Accelerating legacy applications with spatial computing devices

The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04925-2

Accelerating legacy applications with spatial computing devices

Paolo Savio[1] · Alberto Scionti[1] · Giacomo Vitali[1] · Paolo Viviani[1] · Chiara Vercellino[1] · Olivier Terzo[1] · Huy-Nam Nguyen[2] · Donato Magarielli[3] · Ennio Spano[3] · Michele Marconcini[4] · Francesco Poli[4]

Accepted: 1 November 2022

Abstract

Heterogeneous computing is the major driving factor in designing new energy-efficient high-performance computing systems. Despite the broad adoption of GPUs and other specialized architectures, the interest in spatial architectures like field-programmable gate arrays (FPGAs) has grown. While combining high performance, low power consumption and high adaptability constitute an advantage, these devices still suffer from a weak software ecosystem, which forces application developers to use tools requiring deep knowledge of the underlying system, often leaving legacy code (e.g., Fortran applications) unsupported. By realizing this, we describe a methodology for porting Fortran (legacy) code on modern FPGA architectures, with the target of preserving performance/power ratios. Aimed as an experience report, we considered an industrial computational fluid dynamics application to demonstrate that our methodology produces synthesizable OpenCL codes targeting Intel Arria10 and Stratix10 devices. Although performance gain is not far beyond that of the original CPU code (we obtained a relative speedup of $\times 0.59$ and $\times 0.63$, respectively, for a single optimized main kernel, while only on the Stratix10 we achieved $\times 2.56$ by replicating the main optimized kernel 4 times), our results are quite encouraging to drawn the path for further investigations. This paper also reports some major criticalities in porting Fortran code on FPGA architectures.

Keywords FPGA · High-performance computing · Spatial computing

Avio Aero is a GE Aviation business which operates in the design, manufacture and maintenance of civil and military aeronautics subsystems and systems.

✉ Alberto Scionti
alberto.scionti@linksfoundation.com

Extended author information available on the last page of the article

Published online: 29 November 2022

1 Introduction

Energy efficiency has become a major concern in all the computing domains, since the ever-growing demand for performance is followed by an increase in the energy consumption of computing infrastructures. Load-Store Architectures (LSA) dominated the computing landscape for long time, but they have been found to be inefficient. Indeed, a large fraction of the energy is spent by instructions (especially by floating-point instructions—[1]) and by moving data in and out of register files, caches, as well as it is consumed also by the control logic. These control-driven architectures start struggling in keeping the performance pace when deep learning and other highly parallel algorithms have to be executed. As such, conventional CPUs are limited by a relative small parallelism exposed by the few cores, and by their relative high power consumption. Domain-specific architectures (DSAs) like GPUs and deep-learning focused devices (e.g., Google TPU, Nvidia Tensor-Cores) overcome such limitations by tailoring to a specific class of computations. However, such advantages still remain restricted to specific application domains, generally making DSAs poor performing in others. Since their infancy, field-programmable gate arrays (FPGAs) were used as a means for fast prototyping application-specific integrated circuits (ASICs) or to speed the development of the software stack for a given architecture. The fast progress of manufacturing processes allows FPGA manufacturers embedding an ever-increasing number of functional blocks, leading to reconfigurable fabrics capable of accommodating very complex overlay computing architectures [2, 3]. All this is making FPGAs more attracting, thus leading to an increased number of applications successfully ported on such *spatial computing* devices.

All the major FPGA manufacturers offer high-level synthesis (HLS)-based frameworks (i.e., Intel's OpenCL-AOCL and most recently Intel oneAPI, Xilinx's HLS-SDAccel, Maxeler's dataflow compiler) to avoid the application developers the burden of optimizing their code both for (execution) time and space (i.e., minimizing the number of on-chip resources that are used and also limiting the off-chip communication as much as possible). HLS tools are generally based on high-level programming languages and frameworks, like C/C++ and OpenCL; however, there is a lack of libraries of reusable components. Despite this, writing high-performance parallel code targeting modern FPGA fabrics is still a challenging task. Notably, fBLAS [4] represents an example of the effort for offering a high-level library of reusable components, which preserve performance across various target devices. Here, the challenge is tougher than for DSAs since the interfaces of the components must conform to specific (high-level) standards to correctly exchange data each others. This introduces additional constraints to the compiling tool-chain that is in charge of translating high-level application codes into a register-transfer level (RTL) representation. Further, fBLAS re-usability is restricted to only C/C++ applications. Meta-programming frameworks have been proposed to ease the exploitation of spatial computing devices (e.g., Maxeler), but their broader adoption is still far, as well as their capability of coping with the requirements of the large body of legacy scientific codes that are in use today.

As a matter of fact, Fortran-based applications remain out of the support of such frameworks.

While heterogeneous-oriented programming frameworks provide a clear pathway for 'code portability,' 'performance portability' challenge across multiple devices is not yet fully addressed. For instance, Zohouri et al. [5] showed that porting OpenCL code optimized for GPU devices on FPGAs ended in a functionally correct code on the latter devices, but paid in terms of performance. Conversely, code optimized for FPGAs allowed to achieve better power efficiency. Recently, Intel introduced the *oneAPI* framework [6], which leverage on high-level coding to smoothly support CPUs, GPUs and FPGAs. Similarly, AMD offers tools easing the code acceleration using GPUs and FPGAs.

The main drawback of these frameworks is the limited support of high-level languages that is restricted to C/C++ (although, AMD supports Fortran acceleration but only targeting specific features available on their processors' lineup). The problem is exacerbated by the fact that a large body of scientific applications is written using the various standards of the Fortran programming language. To fill the gap between these (legacy) applications and the capability of exploiting performance boost given by heterogeneous devices, effective methodologies to connect application codes with the lower programming substrate should be considered. Aimed in part as an experience report, the main purpose of this work is to define a methodological approach to guide the reader in the way of accelerating legacy Fortran applications using FPGA devices. The work presented in this paper arose in the context of funded European projects, where one of the objectives was that of evaluating the performance improvement of HPC applications with heterogeneous hardware, including FPGAs. As such, we had access to a specific licensed code used to support the simulation and design of aeronautical components. (More specifically, we targeted a specific application in the computational fluid dynamics domain—CFD.) Despite we were limited in the number of target routines that could be disclosed, experimental results provided a glimpse of the potential benefit in using FPGA devices for accelerating Fortran codes, as we achieved a relative speedup over the CPU version of $\times$ 2.56 (Intel Stratix10) when the main kernel is replicated multiple times, although the performance remained quite below in the case replication strategy is not used for both the devices we considered (i.e., $\times$ 0.63 for the Intel Stratix10 and $\times$ 0.59 for the Intel Arria10). Similarly, we show a power consumption estimation in the order of three fourth of the maximum nominal one (i.e., around 200W for the Stratix10 case and around 50W for the mid-range Intel Arria10 device). Furthermore, we show that with our methodology, a synthesizable code is achievable, as well as the approach can be largely automatized. Although these results are still not able to fully demonstrate performance advantage of the FPGA version of the targeted code, they are encouraging for driving further investigations.

1.1 Paper contribution

The large base of applications daily used by scientists and engineers is written using high-level programming languages which are not able to exploit performance and

energy efficiency benefits brought by modern hardware accelerators. One notable example can be found in Fortran applications, for which the support offered by heterogeneous-oriented programming frameworks is poor. Indeed, while some sort of support can be found for GPU targets, FPGAs totally lack any such support.

This paper has the major contribution in filling the gap between CPU-focused Fortran applications and the capabilities offered by modern FPGA devices. While experimental demonstration targeted Intel mid-range and high-end devices, the proposed methodological approach is not stuck to any specific feature of such devices, being thus portable across vendors. Specifically, we derived a methodology to map a selected Fortran routine (acceleration target) on an equivalent HLS synthesizable code, which can be accelerated on a FPGA device. To this purpose, our methodology is composed of three main steps, as follows:

- (Automatic) translation of the original Fortran routine to be accelerated into a HLS-friendly language (OpenCL and C/C++ in our setup);
- Creation of the additional FPGA kernel(s) and data handling support code;
- Creation of data wrappers to properly map Fortran data structures with C/C++ types and data structures.

The remainder of this paper is as follows. Section 2 introduces the selected test vehicle routine as the target for FPGA acceleration. Its subsection details the proposed methodology for mapping Fortran legacy code on the FPGA high-level synthesis (HLS) environment. Section 3 provides the description of the used experimental setup, and in the subsection, we discuss the synthesis results. Here, we provide an overview of the performance achieved by the FPGA versions, along with an estimation of the power consumption on the two experimental devices. Section 4 provides an overview of the most influencing works available in the literature, while Sect. 5 concludes the work and gives a view on future activities.

2 The FPGA acceleration test case

This work was born in the context of European-funded projects, with the main purpose of investigating on porting (large) scientific and engineering applications on heterogeneous high-performance computing infrastructures. As such, we aimed at evaluating diverse acceleration platforms as target of our investigation, thus considering the FPGA as one interesting case. To this purpose, we surveyed pilot applications being part of the LEXIS project[1] to determine a suitable test case. As such, this initial survey resulted in the selection of a Fortran code currently used in the CAE tool supporting the computational fluid dynamics investigations for the aeronautical turbomachinery test-bed included within the LEXIS Aeronautics Large-Scale Pilot led by Avio Aero. Specifically, the code that has been chosen is a ***smoothing routine*** of the *Traf* program [7, 8], which is a computational fluid dynamics (CFD)

[1] https://lexis-project.eu/web/.

solver for three-dimensional Reynolds-averaged Navier–Stokes equations, developed at the University of Florence, with a special focus on turbomachinery applications. An implicit residual smoothing strategy is used to accelerate the convergence. As the name suggests, the routine acts on a set of multi-dimensional arrays which are produced in other elaboration steps and "smooths" them by performing operations closely related to an implicitly computed weighted average. These multi-dimensional arrays are closely linked to a computational grid and bring information related to physical parameters of the main simulation. The purpose of such smoothing operation is to filter out non-physical oscillations that may appear in the residual field. Worth to mention here is the fact that the access to the code of the selected application was subject to a license, which restricted the number of routines that could be disclosed. Also, the majority of the routines shown inter-dependencies which made their use as an initial test case for our methodology more complex. To this end, we looked at to the *smoothing routine*, that did not show such drawbacks.

In fact, this routine caught our attention, since it shows interesting features that helped us to focus on the porting aspects of the code to the FPGA-based environment. Specifically, the routine is short and self-contained (does not rely on external libraries) and uses only easy mathematical operations (sums, multiplications and divisions) on arrays of single precision floats. As such, for the sake of simplicity, the whole application has been stripped down to a bare minimum, keeping only a Fortran main entry point, which is in charge of allocating the data arrays, loading input test data from a file and launching the routine. One important aspect concerns also the way the input dataset has been generated. This is done by executing external Fortran routines that are not the target for the acceleration. Specifically, input data are the outcome of physical simulation processes which limited our capability of enlarging the overall dataset or modifying the structure data are organized in.

Interestingly, some test (not limited to the specific routine targeted in this work) has been performed using the IT4Innovations HPC facilities which allowed to demonstrate that a proper refactoring of the code could provide a significant performance boost using hardware acceleration. Specifically, nodes equipped with multiple Nvidia V100 (using 16 accelerators, 4 cards per node) and Nvidia A100 (using 32 accelerators, 8 cards per node) GPUs have been used to provide a speedup up to $\times 21.4$, with regard to the original code (as reported in a recently accepted paper [9]). While these experiments showed us the great potential benefit of using hardware acceleration, the high power consumption of such GPUs drove us to consider reconfigurable devices as a further direction of investigation for a more power-efficient acceleration solution. In fact, each V100 accelerator draws up to 300W, while each A100 consumes up to 400W against 225W and 75W, respectively, for the high-end (Intel Stratix10) and mid-range (Intel Arria10) boards tested here.

2.1 Proposed methodology

The process of porting legacy Fortran code to modern FPGA-based environment requires several steps aimed at refactoring the code in such a way specific features may become more exploitable by low-level HLS tools. In particular, code

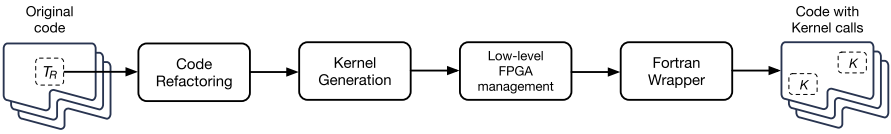

Fig. 1 Proposed transformation pipeline: the target routine (T_R) in the original code passes through four main porting steps, ending in a code where T_R is replaced with kernel calls (K)

refactoring is the first step needed to properly derive a kernel (i.e., the portion of the code that will be executed by the accelerator). To this purpose and when Fortran code is targeted, code refactoring implies mostly removing global variables (which prevent to properly deal with the separated memory spaces of the host CPU and accelerator) and to avoid use of variable names that contains indication of the variable types. Such operations are referred as code normalization and are necessary to properly translate the Fortran code into an equivalent C-based code. Other operations may involve the application of OpenCL annotations (used to drive the HLS compilation to extract as much as possible parallelism), pipelining and vectorization. While code normalization can be achieved mostly in an automatic manner, some code improvements for performance gain (e.g., monolithic kernel code split into pipelined kernels) may require manual intervention. Despite this limitation, the methodology steps allow to derive a properly synthesizable code. The other steps involve the creation of the logic to manage data transfers between the host CPU and the FPGA board, instantiating a Fortran wrapper (the code linking to the C-based/OpenCL code that drives the FPGA board and the communication with the host), and replacing all the calls to the original Fortran code with equivalent ones to the Fortran wrapper. To summarize, our methodology is based on the following main steps:

- *Step-1*: Code refactoring (normalization);
- *Step-2*: Translate the computing routine into a HLS-friendly language (OpenCL in our setup), including the use of annotation for performance improvement;
- *Step-3*: Create the C-based support code (FPGA kernel and data handling);
- *Step-4*: Create a Fortran wrapper around FPGA management and kernel handlers, as well as to adapt Fortran data structures and types to C ones;
- *Step-5*: Replace original routine calls with wrapper calls.

Figure 1 shows our proposed pipeline for the effective translation of (legacy) Fortran routines into the equivalent C-based/OpenCL code. Once isolated the target routine (T_R), the first step consists in translating the original Fortran code to an equivalent code targeting HLS. Here, some major challenges can be found in moving from Fortran code to OpenCL or even to HDL code. As mentioned in Sect. 1, there is almost complete lack of tools capable of automating this phase, except few examples (i.e., works [10, 11] as reported in Sect. 4) and considering some limitations. By the way, large fraction of scientific and engineering codes contains loops, which are thus one of the main targets of the translation process.

Here, label-based loops can be converted into more manageable do-loops. This latter form can be (automatically) translated in the equivalent C-based version. When the targeted accelerator is an FPGA, the generated C-based code should be compliant with the requirements coming from the device vendor. Furthermore, the HLS tools generally consider specific dialects of the OpenCL standard, i.e., each vendor supports a subset of the general OpenCL standard, to which some custom extensions (custom annotations) are added. While the customization is different from vendor to vendor, a 1-to-1 mapping between annotations of one vendor and those of another is possible. Based on this, the final form of the kernel may vary, depending on the specific targeted FPGA device. Despite, the space for automating the translation process, some very irregular codes may still require manual intervention to generate a proper synthesizable code. This is the case of our test vehicle, which shows a quite irregular pattern of accessing the input arrays (i.e., irregular memory accesses), leading to a not very-optimized kernel using such automation approach. Indeed, multi-dimensional Fortran arrays (input) are allocated at run-time, while in OpenCL, we have to define the array size at compilation-time (we had to analyze run time execution of the original code to determine the proper size of the arrays); moreover, the number of array indexes (which is 4 in our case) required us to explicitly write the address generation logic inside each buffer (i.e., reserved memory regions in the SDRAM of the FPGA target board, used to communicate with the host; on the host side, similar memory regions are created for the same purpose).

A second challenge can be found in the way the original code is written (often this aspect refers to the body of a loop). Here, the code should be first normalized to better expose features that allow to generate (efficient) RTL structures. To address this latter challenge, tools like those described by [12] and in [13] can be used. In particular, these works driven us in the implementation phase of our methodology. Among the operations involved in the code normalization phase, we can highlight: (1) the need of translating all non-program code units into modules which are then used through an explicit export declaration; (2) the need of removing some deprecated features largely exploited in old legacy codes (e.g., Fortran 77), such as implicit typing, the absence of a module system, or the absence of intended access declarations for subroutine arguments; and (3) the need of removing the use of global variables which need to be passed as inputs in the kernel function(s). This latter point, in fact, is directly connected to way memory objects are used in a heterogeneous execution system. The general accepted model when dealing with hardware accelerators is that of keeping host memory (physically) separated from that of the accelerator. Then, the code to be accelerated is embedded into a *kernel* that is then implicitly or explicitly called in the original code (*K* blocks in Fig. 1). As such, the kernel code is directly executed on the acceleration device, while the host code abstracts the necessary machinery for transferring data in and out the accelerator. Thus, such model implies that host memory and accelerator (global) memory were managed and addressed as physically separated memories. Moving data from the host to the accelerator (and vice versa) is done by an explicit copy operation.

2.1.1 Implementation approach

The first implementation of the ***smoothing routine*** (we refer to this as *smoothcl*—baseline) in OpenCL was a simple, naïve translation of the Fortran code to a single OpenCL kernel, with very limited optimizations. The specific implementation followed the OpenCL guidelines for a single work-item kernel; thus, the original loops were explicitly preserved and synthesized as pipelined data paths and finite state machines (FSMs). Indeed, we found *NDRange* kernel structure providing any advantage, since the high dimensionality of the arrays and the specific memory access patterns prevented the synthesized circuit to achieve good performance. In fact, while NDRange kernels provide an easy means for parallelizing the operations performed on multi-dimensional arrays, they need to use the (slow) global memory of the device to synchronize each others. Furthermore, the FPGA vendor suggests the use of single work-item structure to perform more aggressive optimizations on the synthesized circuit. Given that, the loops were handled as streams of single operations performed in parallel on the elements of the data arrays (vectorization). Several iterations of the OpenCL code compilation, synthesis and running session on the target devices have been done in order to fulfill the bare minimum performance requirements (i.e., performance is in line with that of the CPU-only counterpart) for the routine and to reduce as much as possible the amount of FPGA resources consumed. Multiple copies of the processing kernel can be instantiated to further boost the performance of the accelerator.

A C-based FPGA management code takes care of all the operations needed to support the FPGA board programming, as well as those related to kernel discovery, declaration and execution. Furthermore, this code is in charge of managing the creation of data buffers within the on-board SDRAM banks, and to manage the data transfer to and from the FPGA board (with regard to the host). The kernel and FPGA management codes, although fairly repetitive in their structure, count for a quite larger number of lines if compared to the original Fortran code (i.e., more than a thousand lines of code compared to less than 200 lines of Fortran code). When moving on multiple parallel kernels, the lines of code ratio further increase. (Indeed, the C-based code has to take care also of the concurrent execution and needed synchronization among the kernels.)

The latest step of our proposed methodology consists in writing a Fortran wrapper, which is in charge of directly linking the FPGA management code to the Fortran entry point. Thanks to this wrapper, we can effectively replace the original residual smoothing routine code. More specifically, this wrapper unpacks the original data structure, isolates the arrays that need to be copied to the board (global memory) for the smoothing operation, and simply calls the FPGA management code (which in turns embeds the call to the FPGA kernels). The processed data arrays are then copied back from the board (global memory) to the host memory structure. Figure 2 shows the architecture resulting from the Fortran routine translation into the FPGA-accelerated kernels. The synthesized acceleration kernels—i.e., RTL blocks in the figure—contain the FSMs responsible for the operations performed on the data arrays. Through the FPGA Board Support Package (BSP), the kernels have access to the PCIe interface and the memory controller logic, which are needed to manage correctly the access to the global memory

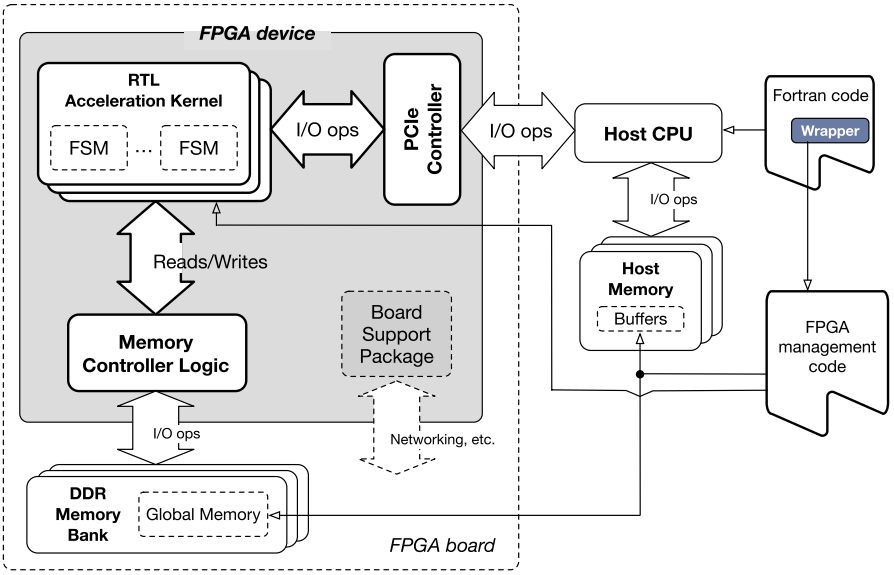

Fig. 2 Resulting architecture for accelerating legacy Fortran code on the FPGA target device. A wrapper allows to directly call the FPGA management code used to drive the communication between the host and the FPGA, as well as to manage kernels execution

Table 1 FPGA boards used for evaluation

FPGA		ALMs	FFs	M20Ks	DSPs	DRAM
Arria10 GX1150	Total	427×10^3	1.7×10^6	2.7×10^3	1518	2×8 GB
	Avail.	392×10^3	1.5×10^6	2.4×10^3	1518	2×8 GB
Stratix10 GX2800	Total	933×10^3	3.7×10^6	11.7×10^3	5760	4×8 GB
	Avail.	692×10^3	2.8×10^6	8.9×10^6	4468	4×8 GB

and to communicate with the host. Optimization of the kernel structure led to the generation of several DMA engines to cope with the memory access patterns generated by a multiple kernel instantiation. On the host side, communication buffers are created in the memory pool. FPGA management code, which is directly linked to the Fortran code through the wrapper, manages the data exchange between host and FPGA global memory, and the execution of the acceleration kernels.

3 Experimental setup

Experimental setup was based on Bittware boards based on two different FPGA devices (i.e., the Intel Arria10—mid-range device, and the Intel Stratix10—high-end device), whose main features are reported in table 1. These boards were made

available in the context of the European-funded projects that supported this work, through HPC heterogeneous infrastructures.[2,3] These FPGA devices embed different amounts of reconfigurable resources, as well as they embed dedicated hardened features providing specific functionalities to the application layer. Reconfigurable resources can be divided in: *Adaptive Logic Modules* (ALMs), *Flip-Flops* (FFs), *Memory Blocks* (M20Ks), and *Digital Signal Processors* (DSPs).

With each board, the vendor provides the specific Board Support Package (BSP), which is a predefined circuitry loaded on the FPGA fabric with the purpose of managing the interconnection with other external components (e.g., access to the PCIe interface, access to networking interface, etc.) which are present on the board. BSP consumes a fraction of the reconfigurable resources, so that the actual number of ALMs, FFs, M20Ks and DSPs available to the application is less than what reported by the device data-sheets. For both the board models, the connection to the host device is done via PCIe configured in $\times 8$ mode. Interestingly, the DSPs embedded in the Intel devices expose the hardware support for single precision floating point operations, which has been extensively exploited in our test case application. Furthermore, the ALMs are flexible reconfigurable logic blocks, which in turn contains a lookup table block (LUT), two full adders, carry-chain, and register-logic. Thanks to such flexibility, the ALM resources can be used to implement both combinational- and register-functions. On the more advanced FPGA device (Intel Stratix10), there is a second kind of on-chip memory blocks, referred to as MLABs. Similarly to M20K blocks, MLABs can be configured in different ways (single port, dual port, FIFO, ROM, shift register); however, the main difference with M20Ks remains in their arrangement which is, in the MLAB case, better suited for wide and shallow on-chip storage. It is also of worth mentioning that ALM blocks also support their fracturing. By this, each building block can be configured to work as two separated logic blocks although halving the number of available resources.

Experiments were carried out on three different cluster nodes (host systems) based on the Intel Xeon CPU architecture. The nodes were equipped with a different amount of main memory and different CPU versions. Worth to mention is the fact that, the CPUs on board of these nodes had a large amount of cache memory. This led us to suspect that the small performance advantage shown by the FPGA accelerated code was masked by the large cache memories (L3 cache size was in the order of few tens of MB) able to accommodate large chunks of the input data set. From a software perspective, for synthesizing FPGA kernels we relied on the Intel HLS tool-chain, which is based on the Intel FPGA SDK for OpenCL v19.1. On the host side, the application code was compiled using an open-source tool, specifically the GCC compiler. In all the cases, the operating system of the nodes was a recent Linux version.

The original Fortran code (smoothing routine) consisted in many do-loops counting for an overall number of lines equals to nearly 200, and computing the implicit weighted average of the values carried by multi-dimensional input arrays. Even the

[2] https://lexis-project.eu/web/.
[3] https://www.acrossproject.eu.

Table 2 FPGA resource consumption (area estimation) targeting the Intel Arria10 device

Kernel function	ALUTs	FFs	M20Ks	DSPs
Smoothcl core	115,578	281,998	2019	124
Global interconnect	21,305	43,500	61	0
Board interface	21,305	43,500	61	0
FPGA fabric resources consumption				
% Of used resources	33%	36%	100%	14%

structure of the loops is quite similar to each other. While the following considerations apply to all the loops being part of the smoothing routine, for the sake of simplicity, we provide in Listing 1 an illustrative example of one of these do-loops. Specifically, on each iteration, the multi-dimensional data arrays are accessed, and their value is updated by applying division operator (see line 2). A very direct translation of this simple loop is provided in Listing 2, whereas the multi-dimensional arrays access is split on multiple lines (lines 3, 4 and 5), and do-loop is replaced with a for-loop.

Listing 1 Simple Fortran code with a simple loop.

```
do i = 2, nxp                                                    1
  r(i, nyp, k, l) = r(i, nyp, k, l) / ac(i, nyp - 1, myk)        2
end do                                                           3
```

Listing 2 OpenCL translation of the simple Fortran loop provided in Listing 1

```
for(i = 1; i < nxp; i++)                                         1
{                                                                2
  *(r + (i) + (nyp)*nxp + k*nyp*nxp + l*nxp*nyp*nzp) =           3
      *(r + (i) + (nyp)*nxp + k*nyp*nxp + l*nxp*nyp*nzp) /       4
      *(ac + (i) + (nyp-1)*nxp+myk*nyp*nxp);                     5
}                                                                6
```

The code demonstrates the complexity of the indexing used to point the actual elements of the arrays. Another point to mention is that in the OpenCL version, the access to those data arrays is done through memory pointers.

3.1 Synthesis results

Table 2 is derived from the standard report coming out of FPGA synthesis tool targeting the Intel Arria10 device; it shows the resources (the main building blocks) that have been used to map the kernel functions to a properly working digital circuit: ALUTs are derived mostly from the ALM blocks and are used to generate Boolean functions, including basic Boolean gates (AND and OR) with multiple inputs. FFs

Table 3 FPGA resource consumption (area estimation) targeting the Intel Stratix10 device

Kernel function	ALUTs	FFs	M20Ks	DSPs	MLABs
smoothcl_kernel1	11,627	19,962	108	11.	105
smoothcl_kernel2	87,955	198,135	921	76.5	1613
smoothcl_kernel3	49,804	122,626	619	32.	815
smoothcl_kernel4	72,097	163,350	809	42.5	1102
Global interconnect	40,372	62,912	104	0.	0
Board interface	474,980	949,960	2768	1047	0
System description	115,578	281,998	2019	124	0
FPGA fabric resources consumption					
% Of used resources	39%	24%	45%	21%	(*)

(*) the Intel reporting tool does not provide the percentage of used MLABs

are mapped to registers, while the DSPs are used to implement the most expensive mathematical operations such as summation and multiplications (i.e., adders and multipliers blocks). Finally, M20K memory blocks are used as an embedded on-chip RAM to quickly map data arrays on top.

It is worth noting here that the result on the resource usage done by the synthesis tool is generally an over-estimation of the actual resources required by the kernel after the placing and routing subsequent step. Nevertheless, this estimation provides an important insight of the FPGA resource occupation of the kernel(s) and generally is not too dissimilar from the actual value. Given that, on-chip RAM memory blocks seem to be the limiting resource at this point. Indeed, after some optimization steps and using OpenCL pre-processor pragmas to automatically replicate sections of the code only resulted in a design consuming all the M20K memory blocks (i.e., 100% of the RAM blocks was used) in the FPGA fabric. In this case, the kernel replication strategy for improving performance was not feasible on such mid-range device. Worth to say, the naïve kernel implementation (i.e., the one obtained by translating the original Fortran smoothing routine into an equivalent C-based function), as well as most of the applied optimizations, can be automatized. On the contrary, some of the following optimization strategies have been implemented manually, although this does not affect the general methodology we illustrated.

A different optimization strategy consisted in dividing the main kernel core function in four kernel functions (actually mapped as four independent kernels with data pipes in between to speed up data transfers). This strategy allowed us also to better understand the performance bottlenecks and explore the possibility of replicating only critical sections of the routine. In order to simplify the control logic, we decided to reduce the number of nested loops; instead, we run the kernels several times, passing the loop index as argument. This approach did not have significant overload and helped us achieve better overall clock frequencies. Unlike for the mid-range device, on larger FPGA fabrics (i.e., Intel Stratix10 device in our case), a very straightforward strategy to improve the performance of the accelerated code was that of replicating multiple times the main kernels and give them the capability of processing non-overlapped portions of the input data arrays in parallel. As such, the

Table 4 Synthesis results for the simple for-loop-based kernel targeting the Intel Arria10 device

Kernel function	ALUTs	FFs	M20Ks	DSPs
smoothcl_v2.cl:177	2668 (0%)	7331 (0%)	46 (2%)	3.5 (0%)
32-bit Integer Add	11	0	0	0
64-bit Integer Add (×3)	195	0	0	0
And (×2)	22	0	0	0
Floating-point divide	1014	948	3	3.5
Integer compare (×3)	22	2	0	0
Iteration initiation	1	1	0	0
Load (×2)	1004	4100	26	0
Or (×21)	1	1	0	0
State	0	64	0	0
Store	399	2216	17	0

The consumption of resources is broken down into resource consumption of single-generated logic operations

larger amount of resources allowed to replicate up to 4 times the overall kernels pipelines, allowing to achieve performance that are in line with that of the CPU version (see Sect. 3.2 for an in-depth analysis of the performance results). This strategy required to slightly adapt the corresponding control code. Synthesis results reported in Table 3 show that kernel parallelization can be exploited, at least in terms of amount of available resources on the Intel Stratix10 fabric.

Despite these various optimizations, performance evaluation suggested us that the main performance limiting factor was to ascribe to the memory access patterns. Indeed, the selected target routine generates memory accesses causing a large number of stalls incurring in the synthesized pipeline. As such, the overall performance drop down. For instance, Table 4 shows the synthesis results (targeting the Intel Arria10, while similar conclusion can be drawn for the Intel Stratix10 from the synthesis results) for the illustrative example of Listing 2, by detailing on the resource used by each synthesized circuit structure. From these results, it can be seen how raw global memory accesses mostly affect on-chip resources: For each access to global memory using a memory pointer, the synthesis tool instantiates a dedicated DMA engine and the arbitration logic to access the SDRAM controller. We identified all this logic as the main consumer of device resources as well as the main source of lowering the synthesized circuit clock frequency. Indeed, the large number of DMA engines concurrently trying to access the SDRAM controllers creates a very high number of contentions bringing to stalling the pipeline until data become available. To alleviate this, we introduced an input caching mechanism, which reduced the overall number of stalls, albeit their complete removal was not achieved. To this end, several M20K blocks are grouped and paired to the DMA engines.

Worth to note is that although resources on the FPGA fabric (Intel Stratix10) were enough to further replicate the kernels, experiments suggested that 4 instances were able to saturate the bandwidth toward the global memory due to these multiple concurrent accesses.

To further circumvent this limitation, we opted for the strategy of splitting the loops body into separated operational steps, each wrapped by a separated loop. Globally, the operations performed by this new version of the translated kernel are equivalent to those done by the single loop version, while they contributed to reduce the number of concurrent access to the global memory. For instance, the code shown in Listing 3 is obtained after the splitting of the code reported in Listing 2.

Listing 3 Improved version of the OpenCL kernel mapping the simple Fortran loop provided in listing 2

```
float* tmpac;
float* tmpr;

// do some other work and vector allocation
// ...

// kernel loop split
// step 1
for(int i = 1; i < nxp; i++)
{
   tmpac[i] = *(ac + (i) + (nyp-1)*nxp+myk*nyp*nxp);
}

// step 2
for(int i = 1; i < nxp; i++)
{
   tmpr[i] = *(r + (i) +
             (nyp)*nxp + k*nyp*nxp + l*nxp*nyp*nzp);
}

// step 3
for(int i = 1; i < nxp; i++)
{
   tmpr[i] = tmpr[i] / tmpac[i];
}

// step 4
for(int i = 1; i < nxp; i++)
{
   *(r + (i) + (nyp)*nxp + ridx) = tmpr[i];
}
```

Through this splitting strategy, the following execution pattern is achieved. First step (corresponding to the first for-loop) manages the gathering of the data scattered around the global memory (remember that we are dealing with multi-dimensional arrays); the second step builds a cache using local memory blocks (M20Ks), while the third step performs the math operations on locally stored data. Finally, we write back results all together on the global memory. Through this implementation, we got a significant FPGA clock speed update, so that synthesized circuit moved over 200 MHz on the largest device. While the pattern of accesses to the

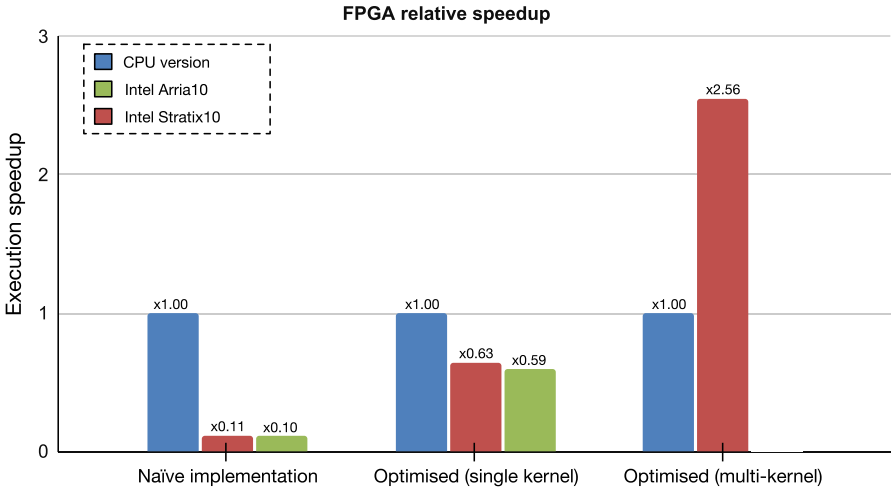

Fig. 3 Performance comparison between Intel Arria10 and Stratix10 kernel implementation: relative speedup over the original CPU-only execution

memory ameliorated, the number of resources in use remained unaffected by these modifications.

Our general observation that, in the selected test vehicle routine, the memory access patterns are the largest limitation factor for the performance, is confirmed by the Intel profiler. (It is used to profile the kernel execution over time.) Indeed, the profiler showed that poor bandwidth occupation toward the global memory (SDRAM), suggesting that memory access scheme provided by the naïve code translation is not the best choice.

3.2 Performance results and analysis

To further characterize our methodology, we analyze here the performance of the synthesized circuit on both the FPGA devices we had access to. The performance results are expressed as relative speedup with respect to the execution of the original code on a single CPU.

The final design of the accelerator can be summarized as a pipelined sequence of smaller kernels performing the operations related to weight-averaging multi-dimensional array values. This kernel-based pipeline contains also input caches to reduce the pressure on the global memory. On the smaller device (Intel Arria10), this configuration consumed nearly all the on-chip memory resources, allowing to achieve a clock frequency of 210 MHz. On the larger device (Intel Stratix10), the above-mentioned configuration has been replicated 4 times and the final clock frequency of the synthesized 223 MHz.

Figure 3 shows the execution speedup (relative to the CPU-only original version) obtained comparing the naïve implementation, the optimized design and the execution with multiple kernels replicated. As the reader can see, although the methodology is able to drive the user toward a synthesizable design, performance is very poor

when compared to the CPU version, when the naïve implementation is considered. Similarly, performance raised up when the various optimization steps are applied, but also in this case, there is no effective speedup for both the considered devices, which is nearly half (i.e., $\times$ 0.59 and $\times$ 0.63, respectively, for the Intel Arria10 and Intel Stratix10) the performance of the CPU version. The strategy of replicating the kernels on the larger device led the performance to greatly improve, ending in a speedup in favor of the larger FPGA device ($\times$ 2.56).

Also, when comparing the execution of the accelerated code with that of the original code running entirely on the host CPU, the CPU version remained in most of the cases faster than the FPGA-versions (only replicating multiple times the kernel gave us an effective advantage). However, it is important to remark that in our experiments, we had access to a small input dataset used for test purposes (several tens of MB in total), which was effectively cached by the CPU cache subsystem. In fact, all the CPU we had access to provided few tens of MB of L3 cache, leading us to suspect that large chunks of the input data could easily fit on the large L3 cache. Also, the synthesized designs never exceeded the decent clock barrier of 250 MHz, while all the used host CPUs run above the GHz. Therefore, we think that a real advantage of the FPGA acceleration would emerge using real input datasets which are larger. This indeed is a planned activity through which we want also to fine-tuning our methodology. Nevertheless, power consumption remains in favor of FPGAs when compared to other acceleration platforms (e.g., GPUs). In particular, regarding the measurement of the power consumption of the two devices, this operation would have required to instantiate specific hardware modules to capture data. As such, this operation would have drawn out further FPGA fabric resources. To overcome this limitation, we can roughly estimate the power consumption as a fraction of the maximum power (data are taken from the datasheet of the boards), which is proportional to the average resources used. Specifically, from the experimental values reported in Sect. 3.1, for both the devices, the average resources consumption is not higher than three fourth, thus setting the average power consumption to ~206 W and ~56 W, respectively, for the Intel Stratix10 and Arria10.

4 Related works

Computer architecture specialization has been established as the main driving factor for achieving better performance and energy efficiency in the current and upcoming high-performance machines. The viability of spatial architectures (and more specifically FPGAs) as mainstream hardware accelerators for computationally high-demanding applications has been already demonstrated by far, with some notable examples in the literature [14]. For instance, [15] analyzed algorithms largely used in the HPC context and found that 5 out of the 13 analyzed were suitable for being accelerated on FPGA devices, although the required knowledge for their porting clearly shows the need for a higher abstraction level. Weller et al. [16] demonstrated the suitability of FPGA devices to accelerate partial differential equations (PDEs); authors also showed the performance limitations due to porting the kernels across devices of different vendors.

Maxeler provides a high-level Java-based framework to ease the definition of computing kernels and to connect them through data streams [17]; the capability of the framework has been demonstrated with scientific applications belonging to diverse domains (e.g., financial, seismology). Intel and Xilinx have their own high-level synthesis (HLS) programming frameworks (AOCL—[18]—and SDAccel[4], respectively) to eliminate the burden of dealing with lower level architectural aspects (e.g., connecting with the PCIe interface, instantiating control memory logic, etc.). Despite these frameworks leverage on high-level programming languages such as C/C++, the knowledge required to optimize the generated HDL code still remains high, thus preventing their large adoption. Other attempts to create an abstraction layer on top of the HLS compiler tool-chains have been reported in the literature. SparkCL [19] is the attempt to bring spatial architectures to the Spark environment, while in [20] and [21], domain-specific languages (DSL) are used to define finite state machines (FSMs) and to support networking applications. Chisel [22] is a DSL language based on Scala to ease the design and implementation of application-specific architectures. Chisel compiler provides the developers with an easy-to-use environment where they can specify the architecture in a similar way to RTL level. Chisel can target FPGA devices by generating synthesizable HDL code. fBLAS (see [4]) is a portable implementation of the BLAS library targeting FPGA devices; although authors provided implementation only for Intel devices. While there is a growing interest in supporting application code development for FPGA at different levels, big limitations still persist. Among the others, frameworks targeting portability [23] are generally restricted to C/C++ code, since vendors' tool-chains also support C/C++. On the other hand, domain-specific languages are able to catch the needs of only a specific class of applications. As a matter of that, a large body of applications written using high-level programming languages other than C/C++ remains unsupported.

Some notable attempts to fill the gap with these unsupported languages are provided by [12] and in [13]. In the former case, authors rely on a functional programming paradigm to support the generation of RTL code targeting FPGAs. Also, pipelining and vectorization [24] are explored to increase performance of the generated RTL code. The latter automatizes the pipelining and vectorization of the application code, still generating RTL code for FPGA targets. This is one of the few works done, where the input application code is written in Fortran, while experiments targeted (AMD) Xilinx devices. Elements of the proposed tool-chain such as the front-end compiler (which allows for an automatically refactoring of the Fortran code) provided an interesting cue for the development of our methodology. Automatic code refactoring targeting specifically Fortran (along the different language standards) can be found also in the ROSE framework [25] from LNLL supporting the 77-standard), while the CamFort [26] supporting 60-, 77- and 90-standards, and Photran [27] covering Fortran 77- to 2008-standards.

Recent works have been carried out to extend the support to heterogeneous system (including FPGAs) and targeting applications written in Fortran. Recently, Intel

[4] https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/cuu1526001449959.html.

has strongly promoted its heterogeneous programming environment [6]—*oneAPI*—which has been designed to ease code portability across diverse hardware architectures (i.e., CPUs, GPUs, FPGAs, etc.). oneAPI is organized into a layered solution that finds the parallel programming language DC++ at its core. As such, oneAPI offers various tools to map high-level code to diverse low-level back-ends (i.e., OpenCL, and Level Zero interface), including also a CUDA experimental one. This said, oneAPI is able to concurrently support multiple heterogeneous accelerators, by exposing an enriched SYCL programming framework. Despite the high flexibility and large set of suitable target acceleration devices, oneAPI remains a tool mainly devoted to support C/C++ programming, leaving Fortran code less supported. Indeed, Fortran compilation is supported only targeting CPUs and GPUs. In [28], the authors proposed the implementation of an OpenMP-to-FPGA compiler; the main objective of the authors was to provide an easy installation process of the tool chain, support different FPGA boards and vendors, and make a modular and extensible compilation framework. This said, the authors implemented a compilation tool-chain that is able to outline portions of the code that can be directly accelerated by the target FPGA. As such, the source code is annotated with OpenMP: Only a subset of the OpenMP pragmas are converted into an equivalent high-level synthesis (HLS) code. Vendors' tools are used to synthesize the hardware to run on the FPGA. Despite its encouraging results, the proposed tool is limited to C/C++ source codes, since there is not well-known way to generate the low-level intermediate representation (IR) of the function to accelerate as it was in C/C++. While the porting of (large) Fortran code on FPGA appears to be less prominent in the literature, several works attempted to apply methodologies to accelerate Fortran code on GPUs. This said, a proprietary compiler (originally developed by PGI) was developed: The book [29] covers the majority of the technical aspects required to get advantages from the GPU porting. Recently, Nvidia added Fortran to its CUDA offer [30]. Conversely, for our best knowledge, any direct support for AMD GPUs has been made available.

Authors in [11] proposed a complete tool-chain for transforming Fortran code into an OpenCL synthesizable code targeting FPGA. The work is based on a combination of different tools used to: (1) normalizing the source code (i.e., avoiding the deduction of the variable type from the name by explicitly declaring the type, avoiding as much as possible the use of global variable to fit in the separation of memory spaces between the host CPU and the accelerator); (2) converting the target (normalized) Fortran code into an equivalent C-based code; and (3) generate the OpenCL code needed to drive the communication with the accelerator. Thus, the proposed compiler is an end-to-end solution for generating synthesizable FPGA OpenCL code starting from Fortran. Although this is a solution close to our proposed methodology, the paper does not provide more details on how the Fortran code is converted in the proper C-based OpenCL code. In this context, our methodology is in line, by applying simple code transformations to convert target Fortran code into C-based code. Similarly, [31] is a repository providing an open implementation of an OpenCL wrapper for diverse programming languages that do not provide direct support for the code parallelization using the OpenCL facility. As such, the repository provides a wrapper for Perl and Fortran to easily target acceleration devices

(i.e., enables the user to properly define I/O buffers, initialize the devices, etc.), while writing and parallelizing the specific routines are out of the wrapper scope. Authors in [32] largely exploited the features exposed by the Intel oneAPI programming environment to effectively accelerate an HPC code (astrophysics application) on a fully heterogeneous computer system (i.e., a parallel system where each compute node is equipped with GPU and FPGA accelerators). Despite the paper shows a very interesting pathway to effectively mix different accelerators in one single applications, the targeted source code is C/C++. Again, this demonstrates the weakness of the compute accelerator ecosystem in supporting Fortran-based applications.

The work [33] focused on providing a tool for directly translate Python code (actually a Python function is translated) into an equivalent form that can be compiled to a proper FPGA bitstream. To this purpose, the authors use the Numba compiler coupled with Intel OpenCL compiler. While this approach is remarkably useful in all the context where Python code needs to be accelerated, it remains though to apply to Fortran code. Indeed, there is no easy way to generate an intermediate representation (IR, e.g., as used by LLVM) from the Fortran code using available compilers.

In this work [34], a well-known simulation code (*Alya*) in the computational mechanics domain is effectively accelerated using high-performance FPGAs. The targeted code (performing multi-physics simulations) is written in Fortran, where different physical domains are simulated by using specifically designed modules. While the experimental results show how the FPGA implementation provides performance in line with that of a high-end GPU with a lower power consumption, there is only a vague explanation of the methodology applied to translate the original Fortran code into the equivalent C/C++ and OpenCL version. This said, the paper provides interesting guidelines for tuning the FPGA kernel code in such a way maximum performance can be extracted. Compared to our proposed methodology, this work provides interesting hints to optimize the kernel code, while our methodology aims at providing guidelines for translating the original Fortran code into the C-based/OpenCL kernel(s).

5 Conclusion

Recently, FPGA devices entered in the ecosystem of hardware accelerators available within HPC clusters. Some factors driving their growing adoption can be found in the fact that FPGAs are generally less power hungry and provide lower latency in processing data streams, still preserving high flexibility. Unlike well-known GPUs, FPGAs come with more constrains in the way an application developer can exploit these resources. Indeed, the entire flow of coding, testing and debugging is largely time consuming, since kernels must be synthesized into an equivalent digital circuit and then mapped onto the target FPGA device. The situation is even more complex when the developer has to deal with legacy code. In that case, Fortran code is still widely adopted by scientific and engineering communities, while the availability of tools for easing the development of the kernel and their synthesis (high-level synthesis—HLS—tools) is limited to C/C++ programming languages.

Aimed as part of an experience report, the main purpose of this paper is that of guiding the reader to transform (legacy) Fortran code into a synthesizable (and portable) code through a well-defined methodology. To this purpose, several steps have been described, as well as different optimization strategies. With this in mind, most of the operations and transformations described in this work can be automatized, although for some optimization strategies manual intervention has been required. To demonstrate the feasibility of the proposed methodology, we used a test vehicle application which is largely used in the aeronautic domain (CAE), where we isolated a specific routine. We showed that through the proposed approach, we were able to synthesize acceleration kernels targeting both mid-range and high-end FPGA devices (Intel Arria10 and Intel Stratix10) which were made available in the context of two EU projects (H2020 LEXIS project and EuroHPC-JU ACROSS project). Although most of the performance results did not show any advantage with respect to the original code running on the CPU, some interesting and promising results have been achieved. Specifically, only by replicating multiple times the main computing kernels' pipeline we achieved an effective speedup of $\times$ 2.56 when running on the largest device (performance was in part masked by the higher clock frequency of the used CPUs, as well as by the large L3 caches that could accommodate large chunks of the input dataset). Indeed, our test code proved to be limited by memory access bandwidth, due to the non-optimal SDRAM access patterns needed by smoothing data on multi-dimensional arrays. Large performance gains can be obtained by targeting the specific use case and defining a compute architecture using conventional HDL languages (Verilog, VHDL or SystemC), but at the cost of strongly defeating the automation process of translating Fortran code. Power consumption seems to be still in favor of the FPGA accelerators when compared to GPUs, but further optimization is strongly required in order to fully gain advantage. Beside pure performance analysis, these results encourage us to further investigate on the direction already drawn in this work. Large room for further automatizing the process of code refactoring is present and will be part of our future activities.

Future work activities will be also oriented to test the proposed approach by porting larger and complex Fortran code to the FPGA devices (including AMD/Xilinx ones), as well as to test the synthesized kernels with larger (real) datasets that may provide more chances of making emerge the advantages of the hardware acceleration [14, 23, 24].

Acknowledgments This work was supported by the LEXIS Project funded by the EU's Horizon 2020 research and innovation programme (2014–2020) under Grant Agreement No. 825532, and by the EuroHPC-02-2019 ACROSS Project, Grant Agreement No. 955648.

Data availability statement Worth to mention, all the data and code used in the presented study may be accessed after the evaluation of any access requests, due to privacy or other restrictions.

References

1. Horowitz M (2014) 1.1 Computing's energy problem (and what we can do about it). In: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE
2. Al Kadi M (2018) FGPU: a flexible soft GPU architecture for general purpose computing on FPGAs
3. Ma R et al (2021) Specializing FGPU for persistent deep learning. ACM Trans Reconfig Technol Syst (TRETS) 14(2):1–23
4. De Matteis T, de Fine Licht J, Hoefler T (2020) FBLAS: streaming linear algebra on FPGA. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE
5. Zohouri HR et al (2016) Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE
6. https://spec.oneApi.com/versions/latest/index.html
7. Arnone A (1994) Viscous analysis of three-dimensional rotor flow using a multigrid method 435–445
8. Pacciani R, Marconcini M, Arnone A (2019) Comparison of the AUSM+-up and other advection schemes for turbomachinery applications. Shock Waves 29(5):705–716
9. Poli F, Marconcini M, Pacciani R, Magarielli D, Spano E, Arnone A (2022) Exploiting GPU-based HPC architectures to accelerate an unsteady CFD solver for turbomachinery applications. In: IGTI ASME Turbo Expo, June 13–17, Rotterdam, The Netherlands, ASME paper GT2022-82569
10. Vanderbauwhede W, Davidson G (2018) Domain-specific acceleration and auto-parallelization of legacy scientific code in FORTRAN 77 using source-to-source compilation. Comput Fluids 173:1–5
11. Vanderbauwhede W, Nabi SW (2018) Towards automatic transformation of legacy scientific code into OpenCL for optimal performance on FPGAs. arXiv:1901.00416
12. Thomas DB et al (2015) Transparent linking of compiled software and synthesized hardware. In: 2015 Design, Automation and Test in Europe Conference and Exhibition (DATE). IEEE
13. Nabi SW, Vander bauwhede W (2019) Automatic pipelining and vectorization of scientific code for FPGAs. Int J Reconfig Comput 2019
14. Nguyen T et al (2022) FPGA-based HPC accelerators: an evaluation on performance and energy efficiency. Concurr Comput Pract Exp 34(20):e6570
15. Escobar Fernando A, Chang Xin, Valderrama Carlos (2015) Suitability analysis of FPGAs for heterogeneous platforms in HPC. IEEE Trans Parallel Distrib Syst 27(2):600–612
16. Weller D et al (2017) Energy efficient scientific computing on FPGAs using OpenCL. In: Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays
17. Pell O et al (2013) Maximum performance computing with dataflow engines. In: High-performance computing using FPGAs. Springer, New York, pp 747–774
18. Czajkowski TS et al (2012) From OpenCL to high-performance hardware on FPGAs. In: 22nd International Conference on Field Programmable Logic and Applications (FPL). IEEE
19. Segal O et al (2015) Sparkcl: a unified programming framework for accelerators on heterogeneous clusters. arXiv:1505.01120
20. Agron J (2009) Domain-specific language for HW/SW co-design for FPGAs. In: IFIP Working Conference on Domain-Specific Languages. Springer, Berlin, Heidelberg
21. Kulkarni C, Brebner G, Schelle G (2004) Mapping a domain specific language to a platform FPGA. In: Proceedings of the 41st Annual Design Automation Conference
22. Bachrach J et al (2012) Chisel: constructing hardware in a scala embedded language. In: DAC Design Automation Conference 2012. IEEE
23. Cole M (2004) Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput 30(3):389–406
24. Weinhardt M, Wayne L (1999) Memory access optimization and RAM inference for pipeline vectorization. In: International Workshop on Field Programmable Logic and Applications. Springer, Berlin, Heidelberg

25. Liao C et al (2010) A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. International workshop on OpenMP. Springer, Berlin, Heidelberg
26. Orchard D, Rice A (2013) Upgrading fortran source code using automatic refactoring. In: Proceedings of the 2013 ACM workshop on Workshop on refactoring tools
27. Overbey J et al (2005) Refactorings for Fortran and high-performance computing. In: Proceedings of the second international workshop on Software engineering for high performance computing system applications
28. Mayer F et al (2022) The ORKA-HPC compiler-practical OpenMP for FPGAs. In: International workshop on languages and compilers for parallel computing. Springer, Cham
29. Fatica M, Ruetsch G (2014) CUDA Fortran for scientists and engineers
30. https://developer.nvidia.com/cuda-fortran
31. https://github.com/wimvanderbauwhede/OpenCLIntegration
32. Kashino R et al (2022) Multi-hetero acceleration by GPU and FPGA for astrophysics simulation on oneAPI environment. In: International Conference on High Performance Computing in Asia-Pacific Region
33. Uguen Y, Petit E (2018) PyGA: a Python to FPGA compiler prototype. In: Proceedings of the 5th ACM SIGPLAN international workshop on artificial intelligence and empirical methods for software engineering and parallel computing systems
34. Brown N (2021) Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations. In: 2021 IEEE/ACM international workshop on heterogeneous high-performance reconfigurable computing (H2RC). IEEE

Authors and Affiliations

Paolo Savio[1] · Alberto Scionti[1] · Giacomo Vitali[1] · Paolo Viviani[1] · Chiara Vercellino[1] · Olivier Terzo[1] · Huy-Nam Nguyen[2] · Donato Magarielli[3] · Ennio Spano[3] · Michele Marconcini[4] · Francesco Poli[4]

Paolo Savio
paolo.savio@linksfoundation.com

Giacomo Vitali
giacomo.vitali@linksfoundation.com

Paolo Viviani
paolo.viviani@linksfoundation.com

Chiara Vercellino
chiara.vercellino@linksfoundation.com

Olivier Terzo
olivier.terzo@linksfoundation.com

Huy-Nam Nguyen
huy-nam.nguyen@atos.net

Donato Magarielli
donato.magarielli@avioaero.it

Ennio Spano
ennio.spano@avioaero.it

Michele Marconcini
michele.marconcini@unifi.it

Francesco Poli
francesco.poli@tgroup.unifi.it

[1] LINKS Foundation, Via P. G. Boggio, 61, 10138 Turin, Italy

[2] ATOS, Rue de Provence, 1, 38130 Échirolles, France

[3] Via I Maggio, 99, 10040 Rivalta di Torino, Italy

[4] Department of Industrial Engineering, University of Florence, Via di Santa Marta, 3, 50139 Florence, Italy