Hindawi
Scientific Programming
Volume 2019, Article ID 7807860, 18 pages
https://doi.org/10.1155/2019/7807860

Research Article

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

Mike Ashworth, Graham D. Riley, Andrew Attwood, and John Mawer

Department of Computer Science, University of Manchester, Manchester M13 9PL, UK

Correspondence should be addressed to Mike Ashworth; mike.ashworth.compsci@manchester.ac.uk

Received 29 March 2019; Accepted 31 July 2019; Published 13 October 2019

Guest Editor: Qiang Guan

In recent years, there has been renewed interest in the use of field-programmable gate arrays (FPGAs) for high-performance computing (HPC). In this paper, we explore the techniques required by traditional HPC programmers in porting HPC applications to FPGAs, using as an example the LFRic weather and climate model. We report on the first steps in porting LFRic to the FPGAs of the EuroExa architecture. We have used Vivado High-Level Syntheusywwi to implement a matrix-vector kernel from the LFRic code on a Xilinx UltraScale+ development board containing an XCZU9EG multiprocessor system-on-chip. We describe the porting of the code, discuss the optimization decisions, and report performance of 5.34 Gflop/s with double precision and 5.58 Gflop/s with single precision. We discuss sources of inefficiencies, comparisons with peak performance, comparisons with CPU and GPU performance (taking into account power and price), comparisons with published techniques, and comparisons with published performance, and we conclude with some comments on the prospects for future progress with FPGA acceleration of the weather forecast model. The realization of practical exascale-class high-performance computinems requires significant improvements in the energy efficiency of such systems and their components. This has generated interest in computer architectures which utilize accelerators alongside traditional CPUs. FPGAs offer huge potential as an accelerator which can deliver performance for scientific applications at high levels of energy efficiency. The EuroExa project is developing and building a high-performance architecture based upon ARM CPUs with FPGA acceleration targeting exascale-class performance within a realistic power budget.

1. 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 10^{18} floating point operations per second). Achieving exascale with CPU-based technologies is technically feasible but would result in unacceptable power requirements [1]. Therefore, there has been considerable interest in recent years in novel architecture systems which harness accelerators alongside CPUs to boost performance while delivering substantially improved power efficiency.

Field-programmable gate arrays (FPGAs) have recently attracted the attention of researchers in both academia and industry as a candidate accelerator for large-scale high-performance computing (HPC) applications [2, 3]. Limitations in the programmability and ease-of-use of FPGAs for large-scale scientific computing have recently been alleviated through the development of high-level tools [4]. Escobar et al. have analysed the suitability of a range of computational dwarves for acceleration using FPGAs [5]. Although FPGAs typically run at a lower frequency compared to CPUs and GPUs, they provide extremely competitive performance, especially when taking into account their lower power requirements; for example, Cong et al. have compared FPGA and GPU performance using the Rodinia suite, which is optimized for GPUs, and found that, "for 6 out of the 15 ported kernels, today's FPGAs can provide comparable performance or even achieve better performance than the GPU, while consuming an average of 28% of the GPU power" [6].

Conventional hardware description languages (HDLs) for programming FPGAs have in the past restricted this technology to experienced hardware engineers because of its complexity and the low level of abstraction. However, over the past decade, high-level languages and high-level synthesis (HLS) tools have been proposed to raise the abstraction level and the development accessibility of FPGAs [7–10]. These enable hardware configurations to be generated from high-level descriptions, namely, C, C++, OpenCL, and Java, to utilize the FPGAs with minimal experience of hardware design. Cong et al. have demonstrated that "an HLS solution can achieve an 11%–31% reduction in FPGA resource usage with improved design productivity compared to a hand-coded design" [11].

For example, Xilinx Vivado (previously AutoPilot) provides HLS compilers and tools that transform behavioural descriptions written in C/C++ to RTL languages by supporting all design phases of FPGAs [12]. In another example, Bosch et al. presented an OmpSs programming model that targets heterogeneous systems including CPUs, GPUs, and FPGAs [13]. Similar to OpenMP, OmpSs allows programmers to annotate their applications with compiler directives to express task parallelism with the FPGA configuration files being generated using Xilinx Vivado or Altera Quartus. Lee et al. proposed a directive-based OpenACC to FPGA translation framework, which provides a high-level programming model for utilising FPGAs [14]. This framework builds on the top of the OpenARC compiler, which translates OpenACC to OpenCL, which may then be compiled by the Altera OpenCL compiler. All of these solutions aim to provide portable performance across heterogeneous systems. Compared with the CPUs and GPUs, the time allotted to design solutions on the FPGA is relatively long. However, this time has been reduced significantly by HLS tools, thereby attracting researchers from the wider HPC community to accelerate their large-scale applications on FPGAs.

Linear algebra is ubiquitous in HPC applications across almost all scientific areas, and matrix operations form the basis of higher level algorithms such as the solution of partial differential equations. The Basic Linear Algebra Subprograms (BLAS) were defined in order to standardise a range of matrix operations for use in computational algorithms and have been categorised according to their level of computational intensity. Matrix-vector operations have been categorised as Level 2 BLAS in which results are computed one vector (row or column) at a time [15, 16]. Their implementation may be optimized for increased vector length with vector results being reused in cache before being returned to memory. For computers with hierarchical memory and for parallel computers, the opportunity for a further, higher level of optimization was identified, i.e., the Level 3 BLAS, in which matrix-matrix operations may be blocked such that submatrices are reused in cache or in local memory [17].

Most work on implementation of linear algebra subprograms on FPGAs has focused on matrix-matrix multiplication, a Level 3 BLAS operation. Dou et al. have proposed a general block matrix multiplication algorithm, which enhances data locality and reusability and considers limitations of local storage and I/O [18]. This algorithm is suited to the use of arbitrary matrix sizes and is supported by a scalable linear array of 12-stage pipelined processing elements (PEs). Integrating 39 PEs onto a Virtex xc2vp125-7 FPGA running at 200 MHz reaches a performance of 15.6 Gflop/s for double precision with 1.6 MB local memory and 400 MB/s external memory bandwidth.

Kumar et al. present two designs for an FPGA accelerator for IEEE 754 double-precision floating point matrix multiplication on a Virtex-5 SX240T FPGA [19]. The first design is limited by a requirement for high I/O bandwidth; the second reduces this by enhancing PE utilization. Xilinx ISE 10.1sp1 and ModelSim 6.2e were used to implement and simulate the designs. A simulated performance of 29.8 Gflop/s is reported for 40 PEs at a frequency of 373 MHz with 750 MB/s and 5.9 GB/s bandwidth requirements for design one and two, respectively.

Jovanovic and Milutinovic present the architecture and implementation of an FPGA accelerator for matrix-matrix multiplication using double-precision IEEE floating point arithmetic [20]. The algorithm is blocked to minimize resource utilization and maximise clock frequency. The authors compare the design with matrix multiplication from high-performance libraries, such as MKL, ACML, GotoBLAS, and Atlas implemented on Intel and AMD microprocessors. The FPGA design outperforms the CPU's with a reported performance of 203.1 Gflop/s using 252 PEs running at 403 MHz.

We also note that another body of work has focussed on the implementation of sparse matrix operations on FPGAs, e.g., [21], but that sparse methods are different in computational characteristics from the dense matrix operations considered here.

The use of lower precision data formats can reduce resource utilization, reduce memory bandwidth requirements, and increase circuit frequencies, thus delivering significantly higher performance for computationally intensive applications. Sun et al. investigate the use of mixed-precision algorithms in order to utilize reduced-precision data formats wherever possible without losing accuracy [22]. They have implemented a direct LU solver with iterative refinement with a choice of three different precisions for the iteration loop: standard 64-bit double-precision and 32-bit and 16-bit floating point representations. Use of reduced precision delivers two to three times the performance, can be run with a higher clock frequency (140 MHz for 16-bit vs. 120 MHz for 64-bit), and uses fewer resources (32 embedded multiplier blocks for 16-bit vs. 128 for 64-bit).

The work described in this paper has been carried out in the context of the EuroExa project which is described in Section 2. Section 3 of this paper describes the LFRic weather model focussing on the use of matrix-vector updates and how they have been extracted for FPGA acceleration. In Section 4, we describe the FPGA evaluation platform, the details of the implementation and the optimization of the matrix-vector code for FPGAs. Section 5 reports performance results, and in Section 6, we discuss the results including sources of inefficiencies, comparisons with peak performance, comparisons with CPU and GPU performance taking into account power and price, comparisons with published techniques, and comparisons with published performance and conclude with some comments on the prospects for future progress with FPGA acceleration of the weather forecast model.

2. The EuroExa Project

The EuroExa project (http://www.euroexa.eu; EuroExa has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement no. 754337) titled "Co-Designed Innovation and System for Resilient Exascale Computing in Europe: From Applications to Silicon" proposes an HPC architecture that is 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. EuroExa combines state-of-the-art computing components using a groundbreaking system architecture, which applies the design flexibility of UNIMEM [23], delivers high levels of performance to the selected applications, and balances compute resources with the resource demands of applications. Through codesign between the enabling technologies, the system software, and the applications, EuroExa is delivering an innovative solution that achieves both extreme data processing and extreme computing. This solution will be demonstrated through the design, construction, testing, and evaluation of three testbed systems throughout the duration of the project. This will enable EuroExa to deliver a recipe for the creation of an exascale computer by 2021.

In order to demonstrate the efficacy of the design, the EuroExa partners are assessing performance using 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 [24]. High-quality forecasting of weather and climate 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 and climate research communities.

In order to prepare for execution on the EuroExa testbed systems, we have been porting key components of the LFRic model to a Zynq UltraScale+ ZCU102 evaluation platform [25]. The approach is to study the LFRic code at three levels: the full application, compact applications or "mini-apps," and key computational kernels. An example of such a kernel is the matrix-vector product which contributes significantly to the execution time in the Helmholtz solver and elsewhere (see Section 3.2), and this kernel forms the focus of this paper.

The EuroExa project has identified five programming models which may be used to implement HPC applications on the Xilinx FPGA hardware. They are (listed alphabetically) as follows:

(i) Maxeler MaxCompilerMPT development environment [26, 27]

(ii) OmpSs@FPGA [28]

(iii) OpenStream [29]

(iv) SDSoC [30] or SDAccel [31] with OpenCL

(v) Vivado High-Level Synthesis and Vivado Design Suite [32]

For this work using the LFRic model, we have chosen to use Vivado High-Level Synthesis (HLS) to generate IP blocks to run as part of a Vivado design on the UltraScale+ FPGA. HLS allows direct synthesis from the high-level C code, which can be obtained from the weather model code, and was available at the time of the research for our target architecture, the Xilinx UltraScale+ development platform (see Section 4.1). Other partners within the EuroExa project are looking at the other programming models listed above, and therefore, across the project as a whole, we will be in a good position to compare the performance, ease-of-use, robustness, and maturity of the tools.

3. LFRic Weather and Climate Model

3.1. Overall Description. LFRic is a new atmospheric model, being developed at the Met Office in the United Kingdom, which supports both weather forecasting and climate simulations. 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 precursor 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 [33].

The GungHo code has also been developed specifically to maintain performance at high and low resolutions 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. The science code is written conforming to a specific application programming interface (API), and the PSyclone code generation tool is used to automatically generate the code targeting different computer architectures. The LFRic weather and climate model is based on the GungHo dynamical core with its PSyclone software technology [24].

LFRic uses data decomposition across parallel multinode clusters with halo exchanges between subdomains carried out using the message passing interface (MPI). This paper describes an approach to FPGA acceleration of a simple kernel that represents the situation on a computing element of a multinode cluster. For a full forecast model running on a large-scale multinode parallel system, this would be multiplied up many times. The relationship of this work to the MPI decomposition and to halo exchange between subdomains is further discussed in Section 6.6.

3.2. 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. The version of LFRic used for this work implements only parts of the scientific model, namely, the dynamics and individual kernels. LFRic dynamics was still under development at the time of this work, and important optimizations to its algorithmic performance such as provision of a multigrid preconditioner were not complete. Furthermore, additional science modules such as physics, ocean coupling, and data assimilation will also need to be addressed in the future.

Profiling was carried out on the Met Office collaboration system, a Cray XC40, running on a single core. Running the model with gprof and piping the output first into gprof2dot.py and thence into dot, the call graph is produced, as shown in Figure 1. The boxes in the call graph are coloured according to the amount of CPU time taken, with red being the highest and blue the lowest, inclusive of called routines.

Most of the CPU time is spent in the Helmholtz solver that is used to compute the pressure. Two leaf nodes, shown expanded in Figure 2, account for greater than 50% of the CPU time for this test case. 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 multiplication 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, as lower precision arithmetic operations consume fewer resources and can operate at higher clock frequencies [22]. 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, e.g., [34]. For this work, although the current LFRic code uses double precision only, we have tested our implementation with both single and double precision, in anticipation of the benefits of a mixed-precision solution (however, see the conclusions reported in Section 4.7).

3.3. Matrix-Vector Updates in LFRic. The matrix-vector updates have been extracted into a kernel test program and converted to C. There are dependencies between some of the updates across the horizontal mesh, and a graph colouring scheme is used in LFRic such that nodes within a single "colour" have no dependencies and can be computed simultaneously. Adams et al. describe how colouring is used to produce independent computations for multithreading with OpenMP [24]. This parallelism can also be exploited for the FPGA acceleration. 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 in which the cube-sphere grid has 6 "faces" each consisting of 12×12 finite-element cells, giving rise to 864 cells in the horizontal, extruded into vertical columns with 40 vertical levels (see Figure 1 in [24]). Today's global weather models are run with a resolution of order 10 km which using this grid would have around 6 million cells in the horizontal. These models would typically be distributed across thousands of nodes of a highly parallel multiprocessor, so the test grid is a good example of the subgrid size that may be found running on a single CPU.

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 8×6; this can increase if higher order methods are employed in the solution of the finite-element discretisation. Each update therefore consists of an x-vector of 6 elements, a matrix of 8×6 elements, and an output or left-hand-side (lhs) vector of 8 elements. The lhs-vector is updated, so it 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 are therefore $(8 + 6 + 48) * 864 * 40 * 8\text{B} = 17\,\text{MB}$ of input data and $8 * 864 * 40 * 8\text{B} = 2\,\text{MB}$ of output data. We will describe in Section 4.6 how the data are managed given the local memory constraints of the FPGA design.

4. Porting and Optimization for FPGA Acceleration

4.1. Development Platform and Environment. In preparation for access to the EuroExa testbed systems, we have been using a Xilinx Zynq UltraScale+ ZCU102 evaluation platform [25]. At its heart, this board contains a multiprocessor system-on-chip (MPSoC) comprising, in addition to other processors, an ARM Cortex A53 quad-core CPU running at 1.3 GHz and a Zynq UltraScale XCZU9EG-FFVB1156 FPGA. The FPGA contains some 600k logic cells, 2,520 DSP (digital signal processing) slices, and around 3.5 MB of BRAM (block random access memory, which is a small, fast memory implemented within the FPGA fabric). The ARM CPU is running Ubuntu 16.04.5, and we are using Vivado Design Suite [35] and Vivado HLS [36], both at version level 2017.4, to generate IP blocks and bitstreams for the FPGA.

4.2. Starting Code. LFRic is written in Fortran making use of many features from the Fortran 2003 standard [37]. Vivado HLS does not accept Fortran code, so the matrix-vector kernel was translated into C. We have hard-coded matrix sizes and loop lengths using #define statements so that HLS can produce its most efficient implementation and the requirement for control logic is minimized.

The resulting starting code after translation to C and with fixed data sizes is shown in Figure 3.

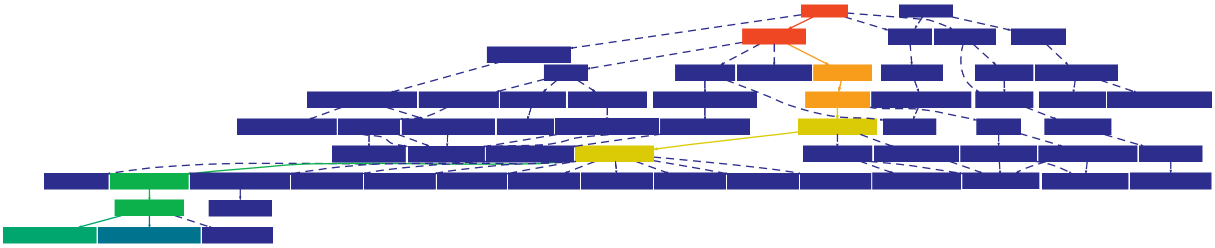


Figure 1: Call graph produced from gprof for the LFRic baroclinic test case. The colour coding shows which subroutines take the most CPU time as follows: red 90%–100%, orange 80%–90%, yellow 60%–80%, green 40%–60%, aquamarine 20%–40%, teal 10%–20%, and blue 0%–10%.

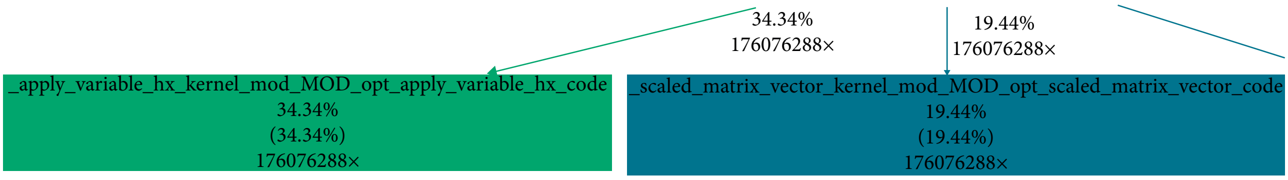

34.34%
176076288×
19.44%
176076288×
_apply_variable_hx_kernel_mod_MOD_opt_apply_variable_hx_code
34.34%
(34.34%)
176076288×
_scaled_matrix_vector_kernel_mod_MOD_opt_scaled_matrix_vector_code
19.44%
(19.44%)
176076288×

Figure 2: Expanded view of two leaf nodes of the call graph which account for more than 50% of the CPU time.

```
#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;
}
```

Figure 3: Starting code for a matrix-vector multiplication for NK vertical levels translated into C for entry into Vivado HLS.

4.3. Optimized Code in Vivado HLS. Vivado HLS includes a compiler which analyses C code, schedules operations in time sequence on the FPGA hardware (synthesis), and then writes the register-transfer level (RTL) code which forms an IP (intellectual property) block which can be added to an IP repository for later inclusion in a design using Vivado Design Suite.

On completion of the C synthesis process, Vivado HLS produces a synthesis report which contains performance metrics. Metrics include the task latency (the time from the start to the finish of the task) and the task interval (the time between the start times of two consecutive tasks), both measured in clock cycles. One 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 synthesised design; thus, it is always possible to convert performance in clock cycles into expected timings. 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, it is possible to optimise the code without executing it, achieving a substantial reduction in the reported latency. Objectives of the optimization were the following:

(i) To achieve streaming of data in and out of the IP block with a target of one 64-bit word per clock cycle

(ii) To achieve pipelining of the arithmetic operations and overlapping of multiplications with additions to achieve one 64-bit multiplication and one 64-bit addition every cycle

(iii) To minimize use of resources on the FPGA

The following optimizations were carried out:

(i) Loops were swapped to make the k-index over the vertical levels the innermost loop

(ii) 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 with a length of 40 elements

(iii) The HLS UNROLL pragma was applied to the innermost loops; unrolling by hand was also tried but was shown to result in no additional benefit

(iv) The HLS PIPELINE pragma was applied to the outermost loop

(v) The code just computes the matrix-vector product, without updating the left-hand-side (lhs) array; the update can be performed on the ARM CPU

(vi) 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, and max_write_burst_length = 64 were used

(vii) Data read from and written to the subprogram arguments were copied into and copied out from local working arrays (local arrays are implemented using BRAM_18K logic elements within the matrix-vector IP block) using memcpy

(viii) The input array x is constant for iterations of the outer df loop, so x is copied into its local array once at the start; slices of the matrix are copied in and columns of the output lhs array are copied out at each iteration of the loop

A comprehensive set of optimizing transformations for developing high performance FPGA designs using HLS is given in [38]. The resulting code is shown in Figure 4.

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. The three array arguments are specified as AXI (Advanced Extensible Interface) master ports (m_axi). AXI is an advanced microcontroller bus architecture from ARM Ltd. The three interfaces are bundled together into a single port using the bundle - option which helps to simplify the interconnection network. Specifying the return argument with an interface of s_axilite produces a slave port with the AXILite protocol, which is the connection for the “registers” used to control the block (see Section 4.5).

The last two optimizations in the list above are 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 are streaming at one word per cycle. We note in particular the benefit of the use of memcpy; HLS recognises memcpy and implements it using “burst mode” [36]. The presence or absence of data streaming in burst mode can be seen by attaching in the Vivado design an Integrated Logic Analyzer (ILA) IP block to the data path between the matrix-vector block and its BRAM. Screenshots for the two cases, without and with the data streaming optimizations, are shown in Figures 5(a) and 5(b).

The analysis reports produced by Vivado HLS include a Performance tab with a timeline which shows the issuing and duration of operations such as reads, multiplications, and additions. Examination of this timeline shows that, following an initial start-up phase, an addition (dadd) and a multiplication (dmul) are being issued every cycle.

Metrics reported in the HLS synthesis report detail the utilization of logic elements on the FPGA chip, specifically the numbers of BRAM elements (BRAM_18K), digital signal processors (DSP48Es), flip-flops (FFs), and look-up tables (LUTs). Performance and utilization data from this report are shown in Table 1 for a range of target clock periods. For slow clocks (50 ns and greater), HLS uses three times the number of DSP48Es.

Examination of the performance timeline in Vivado HLS shows that while it is at times able to carry out three additions and three multiplications in one cycle, the effect on performance is negligible as the execution of the whole is load/store bound. For clock periods greater than 2 ns, HLS uses three DSP48Es for a double-precision addition and eleven DSP48Es for a double-precision multiplication. At high clock speeds (clocks of 2 ns and less), fewer DSP48Es are used. The implication is that, at high clock speeds, timing constraints cannot be met using some of the DSP48Es and that their function is replaced by increased use of FFs and LUTs.

Table 2 shows the number of logic elements used by the matrix-vector code synthesised with a 2 ns clock period, compared with the total number of elements available on the ZU9 FPGA. The matrix-vector block uses 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.

Eight BRAM_18K logic elements are used: four as a buffer for the master m_axi port for streaming the data arrays and four for the local array space.

Using the starting code in Figure 3, Vivado HLS reports a latency of 69,841 clock cycles. The optimized code, synthesised with a target clock cycle of 2 ns, produces a latency of just 2,334 cycles, an improvement of a factor of 30. The actual estimated clock period of 2.89 ns (346 MHz) gives a guideline to the maximum frequency at which designs including this IP block may be expected to operate correctly.

4.4. Analysis of Execution Time for the Matrix-Vector Kernel. For this data case, with a matrix of size 8×6 and 40 vertical levels, the matrix-vector kernel executes $2 \times 8 \times 6 \times 40 = 3840$ floating point operations (flops); that is, there are one addition and one multiplication for every element of the matrix. Detailed examination of the performance timeline affords the following analysis of the reported latency. The latency of 2334 cycles (at a clock period of 2 ns) breaks down as follows: There is an initial period of 371 cycles comprising a start-up of 11 cycles followed by reading the 240 elements of the x matrix into the local array (local BRAM) in the burst mode at one element per cycle (240 cycles) and loading from the local array into registers at two elements per cycle (120 cycles).

Then, the outer loop iterating over df is executed with a loop count of 8. The loop iteration comprises a pipeline

```
#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;
}
```

Figure 4: Optimized code for a matrix-vector multiplication for NK vertical levels.

which performs 240 additions and 240 multiplications using one cycle for each combined multiply-add, so there are 240 cycles for each iteration of the loop. There is a starting/finishing cost of 43 cycles to carry out reads to fill the pipe (24 cycles) and writes to empty it (19 cycles), but the iterations can be executed at an interval of 240 cycles, so the 43-cycle "start-up" cost is paid once only for the loop. Eight times 240 plus a pipeline start-up of 43 cycles and an initial reading phase of 371 cycles yields a total of 2334 cycles.

An execution time of 2334 cycles corresponds to 1.65 flops per clock cycle, the overheads described above effecting a reduction from the ideal target of 2.0 flops per cycle.

4.5. Integrating the Matrix-Vector Blocks into a Vivado Design.

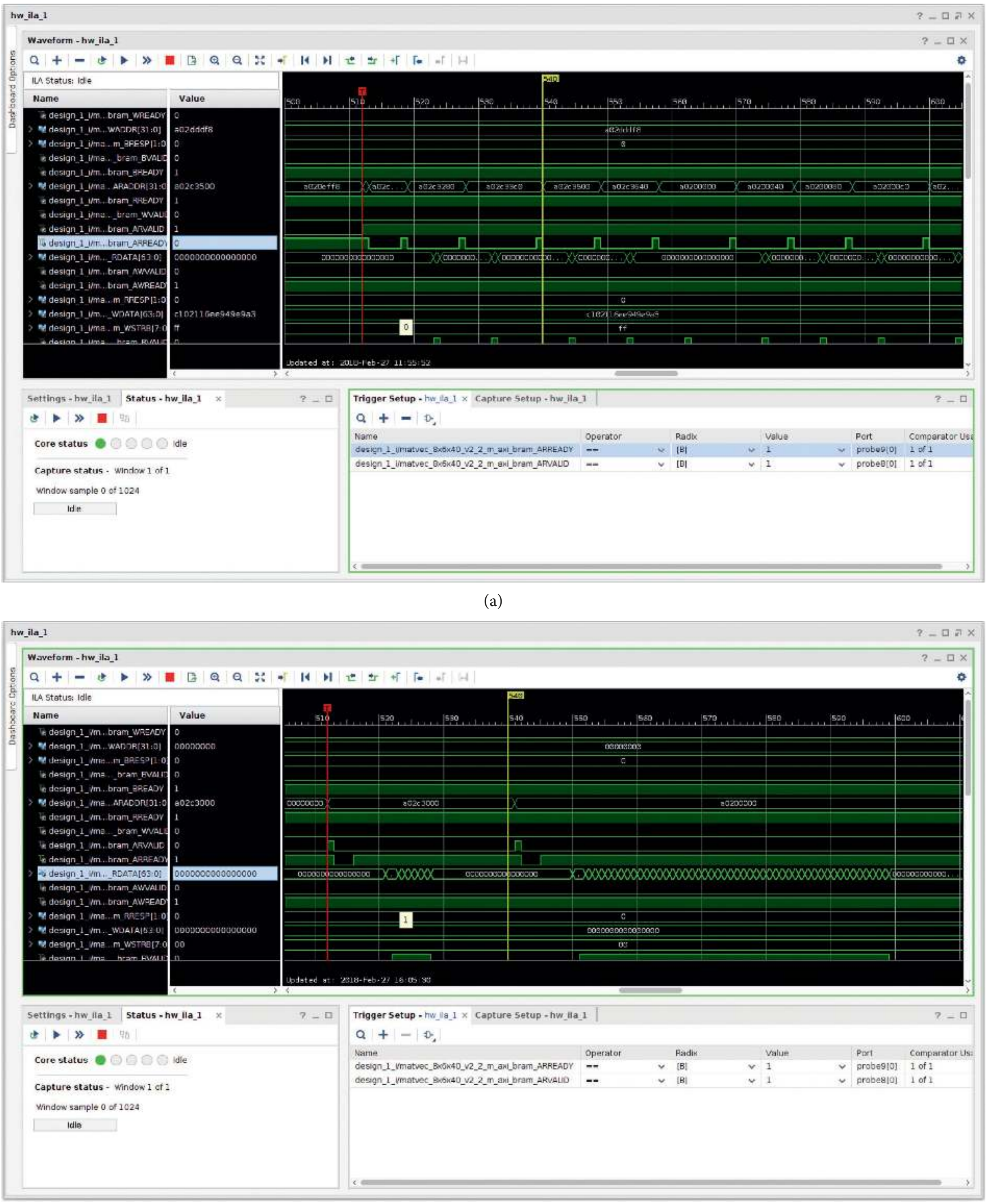
In order to execute the matrix-vector IP block, it needs to be incorporated into a block design using Vivado Design Suite [35]. We use IP blocks from the Vivado IP Catalog [39] in order to provide functions in the design for data handling, interface with the ARM CPU, BRAMs, clock control, etc. There are many ways to do this, and considerable effort was expended in comparing different options for the design. We present here two designs which incorporate a number of matrix-vector IP blocks in order to increase the available parallelism and which illustrate some of the trade-offs which need to be considered.

Both designs contain the following:

(i) A number, *n*blocks, of matrix-vector IP blocks

(ii) The same number, *n*blocks, of Block Memory Generator blocks to provide BRAM block memory, one memory block per matrix-vector block

(iii) The same number, *n*blocks, of AXI BRAM Controller blocks to provide an AXI protocol interface for each memory block



hw_ila_1

Waveform - hw_ila_1

ILA Status: Idle

Name	Value
design_1_i/m...bram_WREADY	0
design_1_i/m...WADDR[31:0]	a02dddf8
design_1_i/ma...m_BRESP[1:0]	0
design_1_i/ma..._bram_BVALID	0
design_1_i/m...bram_BREADY	1
design_1_i/ma...ARADDR[31:0]	a02c3500
design_1_i/m...bram_RREADY	1
design_1_i/ma..._bram_WVALID	0
design_1_i/m...bram_ARVALID	1
design_1_i/m...bram_ARREADY	0
design_1_i/m..._RDATA[63:0]	0000000000000000
design_1_i/m...bram_AWVALID	0
design_1_i/m...bram_AWREADY	1
design_1_i/ma...m_RRESP[1:0]	0
design_1_i/m..._WDATA[63:0]	c102116ee949e9a3
design_1_i/ma...m_WSTRB[7:0]	ff

Updated at: 2018-Feb-27 11:55:52

Settings - hw_ila_1 | Status - hw_ila_1

Core status ● ○ ○ ○ ○ Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

Idle

Trigger Setup - hw_ila_1 | Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port	Comparator Usa
design_1_i/matvec_8x6x40_v2_2_m_axi_bram_ARREADY	==	[B]	1	probe9[0]	1 of 1
design_1_i/matvec_8x6x40_v2_2_m_axi_bram_ARVALID	==	[B]	1	probe8[0]	1 of 1

(a)

hw_ila_1

Waveform - hw_ila_1

ILA Status: Idle

Name	Value
design_1_i/m...bram_WREADY	0
design_1_i/m...WADDR[31:0]	00000000
design_1_i/ma...m_BRESP[1:0]	0
design_1_i/ma..._bram_BVALID	0
design_1_i/m...bram_BREADY	1
design_1_i/ma...ARADDR[31:0]	a02c3000
design_1_i/m...bram_RREADY	1
design_1_i/ma..._bram_WVALID	0
design_1_i/m...bram_ARVALID	0
design_1_i/m...bram_ARREADY	1
design_1_i/m..._RDATA[63:0]	0000000000000000
design_1_i/m...bram_AWVALID	0
design_1_i/m...bram_AWREADY	1
design_1_i/ma...m_RRESP[1:0]	0
design_1_i/m..._WDATA[63:0]	0000000000000000
design_1_i/ma...m_WSTRB[7:0]	00

Updated at: 2018-Feb-27 16:05:38

Settings - hw_ila_1 | Status - hw_ila_1

Core status ● ○ ○ ○ ○ Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

Idle

Trigger Setup - hw_ila_1 | Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port	Comparator Usa
design_1_i/matvec_8x6x40_v2_2_m_axi_bram_ARREADY	==	[B]	1	probe9[0]	1 of 1
design_1_i/matvec_8x6x40_v2_2_m_axi_bram_ARVALID	==	[B]	1	probe8[0]	1 of 1

(b)

Figure 5: Waveforms from a Vivado Integrated Logic Analyzer (ILA) block inserted into the design, without (a) and with (b) streaming optimizations.

(iv) A Zynq UltraScale+ MPSoC IP block, which provides an interface to the ARM processor, through two AXI4 high-performance master full-power domain ports (HPM0 FPD and HPM1 FPD)

(v) A Clocking Wizard IP block, which provides a custom clock and is used to vary the clock speed provided to the other blocks

(vi) A Processor Reset System block

Table 1: Performance and utilization metrics from the Vivado HLS synthesis report for the double-precision matrix-vector code for a range of target clock periods.

Target clock (ns)	Estimated clock (ns)	Latency (clocks)	DSP48E total	DSP48E dadd	DSP48E dmul	FF	LUT
100	87.50	2306	42	3	11	28960	9023
50	43.75	2306	42	3	11	28960	9023
20	17.50	2312	14	3	11	20423	6751
10	8.75	2315	14	3	11	20742	6889
5	4.99	2321	14	3	11	21466	6860
2	2.89	2334	10	0	10	23199	7203
1	1.96	2336	10	0	10	23570	7200
0.5	1.96	2336	10	0	10	23570	7200

Table 2: Number of logic elements used by the matrix-vector code synthesised with a 2 ns clock period, compared with the total number of elements available on the ZU9 FPGA.

Logic element	BRAM_18K	DSP48E	FF	LUT
Matrix-vector block	8	10	23199	7203
Available	1824	2520	548160	274080
Percentage used	0.44	0.4	4.23	2.63

The differences between the two designs then arise in the way in which the matrix-vector blocks, the BRAMs, and the Zynq MPSoC are interconnected. The basic requirement is for 64-bit data paths for the following connections:

(i) Master Zynq port to the slave port on each matrix-vector IP block so that the ARM CPU has access to the control registers

(ii) Master Zynq port to a slave port on each BRAM controller so that the ARM CPU can access the BRAMs

(iii) Master port on each matrix-vector IP block to a slave port on each BRAM controller so that each matrix-vector IP block can access data in the BRAMs

4.5.1. Design 1: Full Interconnection. This design simply uses AXI Interconnect blocks to provide complete interconnection between all matrix-vector blocks, all BRAMs, and the Zynq MPSoC. For small numbers of blocks ($n$blocks ≤ 8), this can be achieved with a single AXI Interconnect block. We set up an AXI Interconnect block with $n$blocks+1 slave ports connected to the master ports of the $n$blocks matrix-vector blocks and one of the master ports on the Zynq, and 2 ∗ $n$blocks master ports connected to the slave ports of the matrix-vector blocks and the slave ports of the BRAM controllers.

AXI Interconnect blocks support a maximum of 16 master ports and 16 slave ports, so for $n$blocks >8, two AXI Interconnect blocks were used each using a different master port on the Zynq and each servicing $n$blocks/2 matrix-vector blocks and $n$blocks/2 BRAM controllers. An example of this design with four matrix-vector blocks is shown in Figure 6.

Having completed the design, the developer instructs Vivado Design Suite to perform the following series of steps:

(i) Synthesis: transforming an RTL-specified design into a gate-level representation

(ii) Implementation: placing and routing the netlist onto device resources, within the logical, physical, and timing constraints of the design

(iii) Bitstream generation: generating a bitstream for the Xilinx device configuration

There are two types of design constraints: physical constraints, which recognise limitations in the mapping of logical design objects to device resources, and timing constraints, which define the frequency requirements for the design. Different physical and timing constraints might be needed for different target devices. A timing report can be run on the synthesised design so that problems can be fixed before implementation.

Vivado carries out a static timing analysis which computes the expected timing of the circuit without requiring a simulation of the full circuit. The timing report summarizes any negative slack found in the analysis; negative slack is generated when a path is too slow, and the path must be sped up (or the signal must be delayed) if the whole circuit is to work at the desired speed. In particular, the report provides figures for the worst negative slack (WNS), the worst slack of all the timing paths, and the total negative slack (TNS), the sum of all WNS violations [40]. If the design violates these timing constraints, bitstream generation will fail.

4.5.2. Design 2: Reduced Interconnection. Testing Design 1 showed that the maximum clock speed at which the design could be run before timing violations induce failure was relatively low, just 250 MHz. As we have provided each matrix-vector IP block with its own BRAM, it does not need to access the other BRAMs and full interconnection between every matrix-vector block and every memory is not required. Design 2 implements a simplified design while still providing the required connectivity described above. In addition, AXI Interconnect blocks have a high degree of internal complexity as they include capabilities for converting between different AXI protocols and between paths having different data widths. In this design, we used, where possible, AXI Crossbar blocks, which are internally simpler.

As we are running with double-precision floating point data, all data paths in the design were set to 64-bit. However,

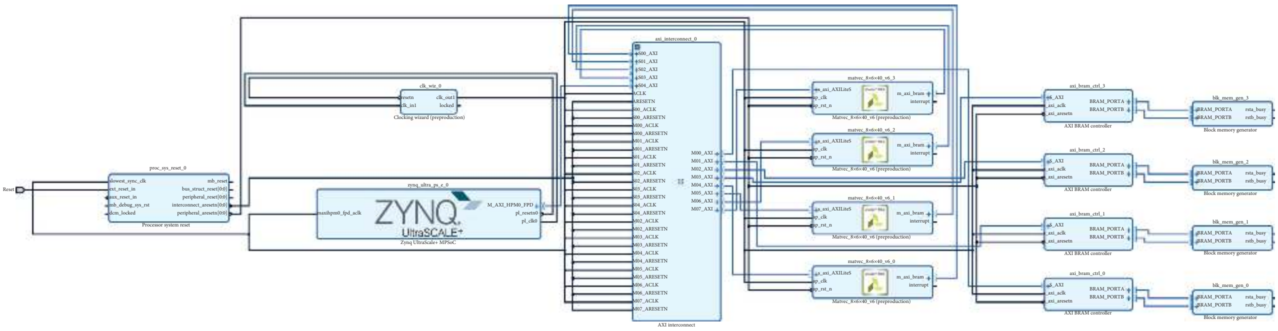


Figure 6: Design 1, the full interconnect design, with four matrix-vector blocks and four BRAM blocks implemented in Vivado Design Suite.

the master ports on the Zynq are fixed at 128-bit. We therefore start with an AXI Interconnect block to provide conversion from 128-bit wide HPMx FPD ports to 64-bit wide data paths to the matrix-vector blocks and the BRAM blocks. From one master port on the AXI Interconnect, an AXI Protocol block provides conversion between AXI4 and AXI4Lite for the slave ports on the matrix-vector blocks. This design differs from Design 1 in that we use the two master ports on the Zynq for different purposes; one services the control registers of the matrix-vector blocks, and the other accesses the BRAM blocks.

For the control registers, one AXI Crossbar switch allows the MPSoC HPM0 FPD port to fan out to the slave ports on multiple matrix-vector blocks. A second AXI Crossbar switch allows the MPSoC HPM1 FPD port to fan out to the slave ports on the BRAM controllers. However, both these latter connections and the master ports on the matrix-vector blocks need access to the BRAMs, so each BRAM controller is attached to a simple AXI Crossbar with two slave ports and one master port. An example of this design with four matrix-vector blocks is shown in Figure 7.

4.5.3. Comparison of the Designs. In the previous two sections, we have produced two designs with twelve matrix-vector blocks. The ZU9 FPGA has around 3.5 MB of BRAM. Using twelve matrix-vector blocks each with its own dedicated BRAM leads to twelve BRAMs of 256 kB each or 3 MB in total, utilizing most of the available BRAM resources. Fewer matrix-vector blocks would provide limited parallelism. Larger numbers of blocks would mean reducing the capacity of the BRAMs to 128 kB. In addition, we are close to the limitations imposed by timing constraints; a trial design with sixteen matrix-vector blocks failed with timing violations.

A comparison of the interconnect requirements of the two designs is as follows:

Design 1: two AXI Interconnect blocks each with 7 slave ports and 12 master ports (7×12)

Design 2: one AXI Interconnect block (2×2)

one AXI Protocol converter
two AXI Crossbar blocks (1×12)
12 AXI Crossbar blocks (2×1)

4.5.4. Managing Memory and Address Spaces. Having successfully completed the steps to generate a bitstream in Vivado Design Suite, we can export the hardware platform specification file. It is most easily viewed using the Xilinx SDK [41], which 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 its execution and is idle. This area also contains the addresses for each of the arguments in the interface to 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 block to the required input and output data locations.

Each HPM FPD master port is associated with an address space; for HPM0 FPD, the address space starts at xA0000000, and for HPM1 FPD, the address space starts at xB0000000. The Vivado Design Suite Address Editor [42] is used to allocate address ranges for each master interface. For Design 2, HPM0 FPD only has access to the slave ports on the matrix-vector blocks, so the address space from xA0000000 is used for them. In contrast, HPM1 FPD only has access to the memories, so it uses the space from xB0000000. The size of the BRAM blocks is also specified in the address editor, by specifying the address range.

Although the 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 discussed in Section 4.3 the utilization report from HLS showing the resources used by a single matrix-vector IP block. Vivado Design Suite also provides a utilization report for the whole design. FPGA utilization for the two designs is shown in Figure 8. While the utilization of DSPs and BRAMs is the same as they derive from, respectively, the number of matrix-vector blocks and the number of BRAM blocks, the simplification of the interconnect in Design 2 relative to Design 1 has resulted in a small but significant reduction in the numbers of FFs and LUTs used. As we shall see when looking at performance in Section 5, this allows faster clock speeds and/or additional spatial parallelism, leading to higher performance.

4.6. CPU Driver Code. The FPGA bitstream generated by Vivado Design Suite is driven by an application code

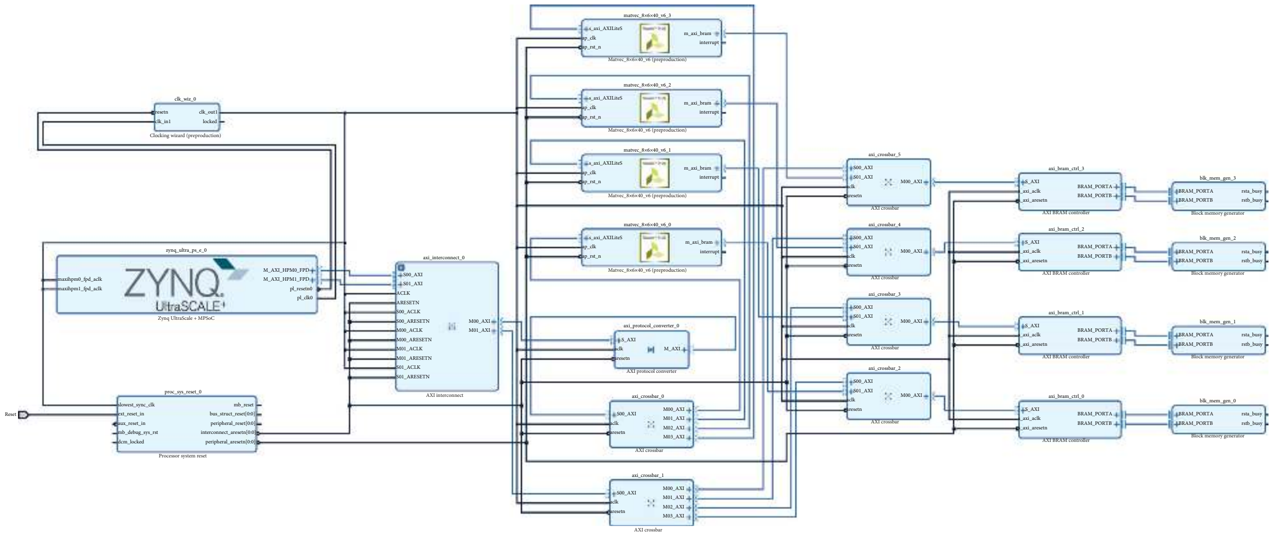
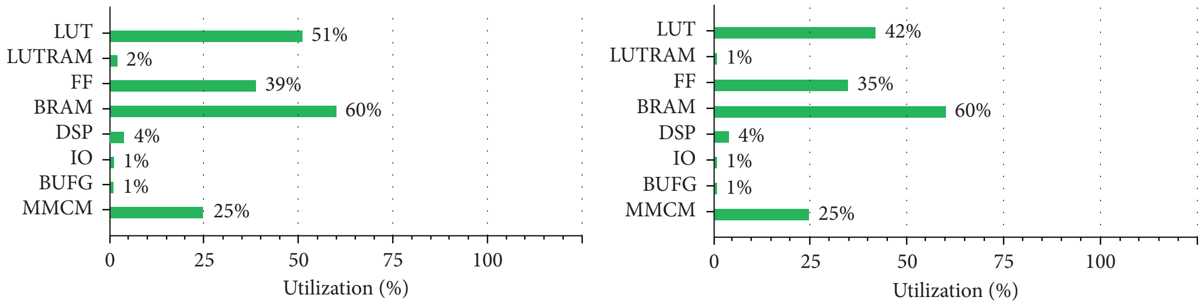


FIGURE 7: Design 2, the reduced interconnect design, with four matrix-vector blocks and four BRAM blocks implemented in Vivado Design Suite.

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

Resource	Utilization	Available	Utilization (%)
LUT	116237	274080	42.41
LUTRAM	817	144000	0.57
FF	194246	548160	35.44
BRAM	544	912	59.65
DSP	112	2520	4.44
IO	1	328	0.30
BUFG	5	404	1.24
MMCM	1	4	25.00



(a)

(b)

FIGURE 8: Comparison of the FPGA utilization reported by Vivado Design Suite for Design 1 (a) and Design 2 (b).

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. This is a preparatory step, prior to running the full LFRic weather model on the ARM with kernels, including matrix-vector multiplications and other kernels, offloaded onto the FPGA.

Provision is made within Vivado HLS for the automatic generation of IP-specific application program interfaces (APIs), specifically C routines which can be used on the ARM processor to control the IP block, e.g., to initialize, start, and monitor status and read and write data [43]. We have chosen to develop our own API, initially in C for the matrix-vector kernel, but it has also been implemented in standard Fortran and tailored to the needs of the LFRic application. The methodology is as follows:

(i) We add devices /uio0 and /uio1 to the Linux OS providing access to the ports HPM0 FPD and HPM1 FPD

(ii) We call mmap to map each device into the user space

(iii) We assign pointers to locations in the user space for the register space for each block, and for the data arrays in each BRAM

(iv) We divide the work into groups of columns which will fit into the FPGA BRAM (see below)

(v) For each group of columns (see below), we perform the following:

(a) Assign to a matrix-vector block
(b) Copy input data from CPU RAM into BRAM
(c) Set the control word "registers" for the block
(d) Start the block by setting AP_START
(e) Wait for the block to finish by watching AP_IDLE
(f) Copy output data from BRAM back to CPU RAM

(vi) In practice, we separate execution time from data copy time by filling all of the BRAMs, then running all *n*blocks matrix-vector blocks, and finally copying output data back and repeat

(vii) We perform a single execution to check correctness by comparing with the standard answer

(viii) We time the code by executing within a repeat loop to amortize start-up costs over a larger body of work

On the UltraScale+, we are limited to around 3.5 MB of BRAM space. The two FPGA designs in Section 4.5 use BRAMs of size 256 kB so that twelve such BRAMs deliver 3 MB of capacity. The code therefore blocks the data for each matrix-vector block so that it fits within its 256 kB of BRAM.

The LFRic data consist of vertical columns comprising a horizontal cell from the finite-element grid with all its 40 vertical levels. With an input x array of 6 elements, an output lhs array of 8 elements, and a matrix of 8×6 elements, each column requires $(8 + 6 + 48) * 40 * 8\,\mathrm{B} = 19840\,\mathrm{B}$ or around 19 kB. We can accommodate just 13 columns within the 256 kB BRAM for each matrix-vector block. The driver code on the ARM contains code to manage the allocation of the 864 total columns in the model across the number of matrix-vector IP blocks given the limitation of 13 columns per block.

4.7. Modifications for 32-Bit Floating Point Data. In order to assess the effect on performance and chip utilization of reduced precision, we repeated the above procedure with 32-bit single-precision "float" data. The only changes required on the FPGA side were changing from double to float in the C code which was passed to Vivado HLS and changing the width of some of the data paths in the Vivado design from 64-bit to 32-bit, i.e., those data paths between the matrix-vector blocks and the BRAM blocks and between the Zynq and the BRAM blocks. The ARM driver code was modified with single-precision versions of the key data-handling subprograms.

Estimated performance reported by Vivado HLS for the single-precision code was 2328 ns compared with 2334 ns for double precision, a saving of only 0.26%. The implementation in HLS is still targeting two flops per cycle irrespective of the precision. Where we hope to make major gains in using single precision is that the logic required for the computation is expected to consume fewer resources on the chip meaning that we can implement a larger number of IP blocks delivering greater parallelism.

Table 3 shows the resource utilised for a single IP block in both single and double precision and for a full 12-block Vivado design (reduced interconnect) in single and double precision. Indeed, the resources utilised are much reduced with single precision, using for the full design only 30% of the DSP48Es, 54% of the FFs, and 64% of the LUTs. However, this does not translate into larger numbers of IP blocks as, once again, the violation of timing constraints was found to limit the number of blocks to 12 and the clock frequency to 333 MHz.

We noted earlier that the use of FPGAs as accelerators offers considerable scope for improved performance using reduced precision, as lower precision arithmetic operations consume fewer resources and can operate at higher clock frequencies [22]. However, our attempt to exploit higher levels of parallelism in single precision by replicating IP blocks across the FPGA has not been successful, raising the question of whether an alternative technique is required to generate more parallelism in this case.

5. Performance

Performance of the matrix-vector code was timed, excluding the data transfers between the ARM CPU and the FPGA. The reason for this is that the need for transferring data depends on the context. The major part of this dataset, 17 MB out of 19 MB, consists of the matrices. In any completed port of the LFRic weather model 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 8×6 matrix-vector multiplication requires $2 \times 8 \times 6$ flops; there are two operations, one addition and one multiplication, for each matrix element.

Performance results are shown in Figure 9 and Table 4. Performance is reported for the two designs, full interconnect and reduced interconnect, showing the scaling in performance with the number of matrix-vector blocks used, at the clock frequencies of 100 MHz, 250 MHz, and 333 MHz. We also include for Design 2 (reduced interconnect) the performance scaling for single precision at 333 MHz. The maximum performance achieved with twelve matrix-vector blocks is 5.34 Gflop/s for double precision and 5.58 Gflop/s for single precision.

The performance for Design 2 (reduced interconnect) exceeds that for Design 1 by 10% at 8 IP blocks, but more importantly, it functions correctly out to 12 IP blocks, whereas Design 1 is limited to 8 IP blocks because of violation of timing constraints.

For Design 2, the speed increase for twelve blocks relative to one block is 10.5x representing a parallel efficiency of 94%, where parallel efficiency on n IP blocks, E_n, is defined as $E_n = T_1/(n \cdot T_n)$, where T_1 is the execution time on one IP block and T_n the execution time on n IP blocks. Scaling with clock speed is also good. With twelve matrix-vector blocks, the performance improves from 1.71 Gflop/s to 5.34 Gflop/s at clock frequencies from 100 MHz to 333 MHz, an efficiency of 94%. We note that the maximum clock frequency for the DSP48 logic cells on the ZU9EG FPGA is 775 MHz [37].

Table 3: ZU9 FPGA resource utilization for the 333 MHz reduced interconnect designs using 12 matrix-vector IP blocks for single and double precision.

	BRAM_18K	DSP48E	FF	LUT
Double-precision IP block	8	10	23199	7203
Single-precision IP block	4	3	11934	6391
Single-precision IP block resource relative to double precision (%)	50	30	51	89
Double-precision 12-block Vivado design	816	120	302606	204616
Single-precision 12-block Vivado design	792	36	162726	131758
ZU9 FPGA available resource	912	2520	548160	274080
Single-precision Vivado design resource relative to double precision (%)	97	30	54	64

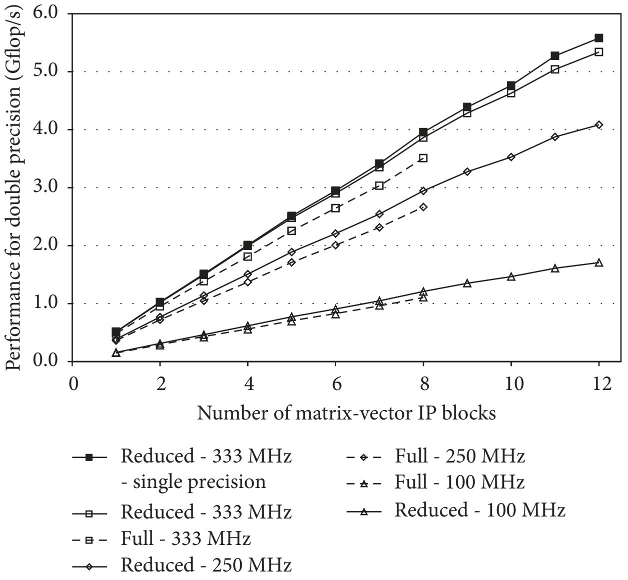


Figure 9: Performance of Design 1 (full) and Design 2 (reduced) for double precision on the UltraScale+ FPGA at different clock frequencies, showing scaling with the number of matrix-vector IP blocks used. Also shown is Design 2 (reduced) for single precision at 333 MHz.

In order to compare FPGA performance against state-of-the-art CPU performance, we have run the matrix-vector kernel, after conversion to C but before modification for FPGAs, on a single node Intel CPU. The CPU is an Intel Broadwell E5-2650 v2 2.60 GHz CPU with 8 cores. As it is part of a Cray multiprocessor system, we used the C compiler from Cray Compilation Environment version 8.5.8, as this generally delivers better performance than GCC. OpenMP was used to exploit the benefits of all eight cores. The speed increase on the Intel CPU from 1 to 8 cores was 5.8x, i.e., 72% parallel efficiency. The performance comparison is shown in Table 5. Peak performance of the Broadwell CPU is simply derived from 16 flops/cycle × 8 cores × 2.6 GHz. Peak performance of the ZCU102 FPGA is from the study in [45]. This figure does not take into account the precision of the data in the computation and so is probably a considerable overestimate for 64-bit precision.

The FPGA performance is 54% of that of the Broadwell 8-core CPU; however, for comparison of CPU and FPGA performance, it is essential to take into account the relative power consumption and the relative price of the two devices. This will be done in Section 6.3.

6. Discussion

6.1. Performance of the Matrix-Vector Kernel. We have described the development and implementation using Vivado HSL and Vivado Design Suite of a matrix-vector multiplication kernel which has been used to process a significant dataset, 864 columns of 40-level data, from the LFRic weather and climate model. A kernel which processes a single column of data (40 double-precision floating point matrix-vector multiplications with an 8×6 matrix) has been implemented as an IP block and replicated twelve times across the logic cells of a Xilinx ZU9EG FPGA. The resulting performance is 5.34 Gflop/s for double precision at a clock speed of 333 MHz on the programmable logic side. This is the rate of computation from BRAM to BRAM on the FPGA, neglecting data transfers from other parts of the system.

There are three factors which determine and which limit the achieved performance for our matrix-vector kernel:

(i) The performance in flops/cycle of an individual matrix-vector IP block

(ii) The number of matrix-vector IP blocks in the design

(iii) The clock frequency used to drive the programmable logic (PL), principally the matrix-vector blocks but also the associated blocks, e.g., the BRAM blocks

The performance of an individual matrix-vector IP block is targeting a peak of 2 flops/cycle but is limited in practice to 1.65 flops/cycle because of overheads associated with data transfers and pipeline start-up costs (according to the performance estimate of Vivado HLS). The number of IP blocks employed and the clock frequency of the PL are limited by timing constraints. In particular, we would like to be able to exploit all the available logic of the FPGA but find that, in practice, these timing constraints place a limitation which is more severe than the amount of resources required. In other words, for our application, timing constraints outweigh resource constraints.

An ideal or peak performance figure, P_0, for this design with twelve blocks running at 333 MHz would be 2 flops/

Table 4: Performance in Gflop/s using different numbers of matrix-vector IP blocks of Design 1 (full) and Design 2 (reduced) for single and double precision on the UltraScale+ FPGA at different clock frequencies.

	Design 2 (reduced interconnect)				Design 1 (full interconnect)		
Number of matrix-vector IP blocks	Single precision	Double precision			Double precision		
	333 MHz	333 MHz	250 MHz	100 MHz	333 MHz	250 MHz	100 MHz
1	515.7	510.6	386.3	157.9	483.1	362.0	148.3
2	1025.5	1016.2	768.7	312.7	950.7	720.7	292.6
3	1512.0	1496.4	1140.2	463.9	1385.9	1048.5	432.7
4	2008.8	1995.7	1508.6	616.8	1808.4	1370.5	562.0
5	2511.3	2477.7	1889.8	772.9	2253.3	1711.1	705.3
6	2947.1	2900.2	2208.0	907.0	2645.9	2007.9	828.2
7	3414.7	3351.6	2545.8	1047.3	3033.7	2314.7	960.4
8	3956.5	3862.6	2944.9	1209.2	3508.4	2666.1	1105.4
9	4390.0	4286.3	3275.1	1351.7	—	—	—
10	4761.8	4629.6	3527.9	1465.6	—	—	—
11	5274.2	5040.3	3875.9	1610.0	—	—	—
12	5580.1	5338.8	4084.6	1707.9	—	—	—

Table 5: Comparison of ZU9 FPGA double-precision matrix-vector performance with Intel multicore CPU performance.

Hardware	Matrix-vector performance (Gflop/s)	Peak performance (Gflop/s)	Percentage peak
ZCU102 FPGA	5.34	600	0.9%
Intel Broadwell E5-2650 v2 2.60 GHz 8-core CPU	9.86	332.8	3.0%

cycle × 12 blocks × 333 MHz = 7.99 Gflop/s. The actual performance, P_a, may be obtained from the ideal performance by two efficiency factors, the single block efficiency, eff_s, and the efficiency with which the blocks are combined in parallel in the design, eff_p; thus,

$$P_a = P_0 \times \mathrm{eff}_s \times \mathrm{eff}_p = 5.34\,\mathrm{Gflop/s}. \tag{1}$$

Again assuming the performance estimate from Vivado HLS is realised in practice, a value of 85% for eff_s is given, which implies the parallel efficiency figure for the design, eff_p, is 79%.

There is a relationship, a trade-off, between the number of blocks and the maximum clock speed. 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 decreases. As the clock speed increases and/or the number of matrix-vector blocks increases, the design reaches a point at which timing constraints become important and timing violations cause the implementation to fail.

The maximum clock frequency at which Design 2 operates correctly is shown in Table 6 for different numbers of matrix-vector blocks. The impact on performance is that although increasing the number of blocks from one to twelve potentially delivers up to a twelvefold increase in performance, the clock speed is reduced from 450 MHz to 333 MHz, a reduction of 74%, so the 12x potential increase is immediately limited to 8.9x. A similar impact on performance is reported by Khayyat and Manjikian who stated that “Increasing the system size eight times, from 8 to 64 arithmetic units, results in halving the maximum achievable frequency from 320 to 160 MHz” [46].

6.2. Peak Performance of FPGAs.

It is usual to compare the sustained performance of an algorithm on a processor with the processor’s peak performance in order to assess the efficiency with which the hardware is being utilised and to look for sources of inefficiency. The peak performance for a fixed-hardware processor such as a CPU or GPU is obtained by summing the number of adders and multipliers, followed by multiplying by the number of floating point results they return per cycle (in streaming or vector mode, if available) and then multiplying by the maximum clock frequency. This represents the maximum processing rate, which can never be attained in practice, but it is approached by the most efficient algorithms, which manage to keep arithmetic hardware busy despite overheads of data transfer, instruction scheduling, etc.

Determining the peak performance of programmable logic, such as FPGA hardware, is less straightforward. The nature of the hardware means that the number of adders and multipliers is unknown, being an output of the implementation. Moreover, an almost limitless number of different floating point representations are available, the choice of which has an effect on floating point performance. The ideal performance derived above for our design is certainly not the peak performance of the FPGA, being dependent as it is on the number of blocks and the clock frequency, which are determined by the design, not by the hardware.

This problem has been addressed by Parker [45], who specifically examined the peak performance capabilities of digital signal processors (DSPs), GPUs, and FPGAs. With regard to our problem, outlined above, of being unable to utilize all the logic resources of the chip because of timing constraints, he wrote “. . . when large numbers of floating point operators are packed together, the result

Table 6: Maximum clock frequency at which Design 2 still operates correctly for different numbers of matrix-vector blocks.

Number of matrix-vector blocks	Maximum clock frequency (MHz)	Matrix-vector performance (Gflop/s)
1	450	0.688
4	400	2.372
8	333	3.863
12	333	5.339

is routing congestion. This causes a large reduction in achievable design clock rates as well as logic usage which is much higher than a comparable fixed-point FPGA design."

6.3. FPGA vs. CPU Comparison. We have reported FPGA performance for the double-precision matrix-vector kernel which is 5.34 Gflop/s and which is 54% of that of an 8-core Intel Broadwell CPU. Clearly for "acceleration," we wish to outperform typical CPUs; however, we have noted that it is critical to take into account the comparison between power consumption and price between these two devices. It has been reported that GPUs achieve a price efficiency ranging from 0.07 to 0.12 €/Gflop/s and power efficiency of up to 20 Gflop/s/W [47], and the use of GPUs as accelerators in multi-CPU systems implies that their efficiencies certainly exceed those of CPUs. For midclass FPGAs, the equivalent figures are a price efficiency of 0.29 €/Gflop/s and a power efficiency which exceeds 70 Gflop/s/W [47].

6.4. Comparison with Published Techniques. Cong et al. [48] argue that FPGA programming with HLS can be made easy by following a simple set of five "best-effort" guidelines or steps. Comparing our work with these guidelines, the following are found:

(i) Data caching: we load the data into BRAM which acts as a fast cache for the FPGA matrix-vector blocks.

(ii) Pipelining: we achieve pipelining using HLS directives.

(iii) PE duplication: we achieve duplication of processing elements using Vivado with multiple IP blocks.

(iv) Computation/communication overlap: we achieve overlap as communication is performed by the ARM CPU which overlaps with computation on the FPGA, though we also discount communication time for well-considered reasons discussed earlier.

(v) Scratchpad reorganization: this is something we have not tried. This technique is mainly aimed at building larger data structures from small data types (e.g., 8-bit) to amortize costs, and as we mainly use 64-bit data, it is probably not worthwhile. Cong et al. say of their 64-bit benchmarks (GEMM and SPMV) that the speed increase achieved with this technique is "limited."

6.5. Comparison with Published Performance. Of the research papers dealing with the implementation of dense linear algebra algorithms on FPGAs, there are many which report the performance of matrix-matrix multiplication (MXM) but few which look at matrix-vector multiplication (MVM). In order to sensibly compare our results with published performance figures, we examine here the characteristics of these two algorithms and the effect on the performance of changes in problem size.

MXM in its standard form with square matrices of rank N requires $2N^3$ flops, $2N^2$ reads, and N^2 writes, so there are $3N^2$ data moves. There are other algorithms, of which the best known is Strassen's method, which result in better scaling of the numbers of flops with matrix size. Other techniques for small matrix multiplication are discussed in [49]. For eight-byte double-precision elements, the computational intensity (the ratio of computation to communication) is $N/12$ flops/byte. Thus, as the size of the matrices increases, the computational intensity increases and problems with data access and data movement become much reduced. A matrix as small as 12×12 gives a computational intensity of one flop/byte, and most papers report performance for matrices much larger than this.

For MVM, however, the situation is different: both flops and bytes scale as $N^2 : 2N^2$ flops and $2N + N^2$ data moves, leading to a computational intensity of $2N/(8(2 + N))$ flops/byte. For a large N, this asymptotically approaches 0.25 flops/byte. Thus, even in the best case, we have to transfer four bytes for every floating point operation.

Therefore, although it is tempting to compare our MVM results with the performance of MXM, it should be remembered that MXM is much more computationally intensive and that MVM performance is always going to be dominated by data transfer costs.

We referred earlier (Section 1) to published performance for matrix-matrix multiplication on FPGAs. Dou et al. reported "reaching a performance of 15.6 Gflop/s," e.g., [18]; however, closer inspection reveals that this is a peak performance calculated as 2 flops/cycle × 200 MHz × 39 PEs = 15.6 Gflop/s. Jovanović and Milutinović wrote about "achieving 203.1 Gflop/s" [20], but this is also a calculated or simulated figure obtained from (2 flops/cycle × 252 PEs − 1) × 403.87 MHz = 203.1 Gflop/s. Kumar et al. claimed that "a sustained performance of 29.8 Gflop/s is possible." Sustained performance usually refers to an actual measurement as opposed to peak performance, but in this case, the performance is also obtained from 2 flops/cycle × 40 PEs × 373 MHz = 29.8 Gflop/s. It would appear that many authors are content to report calculated or peak performance rather than real measurements. The equivalent calculated performance in our case is 2 flops/cycle × 12 PEs × 333 MHz = 8.0 Gflop/s, of which 5.34 Gflop/s is measured. The sources of the overheads which cause this reduction in achieved performance were discussed in Section 6.1.

A final comparison may be made with the paper by Bosch et al. which uses the OmpSs programming model [13]. This model offers a high-level directive-based programming approach which is easier to use than our approach but ultimately uses the Vivado toolset, including Vivado HLS, under the hood. Measured performance is reported for single-precision matrix-matrix multiplication using a blocking procedure which splits larger matrices into blocks of size 128 × 128. The best performance figure reported is 13.2 Gflop/s noting that the OmpSs parallel model utilizes four ARM CPU cores as well as the FPGA. Given that this is single precision compared to our double-precision multiplication and that this is matrix-matrix multiplication compared to our much less computationally efficient matrix-vector multiplication, we believe this sets our result in a good light.

6.6. Prospects for the Weather Forecast Model. We have established a methodology for implementing the matrix-vector multiplication kernel on FPGAs. This methodology is general and extensible so that similar benefits can be brought to other BLAS kernels and to other kernels within the LFRic weather and climate code. In current and future work, we are proceeding to apply these techniques to other LFRic kernels and investigating the effect on the performance of the full LFRic code.

We note that the programming model described here is rather programmer unfriendly requiring some low-level concerns, including address manipulation, setting and examining start and stop bits, and setting widths of data paths. However, as described in Section 3.1, LFRic uses the separation of concerns to ensure that scientists do not have to concern about themselves with parallel, platform-dependent coding. In the future, in order to fully support an FPGA port using this methodology, the PSyclone system could be modified to support automatic generation of the FPGA-dependent code.

LFRic can be run in many configurations representing a range of weather and climate scenarios at low, medium, and high resolutions. The profiling of the baroclinic test case has been shown in Section 3.2. The Helmholtz kernel, apply_hx_variable_code, will now be offloaded to the FPGA. It consists of a series of matrix-vector multiplications and ancillary calculations on six input variables. The only difference in our methodology compared with the matrix-vector kernel is that this time we will write the ARM code in standard Fortran rather than C, in order to fit better with the LFRic programming model. This kernel, together with the matrix-vector kernel, will be implemented in a design with twelve IP blocks capable of running independently, thus exploiting spatial parallelism on the FPGA.

A key issue for the LFRic code in exploiting the acceleration potential of the FPGA, with any accelerator, is reducing the overhead of transferring data between the host CPU and the FPGA. Thus, it makes little sense to look at the performance of one small kernel in isolation where that performance will be dominated by data transfer costs. We need to port a full workflow consisting of a sequence of kernels so that key data structures exist on the FPGA for long periods and ideally are created and used entirely in FPGA memory.

LFRic uses data decomposition across parallel multinode clusters with halo exchanges between subdomains carried out using MPI. A part of a workflow may therefore be represented as follows:

(i) Kernel 1
(ii) Halo exchange for variable x1
(iii) Kernel 2
(iv) Halo exchange for variable x2
(v) Kernel 3
(vi) Halo exchange for variable x3

In offloading a whole workflow, it is therefore essential to take into account the MPI communications required for halo exchange. Initially, the halo exchange will be carried out between host CPUs with data transferred to and from the FPGAs. We note that the amount of data involved for halo exchange is much smaller than the entire data arrays as only boundary data need to be transferred. As a further optimization step, MPI communications will be available directly from FPGA to FPGA, using communication libraries under development in the EuroExa project.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Disclosure

A part of this work was presented as a conference abstract at the Fourth International Workshop on Heterogeneous High-Performance Reconfigurable Computing (H2RC'18) held at SC18 on 11 November 2018 [50].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was carried out with support from the EuroExa project (grant agreement no. 754337), funded by the European Union's Horizon 2020 Research and Innovation Programme.

References

[1] K. Bergman, S. Borkar, D. Campbell et al., "Exascale computing study: technology challenges in achieving exascale systems," Tech. Rep., Defense Advanced Research Projects

Agency Information Processing Techniques Office (DARPA IPTO), Washington, DC, USA, 2008.

[2] N. Trifunovic, V. Milutinovic, J. Salom, and A. Kos, "Paradigm shift in big data supercomputing: dataflow vs. controlflow," *Journal of Big Data*, vol. 2, no. 1, p. 4, 2015.

[3] N. Hemsoth and T. P. Morgan, *FPGA Frontiers: New Applications in Reconfigurable Computing*, Next Platform Press, San Francisco, CA, USA, 2017.

[4] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[5] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of FPGAs for heterogeneous platforms in HPC," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2016.

[6] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 93–96, IEEE, Boulder, CO, USA, April 2018.

[7] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[8] R. Nane, V.-M. Sima, C. Pilato et al., "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[9] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*, Springer, Berlin, Germany, 2016.

[10] C. Y. Lin, Z. Jiang, C. Fu, H. K.-H. So, and H. Yang, "FPGA high-level synthesis versus overlay," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 92–97, 2017.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: from prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[12] S. Churiwala and I. Hyderabad, *Designing with Xilinx® FPGAs*, Springer International Publishing, Basel, Switzerland, 2017.

[13] J. Bosch, A. Filgueras, M. Vidal, D. Jimenez-Gonzalez, C. Alvarez, and X. Martorell, "Exploiting parallelism on GPUs and FPGAs with OmpSs," in *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*, p. 4, ACM, Portland, OR, USA, September 2017.

[14] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: a framework for directive-based high-performance reconfigurable computing," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, pp. 544–554, IEEE, Chicago, IL, USA, May 2016.

[15] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 18–32, 1988.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.

[18] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pp. 86–95, ACM, Monterey, CA, USA, February 2005.

[19] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan, "FPGA based high performance double-precision matrix multiplication," *International Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 322–338, 2010.

[20] Ž. Jovanović and V. Milutinović, "FPGA accelerator for floating-point matrix multiplication," *IET Computers & Digital Techniques*, vol. 6, no. 4, pp. 249–256, 2012.

[21] D. Zou, Y. Dou, S. Guo, and S. Ni, "High performance sparse matrix-vector multiplication on FPGA," *IEICE Electronics Express*, vol. 10, no. 17, article 20130529, 2013.

[22] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance mixed-precision linear solver for FPGAs," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1614–1623, 2008.

[23] Y. Durand, P. M. Carpenter, S. Adami et al., "Euroserver: energy efficient node for european micro-servers," in *Proceedings of the 2014 17th Euromicro Conference on Digital System Design*, pp. 206–213, IEEE, Verona, Italy, August 2014.

[24] S. V. Adams, R. W. Ford, M. Hambley et al., "LFRic: meeting the challenges of scalability and performance portability in weather and climate models," 2018, http://arxiv.org/abs/1809.07267.

[25] Xilinx Inc., "Zynq Ultrascale+ MPSoC," February 2019, https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[26] Maxeler Technologies, "MaxCompilerMPT," February 2019, https://www.maxeler.com/solutions/low-latency/maxcompilermpt/.

[27] O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 4, pp. 60–65, 2011.

[28] J. Bosch, A. Filgueras, M. Vidal, D. Jimenez-Gonzalez, C. Alvarez, and X. Martorell, "Exploiting parallelism on GPUs and FPGAs with OmpSs," in *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems (ANDARE'2017)*, Portland, OR, USA, September 2017.

[29] A. Pop and A. Cohen, "OpenStream: expressiveness and dataflow compilation of OpenMP streaming programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–25, 2013.

[30] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "SDSoC: a higher-level programming environment for Zynq SoC and Ultrascale+ MPSoC," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 4, ACM, Monterey, CA, USA, February 2016.

[31] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[32] Feist, T., Vivado design suite, White Paper 5, 2012.

[33] A. Staniforth, T. Melvin, and N. Wood, "Gungho! a new dynamical core for the unified model," in *Proceedings of the ECMWF Workshop on Recent Developments in Numerical*

Methods for Atmosphere and Ocean Modelling, Reading, PA, USA, September 2013.

[34] 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," *Journal of Advances in Modeling Earth Systems*, vol. 7, no. 3, pp. 1393–1408, 2015.

[35] Xilinx Inc., *Vivado Design Suite User Guide: Getting Started, UG910 (v2017.2)*, Xilinx Inc., San Jose, CA, USA, 2017.

[36] Xilinx Inc., *Vivado Design Suite User Guide: High-Level Synthesis, UG902 (v2017.2)*, Xilinx Inc., San Jose, CA, USA, 2017.

[37] US Fortran Standards Committee, "J3/04-007 working draft," May 2004, https://j3-fortran.org/doc/year/04/04-007.pdf.

[38] J. de Fine Licht, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," 2018, http://arxiv.org/abs/1805.08288.

[39] Xilinx Inc., *Vivado Design Suite User Guide: Designing with IP, UG896 (v2017.2)*, Xilinx Inc., San Jose, CA, USA, 2017.

[40] Xilinx Inc., *Vivado Design Suite User Guide: Design Analysis and Closure Techniques, UG906 (v2017.2)*, Xilinx Inc., San Jose, CA, USA, 2017.

[41] Xilinx Inc., *Zynq UltraScale+ MPSoC Software Developer Guide, UG1137 (v8.0)*, Xilinx Inc., San Jose, CA, USA, 2018.

[42] Xilinx Inc., *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator, UG994 (v2017.2)*, Xilinx Inc., San Jose, CA, USA, 2017.

[43] Xilinx Inc., *Processor Control of Vivado HLS Designs, Application Note XAPP745 (v1.0)*, Xilinx Inc., San Jose, CA, USA, 2012.

[44] Xilinx Inc., *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics, DS925 (v1.14)*, Xilinx Inc., San Jose, CA, USA, 2018.

[45] M. Parker, "Understanding peak floating-point performance claims," Technical White Paper WP-1.0, Intel Corporation, Santa Clara, CA, USA, 2014.

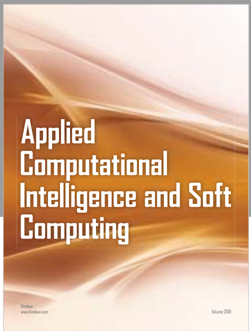
[46] A. Khayyat and N. Manjikian, "Analysis of blocking and scheduling for FPGA-based floating-point matrix multiplication analyse du blocage et de l'ordonnancement d'une multiplication matricielle à virgule flottante sur un FPGA," *Canadian Journal of Electrical and Computer Engineering*, vol. 37, no. 2, pp. 65–75, 2014.

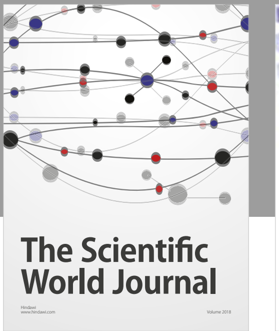
[47] Berton DSP S. L., GPU vs FPGA Performance Comparison, Berton White Paper BWP001 v1.0, 2019, http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf.

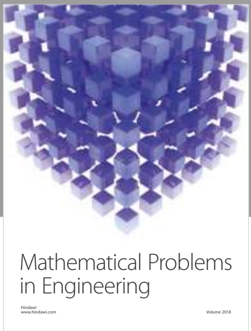
[48] J. Cong, Z. Fang, Y. Hao et al., "Best-effort FPGA programming: a few steps can go a long way," 2018, http://arxiv.org/abs/1807.01340.

[49] C.-É. Drevet, M. Nazrul Islam, and É. Schost, "Optimization techniques for small matrix multiplication," *Theoretical Computer Science*, vol. 412, no. 22, pp. 2219–2236, 2011.

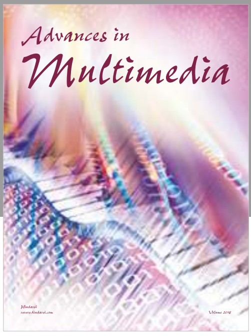
[50] M. Ashworth, A. Attwood, G. D. Riley, and J. Mawer, "First steps in porting the LFRic weather and climate model to the FPGAs of the EuroExa architecture," in *Fourth International Workshop on heterogeneous high-performance reconfigurable computing (H2RC'18)*.

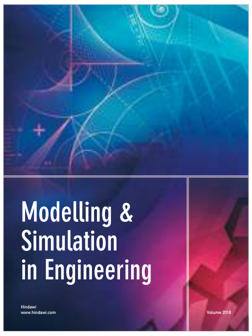

Applied Computational Intelligence and Soft Computing

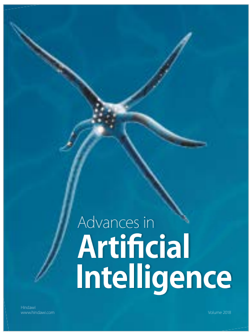

The Scientific World Journal


Mathematical Problems in Engineering

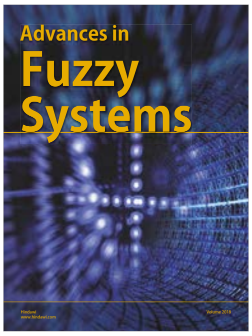
Journal of Engineering

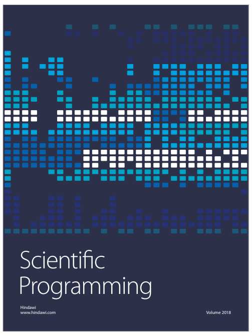

Advances in Multimedia


Modelling & Simulation in Engineering

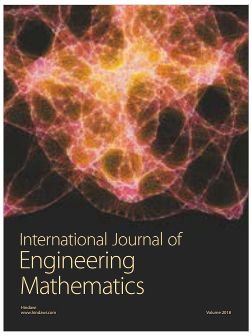

Advances in Artificial Intelligence

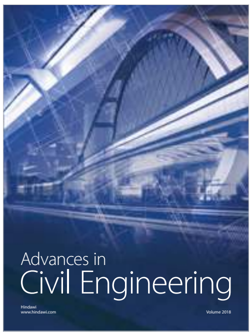
International Journal of Reconfigurable Computing

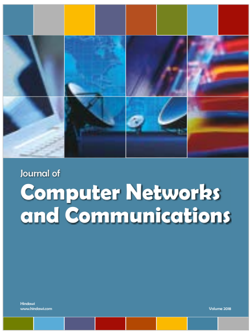

Advances in Fuzzy Systems


Scientific Programming

Advances in Human-Computer Interaction


International Journal of Engineering Mathematics


Advances in Civil Engineering

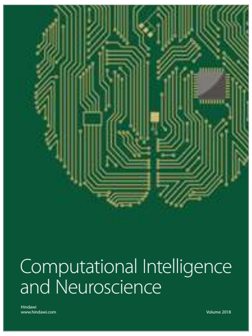

Journal of Computer Networks and Communications

Journal of Robotics

International Journal of Computer Games Technology

International Journal of Biomedical Imaging

Journal of Electrical and Computer Engineering


Computational Intelligence and Neuroscience