First steps in Porting the LFRic Weather and Climate Model to the FPGAs of the EuroExa Architecture

Mike Ashworth
School of Computer Science
University of Manchester
Manchester, M13 9PL, UK
mike.ashworth.compsci@manchester.ac.uk

Andrew Attwood
School of Computer Science
University of Manchester
Manchester, M13 9PL, UK
andrew.attwood@manchester.ac.uk

Graham Riley
School of Computer Science
University of Manchester
Manchester, M13 9PL, UK
graham.riley@manchester.ac.uk

John Mawer
School of Computer Science
University of Manchester
Manchester, M13 9PL, UK
john.mawer@manchester.ac.uk

***Abstract*— The EuroExa project proposes a High-Performance Computing (HPC) architecture which is both scalable to Exascale performance levels and delivers world-leading power efficiency. This is achieved through the use of low-power ARM processors accelerated by closely-coupled FPGA programmable components. In order to demonstrate the efficacy of the design, the EuroExa project includes application porting work across a rich set of applications. One such application is the new weather and climate model, LFRic (named in honour of Lewis Fry Richardson), which is being developed by the UK Met Office and its partners for operational deployment in the middle of the next decade.**

Much of the run-time of the LFRic model consists of compute intensive operations which are suitable for acceleration using FPGAs. Programming methods for such high-performance numerical workloads are still immature for FPGAs compared with traditional HPC architectures. The paper describes the porting of a matrix-vector kernel using the Xilinx Vivado toolset, including High-Level Synthesis (HLS), discusses the benefits of a range of optimizations and reports performance achieved on the Xilinx UltraScale+ SoC.

Performance results are reported for the FPGA code and compared with single socket OpenMP performance on an Intel Broadwell CPU. We find the performance of the FPGA to be competitive when taking into account price and power consumption.

Keywords—Exascale, FPGA, matrix-vector multiplication, weather modeling

I. INTRODUCTION

Many fields in science and engineering which use High Performance Computing (HPC) to obtain high levels of compute performance for simulation and modelling have identified a need to progress towards Exascale levels of performance (order of 1018 floating point operations per second). Achieving Exascale with CPU-based technologies is technically feasible but would result in unacceptable power requirements. Therefore there has been considerable interest in recent years in novel architecture systems which harness accelerators alongside CPUs to maintain and/or boost performance while delivering substantially improved power efficiency.

Field-Programmable Gate Arrays (FPGA) have long been considered as candidate accelerators offering the potential for orders of magnitude improvements in performance per watt and per unit cost over existing technologies. Until recently, difficulties in programming scientific codes on FPGAs have limited their uptake for scientific computing, but advances in high-level programming environments have brought about a resurgence in interest. These advances have primarily been targeting applications in data centres, where there has been a surge in the deployment of FPGAs, e.g. at Microsoft [1]. A range of high-level tools, known as High-Level Synthesis, provide programming environments for porting scientific codes from C, C++ or OpenCL [2].

The EuroExa, project proposes an HPC architecture which is both scalable to exascale performance levels and delivers world-leading power efficiency. This is achieved through the use of low-power ARM processors together with closely coupled FPGA programmable components. The project uses a cost-efficient, modular integration approach enabled by novel inter-die links, with FPGAs to leverage data-flow acceleration for compute, networking and storage, an intelligent memory compression technology, a unique geographically-addressed switching interconnect and a novel ARM-based compute unit. The software platform provides advanced runtime capabilities supporting novel parallel programming paradigms, dataflow programming, heterogeneous acceleration and scalable shared memory access.

In order to demonstrate the efficacy of the design, the EuroExa partners support a rich set of applications. One such application is the new weather and climate model, LFRic (named in honour of Lewis Fry Richardson), which is being developed by the Met Office and its partners for operational deployment in the middle of the next decade. High quality forecasting of the weather on global, regional and local scales is of great importance to a wide range of human activities and exploitation of latest developments in HPC has always been of critical importance to the weather forecasting community.

EuroExa has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement 754337.

The first EuroExa system is being built now and is due for first application access in late-2018. In order to prepare for this we have been porting the LFRic model to a Zynq UltraScale+ ZCU102 Evaluation Platform [3]. The initial approach is to study the LFRic code at three levels: the full application (Fortran), compact applications or "mini-apps" and key computational kernels.

An example of a kernel is the matrix-vector product which contributes significantly to the execution time in the Helmholtz solver and elsewhere. We use Vivado High Level Synthesis (HLS) to generate IP blocks to run as part of a Vivado design on the UltraScale+ FPGA.

II. LFRic Weather and Climate Model

A. Overall Description

LFRic is a new weather and climate model, which is being developed by the UK Met Office and its partners for operational deployment in the early part of the next decade [4]. High quality forecasting of the weather on global, regional and local scales is of great importance to a wide range of human activities. The model supports simulations for both weather and climate scenarios and the climate research community in the UK is also closely involved in the model's development.

The current operational model at the Met Office, the Unified Model, uses a latitude-longitude grid in which lines of longitude converge at the poles leading to problems in performance and scalability, especially on modern highly parallel HPC systems. In a pre-cursor project between the Met Office, the Natural Environment Research Council and the Science and Technology Facilities Council, called GungHo, a new dynamical core was developed using the cube-sphere grid which covers the globe in a uniform way. The GungHo code has also been developed specifically to maintain performance at high and low resolution and for high and low CPU core counts. A key technology to achieve this is Separation of Concerns, in which the science code is separated from the parallel, performance-related code and the PSyclone code generation tool is used to automatically generate code targeting different computer architectures. The LFRic weather model is based on the GungHo dynamical core with its PSyclone software technology.

B. LFRic Profile and Call Graph

LFRic can be run in many configurations representing a range of weather and climate scenarios at low-, medium- and high-resolutions. In order to characterise the performance we ran and profiled a baroclinic test case, which has been developed by the Met Office as a part of their performance evaluation procedure. Profiling was carried out on the Met Office collaboration system, a Cray XC40, running on a single core. By running with gprof and piping the output first into gprof2dot.py and thence into dot, we produced the call graph complete with the amount of CPU time taken at each node in the graph.

Most of the CPU time is spent in the Helmholtz solver that is used to compute the pressure. Two leaf nodes, corresponding to the subroutines opt_apply_variable_hx-_code and opt_scaled_matrix_vector_code, account for greater than 50% of the CPU time. Both of these subroutines spend most of their time performing double-precision matrix-vector multiplication within an outer loop which runs over the vertical levels within the atmosphere. Therefore as a first step to porting LFRic, we are focussing on running a matrix-vector multiply kernel on the FPGA, using data dumped from a real LFRic execution.

We note that the use of FPGAs as accelerators offers considerable scope for improved performance using reduced precision. Most current weather models use double precision throughout, but there is active research in the use of reduced precision for some parts of the computations, see e.g. [5]. For this work we have carried out most of the development using double precision (64-bit double type), but also measured performance for single precision (32-bit float type).

C. Matrix-vector Updates in LFRic

The matrix-vector updates have been extracted from the full LFRic Fortran code into a kernel test program. There are dependencies between some of the updates and a colouring scheme is used in LFRic, such that nodes within a single 'colour' have no dependencies and can be computed simultaneously. This is used to produce independent computations for multi-threading with OpenMP and can be exploited for the FPGA acceleration as well. As with all accelerator-based solutions, a key optimization strategy is to minimize the overhead of transferring data between the CPU and the FPGA.

The test grid is a very coarse representation of the globe (12km resolution) consisting of 864 finite-element cells in the horizontal, extruded into vertical columns with 40 vertical levels. As a part of the Helmholtz solver for determining the pressure, the code performs a matrix-vector update on each cell and for each vertical level. The size of the matrix is 8x6; this can increase if higher order methods are employed. Each update therefore consists of an x-vector of 6 elements, a matrix of 8x6 elements, and an output or left-hand-side (lhs) vector of 8 elements. The lhs vector is updated, so appears as input and output, though the current FPGA implementation only does the matrix-vector product, leaving the update to be performed on the ARM CPU.

For each update there is therefore (8+6+48)*864*40*8B = 17 MB data. Using BRAMs on the FPGA means, with the current ZU9 chip, that we are limited to around 3.5 MB of space. The FPGA designs in section 3.4 use BRAMs of size 256 kB so that twelve such BRAMs use 3MB. The ARM code therefore blocks the data for each matrix-vector block so that it fits within its 256 kB of memory.

III. Porting and optimization for FPGA acceleration

A. Development platform and environment

In preparation for access to the EuroExa TestBed systems we have been using a XilinX ZynQ UltraScale+ ZCU102 evaluation platform. At its heart this board contains an MPSoC comprising, amongst other processors, an ARM Cortex A53 quad-core CPU running at 1.2 GHz and a ZynQ UltraScale XCZU9EG-2FFVB1156 FPGA. The FPGA contains some 600k logic cells, 2,520 DSP slices and around 3.5 MB of BRAM memory. The ARM CPU is running Ubuntu 14.04.5 and we are using Vivado Design Suite and Vivado HLS versions 2017.4 to generate IP blocks and bitstreams for the FPGA.

B. Starting code

LFRic is written in Fortran making use of many features from the latest Fortran standards: Fortran 2003, 2008 and 2018. Vivado HLS does not accept Fortran code so the matrix-vector kernel was translated into C. In order to generate a layout for the FPGA, HLS needs to know all data sizes at compile-time so the matrix sizes and loop lengths have been hard-coded using #define statements. The starting code is shown in Fig. 1.

```
#define NDF1 8
#define NDF2 6
#define NK 40
#define MVTYPE double
int matvec_8x6x40_vanilla
   (MVTYPE matrix[NK][NDF2][NDF1],
    MVTYPE x[NDF2][NK], MVTYPE lhs[NDF1][NK]) {
  int df,j,k;
  for (k=0;k<NK;k++) {
     for (df=0;df<NDF1;df++) {
     lhs[df][k] = 0.0;
     for (j=0;j<NDF2;j++) {
     lhs[df][k]=lhs[df][k]
         +x[j][k]*matrix[k][j][df];
     }

    }
  }
  return 0;
}
```

Fig. 1. Starting code for a matrix-vector multiply for NK vertical levels translated into C for entry into Vivado HLS.

C. Optimized code in Vivado HLS

When running the “C synthesis” process, Vivado HLS reports the expected performance in terms of the "Task Latency", the time from start to finish of the task, and the "Task Interval", the time between the start times of two consecutive tasks, both measured in clock cycles. One input parameter to be set in HLS is the target clock cycle, and HLS also reports the estimated clock cycle based upon timings achieved in the synthesized design, thus it is always possible to convert performance in clock cycles into real timings. For this code, the interval is always the same as the latency. There are also messages sent to the console window providing information on optimizations which have or have not been performed, reasons for lack of success in optimization, and the location of the critical path through the code.

Using this feedback from HLS is it possible to optimise the code without executing it, achieving a substantial reduction in the reported latency. The following optimizations were carried out:

- loops were swapped to make the k-index over the vertical levels the innermost loop;
- data arrays were transposed where necessary to ensure that data running over the k-index were sequential in memory; together with the above this ensures a sequential innermost loop of length 40 elements;
- the HLS UNROLL pragma was applied to the innermost loops; unrolling by hand was also tried but shown to result in no additional benefit;
- the HLS PIPELINE pragma was applied to the outermost loop;
- the code just computes the matrix-vector product, without updating the left-hand side array (lhs); the update can be performed on the ARM CPU as a low-cost addition when copying the result data;
- HLS INTERFACE pragmas were added to define the interfaces for the subprogram arguments; in particular the clauses num_read_outstanding=8, max_read_burst_length=64, num_write_outstanding=8, max_write_burst_length=64, were used; and
- data read from and written to the subprogram arguments were copied into and copied out from local working arrays using memcpy.

The resulting code is shown in Fig. 2. The last two optimizations were particularly important. Without them data reads are not streamed, with each word being read independently as though the block is waiting for one read to complete before starting the next. With the optimizations, data is streaming at one word per cycle.

The HLS INTERFACE pragmas in the code define the ports which will be available at the matrix-vector IP block for interconnection in the subsequent Vivado design.

```
#define NDF1 8
#define NDF2 6
#define NK 40
#define MVTYPE double
#include <string.h>
int matvec_8x6x40_v6 (
    const MVTYPE *matrix, const MVTYPE *x,
    MVTYPE *lhs){
#pragma HLS INTERFACE m_axi depth=128 \
    port=matrix offset=slave bundle=bram \
    num_read_outstanding=8 \
    num_write_outstanding=8 \
    max_read_burst_length=64 \
    max_write_burst_length=64
#pragma HLS INTERFACE m_axi depth=128 \
    port=x offset=slave bundle=bram \
    num_read_outstanding=8 \
    num_write_outstanding=8 \
    max_read_burst_length=64\
    max_write_burst_length=64
#pragma HLS INTERFACE m_axi depth=128 \
    port=lhs offset=slave bundle=bram \
    num_read_outstanding=8 \
    num_write_outstanding=8 \
    max_read_burst_length=64 \
    max_write_burst_length=64
#pragma HLS INTERFACE s_axilite port=return
  int df,j,k;
  MVTYPE ml[NDF2][NK], xl[NDF2][NK], ll[NK];
  memcpy (xl, x, NDF2*NK*sizeof(MVTYPE));
  for (df=0;df<NDF1;df++) {
#pragma HLS PIPELINE
    memcpy (ml, matrix+df*NDF2*NK,
      NDF2*NK*sizeof(MVTYPE));
    for (k=0;k<NK;k++) {
#pragma HLS UNROLL
      ll[k] = 0.0;
    }
    for (j=0;j<NDF2;j++) {
      for (k=0;k<NK;k++) {
#pragma HLS UNROLL
        ll[k] = ll[k]+ xl[j][k]*ml[j][k];
      }
    }
    memcpy (lhs+df*NK, ll, NK*sizeof(MVTYPE));
  }
  return 0;
}
```

Fig. 2. Optimized code for a matrix-vector multiply for NK vertical levels.

The three array arguments are specified as AXI[1] master ports (m_axi) and are bundled together into a single port using the bundle option. This helps to simplify the interconnection network. Specifying the return argument with an interface of s_axilite produces a slave port with AXILite protocol, which is the connection for the "registers" used to control the block (see section D).

Using the starting code in Fig. 1, Vivado HLS reports a latency of 69,841 clock cycles. The optimized code produces a latency of just 2,334 cycles; an improvement of a factor of 30. This is with a target clock cycle of 2 ns, i.e. a clock rate of 500 MHz, for which HLS reports an estimated clock of 2.89 ns, corresponding to an estimated maximum frequency of 346 MHz. For this data case, with a matrix of size 8x6 and 40 vertical levels, the matrix-vector kernel executes 2x8x6x40 = 3840 floating point operations (flops); there is one multiply and one add for every element of the matrix. This corresponds to 1.65 flops per clock cycle, showing that the synthesis is successfully pipelining operations and producing some overlap. Ideally we would like each multiply-add to take one cycle, giving two flops per cycle, but with other overheads, time taken to fill pipelines with data etc., we believe this result is close to optimal.

Another report produced by HLS details the utilization of logic elements on the FPGA chip. Data from this report is shown in Table 1, the BRAM_18K elements are block memory, DSP48Es are the Digital Signal Processors, FF are Flip-Flops and LUTs are Look-Up Tables. This shows that the matrix-vector block is using a very small fraction of the space available on the chip, (at most 4% of the FFs) allowing considerable scope for replicating the block for increased parallelism (spatial parallelism on the FPGA).

D. Integrating the Matrix-Vector HLS Blocks into a Vivado design

In order to execute the matrix-vector IP block, it needs to be incorporated into a block design using Vivado Design Studio. There are many ways to do this and considerable effort was expended in comparing different options for the design. The design presented here contains the following:

- a number, nblocks, of matrix-vector IP blocks;
- the same number, nblocks, of Block Memory Generator blocks to provide BRAM block memory, one memory block per matrix-vector block;
- the same number, nblocks, of AXI BRAM Controller blocks to provide an AXI protocol interface for each memory block;
- a ZynQ UltraScale+ MPSoC IP block, which provides an interface to the ARM processor, through two master AXI4 High Performance Master Full Power Domain ports (HPM0 FPD and HPM1 FPD);
- a Clocking Wizard IP block provides a custom clock and is used to vary the clock speed provided to the other blocks;
- a Processor Reset System block
- sufficient AXI Interconnect and AXI Crossbar blocks to provide connectivity

There is considerable flexibility in how blocks are connected with AXI Interconnect and AXI Crossbar blocks to provide routes from each matrix-vector block to BRAM memory, from the ZynQ block to each BRAM memory controller in order to load and retrieve data, and from the ZynQ to each matrix-vector block in order to send control information. For small numbers of blocks, nblocks, this can be achieved with a single AXI Interconnect block. It has nblocks+1 slave ports connecting to the master ports of the nblocks matrix-vector blocks and one of the master ports on the ZynQ, and 2*nblocks master ports connecting the slave ports of the nblocks matrix-vector blocks and the slave ports of the BRAM memory controllers. An example of this design with four matrix-vector blocks is shown in Fig. 3.

However this provides complete interconnection between all matrix-vector blocks and all memories, which is unnecessary and wasteful of logic space. The final design uses a set of nblocks 2x1 (2 slave ports, 1 master port) AXI Crossbars, two 1x8 AXI Crossbars for connection to the ZynQ and a pair of 2x2 AXI Interconnects for protocol conversion. Testing the design showed that the maximum clock speed at which the design could be run before failure was 333 MHz.

The Vivado SDK tool shows the “registers” for each matrix-vector block. This is an area in memory which contains control words whose bits are used to control the block, e.g. AP_START to start the block and AP_IDLE to test whether the block has completed execution and is idle. This area also contains the addresses for each of the arguments in the matrix-vector subprogram, setting the addresses to locations in BRAM for the three arrays matrix, x and lhs prior to starting the block effectively “points” the routine to different input and output data.

The size of the BRAM memory blocks is also specified in the address editor, by specifying the address range. As discussed in section II, this FPGA has just over 3 MB of BRAM, so by setting the BRAM memories to 256 kB each, we can allow up to twelve matrix-vector blocks to operate simultaneously accessing their own memory.

Although a primary goal of the design and optimization process is to minimize execution time, there is also a trade-off against utilization of FPGA resources. We have seen the report from HLS showing the resources used by a single matrix-vector block. Vivado Design Studio also provides a utilization report for the whole design. FPGA utilization for the design is shown in Fig. 4.

TABLE I. UTILIZATION OF FPGA LOGIC ELEMENTS BY THE MATRIX-VECTOR BLOCK

	Logic Element			
	BRAM_18K	**DSP48E**	**FF**	**LUT**
Matrix-vector block	8	14	22114	6935
Available	1824	2520	548160	274080
Percentage used	0.44	0.56	4.03	2.53

[1] The Advanced Extensible Interface (AXI) is an Advanced Microcontroller Bus Architecture from ARM Ltd

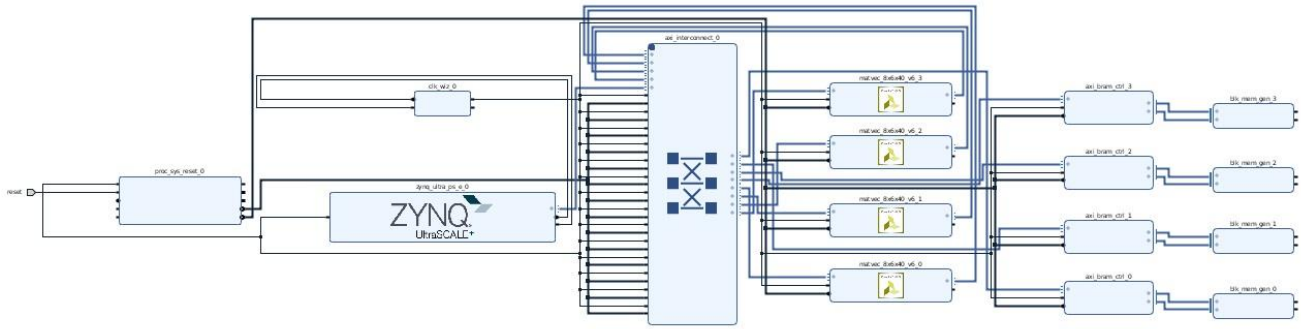

Fig. 3. The Block Design in Vivado Design Studio, with four matrix-vector blocks and four BRAM memory blocks.

E. CPU Driver Code

Although a primary goal of the design and optimization process is to minimize execution time, there is also a trade-off against utilization of FPGA resources. We have seen the report from HLS showing the resources used by a single matrix-vector block. Vivado Design Studio also provides a utilization report for the whole design. FPGA utilization for the design is shown in Fig. 6.

The FPGA code synthesized by Vivado Design Studio is driven by an application code running on the ARM processor. At present this is a driver which exercises the matrix-vector code using real data written out from a run of the LFRic code. A further step is to run LFRic itself on the ARM and have it compute its matrix-vector products on the FPGA. The ARM code is implemented in the following way:

- add devices /uio0 and /uio1 to the OS providing access to the ports HPM0 FPD and HPM1 FPD
- call mmap to map each device into user space
- assign pointers for each data array to locations in user space
- divide the work into groups of cells which will fit into the FPGA BRAM memory (maximum memory is nblocks x 256 kB = 3 MB)
- for each group of cells: assign to a matrix-vector block; copy input data into BRAM; set the control word "registers" for the block; start the block by setting AP_START; wait for block to finish by watching AP_IDLE; and copy output data from BRAM
- in practice we separate execution time from data copy time by filling all of BRAM, then running all nblocks matrix-vector blocks, then copying output data back and repeat
- perform a single execution to check correctness by comparing with a standard answer
- execute the code within a repeat loop to time the code

IV. PERFORMANCE

A. FPGA Performance

Performance of the matrix-vector code is timed as described above, excluding the data transfers between the ARM CPU and the FPGA. The reason for this is that whether data is transferred depends on the context. The major part of the 17MB data set, consists of the matrices. In any completed port of the LFRic code to the FPGA system, the matrices will be generated and used on the FPGA and so will never need to be transferred.

Timings are converted to execution rates in Gflop/s knowing that each matrix vector operation requires 2x8x6 flops; two operations, a multiply and an add for each matrix element.

Resource	Utilization	Available	Utilization %
LUT	138839	274080	50.66
LUTRAM	2327	144000	1.62
FF	213365	548160	38.92
BRAM	544	912	59.65
DSP	112	2520	4.44
IO	1	328	0.30
BUFG	4	404	0.99
MMCM	1	4	25.00

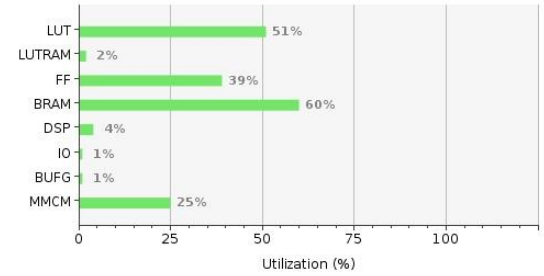

Fig. 4. FPGA utilization for the design reported by Vivado Design Studio.

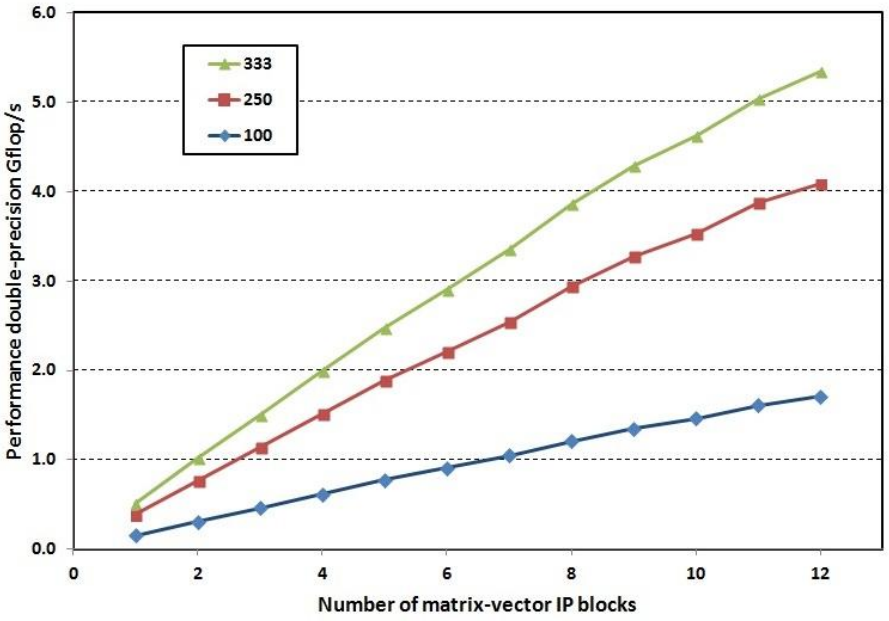

Fig. 5. Performance of the design on the UltraScale+ FPGA at different clock frequencies (MHz), with the number of matrix-vector blocks used.

Performance results are shown in Fig. 5. which shows the scaling in performance with the number of matrix-vector blocks used, with a clock frequencies of 100 MHz, 250 MHz and 333 MHz. The maximum performance achieved with twelve matrix-vector blocks is 5.34 Gflop/s. The speed-up for twelve blocks over one block is 10.5x representing a parallel efficiency of 94%.

Scaling with clock speed is very good but not perfect. With twelve matrix-vector blocks, the performance improves from 1.71 GFlop/s to 5.34 GFlop/s from 100 MHz to 333 MHz, an efficiency of 94%.

The FPGA performance is limited by three factors: the speed of an individual matrix-vector block, the number of blocks laid out on the FPGA hardware and the clock speed.

As we have seen the performance of a single matrix-vector block is running at 1.65 flops/cycle, close to the theoretical maximum of two flops/cycle. We have investigated putting more parallelism into a single block using the HLS dataflow pragma to enable concurrent and independent matrix-vector products to proceed in parallel. It turns out that doing that is exactly analogous to increasing the number of matrix-vector blocks.

When increasing the number of matrix-vector blocks or increasing the clock speed, there comes a point at which Vivado fails to generate a working design, reporting that timing constraints have not been met. There is a trade-off in that with fewer matrix-vector blocks a slightly higher clock speed is possible. Thus, for a simple design we can run the code at a higher clock speed, but as the number of matrix-vector blocks and memories increases, the complexity of the design increases and the maximum clock speed deceases. Thus we find that it is not possible to use all the real estate on the chip for this kernel.

B. Single Precision Performance

Building the matrix-vector block in Vivado HLS with floats instead of doubles yields a Task Latency of 2328 cycles compared to 2334 for doubles; almost identical. However, the utilization report shows that less space on the FPGA is being used, with Flip-Flops, the limiting factor, being down to 11935 from 22114 (54%). The estimated clock speed is up slightly to 377 MHz. Thus all other factors being equal we should be able to run with more matrix-vector blocks and at a higher clock frequency.

In practice we found that due to the recurrence of timing constraint errors in the Vivado design, it was not possible to achieve significant improvements in the number of blocks or the clock speed. With twelve blocks and at 333 MHz the single precision code runs at 5.58 Gflop/s compared to 5.34 Gflop/s for double precision.

C. Performance Compared to Xeon CPUs

The matrix-vector kernel was also run on the Met Office collaboration system 'monsoon' a Cray XC40 system with 12-core Intel Broadwell E5-2650 v4 2.2GHz CPUs. The original matrix-vector code (before FPGA optimization) was run using the Cray C compiler version 8.3.4. This code uses DGEMV calls which bring in the Cray scilib optimized library. This code uses OpenMP as described above to use all 12 cores and provides a socket for socket comparison with the ZU9 FPGAs.

Performance results are shown in Table 2. The Intel Xeon Broadwell performance is 12.56 Gflop/s compared to 5.34 Gflops for the FPGA (43%). However when we factor in the relative price and relative power consumption of the two devices we find that the argument in favour of the FPGA is compelling.

TABLE II. PERFORMANCE OF THE MATRIX-VECTOR KERNEL ON ZU9 FPGAS CMPARED TO XEON CPUS

System	Processor	Matrix-vector performance (Gflop/s)
Xilinx ZCU102	ZU9 FPGA	5.34
Cray XC40	E5-2650 v4 2.2GHz 12 cores	12.56

V. DISCUSSION

We have demonstrated the implementation of a matrix-vector kernel on an FPGA testbed system as a precursor to porting the full FRic weather model onto the EuroExa architecture. We have achieved performance figures of 5.34 Gflop/s in double precision and 5.58 Gflop/s in single precision. Work is continuing to improve the design with the aim of achieving improved performance with the same or decreased utilization of resources on the FPGA.

Another important step is to integrate this matrix-vector code within the LFRic weather model and demonstrate that a performance gain can be achieved by acceleration of the matrix-vector components of the workload. Other kernels will also be ported, especially those associated with generation of the matrices, so that, as mentioned above, the matrices can be generated and used entirely on the FPGA, removing the need for transfer of their data. We will work to port entire workflows, consisting of a series of kernels, from the LFRic model onto the FPGA. Porting such a workflow may result in better utilization of the FPGA resources than we have been able to achieve with single kernel.

In between workflows it will be necessary to reconfigure the FPGA and we note the considerable body of work that has been done in partial reconfiguration, which allows reconfiguration costs to be overlapped with useful work, e.g. [6].

REFERENCES

[1] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture", in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), DOI: 10.1109/MICRO.2016.7783710

[2] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, issue 10, October 2016, pp. 1591-1604, DOI: 10.1109/TCAD.2015.2513673

[3] Xilinx Inc. 2017. Zynq Ultrascale+ MPSoC. (2017). https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[4] S.V. Adams, R.W. Ford, M. Hambley, J.M. Hobson, I. Kavcic, C.M. Maynard, T. Melvin, E.H Mueller, S. Mullerworth, A.R. Porter, M. Rezny, B.J. Shipway, and R. Wong, "LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models", arXiv:1809.07267v1, 2018.

[5] P.D. Düben, F.P. Russell, X. Niu, W. Luk, and T. N. Palmer, "On the use of programmable hardware and reduced numerical precision in earth-system modeling", J. Adv. Model. Earth Syst., 7, 1393–1408, DOI:10.1002/2015MS000494.

[6] D. Koch, Partial Reconfiguration on FPGAs, Springer, ISBN 978-1-4614-1225-0