International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Special Issue Paper

CFD code adaptation to the FPGA architecture

The International Journal of High Performance Computing Applications
2021, Vol. 35(1) 33–46

DOI: 10.1177/1094342020972461
journals.sagepub.com/home/hpc
SAGE

Krzysztof Rojek, Kamil Halbiniak and Lukasz Kuczynski

Abstract

For the last years, we observe the intensive development of accelerated computing platforms. Although current trends indicate a well-established position of GPU devices in the HPC environment, FPGA (Field-Programmable Gate Array) aspires to be an alternative solution to offload the CPU computation. This paper presents a systematic adaptation of four various CFD (Computational Fluids Dynamic) kernels to the Xilinx Alveo U250 FPGA. The goal of this paper is to investigate the potential of the FPGA architecture as the future infrastructure able to provide the most complex numerical simulations in the area of fluid flow modeling. The selected kernels are customized to a real-scientific scenario, compatible with the EULAG (Eulerian/semi-Lagrangian) fluid solver. The solver is used to simulate thermo-fluid flows across a wide range of scales and is extensively used in numerical weather prediction. The proposed adaptation is focused on the analysis of the strengths and weaknesses of the FPGA accelerator, considering performance and energy efficiency. The proposed adaptation is compared with a CPU implementation that was strongly optimized to provide realistic and objective benchmarks. The performance results are compared with a set of server CPUs containing various Intel generations, including Intel SkyLake-based CPUs as Xeon Gold 6148 and Xeon Platinum 8168, as well as Intel Xeon E5-2695 CPU based on the IvyBridge architecture. Since all the kernels belong to the group of memory-bound algorithms, our main challenge is to saturate global memory bandwidth and provide data locality with the intensive BRAM (Block RAM) reusing. Our adaptation allows us to reduce the performance per watt up to 80% compared to the CPUs.

Keywords

CFD, FPGA, energy efficiency, parallel computing, code adaptation, numerical weather prediction

1. Introduction

FPGA devices have proven achievements in many research and business areas, including cryptography, network routing algorithms, machine, deep learning and video processing (Xilinx Intellectual Property, 2020). The new challenge is to indicate the usability of FPGA for a new area such as CFD (Alveo Accelerator Card Applications, 2020).

The goal of this paper is to design and develop a proof of concept that allows users to evaluate the performance and energy efficiency of FPGA cards for CFD codes. CFD is an important branch of fluid dynamics (Xiang et al., 2017). It applies various kinds of discrete mathematical methods to analyze and simulate problems in fluid mechanics with the use of a computing machine. CFD facilitates the researchers to quantify and predict the effects of heat flow, mass transfer, phase change, chemical reaction, mechanical movements, and stresses in the displacement of solids (Zhai, 2019). Moreover, it enables the building service engineers and architects to offer contented and safe human environments, power-plant designers to attain maximum efficiency and reduce the release of pollutants. It also supports chemical engineers to maximize the yields from reactors and processing equipment, land-air and marine vehicle designers to achieve maximum performance at low cost and low risk-and-hazard, analysts and safety engineers to predict the amount of damage (to structures, equipment, human beings, animals and vegetation) by natural or human-made disasters (fires, explosions, and blast waves). Besides, the CFD simulation also allows the metropolitan authorities, meteorologists, and oceanographers, as well as petroleum engineers to detect and predict the fluid dynamics in their field.

A lot of current benchmarks examines general-purpose solutions that require very complex customization to be applied in the target solvers. It makes many promising

Czestochowa University of Technology, Czestochowa, Poland

Corresponding author:
Krzysztof Rojek, Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland.
Email: krojek@icis.pcz.pl

results disappointing when the solution is adapted to the final use-case. In this work, we focus on the development of fully functional kernels in a real-life scenario. The advantage of the proposed approach is to support a scientific scenario that is not only a solution, adjusted to the computing architecture but also a software compatible with the EULAG (Rojek, 2018) geophysical model.

Our research is focused on stencil algorithms that are known from its memory-bound nature. In our research, we unveil the power of one of the most advanced FPGA cards, which is Xilinx Alveo U250 FPGA (appeared in October 2018) (Xilinx—Adaptable, Inteligent, 2020). Our goal is to provide highly efficient algorithms that can be simply deployed to weather forecast models without the need of rebuilding our kernels. For this reason, we take into account not only a basic version of an algorithm but also its boundary conditions, considering the computational domain as a part of a much bigger computing model and provide all the required, user-flexible configurations that are not always well adaptable to the architecture.

EULAG (Rojek et al., 2017a) is an established computational model developed by the group headed by Piotr Smolarkiewicz for simulating thermo-fluid flows across a wide range of scales and physical scenarios, such as numerical weather and climate prediction, simulation of urban flows, areas of turbulence, and ocean currents, etc. EULAG is a representative of the class of anelastic hydrodynamic models. The dynamic core of the EULAG model consists of two parts, including the second-order step of the non-linear iterative upwind advection scheme (non-oscillatory forward in time) called MPDATA (Multidimensional Positive Definite Advection Transport Algorithm) and advanced elliptic solver GCR (Generalized Conjugate Residual) (Prusa et al., 2008). Recently, the dynamical core of EULAG has been implemented into COSMO (Consortium for Small-scale Modeling) weather prediction framework and is expected to be in operational use (Cosmo Public area, 2020).

Our work is focused on 4 CFD kernels, including (i) the first-order step of the non-linear iterative upwind advection scheme (part of MPDATA), (ii) computation of the psuedovelocity for the second pass of the upwind algorithm in MPDATA, (iii) the divergence part of the matrix-free linear operator formulation in the iterative Krylov scheme (part of GCR), and (iv) tridiagonal Thomas algorithm for vertical matrix inversion inside preconditioner for the iterative solver (part of GCR). All the kernels use a 3D compute domain consisted of 7 to 11 arrays. The original contribution of this work includes:

- systematic adaptation of four CFD kernels to the FPGA architecture, where all of them are customized to use in a real-scientific scenario, compatible with the EULAG solver;
- performance and energy efficiency benchmarks of the FPGA and CPU platforms based on a group of kernels characterized by different complexity and parallel strategy that classifies algorithms into less and more adaptable to FPGA based on their structure;
- investigation of strengths and weaknesses of the Xilinx Alveo U250 FPGA as a future candidate to accelerate current HPC codes.

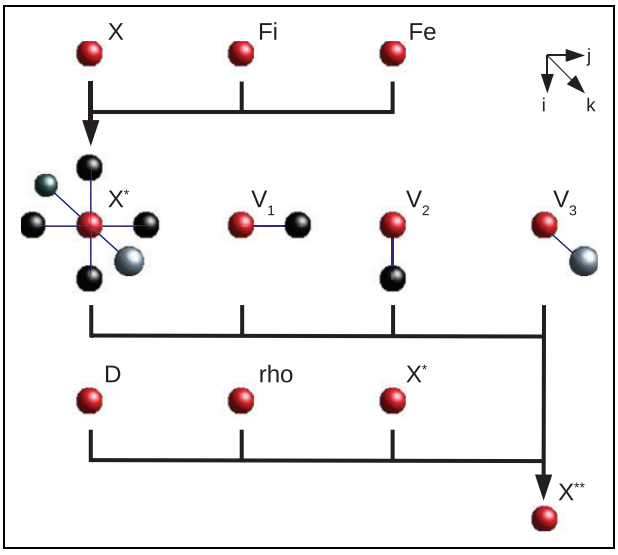

Figure 1. Data dependency of the advection algorithm.

2. CFD kernels

The MPDATA and GCR parts have different characteristics in terms of parallelization, scalability, and adaptability (Iserte and Rojek, 2020). For this reason, we investigate four kernels, where two of them are part of MPDATA and two are part of GCR. To provide the compatibility with EULAG, the general-purpose scheme of advection or GCR has been extended by additional quantities as forces acting on transported substance (implosion and explosion) and density vectors of the substance. Also, they allow users to fully configure border conditions in two scenarios, including periodic or open. All the kernels operate on the structured grid.

2.1. Advection

The first kernel represents the first-order-accurate step of the advection scheme (Smolarkiewicz, 2006). The advection algorithm is responsible for simulating the movement of some material, dissolved or suspended in the fluid. The algorithm flow with its data dependency graph is shown in Figure 1. The code snippet of this kernel is shown in Listing 1. This code is responsible for computing a donor-cell scheme and is used both in the advection, as well as in pseudovelocity. The presented implementation takes eight input arrays, including X— nondiffusive quantity (for example, temperature of water vapor, ice, precipitation, etc.), V_1, V_2, V_3—each of them stores the velocity vectors in one direction: i, j, and k, respectively, F_i, F_e—implosion and explosion forces acting on a structure of X, D corresponds

Listing 1. The code snippet of the donor-cell scheme.

```
float donor(float y1, float y2, float a)
{ return max(0.0f, a)*y1 -
       (-min(0.0f, a)*y2);
} // (...)
for(int i=0; i<np; ++i){
 for(int k=0; k<lp; ++k){
  for(int j=0; j<mp; ++j){
    const realX f1ijkp=
    donor(xIn(i  ,j,k), xIn(i+1,j,k),
          v1(i+1,j,k));
    const realX f1ijk =
    donor(xIn(i-1,j,k), xIn(i  ,j,k),
          v1(i  ,j,k));
    const realX f2ijkp=
    donor(xIn(i,j  ,k), xIn(i,j+1,k),
          v2(i,j+1,k));
    const realX f2ijk =
    donor(xIn(i,j-1,k), xIn(i,j  ,k),
          v2(i,j  ,k));
    const realX f3ijkp=
    donor(xIn(i,j,k  ), xIn(i,j,k+1),
          v3(i,j,k+1));
    const realX f3ijk =
    donor(xIn(i,j,k-1), xIn(i,j,k  ),
          v3(i,j,k  ));
    xOut(i,j,k)=rho(i,j,k)*
      (xIn(i,j,k)-
      ( f1ijkp-f1ijk+f2ijkp-
        f2ijk+f3ijkp-f3ijk )
      /D(i,j,k));
  } } }
```

to the density, *rho* defines an interface for the coupling of COSMO and EULAG dynamic core and is used to provide the transformation of the X variable. The kernel returns a single X^{**} array that stores values of the X field updated in the current time step.

2.2. Pseudovelocity

The second kernel is responsible for computing the psuedovelocity for the second pass of the upwind algorithm. It returns an approximation of the relative velocity and is required to provide the second-order-accurate advection (Rojek and Wyrzykowski, 2017, Smolarkiewicz et al., 2014). The data dependency graph of this algorithm is shown in Figure 2. This kernel is the most compute-intensive from all others considered in this work. It operates on V_1, V_2, V_3, D, X^{**} input arrays and returns pseudovelocities F_1, F_2, and F_3 in i, j, and k dimension, respectively.

2.3. Divergence

The third kernel is responsible for computing the divergence part of the matrix-free linear operator formulation in the iterative Krylov scheme (Smolarkiewicz et al., 2014).

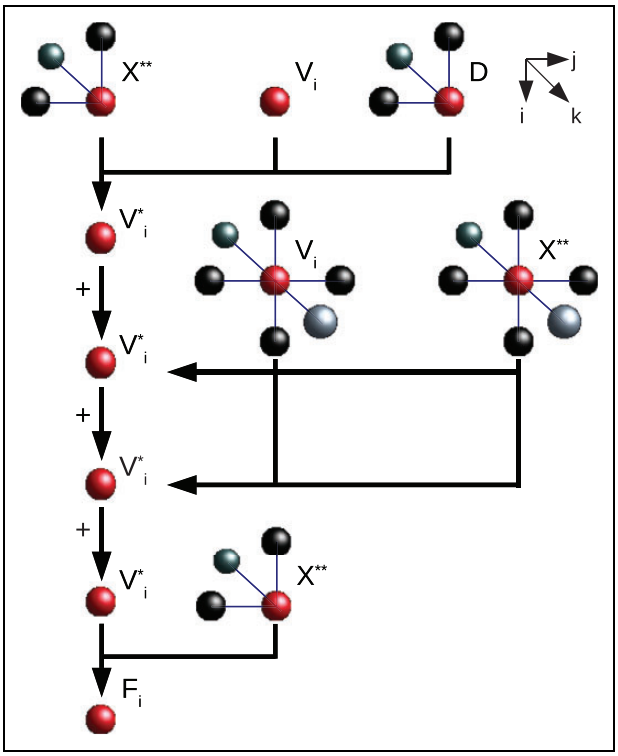

Figure 2. Data dependency of the pseudovelocity algorithm, where i is from 1 to 3.

This kernel measures the amount of fluid required to flow in/out of a certain point in a vector field. The output r represents the divergence. The data dependency graph of this algorithm is shown in Figure 3. This Kernel is the most memory-bound, since it takes nine input arrays, returns one, and executes relatively a small number of computation. To provide the capability of this kernel with the EULAG model, we need to extend a basic divergence scheme. Here the kernel takes as input *rho*, the same as in the advection scheme, $g1$, $g2$, $g3$—arrays of the gradient in i, j, and k dimension of the compute domain, *pex*, $pexr0-1$, *pexc*—arrays of pressure (see more details in Smolarkiewicz et al., 2014), *helmc*—evaluations of the Helmholtz operator.

2.4. Thomas algorithm

An important part of the GCR solver is the preconditioner that accelerates the convergence of the variational scheme (Ciznicki et al., 2014). The preconditioner employs the sequential Thomas algorithm to solve tridiagonal systems of equations with the right-hand side consisting of the vertical divergence of the generalized vertical gradient (Piotrowski et al., 2016). The Thomas algorithm is a simplified form of the Gaussian elimination algorithm for the tridiagonal system of equations. The preconditioner operates on the diagonal part of the full linear problem. Effective preconditioning lies at the heart of multiscale flow simulation, including a broad range of geophysical applications. This kernel is the most difficult to parallelize since

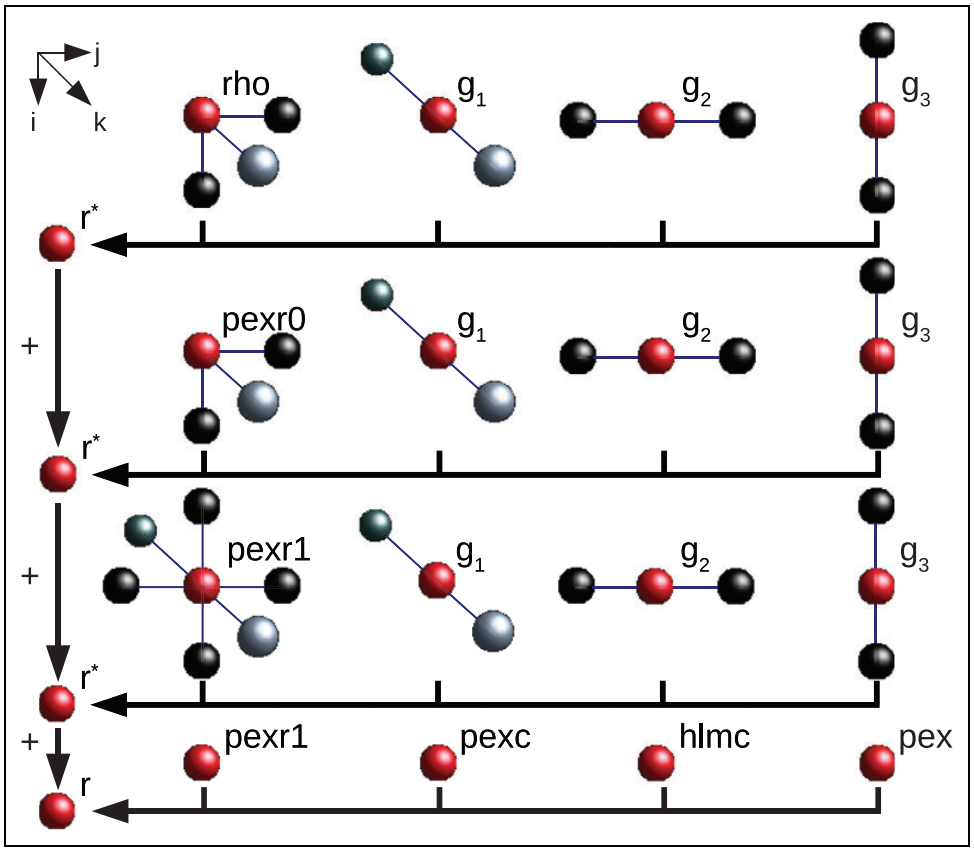

Figure 3. Data dependency of the divergence algorithm.

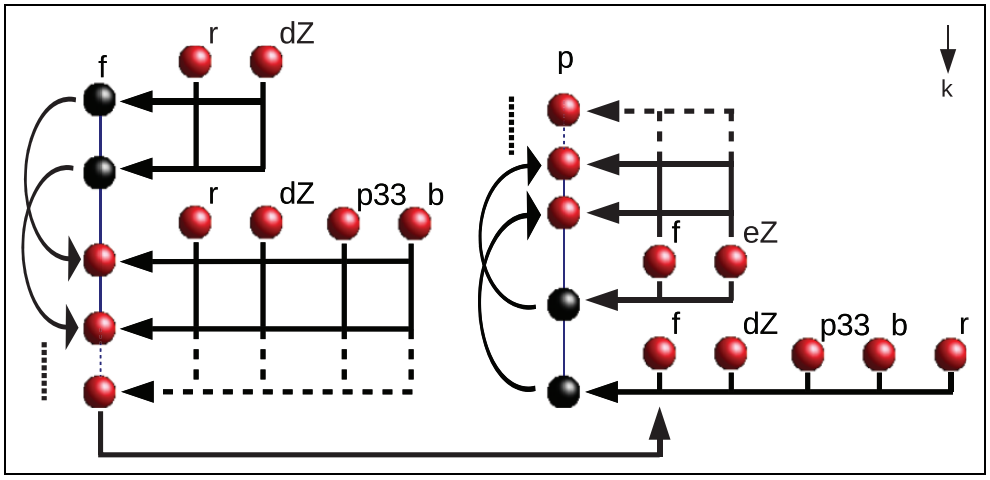

Figure 4. Data dependency of the Thomas algorithm.

it consists of two steps, where every step has data dependencies between iterations of itself. The implementation is consistent with the GCR solver adapted to the EULAG model that is based on the vertical version of the Thomas algorithm. The data dependency graph of this algorithm is shown in Figure 4. Here *f*, *dZ*, *p*33, represent diagonals in the first step, *rho*, and *b* are extensions provided for the compatibility with the COSMO and EULAG models, *eZ*, *f* are used in the second step as a diagonal part of array together with *p* that is also the output array. More details about the Thomas algorithm in the context of the EULAG model are in Piotrowski et al. (2016).

3. Architecture of the Xilinx Alveo U250 FPGA and programming environment

The Xilinx Alveo U250 FPGA is built on the 16 nm UltraScale architecture (SDAccel Environment Programmers

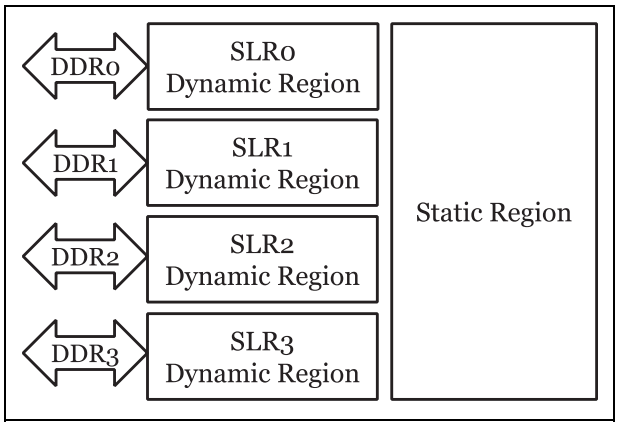

Figure 5. Architecture of the Xilinx Alveo U250 FPGA.

Guide, 2019). The key device components that require special attention during the adaptation process are described below. The off-chip memory is organized into four DDR4 memory banks, each of size 16 GB (64 GB of off-chip memory capacity), connected to one from four Super Logic Regions (SLRs). All the SLRs consist of 1341K Look-Up Tables (LUTs) responsible for arithmetic and logic operations, 2749K registers and 2K blocks of RAM (BRAM), each of size 36 KB. Four SLRs with memory banks create the dynamic region available for creating custom accelerators. The clock frequency can vary from 60 MHz to 300 MHz depending on the instruction flow. The off-chip peak memory bandwidth is 77 GB/s (about 47 GB/s in practice—see section devoted to the platform description for more details). The remaining part of the FPGA card includes the static region containing a deployment shell that handles device bring-up and configuration over Peripheral Component Interconnect Express (PCIe). The thermal design power (TDP) of this card is 225 W. However, it can be significantly reduced, especially for memory-bound algorithms. The architecture of the Alveo U250 is shown in Figure 5.

As a development environment, we selected an Open Computing Language (OpenCL) framework (OpenCL Overview, 2020), as the most portable across different platforms that allows us to provide future extensions of our code onto a GPU architecture. The OpenCL standard distinguishes the global memory which is shared by all SLRs and is represented by the DDR4 memory banks of the Alveo card. The kernels are processed within SLRs that in OpenCL are the equivalents of compute units. Data stored in the global memory are persistent between all kernel calls. The next storage is a local memory that is private to each compute unit and is represented by the BRAM memory of FPGA. Finally, private memory is local to each thread created within a compute unit and is represented by the register file. OpenCL applications run on the Host, which submit work to the compute devices. The algorithm processed by the compute device is called a kernel. Each kernel is managed by the host layer that submits work (kernels, memory copies) to the FPGA. The submissions within a single queue are executed synchronously and can be overlapped with the submissions managed by the other queue. In contrast to the GPU programming model, here we create a single thread per each compute unit. The number of compute units corresponds to the number of SLRs.

4. Systematic adaptation of the kernels to FPGA

Alveo is an accelerator card designed to meet the constantly changing needs of the modern data center, supporting any workload type while reducing the overall cost of ownership. To confirm the designer's assumptions, it is important to verify not only performance and power metrics but also validate an attainable performance, platform limitations and estimate adaptive possibilities of the kernels to this architecture.

Our implementation is based on the Xilinx SDAccel Development Environment (SDAccel Environment Programmers Guide, 2019) that provides a framework for developing and delivering FPGA-accelerated data center applications using standard programming. At this moment, there are some limitations to the OpenCL standard and some extensions available in SDAccel. The allocation of the UltraRAM memory is not supported. We are not able to use vector data types of type double(n), nor half-precision. For this reason, we process the data using single-precision arithmetic with vector data types of size 16 (float16). The extension of OpenCL defines the blocked version of a new memory object called a pipe. The pipe stores data organized as a FIFO. Pipes can be used to stream data from one kernel to another inside the FPGA device without having to use the external memory, which greatly improves the overall system latency. To update the data between the memory banks it is required to exchange halo areas (borders of sub-domains) between neighboring sub-domains. For this purpose, we utilized the pipe object. The limitation of this object is that pipe must be statically defined outside of all kernel functions and must be declared in lower case alphanumeric. In our version we use pipes of size 512, that can be defined in the following way:

*pipe int p*0 *__attribute__*((*xcl_reqd_pipe_depth*(512)));

All the kernels use a 3D compute domain consisted of 7 (Thomas) to 11 (pseudovelocity) arrays. The kernels computations are performed with a stencil fashion (to compute a single element of computing domain it is required to access neighboring elements). Since all the kernels belong to the group of memory-bound algorithms, our main challenge is to provide the highest utilization of the global memory bandwidth. For this reason, we design here a set of methods that allow us to efficiently map the kernels onto FPGA, including:

- design of the management layer that is executed on a single CPU core;
- data access optimizations based on 2.5-dimensional blocking (Rojek et al., 2017b) to improve data

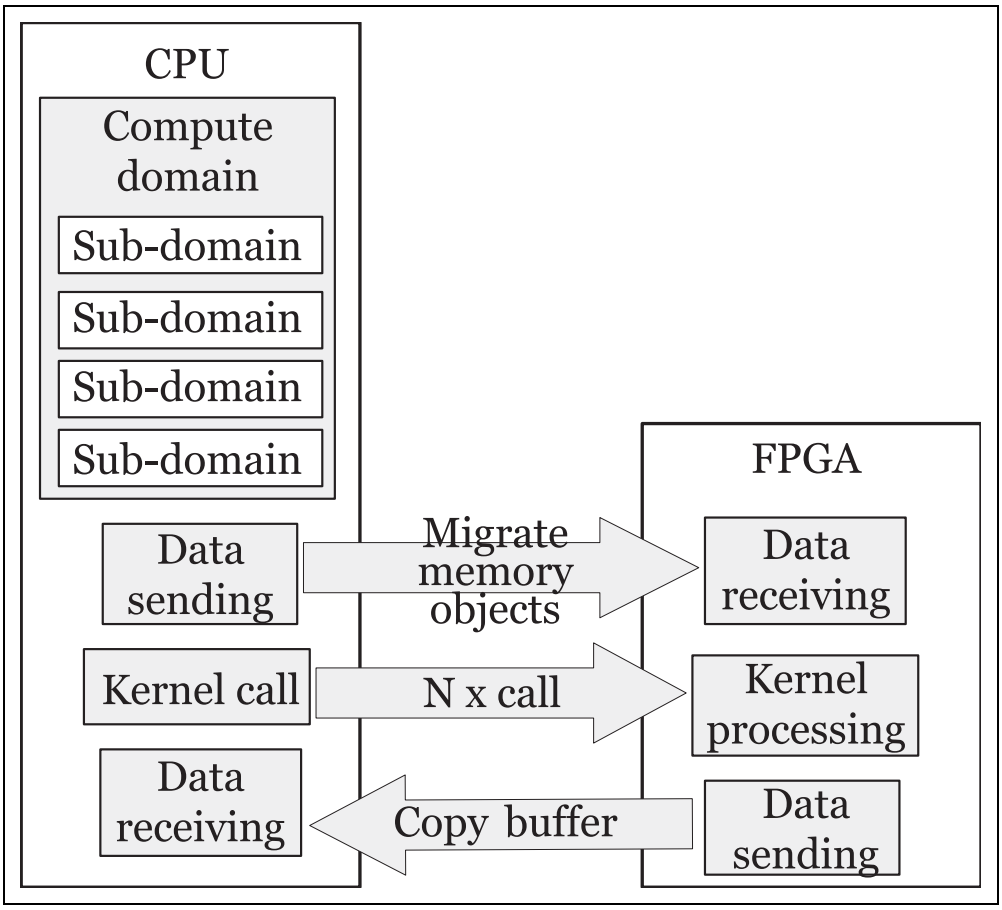

Figure 6. Design of the management layer.

locality and minimize data transfer between the global memory and BRAM;
- utilization of the burst memory access to hide the memory access latency and improve bandwidth usage, as well as efficiency of the memory controller;
- reduction of memory pins—31 available handlers to the global memory.

4.1. Design of the management layer

The process of the adaptation begins with design the management part of the application that is based on the same engine for each kernel. Here we create two layers, including the host layer designed for a single CPU-core execution and the device layer for a kernel execution. Our basic implementation supports a memory allocation with the Xilinx recommendations, where all arrays are 4 KB aligned. We regularly flush the command queue and use the OpenCL method called Migrate memory object to send a set of arrays from the host memory to the FPGA global memory. To receive a single array from the device we use a copy buffer.

The Alveo U250 FPGA consists of four global memory banks, where each of them is connected to a single SLR. To address this design the compute domain is divided into four sub-domains, where each of them is assigned to a separate memory bank. Each kernel is distributed across four compute units assigned to a different SLR. In this way, the memory transfers between the global memory and compute units occurs only between connected pairs of SLR and memory bank. To exchange data between kernels we create six pipe objects, where a pair of them is used to data exchanging between the first and second kernel, another pair between second and third, and the last pair between third and fourth kernel. We use a pair of the pipe objects, where the first one is dedicated for sending, while the second one is dedicated for receiving the halo area. The management part of the application is shown in Figure 6, while the communication within the device layer is shown in Figure 7.

4.2. 2.5-Dimensional blocking

To minimize the global memory traffic we utilize a fast BRAM memory. The characteristic of stencil computation requires to access a single element of each array many times to compute a single cell. Since there is not enough memory space to store a reasonable size of 3D blocks of compute domain, we apply a 2.5D blocking technique (Rojek et al., 2017b) to provide data locality. For this purpose, we only store a small set of 2D planes for each array (see Figure 8) as a queue of planes. After each iteration that

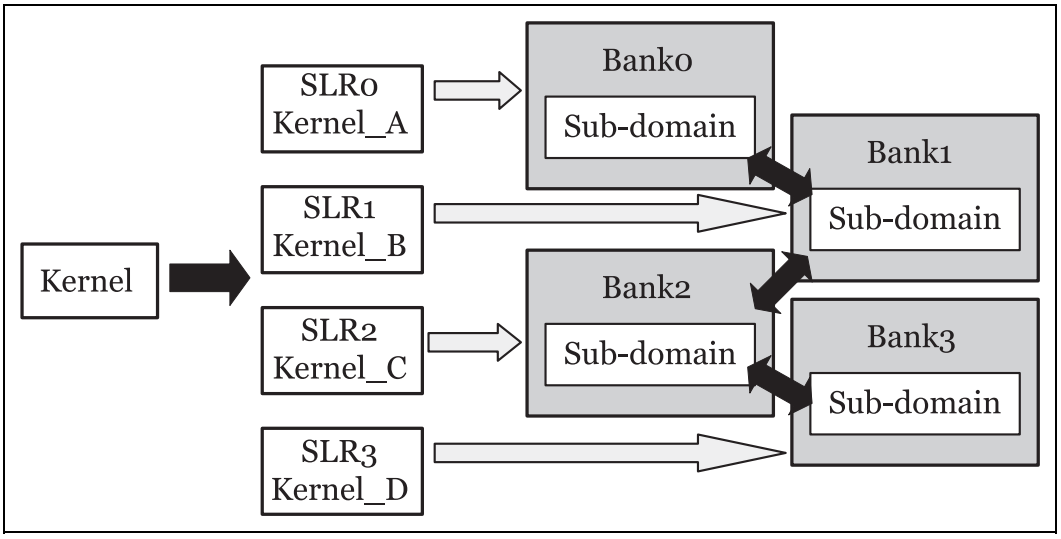

Figure 7. Communication within the kernel layer.

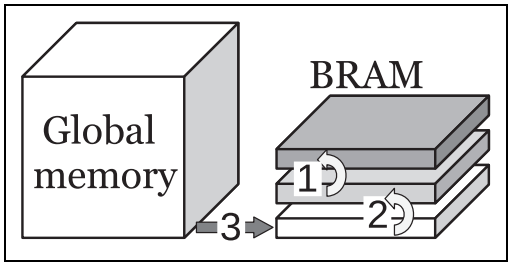

Figure 8. 2.5D blocking on BRAM: (i) 2D planes are migrated within the BRAM (step 1, 2); (ii) a single plane is downloaded from the global memory (step 3).

traverses the algorithm across the third dimension, only a single plane is downloaded from the global memory, while others migrate across the queue. In this way, the global memory traffic is significantly reduced, while the BRAM usage is increased.

Considering a stencil dependent on r top and bottom elements and a compute domain of height N, we can reduce the memory traffic $\frac{3 \cdot r \cdot N}{3 \cdot r + N - 1}$ times. When N is large and $r = 1$ the global memory traffic is reduced three times.

4.3. Burst memory access

To provide a high memory bandwidth it is required to use data access in a burst memory mode. For this reason, a few conditions need to be fulfilled: (i) loop pipelining, (ii) using 512-bit memory access, (iii) limit the number of data accesses per iteration.

The first optimization is to create a pipeline across iterations of a loop. It can be implemented with the optimization directive *__attribute__((xcl_pipeline_loop))*. This method is used to enhance the performance of the hardware function, allowing pipelining which substantially increases the performance of the function. The pipeline is responsible for processing a single iteration in each clock cycle. To ensure fully pipelining it is required to remove dependencies between the iterations and use up to two data accesses per iteration for each array.

Another optimization is to generate a 512-bit AXI4 memory interface for global memory access. For this purpose, we reorganize the computation to utilize vector data types of size 16. Since the kernels belong to a group of stencil algorithms, where a set of neighboring elements need to be accessed to compute a single output, we need to reorganize data by shifting them to the left or right within a 16-elements vector. We can not use here a standard OpenCL shuffle routine, because it is not supported by the Xilinx compiler. For this reason, we implemented our version of this instruction that shifts data by a single element to the left/right within the vector. The method of shifting elements to the left is shown in Listing 2. Here we move every element to its neighbor. To provide a parallelization for each iteration we use a full loop unrolling.

Listing 2. Shift elements by 1 to the left within a vector.

```
__attribute__((always_inline)) float16
shiftLeft(const float16 a,
          const float b) {
  const float *ptr1=(float*)&a;
  float16 out;
  float *o=(float*)&out;

  __attribute__
  ((opencl_unroll_hint(15)))
  for(int i=0; i<15; ++i)
    o[i]=ptr1[i+1];
  o[15]=b;
  return out;
}
```

The proposed routine is fully pipelined (each routine call can be started in every clock cycle) and takes two clock cycles only. The information of this timeline can be extracted from the Xilinx compiler that generates the following log generated for this routine.

```
Pipelining result : Target II = 1,
                    Final II = 1,
                    Depth = 2.
```

Here, the target, final, and depth parameters represent the number of clock cycles resulted from the dependency, latency between the iterations, and depth of the pipeline, respectively.

The disadvantage of BRAM is that they have a limited number of data ports, typically up to two. It means that the number of accesses per iteration needs to be limited up to two to fully utilize the pipelining. The right memory access can be ensured by using the array partition into smaller arrays, improving the data structure by providing more data ports and allowing a higher performance pipeline. For some codes, it is required to redesign them by splitting the loop into two or more, where each loop operates on the right number of memory ports. Thanks to an appropriate loop splitting, we achieved a fully pipelined instructions flow, where a depth of pipelines takes up to 92 clock cycles for different blocks of instructions and takes:

```
Pipelining result:
Target II = 1, Final II = 1, Depth = 14.
Target II = 1, Final II = 1, Depth = 22.
Target II = 1, Final II = 1, Depth = 14.
Target II = 1, Final II = 1, Depth = 14.
Target II = 1, Final II = 1, Depth = 14.
Target II = 1, Final II = 1, Depth = 3.
Target II = 1, Final II = 1, Depth = 92.
Target II = 1, Final II = 1, Depth = 22.
Target II = 1, Final II = 1, Depth = 22.
Target II = 1, Final II = 1, Depth = 73.
...
```

The depth parameter influences LUTs utilization. A big disproportion of its value can have an impact on low resources utilization. However, the loop splitting eliminates the pipeline conflict and allows us to provide the burst memory access for all the arrays. The execution time t expressed in the number of clock cycles required to compute a single loop can be estimated based on the following equation:

$$t = n \cdot f + d - f \tag{1}$$

Here, n is the number of iterations per loop, f is the parameter *final* of a pipeline, while d is depth of the pipeline. To reduce the impact of the pipeline depth on the execution time we use the loop blocking technique with a block of size 2048. In this way, the highest overhead of the pipeline depth (92 clock cycles) takes $2048/(2048 + 92 - 1) * 100 < 5\%$ of the execution time.

4.4. Reduction of memory pins

In the Alveo U250 FPGA, we have 31 available memory pins. The pins are reserved by pointers to the global memory used as kernel arguments. They are also reserved by the returned value of the kernel, and finally by the pipe objects used to exchange data between the kernels. In this way, we can not set from 7 to 11 input/output arrays by separate pointers. Assuming that we create 4 kernels and 6 pipe objects, the number of kernel arguments can not exceed 5. To solve this problem we packed a set of arrays into a single memory buffer. In this way, we reduce the number of memory pins. The code snippet of the pointers management on the buffer of the input arrays is shown in Listing. 3.

Listing 3. Snippet of code responsible for the memory management with pin reduced. The offset object is used to move to another array.

```
//Declaration of the function
void advection(__global float16 * u1,
               __global float16 * u2,
               __global float16 * u3,
               (...)
//Arguments list
#define argList __global float16 *buf
(...)
//Function call
advection(buf, buf+offset, buf+offset,
(...)
```

5. FPGA platform and testing methodology

The efficiency of the proposed adaptation is examined on a server equipped with a single Xilinx Alveo U250 FPGA card and two Ivy Bridge-based Intel Xeon E5-2695 CPUs (12 cores clocked at 2.4 GHz each). To provide all the hardware requirements for the development and deployment of the code, the server contains 500 GB of hard disk space and 80 GB of RAM. The runtime environment is based on the Ubuntu 16.04 OS, Xilinx runtime shell for the FPGA management, as well as Xilinx SDAccel Integrated Development Environment framework (version 2018.3).

In this work, we compare the performance of the FPGA card with a set of Intel Xeon CPUs, including:

- 1x and 2x Intel Xeon E5-2695-based Ivy Bridge CPUs;
- 1x Intel Xeon Gold 6148 CPU with Skylake microarchitecture (20 cores clocked at 2.4 GHz);
- 1x Intel Xeon Platinum 8168 CPU-based Skylake (24 cores clocked at 2.7 GHz).

To provide objective performance comparisons, we use highly tuned CPU versions of the CFD kernels. The CPU implementation is tuned concerning performance and energy consumption. However, we focused here on the

code optimizations, without methods of platform adaptation, such as Dynamic Voltage and Frequency Scaling (DVFS) (Rojek et al., 2017c). The CPU versions are implements using OpenMP (OpenMP—Home, 2020) to utilize all available cores of CPUs. Beside the parallelization, the CPU versions are optimized with a set of commonly used techniques, including:

- thread affinity to avoid thread migration across CPU cores;
- loop transformations to provide cache reusing and auto-vectorization;
- memory alignment for efficient data access;
- compiler optimizations.

In the CPU benchmarks, all the codes are compiled using the GNU G++ compiler (ver. 5.4.0).

The performance of all the studied CFD kernels is described using a throughput expressed in GB/s. The throughput represents the size of the output data returned in every time step of the simulation per second. To reduce the measurement error, all the experiments are repeated 10 times, and the median values are presented as the final result.

Beyond the performance, we have also investigated the power dissipation for the Alveo card, as well as all the considered CPUs. To compare the results obtained on the platforms we use the performance per watt metric. To measure power consumption for the FPGA accelerator we utilize the Xilinx Board Utility and Vivado Design Suite tools. At the same time, for the CPUs, the power dissipation is measured using the Intel RAPL (Running Average Power Limit) (Rojek et al., 2017a) infrastructure.

It should be noted that in the case of the FPGA, the power measurements include the power dissipation of the FPGA, the host memory, as well as the CPU, where the management layer is executed using a single CPU core. The CPU benchmarks are measured based on the power dissipation of all the CPU cores and host memory.

6. Experimental results

6.1. Benchmarking CFD codes

The first part of the performance examination is based on the estimation of the scalability of each kernel depending on the size of the compute domain. All the tests are executed for 5000 time steps with different sizes of the compute domain. The conclusion here is that the performance quickly converges to the maximum throughput for all the kernels except the Thomas algorithm. The difference between the performance achieved for a small compute domain and the big once's varying from 5% for pseudovelocity up to 50% for Thomas. The scalability of the kernels is shown in Figure 9.

The second part includes the comparison of performance results between all the platforms, including FPGA and CPUs. In this test, we configure the kernels with the compute domain size of $1020 \times 512 \times 64$ and 5000 time steps. The results are shown in Figure 10. The performance of advection and pseudovelocity kernels is the highest on the FPGA card. The structure of those algorithms allows programmers to intensively reuse BRAM, which reduces global memory traffic. The different situation is in the divergence algorithm, where data dependencies enforce to keep a large set of arrays for the entire process of simulation in BRAM. For this algorithm, the CPU platform is

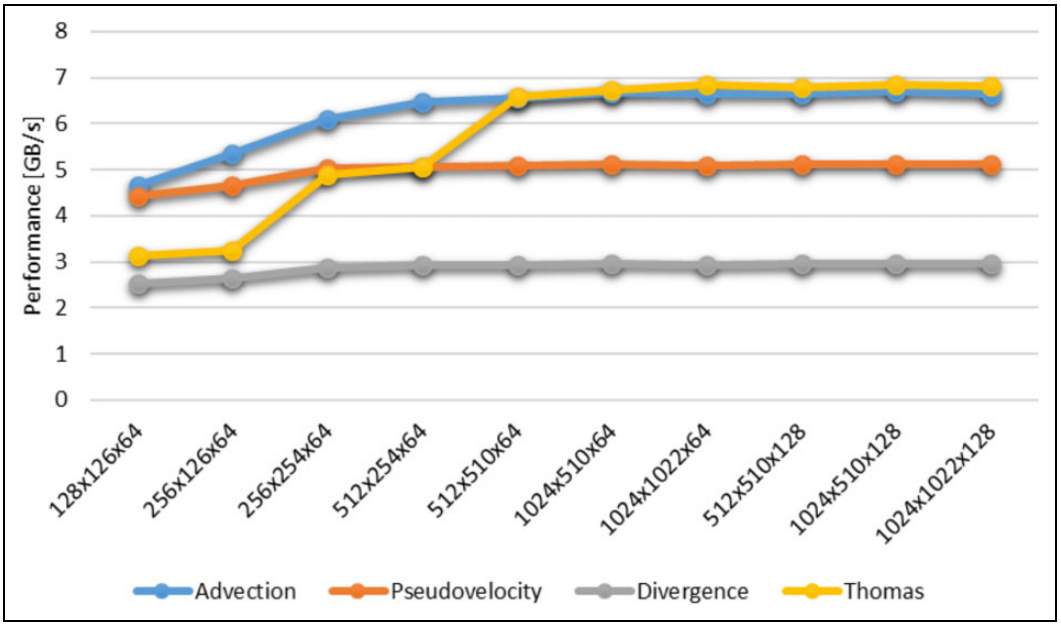

Figure 9. The comparison of performance achieved for all the CFD kernels executed on Xilinx Alveo U250 FPGA with different sizes of the compute domain.

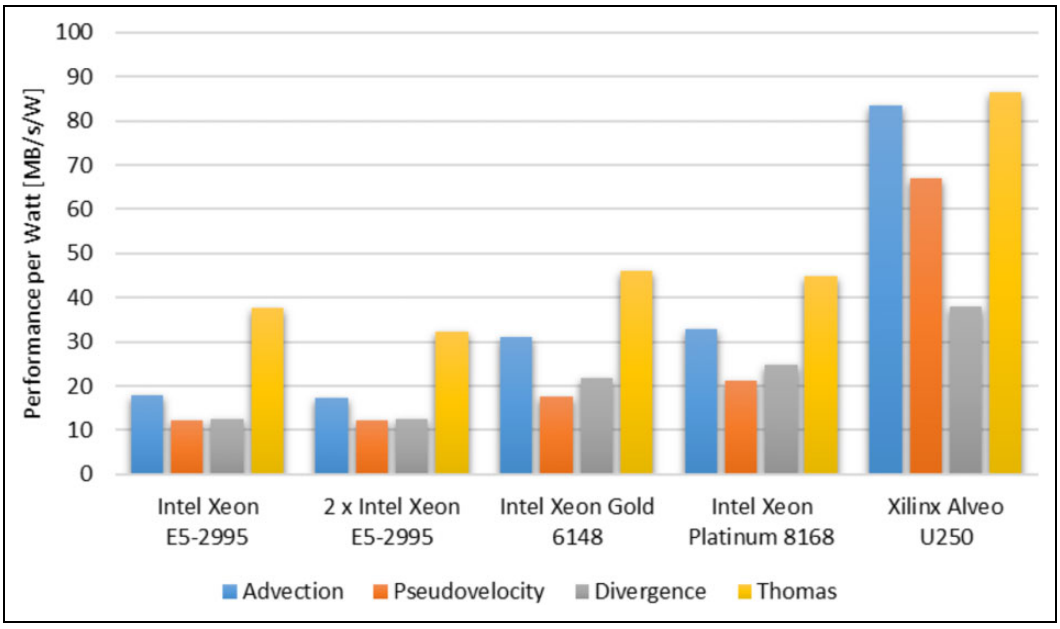

Figure 10. Performance achieved for all the CFD kernels on different platforms.

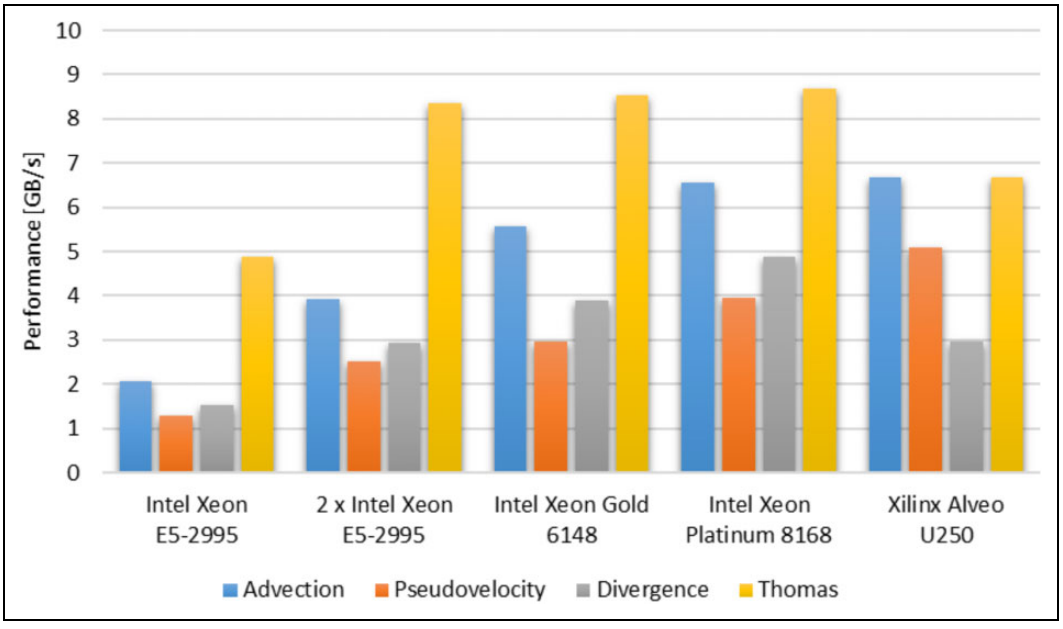

Figure 11. Performance per watt achieved for all the CFD kernels on different platforms.

more appropriate than FPGA due to a larger size of cache memory. The performance results achieved for the divergence algorithm show that the FPGA provides higher performance of computations compared only with the platform equipped with the Intel Xeon E5-2695 CPUs. From the other side, the Thomas algorithm, which is based on the vertical implementation is poor parallelizable. For this reason, we do not notice the significant performance gain between the CPU platforms, and we achieve relatively low performance for FPGA. Due to the low parallelization, a more important feature is a clock frequency, which is much higher in CPU than in FPGA.

The last part is devoted to the analysis of the performance per watt ratio achieved for all the platforms. Here, the memory-bound nature of the algorithm results in low power consumption of the FPGA card. The comparison of this metric is shown in Figure 11. Here, all the kernels achieve higher performance on FPGA than on CPUs. The conclusion is that the CFD codes are well adaptable for Alveo FPGA cards, especially in terms of performance per watt ratio. A low clock frequency (300 MHz) allows us to significantly reduce the power dissipation of FPGA and overtake the CPU platforms even for the kernels that are not well adaptable to the FPGA architecture.

Table 1. Comparison of performance (ratio *S*) and performance per watt (ratio *R*) metrics between FPGA and CPUs.

	Advection		Pseudovelocity		Divergence		Thomas	
Computing device	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*
Intel Xeon E5-2695	3.22	4.67	3.93	5.43	1.93	3.05	1.37	2.31
2 × Intel Xeon E5-2695	1.70	4.86	2.02	5.46	1.01	3.07	0.79	2.67
Intel Xeon Gold 6148	1.20	2.70	1.71	3.77	0.76	1.74	0.78	1.87
Intel Xeon Platinum 8168	1.02	2.54	1.29	3.15	0.61	1.54	0.77	1.93

6.2. Discussion

In general, all the CFD kernels contain suitable properties for the FPGA architecture. The summary of the FPGA-CPUs comparisons is shown in Table 1. Here we include the performance and performance per watt comparison of the results achieved using Xilinx Alveo U250 FPGA and achieved on the remaining CPU platforms. The results are expressed as a speedup, defined as $S = P_{FPGA}/P_{CPU}$, where P_{FPGA} and P_{CPU} refer to the performance of achieved on the FPGA card and CPU platform, respectively. The R ratio is defined as $R = R_{FPGA}/R_{CPU}$, where R_{FPGA} and R_{CPU} correspond to performance per watt of FPGA and CPUs, respectively.

In detail, we distinguish three core differences between the code adaptation to CPU and FPGA. The first one is a large cache memory space of CPU versus a relatively small but ultra-fast BRAM of FPGA. It gives an advantage for kernels where the same memory space can be reused to store different arrays of computing domain depending on the kernel stage. To this group belongs advection ($S \in [1.02; 3.22]$, $R \in [2.54; 4.67]$) and pseudovelocity ($S \in [1.29; 3.93]$, $R \in [3.15; 5.43]$). Here a data locality is sufficiently provided by the 2.5D blocking technique that allows us to reduce data traffic between the global and local memory. In contrast, the divergence kernel ($S \in [0.61; 1.93]$, $R \in [1.54; 3.05]$) requires to keep many arrays of computing domain in the local memory through the entire process. This property makes the algorithm more convenient for implementation using the CPU-based programming model. The second differentiator of FPGA is lower frequency but higher parallelism than in CPU. A really large set of FPGA LUTs allows us to efficiently overlap computations with data transfers by using an intensive pipelining with dual memory access. For most kernels, this method provides a higher performance on FPGA clocked at 300 MHz than the considered CPUs clocked from 2.4 to 2.7GHz. Only for the Thomas operator ($S \in [0.77; 1.37]$, $R \in [1.93; 2.31]$), the parallelization could not be sufficiently utilized both on CPU and FPGA due to the data dependencies. For this reason, the CPU-based platforms obtained higher performance than FPGA thanks to higher clock frequency. From the energy efficiency perspective, the low frequency and memory-bound nature of the algorithm allow us to achieve much better results on FPGA than CPU. The third difference is the memory hierarchy that is especially important for the memory-bound algorithm. In the considered CFD algorithms the bottleneck is the global memory access. FPGA contains integrated global memory with the accelerator and provides a parallel utilization of all memory banks by direct access from kernel code. Thanks to this solution the attainable bandwidth of the FPGA global memory is higher than the bandwidth of external RAM used by CPU.

7. Related work

To the best of our knowledge, this is the first work describing a real-scientific CFD code adaptation to the Xilinx Alveo FPGA cards. The proposed adaptation uses a high-level synthesis based on the OpenCL standard. In contrast to basic applications, our adaptation operates on a large set of arrays, which implies complex data dependencies and requires appropriate management of hardware resources.

There are some works that focus on the adaptation of stencil computations to the FPGA architecture using an OpenCL-based solution. In Waidyasooriya et al. (2017), Sano et al. (2014), Luzhou et al. (2012), Okina et al. (2016), and Waidyasooriya and Hariyama (2016) the authors propose libraries containing basic stencils, including Jacobi (in all the papers excluding Waidyasooriya and Hariyama, 2016) and FDTD (in Waidyasooriya and Hariyama, 2016; Waidyasooriya et al., 2017). The results show that the FPGA is a promising architecture for the presented algorithms. However, there is no investigation of real-scientific scenarios that usually require to provide strong customization of basic methods and extend them to be compatible with scientific models. More arrays generate new problems with the process of adaptation such as management of memory pins and a complex data dependency scheme.

A very recent paper (de Fine Licht et al., 2019) investigates a high-level synthesis on the FPGA platform. The authors propose a model to optimize the Matrix Matrix Multiplication (MMM) algorithm. Although MMM usually belongs to a group of compute-bound algorithms, the most important part of optimizations is devoted to data movements. This work confirms that a key role in the high-level programming of FPGA is an appropriate data organization and pipeline utilization.

de Fine Licht et al. (2018) provide another OpenCL-based approach to the implementation of MMM and Jacobi to the FPGA architecture. Beyond the pipelining, and data locality the authors focus on vectorization, and streaming

dataflow. The results show strong hardware utilization. However, there are no details about the adaptation process to the FPGA, since the optimizations are only generally described.

A possible alternative to the use of FPGA devices is described in Montella et al. (2016, 2017). The authors focused on techniques and virtualization tools of CPUs or GPUs on low-power devices such as the ARM processors. The authors conclude that applications with heavy computation requirements, small-sized input/output problems, and non-strictly real-time could be the best candidates for low-power devices.

In the last few years, papers devoted to the adaptation of the EULAG model to CPU and GPU architectures have appeared. The CPU-based implementations of solving partial differential equations in 3D space for Numerical Weather Prediction (NWP) problems were presented in Prusa et al. (2008), Rojek et al. (2015), and Wojcik et al. (2012). As the GCR elliptic solver is one of the most significant parts of the EULAG model, several different techniques for porting this solver to a cluster with multicore CPUs have been discussed in (Ciznicki et al., 2014). The proposed techniques rely on porting the original MPI code to a hybrid version which combines MPI and OpenMP. Here we investigated the new architecture to solve this problem. The EULAG model constantly evolves, and the current version is not fully applied to the GPU. For this reason, we provided a performance comparison of the FPGA-based implementation with the CPU-based code.

The adaptation of the MPDATA algorithm to the GPU architecture is presented in Rojek and Wyrzykowski (2017) and Rojek et al. (2017b). However, although both GPU and FPGA can be programmed using OpenCL, the code is not portable. The GPU-based implementation is massively parallel and uses many threads to hide memory latency, while FPGA supports pipelining to utilize LUTs. GPU does not require to explicitly manage memory banks of the global memory. Moreover, FPGA provides limitations with two simultaneously memory accesses to BRAM, and 31 memory pins for kernels handlers, pipes objects, and arguments sent to the kernels. It enforces to strongly redesign the GPU implementation to achieve high performance on the FPGA-based platform.

8. Conclusion and future work

In this work, we proposed a highly optimized CFD kernels for Xilinx Alveo U250 FPGA customized to the scenario compatible with the EULAG geophysical model. Current FPGA devices can be effectively explored with high-level programming languages. The OpenCL standard allows us to saturate the FPGA global memory bandwidth and outperform the strongly optimized CPU version of the kernels.

The adaptation process is challenging because of data access restrictions and a very long compilation time in a hardware mode that in some cases exceeds 20 hours. The software and hardware emulation reduces the compilation time but the hardware mode is crucial to provide full code validation. The disadvantage of current Alveo cards is poor support of double-precision arithmetic for floating-point operation. For this reason, applications that require double-precision arithmetic are very inefficient even for memory-bound algorithms. Another inconvenience is the limitations of OpenCL that do not allow programmers to utilize all the architectural solutions (e.g. UltraRAM is not supported). However, the proposed adaptation allows us to achieve up to 4x speedup over the CPU version of kernels, up to 80% less energy consumption, and up to 6x higher performance per watt. The developed kernels are fully compatible with EULAG geophysical model supporting all the required parameters, including domain settings, border conditions, and hardware characteristics.

Our future work includes further adaptations of CFD codes to the FPGA architecture. Currently, we provide the adaptation of advection to the Xilinx Alveo U280 FPGA (Alveo U280 Data Center Accelerator, 2019) equipped with 8GB of HBM (High Bandwidth Memory). Here the memory is organized in 32 banks of size 256 MB each. The fundamental difference in the code adaptation from Alveo U250 is the need for data reorganization to provide parallel utilization of memory banks. Our preliminary results achieved using U280 show that the advection kernel can be executed 1.2 times faster consuming 25% less energy than using U250. However, our current U280 adaptation is still significantly below the peak attainable performance. Another path of future work is to extend our adaptation across a cluster of FPGA-accelerated nodes.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work is supported in part by the National Science Centre, Poland under grant no. UMO-2017/26/D/ST6/00687.

ORCID iD

Krzysztof Rojek https://orcid.org/0000-0002-2635-7345

References

Alveo Accelerator Card Applications (2020) Available at: https://www.xilinx.com/products/acceleration-solutions.html. (accessed 17 February 2020).

Alveo U280 Data Center Accelerator (2019) Available at: https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf. (accessed 27 February 2020).

Ciznicki M, Kopta P, Kulczewski M, et al. (2014) Elliptic solver performance evaluation on modern hardware architectures. In: Wyrzykowski R, Dongarra J, Karczewski K, et al. (eds)

Parallel Processing and Applied Mathematics. Berlin, Heidelberg: Springer, pp. 155–165.

Cosmo Public area (2020) Available at: http://www.cosmo-model.org. (accessed 17 February 2020).

de Fine Licht J, Blott M and Hoefler T (2018) Designing scalable FPGA architectures using high-level synthesis. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP'18, Vienna, Austria, pp. 403–404. New York, NY, United States: Association for Computing Machinery.

de Fine Licht J, Kwasniewski G and Hoefler T (2019) Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis. In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, Seaside, CA, USA, 23–25 February 2020.

Iserte S and Rojek K (2020) An study of the effect of process malleability in the energy efficiency on GPU-based clusters. *The Journal of Supercomputing* 76: 255–274.

Luzhou W, Sano K and Yamamoto S (2012) Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array. In: Choy OCS, Cheung RCC, Athanas P, et al. (eds) *Reconfigurable Computing: Architectures, Tools and Applications*. Berlin, Heidelberg: Springer, pp. 26–39.

Montella R, Giunta G, Laccetti G, et al. (2016) Virtualizing CUDA enabled GPGPUs on ARM clusters. In: Wyrzykowski R, Deelman E, and Dongarra J, et al. (eds) *Parallel Processing and Applied Mathematics*. Berlin: Springer, pp. 3–14.

Montella R, Kosta S, Oro D, et al. (2017) Accelerating Linux and Android applications on low-power devices through remote GPGPU offloading. *Concurrency and Computation: Practice and Experience* 29(24): e4286.

Okina K, Soejima R, Fukumoto K, et al. (2016) Power performance profiling of 3-d stencil computation on an FPGA accelerator for efficient pipeline optimization. *ACM SIGARCH Computer Architecture News* 43(4): 9–14.

OpenCL Overview (2020) Available at: https://www.khronos.org/opencl. (accessed 19 February 2020).

OpenMP—Home (2020) Available at: https://www.openmp.org. (accessed 25 February 2020).

Piotrowski ZP, Matejczyk B, Marcinkowski L, et al. (2016) Parallel ADI preconditioners for all-scale atmospheric models. In: R Wyrzykowski, E Deelman, and J Dongarra. (eds) *Parallel Processing and Applied Mathematics*. New York, NY: Springer International Publishing, pp. 607–618.

Prusa J, Smolarkiewicz P and Wyszogrodzki A (2008) Eulag, a computational model for multiscale flows. *Computers & Fluids* 37: 1193–1207.

Rojek K (2018) Machine learning method for energy reduction by utilizing dynamic mixed precision on GPU-based supercomputers. *Concurrency and Computation: Practice and Experience* 31: e4644.

Rojek K and Wyrzykowski R (2017) Performance modeling of 3D MPDATA simulations on GPU cluster. *The Journal of Supercomputing* 73(2): 664–675.

Rojek K, Ciznicki M, Rosa B, et al. (2015) Adaptation of fluid model EULAG to graphics processing unit architecture. *Concurrency and Computation: Practice and Experience* 27(4): 937–957.

Rojek K, Quintana-Ortí ES and Wyrzykowski R (2017a) Modeling power consumption of 3D MPDATA and the CG method on ARM and intel multicore architectures. *The Journal of Supercomputing* 73(10): 4373–4389.

Rojek K, Wyrzykowski R and Kuczynski L (2017b) Systematic Adaptation of Stencil-based 3D MPDATA to GPU Architectures. *Concurrency and Computation: Practice and Experience* 29(9): e3970.

Rojek K, Ilic A, Wyrzykowski R, et al. (2017c) Energy-aware mechanism for stencil-based MPDATA algorithm with constraints. *Concurrency and Computation: Practice and Experience* 29(8): e4016.

Sano K, Hatsuda Y and Yamamoto S (2014) Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems* 25(3): 695–705.

SDAccel Environment Programmers Guide (2019) Available at: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/vno1533881025717.html (accessed 26 July 2020).

Smolarkiewicz P (2006) Multidimensional positive definite advection transport algorithm: an overview. *International Journal for Numerical Methods in Fluids* 50: 1123–1144.

Smolarkiewicz PK, Kühnlein C and Wedi NP (2014) A consistent framework for discrete integrations of soundproof and compressible PDEs of atmospheric dynamics. *Journal of Computational Physics* 263(C): 185–205.

Waidyasooriya HM and Hariyama M (2016) FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL. In: *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, Okayama, Japan, 26–29 June 2016, pp. 1–6.

Waidyasooriya HM, Takei Y, Tatsumi S, et al. (2017) OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems* 28(5): 1390–1402.

Wojcik D, Kurowski M, Rosa B, et al. (2012) A study on paralllel performance of the EULAG F90/F95 code. In *Lecture Notes in Computer Science*, Vol. 7204, pp. 419–427.

Xiang Y, Yu B, Yuan Q, et al. (2017) GPU Acceleration of CFD Algorithm: HSMAC and SIMPLE. In: *Procedia Computer Science* 108: 1982–1989. *International Conference on Computational Science, ICCS 2017*, 12–14 June 2017, Zurich, Switzerland.

Xilinx—Adaptable, Intelligent (2020) Available at: https://www.xilinx.com. (accessed 19 February 2020).

Xilinx Intellectual Property (2020) Available at: https://www.xilinx.com/products/intellectual-property.html. (accessed 17 February 2020).

Zhai Z (2019) *Computational Fluid Dynamics for Built and Natural Environments*. Singapore: Springer. ISBN 9789813298194.

Author biographies

Krzysztof Rojek received his PhD in Computer Science from the Czestochowa University of Technology in 2012

and his DSc in 2019. During this period, his research focused on the adaptation of HPC to the parallel processors' architectures such as CPUs, GPUs, and FPGAs. Since 2012, Dr Rojek is a researcher at Czestochowa University of Technology. His current work is directed at AI-accelerated CFD simulations.

Kamil Halbiniak received his MSc degree in Computer Science from the Czestochowa University of Technology in 2015 and his PhD in 2019. Since 2019, Dr Halbiniak is a researcher at Czestochowa University of Technology, Poland. His current work is directed at the development of efficient methods of adaptation of scientific applications to modern HPC computing platforms.

Lukasz Kuczynski received his MSc degree in Computer Science in 2001 from the Czestochowa University of Technology, Poland. In 2010 he received his PhD degree in Computer Science for his dissertation on data management in grid systems. His research focuses on code optimization for multicore architectures, including Intel and ARM.