Received March 24, 2021, accepted April 7, 2021, date of publication April 20, 2021, date of current version May 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3074171

A Highly-Efficient and Tightly-Connected Many-Core Overlay Architecture

RIADH BEN ABDELHAMID[1], YOSHIKI YAMAGUCHI[2], (Member, IEEE), AND TAISUKE BOKU[3], (Member, IEEE)
[1]Graduate School of Science and Technology, University of Tsukuba, Tsukuba 305-8573, Japan
[2]Faculty of Engineering, Information, and Systems, University of Tsukuba, Tsukuba 305-8573, Japan
[3]Center for Computational Sciences, University of Tsukuba, Tsukuba 305-8573, Japan

Corresponding author: Riadh Ben Abdelhamid (ben.abdelhamid.riadh.by@lila.cs.tsukuba.ac.jp)

This work was supported in part by the MEXT through the ''Next Generation High-Performance Computing Infrastructures and Applications Research and Development Program'' (Development of Computing-Communication Unified Supercomputer in Next Generation), and in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP17H01707 and Grant JP18H03246.

ABSTRACT The technology advances of CPU (Central Processing Unit) architecture alternate between generalization and specialization. In the past decade, the general performance has been enhanced while addressing the new brick walls that include power, memory, and ILP (Instruction-Level Parallelism). Thus, it will enter into the era of specialization called adaptable ISA (Instruction Set Architecture) for target applications. Reconfigurable devices such as FPGAs (Field Programmable Gate Array) can offer a solution if the two following issues are addressed. One is the FPGA design is not easy for non-hardware experts, and the other is the process is iterative and lengthy. The most apparent solution to those problems is an overlay that can abstract hardware details while providing a software-like interface. This article presents DRAGON (Dynamically Re-programmable Architecture of Gather-scatter Overlay Nodes), demonstrates its general aspects as well as the way it can be seamlessly integrated into any heterogeneous computing platform. The experimental evaluation of DRAGON reports more than four times better computational efficiency when compared to an Intel Core i9 CPU, in two stencil-based benchmarks.

INDEX TERMS EPR, FPGA, ISA, many-core, overlay architecture, power-efficiency, SIMD, VLIW.

I. INTRODUCTION

There are several metrics to evaluate the performance of a given computer architecture. It can be anything from power-efficiency, peak operations per second, cycles per instruction, latency, and many more. Although general-purpose processors offer the best-in-class versatility to support a wide range of problems, they still lack behind application-specific computer architecture when it comes to sustained performance. Over-optimizing an architecture for a specific set of tasks, would significantly hinder its flexibility and shrink its market audience. Consequently, none of the commercially available processors can efficiently tackle high-performance computing workloads, such as stencil calculations. While these devices are widely being deployed in HPC (High-Performance Computing) systems to accelerate numerical simulations, their sustained performance remains limited in memory-bound computations, mainly because of their poor scalability and inefficient interconnection topologies.

In contrast, FPGAs (Field Programmable Gate Arrays) offer a more appealing feature than just a bare theoretical peak performance number: They have the flexibility to be re-programmed, at the hardware level, as many times as required. Thanks to their versatile nature, they can be efficiently tailored to target a set of new requirements for any specific application needs. Generally, this efficiency allows FPGAs to reach higher performance levels with fewer hardware resources and less consumed power.

Nonetheless, unlike a CPU (Central Processing Unit), FPGAs have abundant physical resources (LUTs, memories, DSP, registers, etc.) that have to be connected on-the-field, through a configuration file, to generate a compute-ready device.

Here, we reach the first downside of FPGAs. In order to produce that configuration file, designers must usually

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

undergo a cumbersome iterative process, from the very first step of elaborating the specifications until providing a fully functional design. Often, this process can last for months, due to the complex nature of HDL (Hardware Description Language) based design as well as the lengthy FPGA compilation steps (placement, routing, etc).

HLS (High-Level Synthesis) tools aim to alleviate hardware design complexity by offering a software-friendly method to generate the final configuration. Although this approach facilitates the task of non-hardware experts, it still generates HDL source files, that, again, have to undergo the lengthy FPGA compilation process. Worse yet, HLS tools are heavily tied to device vendors and thus hinders design portability.

Processor-based overlays [1] are known to overcome these issues by offering a simplified software/hardware interface and even the possibility of dynamic reconfiguration. A Processor-based overlay is a virtual abstraction layer, on top of the physical FPGA fabric, that encapsulates its hardware details while creating a more comprehensive programming interface. In these overlays, the FPGA is statically configured once, while hardware resources are shared in a time-multiplexed manner and the behavior of the system is dynamically changed during run-time through software-like micro-instructions. This would presumably solve two problems simultaneously: The difficulty of programming FPGAs using HDL and the long compilation time. Consequently, this paves the way for non-hardware experts and large communities of software developers to harness the power of FPGAs without worrying about their fine-grained architectural details.

Clearly, previously proposed processor-based overlay architectures offer numerous advantages; yet, there are still major challenges to be addressed. In fact, as explained in details in the next section, these architectures are lacking dedicated hardware support for double-precision floating-point operations. Besides, the complex nature of numerical simulations has led overlay architectures as well as CPU and GPU architectures to struggle to maintain their sustained performance potential somewhere near their advertised theoretical peak and their Effective to peak Performance Ratio (EPR) [2] has generally remained low. In other words there remains a large gap between the theoretical peak performance and what can be really achieved in any given computation. With the increasing challenges of scientific applications, the need to address these issues has become primordial to satisfy the growing computation, communication and precision requirements expected in complex future applications.

Here, we propose a highly-efficient and tightly-connected many-core overlay architecture code-named DRAGON (Dynamically Re-programmable Architecture for Gather-scatter Overlay Nodes). DRAGON abstracts the FPGA fabric details and can be controlled from an OpenCL-based host; thus, facilitating its integration into a computing platform. DRAGON can also be dynamically re-programmed using its custom-designed ISA (Instruction Set Architecture) and achieves higher EPR values when compared to previous works.

DRAGON was first introduced in [3] and is based upon our cumulative work in [4], [5] where the design was partially implemented on FPGA (accelerator part only). Back then, the controller part was not yet finalized and its continually changing behavior had only been emulated through a software environment. Consequently, in order to evaluate the performance outcome, we relied solely on simulations where we abstracted all the details about the controller part and the way in which the Global Memory is being accessed. In contrast, here we provide analysis on the full architecture implementation on FPGA and therefore the sustained, on-chip, execution results. While this renders a direct performance comparison infeasible with our previous work, we still explain the major architecture changes that can benefit the overall system. Our main contributions are stated as follows:

- A highly modular and computationally-efficient overlay architecture, that adopts a custom-design instruction set and implements VLIW (Very Large Instruction Word) and SIMD (Single Instruction Multiple Data) paradigms as well as a custom DAE (Decoupled Access Execute) approach.
- A versatile VLIW-based, processing element architecture that supports both 64-bit long integer and double-precision floating-point fused multiply and accumulate operations, as well as efficient data movement (scatter/gather, broadcast) between nodes in a many-core processor design.
- A base 2D-mesh, switchless, buffered, point-to-point Network-On-Chip, that is both scalable and expandable to higher dimensions.
- A seamless and easy way to integrate an FPGA overlay into a heterogeneous computing system through an OpenCL host.

Finally, This paper focuses on the architectural aspects of DRAGON and while the current implementation uses an HBM2-enabled (High Bandwidth Memory) FPGA, a study of the corresponding bandwidth is outside the scope of this paper.

This paper is structured as follows: Section 2 reviews previous related research and summarizes the motivation behind this work. Section 3 gives an overview of the proposed DRAGON architecture. Section 4 focuses on the hardware aspects. Section 5 focuses on the software part and presents the details of our custom Instruction Set Architecture (ISA). Section 6 presents the conducted experiments to evaluate the performance of the architecture through its implementation on an HBM2-enabled FPGA. Section 7 presents the results and discusses the strengths and pitfalls of the proposed DRAGON architecture. Finally, we draw our conclusion.

II. BACKGROUND AND MOTIVATION

Research about overlays has a long history, and several innovative approaches have been proposed over the past years.

TABLE 1. Review of some previous parallel processing overlays.

Ref	Year	Name	FPU	ALU	Topology
General-purpose					
[6]	2012	reMORPH	None	variable	2D Mesh
[7]	2013	TILT	32-bit	None	Crossbar
[8]	2015	SIMD-Octavo	None	36-bit	SIMD Lanes /Mesh
[9]	2016	GRVI	None	32-bit	2D Torus
[10]	2019	2GRVI	None	64-bit	2D Torus
Application-specific					
[11]	2010	SCMA	32-bit	None	2D Mesh
[12]	2014	SSA	32-bit	None	1D Torus x 1D Mesh
Hybrid approach					
Ours	2020	DRAGON	64-bit	64-bit	N-D Mesh/Torus

The work in [1] surveyed various kinds of overlays and suggested that overlays can be either classified as spatially configured or time-multiplexed. A spatially configured overlay means that the behavior of its functional units is unique and cannot be changed during run-time. In contrast, in a time-multiplexed overlay, these same units do have the ability to adapt their behaviors over time.

Examples of such overlays are given in Table 1 and include the SIMD-Octavo [8], which adopts the SIMD paradigm to create a parallel processor-based overlay. SIMD-Octavo was an extended version of the Octavo soft-processor [14], in which a single stream of instructions was shared among duplicated data-paths.

The most interesting implementation to date is the GRVI Phalanx, which is a massively parallel FPGA overlay [9] built around an efficiently hand-tuned version of the RISC-V processor that uses only 320 LUTs and achieves a clock speed of 375 MHz.

A recent update of GRVI Phalanx, the 2GRVI Phalanx [10], targets a 64-bit version of the RISC-V processor and implements 1332 cores on an HBM2-enabled Xilinx Alveo acceleration board (U280-ES1). Nevertheless, both implementations omitted any computational performance analysis.

The reMORPH overlay is another unique idea that is likely to have the most resource-efficient implementation [6]. With its functional units consuming just 1 DSP, 3 BRAMs, 196 LUTs, and 41 FFs, this makes it highly likely to be the smallest overlay in terms of resource utilization. This allowed reMORPH to target relatively small FPGAs (Xilinx Spartan 6 FPGA), where it was able to implement 40 tiles named CGRM (Coarse-Grained Reconfigurable module).

The reMORPH approach was to map their ALU with a 5-stage pipeline into a hardened DSP unit (using the internal DSP pipeline) while coupling it to Block RAMs that contain the program instructions. This implementation idea has allowed reMORPH to reach a relatively high clock speed of 400MHz.

Previous work such as TILT proposed an overlay built with VLIW in mind. TILT overlays comprise multiple pipelined 32-bit floating-point units, with a configurable depth, and connected through a crossbar [7].

Generally speaking, these overlays had been proposed for general-purpose computing using parallel processing architectures. Their goal had been almost always the same: decreasing the PE (Processing Element) size, increasing the number of PEs and maximizing the operating frequency in order to boost the overall computational performance. Nonetheless, they all have one thing in common, they do not offer dedicated hardware for double precision floating-point operations and apart from TILT, they do not even offer support for single-precision floating-point operations, while at the best case, they propose this feature as a possible extension [9]. The reason is simple, FPUs (Floating Point Unit) are costly in hardware, mostly because of the required wide multipliers, that occupy large areas and limit the achievable clock speed. Specialized overlays came along as an alternative that offers that capability. In particular, specific-purpose architectures that target stencil-based computation acceleration, have been proposed with execution units capable of doing floating-point operations. Yet, in most cases, they remain limited to single-precision in order to maintain higher GFLOP/s and do not offer any integer execution units [11], [12].

Ultimately, all these overlays were proposed without clarifying how they would interact with a host in order to be integrated for example in a computing platform.

Meanwhile, next generation computing would be rather bottle-necked by energy consumption than computational performance.

Seemingly, FPGAs are praised for their power efficiency and may replace general purpose processors in future computing platforms, if they are fast to use (no compilation time), easy to integrate (software interface) and computationally efficient (performance-oriented architectures that achieve a sustained performance close to the theoretical peak) while providing accurate results (at least double-precision floating-point capability).

Here, our proposed DRAGON architecture tries to achieve all these goals and comes as a possible alternative to existing overlays. For example, DRAGON offers both double-precision floating-point capability and 64-bit integer execution units along with a minimal instruction set that is versatile enough to cover a wide spectrum of application domains.

Furthermore, the DRAGON architecture adopts most sort of parallel processing concepts, such as pipeline parallelism, a SIMD model as well as a VLIW design approach. Condensing these concepts into our general purpose architecture allowed DRAGON to achieve high computational efficiency (translated through EPR values) and to compete against specialized overlays and even fixed implementations tailored to a specific problem, as shown in Table. 2.

In addition, DRAGON is programmable through a custom-designed instruction-set and can be controlled from an OpenCL-based host; thus, facilitating its deployment into heterogeneous computing platforms.

Finally, while FPGA devices are increasingly gaining their spot across numerous fields, they still remain far from

TABLE 2. Review of the Effective to Peak Performance Ratio (EPR) of some previous Processing Element-based architectures implementing scientific kernel benchmarks on FPGA.

Ref	Year	FPGA board	Architecture	Description	Programming	Benchmark (2D)	EPR (%)	
							Single precision	Double precision
[11]	2010	DN7000k10PCI	Stencil	Verilog	Assembly	Red-black-SOR Fractional-step method FDTD method	82.5 87.5 80	N/A N/A N/A
[12]	2014	DE3	Stencil	Verilog	Assembly	4-point Jacobi	87.4	N/A
[2]	2017	DE5	Stencil	OpenCL	-	Laplace 4-point Jacobi 5-point Jacobi FDTD	92.8 61.1 68 76.7	N/A N/A 55.7 N/A
[13]	2019	Nallatech385	Stencil	OpenCL	-	Laplace 5-point Jacobi	48.2 55.8	33.7 30.4
This work	2020	Alveo U280	General-purpose	SystemVerilog	Assembly	Laplace 4-point Jacobi 5-point Jacobi	87.4 87.4 89.9	87.4 87.4 89.9

mainstream adoption because of all the previously stated obstacles. In this paper, we hope, through our proposed overlay architecture and integration approach, to bring FPGA closer to mainstream adoption.

III. A BIRD's EYE VIEW ON THE DRAGON ARCHITECTURE

In this section, we provide insights on the general architecture of DRAGON as well as its parallel processing features. Later, we will discuss some of these features as well as the architecture itself in more details.

A high-level overview of DRAGON is depicted by Fig. 1. It shows that DRAGON is split into two main components operating in tandem. The Controller part is responsible of decoding and issuing instructions via a sequencer, interacting with the GM (Global Memory) and transferring data back and forth to and from the Accelerator part. The latter is the computing core of DRAGON and executes those instructions on an array of PEs (Processing Elements) grouped in relatively small clusters (Broadcast Clusters). Then, it sends back the results of the computation to the GM through the Controller's DMA (Direct Memory Access) interfaces. Dragon communicates with the host through a PCIe interface. A PCIe DMA engine exchanges the data between the host and the GM while a dedicated custom DMA is used to transfer program instructions from the GM to the IM (Instruction Memory).

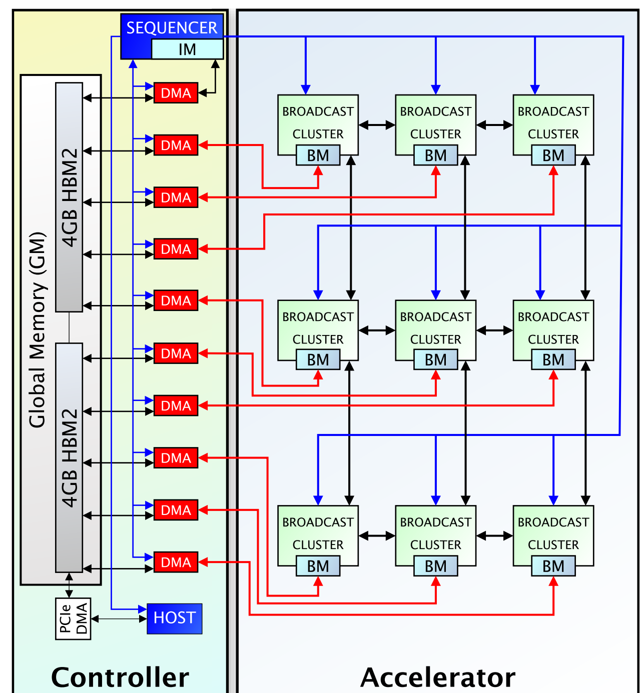


FIGURE 1. Overview of the DRAGON architecture.

A. A SOFTWARE COUPLED HARDWARE DECOUPLED ACCESS EXECUTE APPROACH

DRAGON adopts a programming model where data movement management logic is separated from the execution logic. This approach is known as DAE (Decoupled Access Execute) and was first proposed in [15]. The original work suggests a high degree of decoupling between data movements and execution by implementing two separate streams of instructions interacting with each other through hardware-based queues. While this approach worked well at the moment of publication, it might not be suitable for today's level of complexity in many-core processing systems. First, having two separate and different types of instruction streams requires the implementation of two compilers to generate the machine binary executable file for each stream. Even worse, deadlocks might occur according to the same paper which requires a purge of the program whenever a deadlock error is flagged. While DRAGON adopts the same model, it implements a different logic that solves these issues while keeping the benefits of such an approach. Here, DRAGON uses the same sequencer to issue the two different instruction streams, one computation stream targeting the accelerator and a data movement control stream targeting the DMA engines. Consequently,

a deadlock-free synchronization between both streams is guaranteed while a single compiler is required to generate the executable program. Furthermore, no queues are required and the data is stored in BM (Broadcast Memory), which is an intermediate level of memory that can be randomly accessed from both Accelerator and Controller sides and that serves as a buffer between GMs and LMs (Local Memories).

B. SIMD PARALLELISM

The DRAGON architecture adopts the SIMD data parallel approach. This class of computer architecture consists of broadcasting a single instruction (or operation) to multiple compute units that will execute it, locally, on different sets of data [16]. This paradigm was selected for many reasons. First, it maps well to the fine-grained regular structure of FPGAs. Second, parallelism is straightforward and programming is fairly simple. Third, it simplifies the design (less control is required) and finally and most importantly, it allows designers to fit more compute units thanks to the reduced footprint of the control logic, hence, providing an efficient way to harness the highest levels of performance. From a software perspective, this is the go-to paradigm for many large-scale computational workloads, in order to improve their throughput while preserving, to some extent, simplicity in the programming step. On the other hand, from a pure hardware perspective, SIMD allows designs to consume less power and area, by removing redundant control-dedicated logic.

C. VLIW PARALLELISM

The VLIW approach is often referred to as a design style or philosophy rather than an architectural approach of parallel processing [17]. VLIW-based processing systems explicitly expose instruction parallelism to the programmer, (in the architecture level) rather than imitating ILP-oriented (Instruction Level Parallelism) processors and relying on the compiler to fulfill this task. The adopted VLIW design approach consists of splitting the instruction on each processing element into two packets. The first is dedicated to the computation slot while the second targets the memory slot. This guarantees synchronized overlapping of computations with data movements.

IV. HARDWARE ARCHITECTURE

A. THE DRAGON CONTROLLER

1) THE SEQUENCER

The sequencer is the brain of the DRAGON. As depicted by Fig. 2, it consists of the IM (Instruction Memory) which is a memory that stores program instructions on the FPGA, an AXI (Advanced eXtensible Interface) Lite Control Interface that is linked to the host to obtain or communicate configuration parameters for proper functioning (program size, global memory pointers, start, done and program re-use flags) and a Control Unit that orchestrates the overall operations, both inside the Controller and the Accelerator parts. The host triggers the overlay by setting a start bit into a dedicated register of the AXI Lite Control Interface. The Control Unit continuously polls that bit to check if it was set by the host. Once this start bit is set, the Control Unit starts the boot sequence, when there is a request from the host to store the program instructions into the IM. In the case where the same program is being used with different data, the Control Unit bypasses the boot sequence and directly starts normal operation. During the boot sequence, only one DMA is active. This DMA moves instructions from GM to IM. The size of the program in bytes is set through the host and is used by the Control Unit to configure the IM-dedicated DMA.

After all the program instructions had been stored into the IM, the Control Unit moves into the normal operation state. During this state, a PC (Program Counter) increments every clock cycle to read the program instructions from the IM. These instructions are sent back to the Control Unit that decodes them and issues specific control streams, depending on the decoded opcodes. When the decoded instruction is a data movement operation between GM and BM, the Control Unit will output a DMA control stream to configure the data DMAs while setting NOPs (No Operation) on the dual compute slot stream (DC_slot_stream), the memory slot stream (mem_slot_stream) and the BMC (Broadcast Memory Controller) stream (MC_stream) shown in Fig. 2.

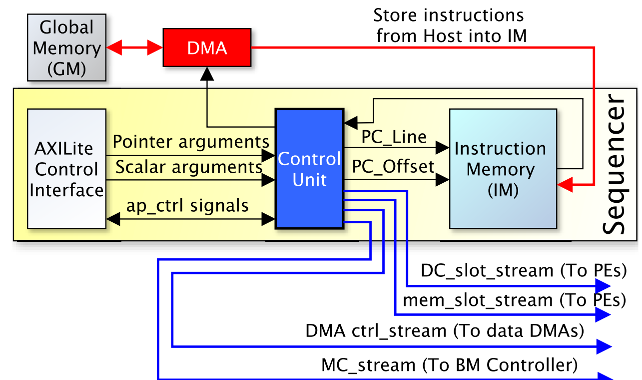


FIGURE 2. Overview of the DRAGON Sequencer.

The sequencer handles loops inside the program and can implement up to seven levels of nesting. The end of the program is flagged by a special instruction (STOP instruction). This instruction should be used when all the data had been processed and sent back to GM. After encountering this instruction, the Sequencer will set a special control register bit into the AXI Lite Control Interface to notify the host about the completion of the program execution.

2) THE INSTRUCTION MEMORY

The IM (Instruction Memory) shown in Fig. 3 stores the executable program loaded from GM during the boot sequence. It uses 16 Ultra RAM banks offering a combined storage capacity of 512 KiB. These banks are arranged into 8 logical

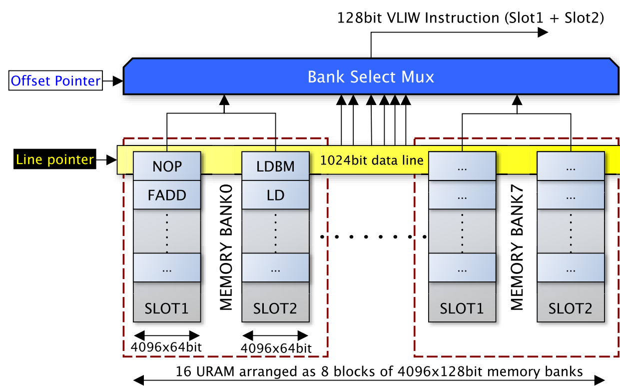


FIGURE 3. Physical and logical layout of the Instruction Memory.

memory banks that can store up to 4096 VLIW instructions, each. A VLIW instruction is 128 bits wide and is composed of 2 64-bit slots.

A physical PC (Program Counter) is used to increment the current IM address at which the program is pointing. This PC is split into two logical parts. The first is the LP (Line Pointer) and is aligned to 128 Bytes. The LP points to the same address line of the 8 logical memory banks. The second is the OP (Offset Pointer) and is used to extract the effective VLIW instruction, among 8 VLIW instructions pointed to by the LP. A specific DMA engine is dedicated to store the program instructions into the IM during the boot sequence. This means that currently the program size is limited to 512 KiB.

As DRAGON evolves, this IM will serve as a program cache and the program instructions will be cached during execution time from on-chip HBM or from an external DDR4 bank that can offer a larger storage.

The current physical layout of the IM was chosen to match the adopted AXI bus width connected to the GM and which is 1024 bits wide. By placing 16 Ultra RAM memories in True Dual Port (TDP) mode, 16 ports can load the program instructions from GM, while the 16 other ports are connected to the bank select multiplexer, that transfers the read instructions to the Control Unit.

3) THE DMAs

A DMA is a data mover engine that can transfer data between the Accelerator's BMs and their respective GM banks. A DMA is configurable through the Control Unit following a request from a DMA-specific instruction. The configuration frame contains the direction of the transfer (read/write from/to GM), the address offset of GM, the address offset of BM and the number of byte bursts that have to be transferred. A single request can result in a maximum of 4 KB of data being transferred.

All the DMAs operate in parallel, apart from a single DMA that is only active during the boot sequence and that is used exclusively to store program instructions into the IM. Using a separate DMA for loading program instructions had been adopted because future generation DRAGON is expected to cache its instruction continuously from the GM, not only during the boot sequence.

4) OpenCL-READY FPGA OVERLAY

DRAGON is designed to offer a convenient alternative for non-hardware experts that intend to use FPGAs for accelerating their applications. While FPGA overlays are a good mean to abstract physical fabric details, mostly, they do not do a great job when it comes to managing the communication with a host. Some very low-level details about firmware programming are obviously required, for instance, when moving data between a host and the FPGA through a PCI-express interface.

Here, we introduce a new approach, that allows to abstract these communication details as well. For this purpose, we packaged our design as an OpenCL kernel as defined and required by Xilinx Vitis RTL kernel flow. This tool considers a carefully packaged RTL (Register Transfer Level) design as a software function prototype that has global memory arguments and scalar arguments.

The global memory arguments are in fact pointers to the address ranges allocated into the HBM memory banks (which are connected to the DMAs as depicted by Fig. 1).

On the other hand, the scalar arguments are the configuration parameters that are transferred to the design through the AXI Lite Control Interface. These scalar inputs provide for example, the necessary information about the size of the program to be stored into the IM, the AXI pointers for every global memory bank or even a set of control signals that can trigger the system in some custom user-defined ways.

The design implements an AXI4 compliant interface to communicate with the GM and was packaged as an RTL kernel along with its XML (Extensible Markup Language) interface definition file. The Vitis framework recognises the packaged IP (Intellectual Property) and prepares the FPGA shell with a static and a dynamic regions as depicted by Fig. 4.

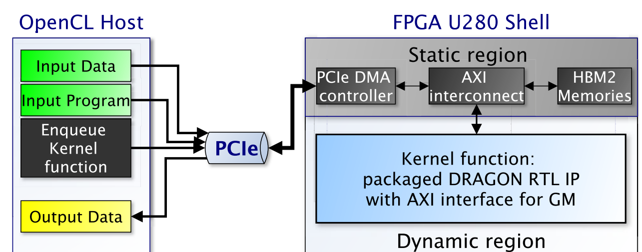


FIGURE 4. Overview of the OpenCL-Host/DRAGON control and communication scheme.

The dynamic region is where the design should be placed and is completely accessible to the IP designer, whereas the static region contains all the infrastructure required by the host to communicate with the packaged IP.

Once the FPGA compilation had been completed, the DRAGON IP can be enqueued as a standard OpenCL task using an OpenCL-based host program. This program

can initialize the data into the GM, instructs the DRAGON "kernel task" to start its execution and moves back the results of the computation to the host. All the PCI-express-based communication details between the host and the GM as well as the details of the AXI interconnect are abstracted and automatically implemented by the Vitis framework.

Here, we consider DRAGON program instructions to be part of the data. First, OpenCL buffers are allocated into the host for all the data that are to be processed as well as the program instructions that are to be stored into the IM. Then, the content of these buffers are transferred to the GM (into the FPGA) and the host sets the start signal (a single-bit register) into the overlay, right after the host OpenCL program enqueues the kernel task. Here, the DRAGON kernel acknowledges the start bit through a handshaking mechanism then triggers the boot sequence when required. The boot sequence allows storing the program instructions into the IM and can be bypassed when the program has to be reused. Once the boot step is completed, the overlay starts the execution of its stored instructions, the input data are read through DMAs, processed into the accelerator and finally, the results of the computation are written back to GM.

To finalize the operation of the kernel, a STOP instruction sets an 'end of program' notification signal that generates an interrupt to the host. This interrupt signal informs the host that the kernel task has been completed and that data are finally ready to be moved from GM to the host memory. This approach offers a higher abstraction scheme for the FPGA and completely hides the communication details from the end users.

B. THE DRAGON ACCELERATOR

1) THE BROADCAST CLUSTER, THE BROADCAST MEMORY AND THE BROADCAST MEMORY CONTROLLER

A modular design requires one or more levels of granularity. The BC (Broadcast Cluster) enhances the modularity of DRAGON and eases its maintenance by implementing a small cluster of 16 tightly-connected PEs (here 2D Mesh interconnection), organized in a 2D grid, and communicating with the BM through a BMC (Broadcast Memory Controller). The BM is at the 3rd level in the memory hierarchy of the DRAGON architecture, the first being the Register File, the second being the LM and the last being the GM. In [3]–[5], the BM was implemented as a single dual-ported block that is composed of 16 Ultra RAMs. The inputs and outputs of the first port were shared by all PEs. The data was loaded from BM to LMs of the same BC in a sequential time-multiplexed manner using masks. This port-sharing scheme was adopted to preserve the broadcast functionality in which the same data could be sent from one location of BM to all LMs in the same clock cycle. Unfortunately, this method hinders a valuable resource which is the higher number of ports that can be used if the BM was split into 16 banks of equal sizes. In the current implementation, we adopted this approach to maximize the internal bandwidth. In fact, every BM is connected to its own HBM bank through a data bus that has a width of 1024 bits. This width was selected to allow transfer of 16 64-bit data to BM, through the DMA side, at every clock cycle. To match this bandwidth within the Accelerator, every single BM bank port was connected to a single PE. In order to preserve the broadcasting feature, we implemented a two stage multiplexing logic as depicted by Fig. 6. The first stage multiplexer has a selector called offset whose value is extracted from the BrOffset field in Fig. 9. It selects a single datum among the 16 outputs of the different BM banks. The second stage has 16 multiplexers, one for each BM bank. These multiplexers will then select between the output of the first stage or the output of the BM bank with the same rank, in which case, all PEs can access their corresponding BM banks separately and concurrently. Nonetheless, in the first case, the broadcast feature is still preserved while the same datum is broadcasted to all PEs through different wires. This datum can come from any BM bank, thus all these banks are accessible to all PEs and the original BM size in previous implementation can be emulated by combining storage capacity of all 16 BM banks.

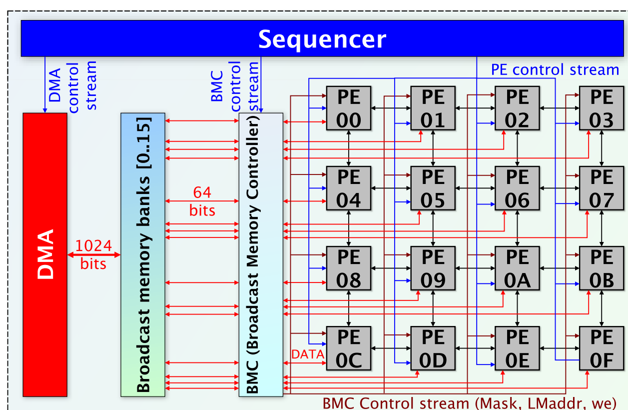


FIGURE 5. Architecture of the Broadcast Cluster.

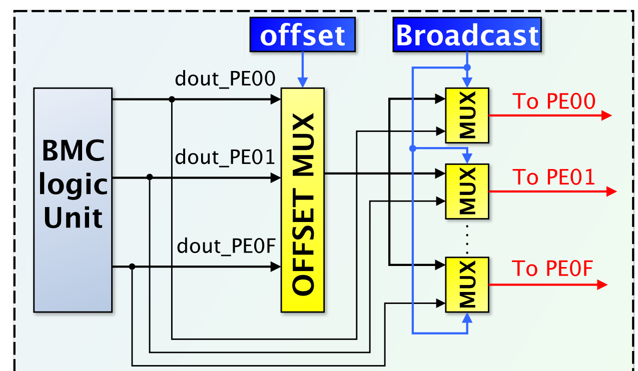


FIGURE 6. Implementation of the data broadcasting feature.

Moreover, in [3]–[5], since the BM data input was shared by all the 16 PEs in the cluster, storing data to BM was done

in a sequential manner. A sampler was used to register the PE data outputs and to serialize them to the BM input port. Here, since the BM is split into 16 different banks, each one of those banks has its own input port and every PE can be connected to that port as depicted by Fig. 5; thus, the PE output bandwidth is 16 times higher. This implementation profits from the large number of HBM banks and can still be implemented on non-HBM FPGAs with the latency cost of time-sharing. The only downside of this approach is that the size of BM is now reduced to 1/16 of its original size. Nonetheless, using the broadcast feature, all data, in all 16 banks can be accessed from any PE, which emulates the original BM size. Finally, The BMC acts as a half duplex DMA that manages bulk data transfers from BM to LM following a call to an LDBM instruction. Similarly to previous implementations in [3]–[5], the same data management logic had been preserved, except, now, the BMC manages 16 separate channels, one for every BM bank/PE pair.

2) THE PROCESSING ELEMENT

While the Sequencer is the brain of DRAGON, the PE is its heart and the core part of its architecture. This PE was designed as a pure computational unit and evolved over time to implement new instructions and adapt to new requirements. It consists of a 64-bit micro-coded programmable compute unit designed with versatility in mind. It provides support for both integer and double-precision floating-point operations as well as a handful set of memory and neighbor communication operations. The PE implements a 7-stage-pipeline and executes statically scheduled VLIW instructions that are split into two parallel execution slots.

The PE contains two compartments as depicted by Fig. 7. In fact, The upper compartment is the DCS (Dual Compute Slot) and the latter is the MS (Memory Slot). The DCS implements a Register File capable of addressing 256 different 64-bit wide registers, a 64-bit integer ALU (Arithmetic and Logic Unit) as well as a double-precision FPU (Floating Point Unit) whose details are illustrated by Fig. 8. This FPU is capable of a low-latency fused multiply-accumulate operation (Only 3 cycles required). This behavior is implemented by chaining a double-precision floating-point multiplier to a double-precision floating-point adder. To accommodate for the short latency, the default rounding mode implemented for all operations is truncation.

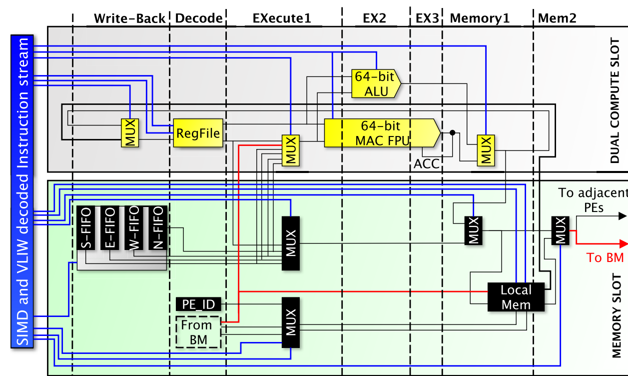


FIGURE 7. Micro-architecture of the Processing Element.

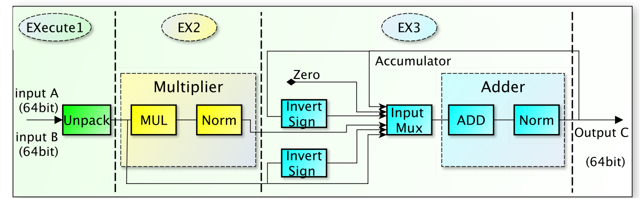


FIGURE 8. Details of the Floating Point Unit.

The FPU, as the totality of the implemented system, is completely written in pure SystemVerilog and do not use any vendor-specific library. This applies also to memory resources such as Ultra RAM that are inferred and not instantiated. The goal is to ensure a better portability and more design freedom.

While the DCS is mostly a computational block, the MS manages incoming and outcoming data operations as well as LM operations. The LM is capable of reading and writing to the Register File in the same clock cycle. The MS contains a set of input buffers that are implemented as cyclic FIFOs capable of continuously storing incoming data from adjacent PEs. The DCS and MS compartments of the PE provide a vessel to implement VLIW-based multiple-issue of instructions (up to 2). For example, A PE can execute a computation through the FPU, scatter the result to adjacent PEs, while loading a new data from LM to the Register File or from BM to LM. This overlapping of computation with the memory transfers allows the PE to achieve the highest levels of efficiency.

A total of 28 instructions are supported by DRAGON. These instructions cover computational integer-based and floating-point-based operations. They also provide support for memory operations such as loads and stores as well as neighbor communication operations such as scatter and gather.

The PE architecture allows operations on operands arriving from the broadcast memory or even from the input FIFO buffers that contain gathered data from the neighboring PEs.

3) THE NETWORK TOPOLOGY

DRAGON is a tightly-coupled architecture where all PEs are inter-connected through a direct network. This interconnection follows a switchless topology, where each PE represents a node and each node has a set of connected neighbors whose number is defined by the dimension level of the network. The DRAGON network interconnection approach is based on buffers that store the incoming data from adjacent or even distant PEs. In the current implementation, a 2D Mesh topology had been adopted to simplify the design in order to allow it to fit into a packaged RTL kernel as described earlier. Nevertheless, we have shown in [4], [5] how the base architecture could implement a 3D Torus or even achieves a 4D Torus interconnection as presented in [3]. The limitation

here comes from the number of physical SLLs (Super Long Lines) that has to cross the SLRs (Super Logic Regions) in the FPGA [18], in order to transfer the data from the HBM banks to their corresponding BCs.

In fact, the current version of DRAGON assigns every BC to a single HBM bank (or a few contiguous banks), through a 1024-bit wide AXI port to transport 16 64-bit data to every 16 PEs in a BC. However, the number of available SLLs was not sufficient for our original implementation with four by four BCs and 4D Torus. This limitation, along with the reduced resources that are automatically allocated by the FPGA shell to the static region in Vitis RTL kernel flow, constrained us to resize our implementation to a grid of three by three BCs and a 2D Mesh network.

Regardless of hardware resource count and from a pure architectural perspective, every PE has to embed a specific number of input buffers, depending on the dimension of the desired topology.

Hence, a 2D Mesh/Torus requires 4 input buffers to accept incoming data from 4 directions, namely North, West, East and South neighboring PEs.

In a 3D Mesh/Torus topology, there are 2 additional distant PEs that would impose adding 2 extra input buffers, one for each newly connected PE.

Similarly, A 4D Mesh/Torus topology requires another two additional input buffers to store the incoming data from the extra two distant PEs, which makes the total input buffers count elevates to eight, four from local and four from remote North, East, West and South directions.

TABLE 3. The DRAGON Instruction Set.

Mnemonic	Function description
NOP	No operation
LDimm	Load 64-bit value in the RegisterFile
Data processing (64-bit Integer), R-Type	
ADD	Integer addition
SUB	Integer subtraction
AND	Bitwise logical AND
OR	Bitwise logical OR
XOR	Bitwise logical XOR
SLL	Shift Logical Left
SRL	Shift Logical Right
MUL	Multiply lower 32 bits of both operands
Data processing (Double-precision Floating-Point), R-Type	
FADD	Floating-point addition
FSUB	Floating-point subtraction
FMUL	Floating-point multiplication
FMACCA	Floating-point Multiply-Add-Accumulate
FMACCS	Floating-point Multiply-Subtract-Accumulate
BM memory transfer operations, BM-Type	
LDBM	Load Data from BM to LM
STBM	Store data from PE to BM
Register-LM memory transfers, LM-Type	
LD	Load from LM into Register File
ST	Store from Register File to LM
Neighbor communication operations, N-Type	
NSG	Scatter/Gather to/from adjacent PEs
BFLUSH	Reset read/write pointers of fifo input buffers
NPASS	PASS data from an input buffer to adjacent PE
NST	Store from an input buffer into LM
Controller-scope-limited instructions, C-Type	
REPEAT	Loop for a number of iterations
BNZ	Check loop counter then branch if not zero
RDGMEM	Configures DMA to pass data from GM to BM
WRGMEM	Configures DMA to pass data from BM to GM
STOP	Flags the end of a program

V. THE INSTRUCTION SET ARCHITECTURE

The proposed DRAGON ISA (Instruction Set Architecture) uses just 28 instructions that are stored in a little-endian order and formatted in a 64-bit wide format. The formatting of these instructions is designed in a way that mostly preserves the position of their fields. This is extremely important to reduce critical path issues for high-fanout signals across many implementations [19]. The fixed width of these instructions, in contrast to dynamically variable CISC-style instructions, forces its effects to be determined at compile time.

Depending on the type of the operation, there are currently five formats to encode the instructions, out of which, there are four being dedicated to the Accelerator and one solely to the Controller. These formats are classified as follows: **R-type**, **LM-type**, **BM-type**, **N-Type** and a Controller-specific **C-Type**. Fig. 9 illustrates the different formats used for encoding the DRAGON instructions.

A. R-TYPE INSTRUCTION

The **R-type** refers to Register-type format and regroups all the instructions that deal with computational operations. In these operations, the first operand comes always from the Register File, whereas the second operand is selected through a multiplexer and can either originate from this Register File, from one of the communication buffers (that store inputs from adjacent PEs) or directly from the BM.

Pseudo-instructions involving an immediate operand could be realized by setting the ''OPSrc'' field of the instruction to a specific value as shown by Fig. 9.

A computational instruction can be followed immediately by a local store to LM or even a scatter-gather operation by setting the ''mode'' field of the instruction.

The **R-Type** includes as well the (LDimm) instruction that allows to load a 64-bit immediate value into the Register File. (LDimm) uses the first VLIW instruction slot to load the upper 16 bits of the immediate value and the second slot to load the remaining bits.

B. LM-TYPE INSTRUCTION

The **LM-type** refers to Local-Memory-type and deals with memory transfer operations that move the data between the LM and the Register File (ST and LD). While R-Type operations can implement pseudo-instructions that store computed results into the LM, the latter can allow a concurrent load operation when the second VLIW slot is set to an (LD) instruction. In other words, storing data to the LM while loading from it to the Register File can be done in the same clock cycle.

Register data operations:			NOP, ADD, SUB, AND, OR, XOR, SLL, SRL, MUL, FADD, FSUB, FMUL, FMACCA, FMACCS					[R-type]
opcode	Src1	mode	Lmaddr	BrOffset	Bmaddr	Src2	RDst	OPSrc
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	8 bit	4 bits
ADDi, SUBi, ANDi, ORi, XORi, SLLi, SRLi, MULi (pseudo instructions), LDimm								
opcode	Src1	mode	Lmaddr	immediateMSB / immediateLSB (SLOT2)		unused	RDst	OPSrc
6 bits	8 bits	2 bits	12 bits	16 bits		8 bits	8 bits	4'b0001
ADDST, SUBST, ANDST, ORST, XORST, SLLST, SRLST, MULST, FADDST, FSUBST, FMULST, FMACCAST, FMACCSST (pseudo instructions)								
opcode	Src1	mode	Lmaddr	BrOffset	Bmaddr	unused	NDst	OPSrc
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	8 bits	4 bits

Local Memory operations:		ST			[LM-type]
Opcode	unused	Lmaddr	unused	Src2	unused
6 bits	10 bits	12 bits	16 bits	8 bits	12 bits
LD					
Opcode	unused	Lmaddr	unused	RDst	unused
6 bits	10 bits	12 bits	24 bits	8 bits	4 bits

Broacast Memory operations:			STBM				[BM-type]
Opcode	unused	mode	Lmaddr	unused	Bmaddr	Src2	unused
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	12 bits
LDBM							
Opcode	Mask_load	mode	Lmaddr	BrOffset	Bmaddr	unused	data_count
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	12 bits

Neighbor communication operations:			NST, NPASS, NSG, BFLUSH				[N-Type]
Opcode	unused	mode	Lmaddr	unused	Src2	NDst	NSrc
6 bits	8 bits	2 bits	12 bits	16 bits	8 bits	8 bits	4 bit

controller operations:		REPEAT, BNZ		[C-Type]
Opcode	Function	iterations		unused
6'b111_111	6 bits	20 bits		32 bits
RDGMEM, WRGMEM				
Opcode	Function	Burst Size	BMOffset	GMOffsetMSB/GMOffsetLSB(SLOT2)
6'b111_111	6 bits	8 bits	12 bits	32 bits
STOP				
Opcode	Function	unused		
6'b111_111	6 bits	52 bits		

FIGURE 9. DRAGON ISA formatting.

C. BM-TYPE INSTRUCTION

The **BM-type** refers to Broadcast-Memory-type and consists of the instructions that deal with memory transfers between LM and BM (LDBM and STBM). An (STBM) instruction stores the data into BM, either from the Register File, from LM or directly from the output of either the FPU or the ALU. A single call to (LDBM) can load a burst of data to LM. The "data_count" field of the instruction, shown in Fig. 9, sets the amount of the data to be transferred. The "Bmaddr" and "Lmaddr" fields set the base addresses of BM and LM, respectively.

A single memory bank can be used to broadcast the data to all PEs by setting the "mode" and "BrOffset" fields to the adequate values.

The "Mask_load" field of the instruction allows targeted data transfer to a subset of PEs inside the BC. The lower four bits of this field instructs the starting PE, whereas the upper four bits instructs the number of PEs to be targeted, counting from the starting PE.

D. N-TYPE INSTRUCTION

The **N-Type** deals with neighbor communication operations such as scatter-gather (NSG), loading data from the input buffers into LM (NST), flushing the read/write pointers of the communication buffers (input FIFO buffers) (BFLUSH) as well as providing the possibility of passing data between 2 PEs (NPASS) in a controlled direction.

The combination of the "NSrc" field bits selects the source adjacent PE data for (NPASS) and (NST) instructions, while the "NDst" field selects the destination buffer for (NSG) instruction.

E. C-TYPE INSTRUCTION

The **C-Type** is a special type that is solely dedicated to the Controller and which regroups just 5 instructions. The corresponding Opcode field is fixed as shown in Fig. 9, whereas the "Function" field is used instead to denote the behavior of the instruction. The (REPEAT) instruction implements a loop over a number of iterations, while the (BNZ) instruction checks the content of the loop counter and exits the loop when it reaches zero. A hardware stack is used to store the values of each loop counter and can implement up to seven levels of nesting.

The (RDGMEM) and (WRGMEM) are two instructions that configure the DMA engines to read (from GM to BM) and write (from BM to GM) data. These two instructions span across two slots because the address offset for the GM has to be on a 64-bit format. Therefore, the first slot contains the upper 32-bit of the GM address offset while the second slot embeds the lower 32-bit part.

The "BMOffset" field from Fig. 9 contains, as its name suggests, the address offset for BM. The number of transaction bursts to be transferred is indicated by the "Burst Size" field and allows up to 256 transaction beats to be issued, which is the maximum allowed by the AXI4 protocol. In the current implementation we use 1024-bit wide (128 Bytes) AXI4 master controllers to interface with the

HBM memories, therefore, we cannot exceed 32 beats (Given that an AXI burst cannot exceed a 4 KB boundary [20]).

Finally, the (STOP) instruction flags the end of the program and notifies the host through a handshaking mechanism, to allow it to start reading the computed results.

Note that, whenever a C-Type instruction is encountered, the instruction streams towards the accelerator contain just (NOP) opcodes. Moreover, since no branch prediction mechanism is being implemented at the current time, an extra (NOP) have to be inserted after each call to (BNZ), in order to stall the pipeline for one cycle, until the branch is solved. We plan to improve these aspects in future iterations.

F. BINARY GENERATION

Since a compiler is still under development, we currently program DRAGON in a challenging yet interesting way. We developed a set of C-based function prototypes that describe all the DRAGON instructions. These function prototypes take as parameters the fields of each instruction, concatenate them and then dump the equivalent hexadecimal value to the generated program file. We use these function prototypes into a C-based program and make use of the powerful constructs of the C language to generate our final target executable. The contents of this file can then be loaded into the FPGA through OpenCL buffers. This code is independent from the number of BCs in the architecture and is the same for configurations implementing any number of BC.

VI. EXPERIMENTAL EVALUATION

A. EXPERIMENTAL ENVIRONMENT

The current implementation of DRAGON targets a Xilinx Alveo U280 acceleration card, featuring a 16 nm Ultrascale+ XCVU37P FPGA with 8 GB of on-chip HBM2 memories split into 2 stacks of 4GB each [21].

We created multiple DRAGON assembly programs that compute the two stencil computation benchmarks shown in Table. 6 and map their execution among all PEs in parallel, for different problem sizes and different number of iterations. Then, for comparison purpose, we created an optimized parallel version (using C++ with OpenMP pragmas and AVX2 compiler directives) of the equivalent computations on both Intel Core i9 and Core i5 CPUs. At first, we will compare the obtained sustained performance of DRAGON as well as its power efficiency to those of the Intel Core i5 and Core i9 CPUs. Later, we will expand our evaluation to compare the computational efficiency of DRAGON to other architectures such as GPUs or even FPGAs from other vendors.

Details about the experimental setup, are given through Table. 4. The current FPGA implementation clocks at 130 MHz.

TABLE 4. Environment setup used in the experiments.

CPU (FPGA host)	Intel Core i9-9900K CPU 3.60GHz (8 cores, TDP=95W)
Operating system (FPGA host)	Ubuntu 18.04.1 LTS
Compiler	g++(7.5.0) (with -fopenmp -O3 -mavx2)
CPU	Intel Core i5-6360U CPU 2.00GHz (2 cores, TDP=15W)
Operating system	MacOS Catalina 10.15.6
Compiler	clang++(10.0.1) (with -fopenmp -O3 -mavx2)
Accelerator	Alveo U280 Data Center Accelerator Card [21]
FPGA Compiler	Xilinx Vitis 2020.2 (64-bit)

B. STENCIL-BASED BENCHMARKS

Molecular dynamics, electromagnetism and particle interactions are examples among many other scientific computing applications that require solving complex partial differential equations. Stencil computing [2] is a powerful tool that is widely used to solve such kind of equations. Stencils operate on a regular N-dimensional grid and iteratively update their cells (grid points) over a certain count of iterations. This process is repeated over a number of successive time iterations where the current cells use exclusively their updated neighborhood points from the previous time-step. This yields a time-step-dependency-free relation between cells which means that the grid cells can be easily partitioned in space and mapped to different computing units provided an efficient communication mechanism to allow them to exchange the partitioned cells boundaries between each other, at every time iteration. Here, the experimental evaluation of DRAGON is based on the sustained performance and power efficiency results of the stencil-based benchmarks whose computational models are given by Table. 6.

VII. RESULTS AND DISCUSSION

A. RESOURCE UTILIZATION

Table. 5 shows the resource utilization of the FPGA, for both the static region (percentages are given relatively to all available resources) and the dynamic region (percentages are given relatively to remaining resources, after excluding those allocated for the static region) in the FPGA.

TABLE 5. FPGA resource utilization summary.

DYNAMIC region					
LUT	**LUTRAM**	**REG**	**BRAM**	**URAM**	**DSP**
DRAGON (9 BCs)					
505202 43.92%	58835 10.27%	281765 11.96%	720 40.09%	304 31.67%	1872 20.75%
Broadcast Cluster (16 x PEs + 1 x BMC + 16 x BM Banks)					
54134	4736	29574	80	32	208
Processing Element					
3297	296	1815	5	1	13

STATIC region (FPGA SHELL)					
LUT	**LUTRAM**	**REG**	**BRAM**	**URAM**	**DSP**
152420 11.7%	27370 4.56%	249387 9.57%	220 10.91%	0 0%	4 0.04%

TABLE 6. Benchmarks used for the experimental evaluation.

Benchmark	Equation
2D Laplace	$U_{i,j}^{t+1} = 0.25 \times (U_{i-1,j}^{t} + U_{i+1,j}^{t} + U_{i,j-1}^{t} + U_{i,j+1}^{t})$
2D 5-point Jacobi	$U_{i,j}^{t+1} = c_0.U_{i-1,j}^{t} + c_1.U_{i+1,j}^{t} + c_2.U_{i,j}^{t} + c_3.U_{i,j-1}^{t} + c_4.U_{i,j+1}^{t}$

The current implementation of DRAGON consists of a grid of three by three BCs and consumes less than half the LUT and BRAM resources available on the dynamic region of the FPGA. BRAMs are used for the Register File and FIFO buffers, whereas URAMs are used for LMs and BMs.

To provide an OpenCL control interface, the design is packed as an RTL-based kernel. This flow instructs the Vivado tool to place a static region inside the FPGA called shell. This shell comprises multiple additional necessary logic for managing HBM memory interfaces and PCIe DMA engine to communicate data with the host. Moreover, the wide AXI buses (1024 bits) used to communicate data between HBM banks and its corresponding BCs limit the number of BCs that can be deployed, due to the limitation of inter-die wires. In fact, the current FPGA uses SSI (Stacked Silicon Interconnect) to connect 3 different chip dies named SLR [18] (Super Logic Region) through the usage of special wires known as SLLs (Super Long Lines). FPGA designs with high connectivity between PEs such as DRAGON, will be mostly bottle-necked by the available amount of SLLs, instead of the available logic resources. Consequently, this complicates placement and routing steps which degrades the performance by negatively impacting the clock speed (currently 130 MHz), in particular when multiple BCs are deployed. In contrast, when a single BC was implemented, the operating frequency could reach 180 MHz. While outside the scope of this paper, it is also interesting to note that our implementation is flexible enough to support multiple types of global memories such as HBM2, DDR4 or even on-chip URAM and BRAM. It is also possible to combine usage of these memories. The only requirement here is to use AXI4 as the external memory interface protocol. Following our OpenCL-based approach, Vitis will automatically implement the necessary memory subsystem to connect the different BCs to the chosen type of global memory.

B. COMPUTATIONAL PERFORMANCE AND POWER EFFICIENCY

Reference [22] shows that Intel Core i9 9900K has a peak performance of 460.8 GFLOP/s and that Intel Core i5 6360U has a peak performance of 64 GFLOP/s. Nonetheless, there is no mention for which clock speed (both processors have dynamic turbo boost feature) or whether this is a single-precision or double-precision performance.

Therefore, we computed the TPP (Theoretical Peak Performance) of the CPU through "(1)".

$$TPP_{\mathrm{CPU}} = FREQ_{\mathrm{CPU}} \times CORES_{\mathrm{CPU}} \times \#OP_{\mathrm{CPU}} \qquad (1)$$

where $\mathrm{FREQ_{CPU}}$ is the clock frequency of the processor, $\mathrm{CORES_{CPU}}$ is the number of processor cores and $\mathrm{\#OP_{CPU}}$ is the number of double-precision operations allowed per clock cycle in every core and which is equal to 8. In fact, both CPUs support AVX2 which allows operations on 256-bit vectors. Therefore, both CPUs allow four 64-bit or eight 32-bit data to be packed; thus, four 64-bit or eight 32-bit parallel operations. They also support fused multiply-add operations which doubles the number of operations to a total of eight for double-precision and sixteen for single-precision. Given the base clock frequency of each CPU, we can verify that the TPP in [22] is given for single-precision. This confirms our approach to calculate it for double-precision and we can deduce that $\mathrm{TPP_{Corei9}} = 230.4$ GFLOPs and $\mathrm{TPP_{Corei5}} = 32$ GFLOP/s. Similarly, each PE of DRAGON has the ability to produce a result for either one 64-bit integer operation or up to 2 double-precision floating-point operations (floating-point multiply-accumulate instruction). Hence, the TPP of DRAGON for FP (Floating-Point) operations $\mathrm{TPP_{DRAGON}}$ is given by "(2)".

$$TPP_{\mathrm{DRAGON}} = 2 \times FREQ \times N_{\mathrm{PE}} \times N_{\mathrm{BC}} \qquad (2)$$

where FREQ is the system clock frequency in GHz and is equal to 0.130 GHz (130 MHz), $\mathrm{N_{PE}}$ is the number of PEs per BC and is equal to 16 and $\mathrm{N_{BC}}$ is the number of BCs in the FPGA and is equal to 9. In current implementation, $\mathrm{TPP_{DRAGON}} = 37.44$ GFLOP/s (double-precision) and 18.72 GOPs for 64-bit integer operations.

In order to update every cell (stencil point), Laplace benchmark requires in average 1 multiplication and 3 multiply-accumulate operations (in total 4 multiplications and 3 additions). On the other hand, the 2D 5-point Jacobi benchmark requires 1 extra multiply-accumulate operation for the central cell point. Consequently, the 2D Laplace benchmark requires 7 operations per cell update, whereas the 2D 5-point Jacobi requires 9 operations. The EP (Effective Performance) is obtained by dividing the number of required operations by the execution time. The calculated EP is depicted by Fig. 10.

Besides, Dragon consumes 35.44W whereas the corresponding TDP (Thermal Design Power) values of the Core i9 and the Core i5 are shown in Table. 4. Consequently, the power efficiency is obtained by dividing the EP by the corresponding power drop. Fig. 11 illustrates the computed power efficiencies based on the EP depicted by Fig. 10.

DRAGON has the ability to accumulate intermediate results in its FPU and can execute computation while scattering and gathering data continuously through reading

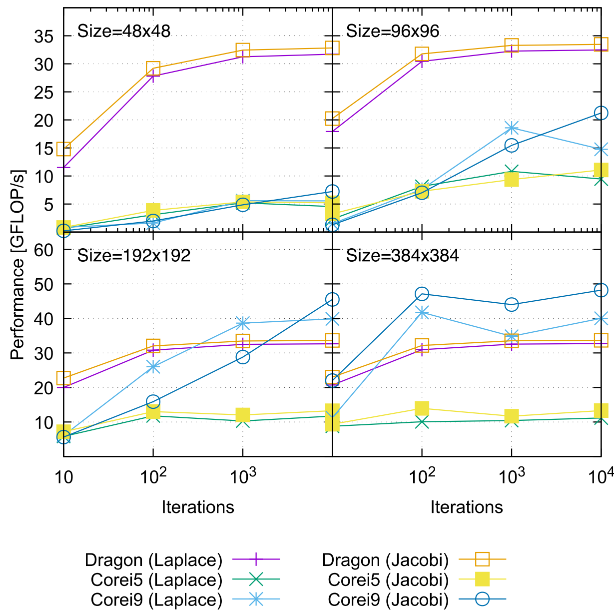


FIGURE 10. Effective Performance of Dragon, Intel Core i9 and Core i5 CPUs with different problem sizes and different number of iterations for 2D Laplace and 2D 5-point Jacobi computations.

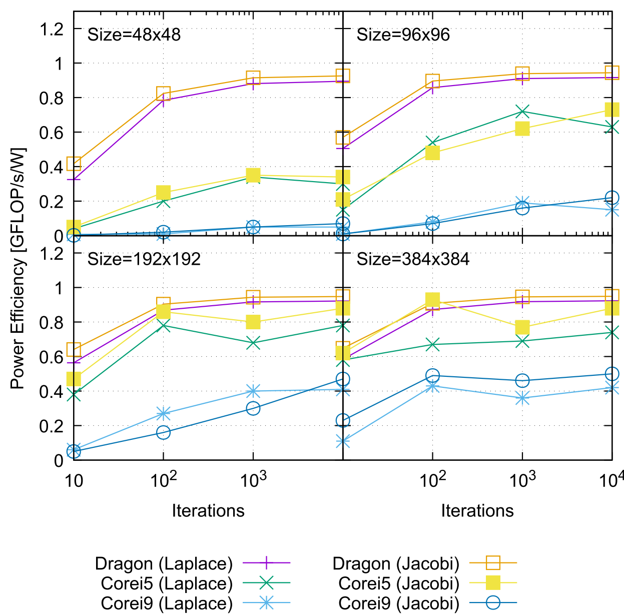


FIGURE 11. Power efficiency of Dragon, Intel Core i9 and Core i5 CPUs with different problem sizes and different number of iterations for 2D Laplace and 2D 5-point Jacobi computations.

and writing into its FIFO buffers; thus, effectively overlapping computation with memory transfers. This explains the fact that DRAGON achieves higher computational efficiency when compared to the Core i9 and the Core i5 processors as shown through EPR values of Table. 7. Overall, DRAGON achieves a higher power efficiency when compared to the Core i9 and the Core i5 CPUs as depicted by Fig. 11. On the other hand, the Core i5 CPU has a low TDP of just 15W and thus manages to have higher power efficiency than that of the Core i9.

The work in [2] reports the single-precision peak performance of a GTX960 GPU which is 2,308.1 GFLOPS. Based on the reported EPRs for single-precision 5-point Jacobi and Laplace benchmarks (7.1% and 3.2%, respectively), the sustained performance is 163.87 GFLOPS and 73.85 GFLOPS, respectively. Given that the TDP is equal to 120W, the corresponding power-efficiencies are 1.36 GFLOPS/W and 0.61 GFLOPS/W, respectively. The number of double-precision cores on this GPU has a rate of 1/32 as compared to single-precision cores which lead to a peak performance of just 72.12 GFLOPS. Assuming the same EPRs are maintained in double-precision computation, the corresponding sustained performances for the 5-point Jacobi and Laplace benchmarks become 5.12 GFLOPS and 2.3 GFLOPS, respectively. Furthermore, the power efficiencies become 0.042 GFLOPS/W and 0.019 GFLOPS/W, respectively. In contrast, DRAGON can achieve 33.66 GFLOPS and 32.74 GFLOPS in double-precision performance for these respective benchmarks. DRAGON consumes 35.44 W at 130 MHz which yields the power efficiency of 0.94 GFLOPS/W and 0.92 GFLOPS/W for these respective benchmarks. This clearly shows the merit of DRAGON in double-precision as compared to GTX960 GPU sustained performance and power efficiency for these benchmarks.

C. COMPARISON WITH RELATED WORKS

The work in [9] and [10] implements a resource-efficient RISC-V overlay, supports only integer computations and abstains from providing any application performance analysis. While RISC-V is an attractive choice for instruction-set-based overlays, we believe it is not adequate for compute-intensive or memory-bound applications found in scientific calculations. While possible ISA extensions might address this kind of computations, they would still require specialized compilers and remain bound by the space of usable instructions, which is intrinsically limited, both in number and width. Clearly, the Load-Store, register-register based approach is not efficient enough to handle data exchange between PEs while consuming a minimal amount of clock cycles.

In contrast, our proposed ISA extends the register-register approach to allow PEs to operate directly on incoming data from external inputs, such as adjacent PEs (through FIFO buffers), or even from an external larger memory (such as BM), through a broadcasting approach. Moreover, our VLIW-based micro-architecture implementation allows PEs to execute more operations with less instructions as depicted by Fig. 12; thus, efficiently overlapping data movements with computations and consequently boosting the computational efficiency. For example, Fig. 12 illustrates how a single VLIW instruction with two slots, can hide data transfer cost between adjacent PEs as well as between the local memory and the Register File (in both directions), by overlapping these memory operations with effective computations; thus,

TABLE 7. Comparison of the EPR(%) with CPU, GPU and FPGA using 2D stencil computation benchmarks.

Benchmark	FPGA				CPU			GPU
	DRAGON	Nallatech385 [13]	DE5[b] [2]	395-D8[b] [2]	i7-4960X[b] [2][a]	i5-6360U	i9-9900K	GTX960[b] [2]
5-point Jacobi	**89.9**	30.4 (55.8[b])	68	15.8	38.1	43.6	20.9	7.1
Laplace	**87.4**	33.7 (48.2[b])	92.8	11.7	18.3	36.8	17.4	3.2

[a]We recalculated and updated the original EPR value given by the work in [2] based on the official Intel data [22] that provides the peak performance of the Core i7-4960X.
[b]EPR is relative to the performance in Single-Precision.

reducing overall requirement for five different operations to a single clock cycle.

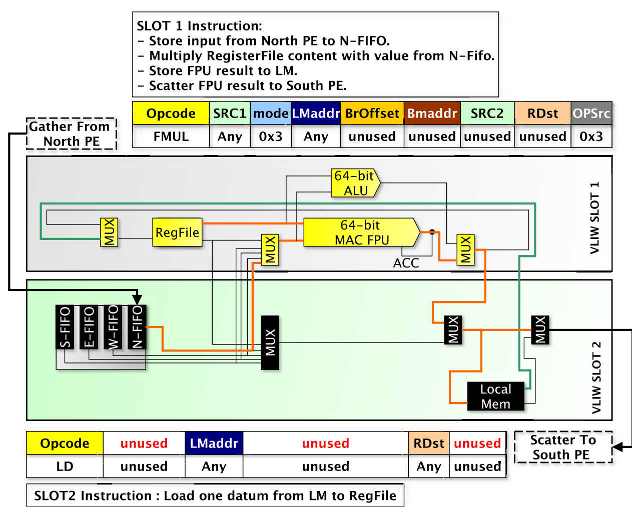


FIGURE 12. VLIW in action: more operations with less instructions.

The work in [12] claims to introduce the concept of domain-specific programmable design for stencil computation. This work aims to create a specialized multi-FPGAs overlay for pipelined, iteration-parallel 2D and 3D stencil calculations. This overlay implements a master and multiple slave FPGAs and uses multiple sequencers to interpret custom instruction-set micro-instructions to control PE operation. While this enhances problem-size flexibility, it introduces an extra burden on programmers by forcing them to configure corner-case sequencers using different micro-instruction programs. In contrast, our architecture is able to implement the same 2D benchmarks using a single program and a single sequencer. Furthermore, while the authors in [12] claim that their approach applies to double-precision floating-point computations, their implementation and evaluation focus only on single-precision floating-point. Their implementation uses Terasic DE3 boards (ALTERA Stratix III EP3SL150) and achieves a TPP of 25.5 GFLOP/s for master FPGA (with 96 PEs) and 34 GFLOP/s for slaves (with 128 PEs) at 133 MHz for both. Our work, not only adopts a double-precision implementation but extends that to support 64-bit integer operations as well. While the current implementation of DRAGON uses a higher precision and reaches 37.44 GFLOP/s (with 144 PEs) at just 130 MHz, a direct performance comparison may be unfair because of the gap in FPGA technology and size. Besides, despite not being tailored to a specific application, DRAGON managed to achieve the same EPR for a 4-point Jacobi benchmark (87.4% for both DRAGON and the work in [12]). In addition, the work in [12] adopts a USB interface to transfer data from the host to the FPGA, which may limit the sustained performance for a streaming architecture. In contrast, our work not only uses a faster PCIe interface for host-FPGA transfers, but also allows controlling the FPGA overlay through an OpenCL host program; thus, ensuring easy integration within computing platforms.

The work in [2] proposes a customized OpenCL-based design for single-precision FPGA implementation of the benchmarks in Table. 6. Generally, a target-specific implementation achieves higher performance than general purpose overlays. This gap in performance comes at the cost of a fixed problem-size as well as a longer design and performance tuning time. Compared to our benchmark implementation, the work in [2] achieves a higher clock frequency (296 MHz for 2D 5-point Jacobi benchmark on DE5), implements larger problem sizes and reports better performance when targeting single-precision (181.9 GFLOP/s, at best, for a 2D laplace benchmark). Nonetheless, a direct performance comparison may be unfair because of the impact of doubling the floating-point precision on the overall area and operating frequency. The authors in [2] reported the double-precision floating-point performance for a single benchmark (2D 5-point Jacobi) while noticing that, in general, the double-precision performance do not exceed 25% of what single-precision could achieve. For example, the implementation of the 2D 5-point Jacobi benchmark on a DE5 board [2] reached 27.2 GFLOP/s in double-precision, while on a 395-D8 board [2] it reached 40.7 GFLOP/s.

Equation "(12)" in [2] defines the Effective to Peak Performance Ratio (EPR) as the ratio of EP to TPP. This performance metric directly points to the computational efficiency of a given architecture, regardless of the underlying implementation details. The single-precision TPP of the Nallatech 385 board used in [13] is given by [23] and is equal to 1366 GFLOP/s; hence, we estimated the double-precision performance to nearly one fourth of that

and thus to 341.5 GFLOPs. Using these two values, we computed the EPR for the floating-point performance of the work in [13] (using skewed grid) and reported our findings in Table. 7. Note that a higher estimation on this value would reduce further the calculated EPR of the FPGA. By default, Table. 7 shows the EPR percentage based on the reported double-precision performance. A note was added to indicate where the EPR is based instead on the single-precision performance.

Interestingly, despite not being tailored to a specific application, Table. 7 shows that our best measured EPR achieves comparable or even higher results than those reported in [2] and [13], regardless of the implemented floating-point precision. This highlights the merit of our overlay architecture and in particular the efficiency of its underlying custom-designed instruction-set.

VIII. CONCLUSION

This work presented a high-performance computing architecture codenamed DRAGON (Dynamically Re-programmable Architecture for Gather/scatter Overlay Nodes). This architecture implements a custom instruction set specifically designed for extracting the highest levels of parallel performance while preserving domain flexibility. DRAGON offers both 64-bit integer and double-precision floating-point computing capability in the same PE. DRAGON is a promising architecture that leverages benefits from CPU and GPU worlds, condenses an overload of parallel execution paradigms and is in a continuous evolution process. The merit of DRAGON was demonstrated through two memory-bound stencil-based benchmark which show that the implemented version with only 144 PEs can outperform an Intel Core i9 under certain conditions, and achieves better power efficiency as well as higher EPR while operating at a clock speed that is nearly 28 times lower. This offers promises that an ASIC (Application-Specific Integrated circuit) implementation would exceed the sustained performance of even the most performing CPUs by at least an order of magnitude. Ultimately, DRAGON is implemented on an HBM2-enabled FPGA and exploits Vitis RTL kernel flow to seamlessly integrate any heterogeneous platform as an acceleration kernel that can be controlled with an OpenCL-based host. Arguably, this is an extremely significant milestone in the great challenge towards bringing FPGAs closer to mainstream adoption. Our future goals will include developing a compiler to ease the programming task, scaling the architecture into a multi-FPGA implementation and finally exploring the possibility of an ASIC tapeout.

REFERENCES

[1] X. Li and D. L. Maskell, ''Time-multiplexed FPGA overlay architectures: A survey,'' *ACM TODAES*, vol. 24, no. 5, pp. 1–19, Jul. 2019, Accessed: Feb. 26, 2021, doi: 10.1145/3339861.

[2] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, ''OpenCL-based FPGA-platform for stencil computation and its optimization methodology,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1390–1402, May 2017.

[3] R. B. Abdelhamid, Y. Yamguchi, and T. Boku, ''Condensing an overload of parallel computing ingredients into a single architecture recipe,'' in *Proc. IEEE 31st Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2020, pp. 25–28.

[4] R. Ben Abdelhamid, Y. Yamaguchi, and T. Boku, ''MITRACA: Manycore interlinked torus reconfigurable accelerator architecture,'' in *Proc. IEEE 30th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2019, p. 38.

[5] R. Ben Abdelhamid, Y. Yamaguchi, and T. Boku, ''MITRACA: A next-gen heterogeneous architecture,'' in *Proc. IEEE 13th Int. Symp. Embedded Multicore/Many-Core Syst. Chip (MCSoC)*, Oct. 2019, pp. 304–311.

[6] K. Paul, C. Dash, and M. S. Moghaddam, ''ReMORPH: A runtime reconfigurable architecture,'' in *Proc. 15th Euromicro Conf. Digit. Syst. Design*, Sep. 2012, pp. 26–33.

[7] K. Ovtcharov, I. Tili, and J. G. Steffan, ''A multithreaded VLIW soft processor family,'' in *Proc. IEEE 21st Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2013, p. 232.

[8] C. E. Laforest and J. H. Anderson, ''Microarchitectural comparison of the MXP and octavo soft-processor FPGA overlays,'' *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, pp. 1–25, Jul. 2017.

[9] J. Gray, ''GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator,'' in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 17–20.

[10] J. Gray, ''2GRVI Phalanx: A 1332-core RISC-V RV64I processor cluster array with an HBM2 high bandwidth memory system, and an OpenCL-like programming model, in a Xilinx VU37P FPGA [WIP report],'' in *Proc. Int. Workshop (H2RC)*, Denver, CO, USA, Nov. 2019. Accessed: Feb. 26, 2021. [Online]. Available: https://h2rc.cse.sc.edu/2019/papers/lightning_2_1_Gray.pdf

[11] K. Sano, W. Luzhou, Y. Hatsuda, T. Iizuka, and S. Yamamoto, ''FPGA-array with bandwidth-reduction mechanism for scalable and power-efficient numerical simulations based on finite difference methods,'' *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 1–35, Nov. 2010, doi: 10.1145/1862648.1862651.

[12] K. Sano, Y. Hatsuda, and S. Yamamoto, ''Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 695–705, Mar. 2014.

[13] H. M. Waidyasooriya and M. Hariyama, ''Multi-FPGA accelerator architecture for stencil computation exploiting spacial and temporal scalability,'' *IEEE Access*, vol. 7, pp. 53188–53201, Feb. 2019, doi: 10.1109/ACCESS.2019.2910824.

[14] C. E. LaForest and J. G. Steffan, ''OCTAVO: An FPGA-centric processor family,'' in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2012, pp. 219–228.

[15] J. E. Smith, ''Decoupled access/execute computer architectures,'' *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, Nov. 1984, doi: 10.1145/357401.357403.

[16] T. Sterling, M. Anderson, and M. Brodowicz, *High Performance Computing: Modern Systems and Practices*. Cambridge, MA, USA: Morgan Kaufmann, 2018, pp. 69–73.

[17] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. San Francisco, CA, USA: Morgan Kaufmann, Dec. 2004.

[18] Xilinx. *UltraFast Design Methodology Guide for Vivado Design Suite*. Accessed: Feb. 26, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_%3/ug949-vivado-design-methodology.pdf

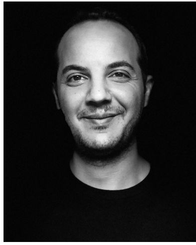
[19] A. Waterman, ''Design of the RISC-V instruction set architecture,'' Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkley, CA, USA, Jan. 2016. Accessed: Dec. 16, 2019. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[20] *AMBA AXI ACE Protocol Specification, ARM*. Accessed: Feb. 26, 2021. [Online]. Available: https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_%protocol_spec.pdf

[21] *Alveo U280 Data Center Accelerator Card, Xilinx*. Accessed: Feb. 26, 2021. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#specif%ications

[22] *APP Metrics for Intel Microprocessors, Intel*. Accessed: Feb. 26, 2021. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/%APPfor-Intel-Core-Processors.pdf

[23] *UltraFast Design Methodology Guide for the Vivado Design Suite, Intel*. Accessed: Feb. 26, 2021. [Online]. Available: https://www.intel.fr/content/dam/www/programmable/us/en/pdfs/literatu%re/pt/arria-10-product-table.pdf



RIADH BEN ABDELHAMID received the bachelor's degree in electrical engineering (electronics and micro-electronics major) from the National Engineering School of Tunis, in 2010, and the M.Eng. degree in computer science from the University of Tsukuba, in March 2020, where he is currently pursuing the Ph.D. degree. From 2010 to 2015, he has worked as an FPGA Design and Verification Engineer in a large Europe-based avionics company, where he took a leading role in designing and verifying safety critical systems, including a flight control system. Since 2015, he has been working with Synopsys as a Subcontractor on their flagship FPGA emulation system ZEBU. His research interests include many-core processor architectures and overlays, high-performance computing, and reconfigurable accelerators. He is also an Enthusiast about making his own many-core processor chip start-up. In 2017, he was selected as the Japanese Government Scholarship (MEXT) recipient to study in Japan.

YOSHIKI YAMAGUCHI (Member, IEEE) received the M.S. degree in science and engineering and the Ph.D. degree in engineering from the University of Tsukuba, in 2000. From April 2000 to March 2003, he was a JSPS Research Fellow with the University of Tsukuba. From April 2003 to March 2005, he was the Research Scientist of the Yokohama Institute, RIKEN (as the Visiting Scientist from April 2005 to March 2008). Since April 2005, he has been an Assistant Professor with the Graduate School of System and Information Engineering, University of Tsukuba. He also held a position as a Visiting Academic with the Department of Computing, Imperial College London, from June 2010 to March 2011. Since July 2011, he has been a collaborative Fellow with the Center for Computational Science, University of Tsukuba. His research interests include reconfigurable architecture, real-time applications, and power-efficiency computing for image, sound, and bioinformatics applications. He is also interested in heterogeneous computing, including FPGA, GPU, and CPUs.

TAISUKE BOKU (Member, IEEE) received the master's and Ph.D. degrees from the Department of Electrical Engineering, Keio University. After his career as an Assistant Professor with the Department of Physics, Keio University, he joined the Center for Computational Sciences (formerly the Center for Computational Physics), University of Tsukuba, where he is currently the Director and the HPC Division Leader. He has been working there more than 25 years for HPC system architecture, system software, and performance evaluation on various scientific applications. In these years, he has been playing the central role of system development on CP-PACS (ranked as number one in TOP500 in 1996), FIRST, PACS-CS, HA-PACS, and Cygnus as the representative supercomputers in Japan. He is also the Director of the Center for Computational Sciences, University of Tsukuba, and the Vice Director of the Joint Center for Advanced HPC (JCAHPC). He is also the President of HPCI Consortium, which is the global community in Japan covering all the supercomputer resource operation centers and major computational science research communities. He is a member of the System Architecture Working Group of Post-K Computer development. He received the ACM Gordon Bell Prize in 2011.