11. Ashworth, et al., First steps in Porting the LFRic Weather and Climate Model to the FPGAs of the EuroExa Architecture.