IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 27, 2020, accepted November 15, 2020, date of publication November 24, 2020,
date of current version December 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3040081

In-FPGA Instrumentation Framework for OpenCL-Based Designs

HACHEM BENSALEM[1], (Graduate Student Member, IEEE), YVES BLAQUIÈRE[1], (Member, IEEE), AND YVON SAVARIA[2], (Fellow, IEEE)
[1]Department of Electrical Engineering, École de Technologie Supérieure, Montreal, QC H3C 1K3, Canada
[2]Department of Electrical Engineering, Polytechnique Montréal, Montréal, QC H3T 1J4, Canada

Corresponding author: Hachem Bensalem (hachem.bensalem.1@ens.etsmtl.ca)

This work was supported in part by the CMC Microsystems, the Natural Sciences and Engineering Research Council (NSERC) of Canada, and in part by the Canadian Fund for Innovation (the CFI-EmSysCan Project).

ABSTRACT The productivity achieved when developing applications on high-performance reconfigurable heterogeneous computing (HPRHC) systems is increased by using the Open Computing Language (OpenCL). However, the hardware produced by OpenCL compilers in field-programmable gate arrays (FPGAs) can result in severe performance bottlenecks that are challenging to solve. The problem is compounded by the fact that the generated netlist details are disorganized, making them mostly unreadable and only partially visible to designers. This paper proposes an in-FPGA instrumentation method and a new framework for extracting the FPGA-cycle-accurate timing performances of OpenCL-based designs. The results clearly show that the chosen execution model for OpenCL-based designs strongly affects the timing performance when it is not properly implemented. Our framework is implemented on an HPRHC platform that contains a CPU and two Arria10 FPGAs, and it is evaluated with a wide variety of benchmarks with different complexities. After testing on the reported benchmarks, the average logic overhead for one inserted instrument is 0.2 % of the total amount of adaptive look-up tables (ALUTs) and 0.1 % of the total registers in an FPGA. This resource utilization is between 1.5 and six times lower than those reported in the best previously published works. The scalability of the framework is also evaluated by inserting up to 50 instruments. The experimental results show that the average logic utilization per instrument is 0.19 % of the ALUTs and 0.17 % of the registers in the FPGA when 50 instruments are inserted.

INDEX TERMS OpenCL, FPGA, instrumentation, high-performance reconfigurable computing, HLS, timing performance.

I. INTRODUCTION

Field-programmable gate arrays (FPGAs) have progressively evolved as powerful accelerators for high-performance reconfigurable heterogeneous computing (HPRHC) systems. The success of FPGAs is largely due to their inherent flexibility, reconfigurable architecture, and high computing capabilities [1]. Application development using hardware description languages (HDLs) is time-consuming, and as a result, it sometimes cannot fulfill time-to-market constraints. This problem is made worse with the increase in application complexity. This is why FPGA vendors have heavily invested in high-level synthesis (HLS) and associated tools to make FPGAs more easily programmable using high-level languages (HLLs) for general-purpose computing [2]. HLS tools offer interesting options for overcoming this design complexity, and they are mainly used to generate register transfer level (RTL) specifications from some HLLs [3]. An important challenge of HLS tools is to generate high-quality HDL specifications, which depend not only on the objectives and the specificities of the implemented optimization methods [4] but also largely on the adoption of good coding styles and practices. OpenCL is a popular, open-standard high-level language that allows software designers to develop applications executing in parallel on HPRHC platforms, which may be graphic processing units (GPUs) or FPGAs. An OpenCL-to-FPGA compiler [5], a type of HLS compiler, promises better development productivity than traditional FPGA development [6]. However, writing an optimized OpenCL design is still challenging because

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

the resulting netlists may have very significant performance bottlenecks when running on FPGAs [7]. To resolve this problem and to predict bottlenecks, researchers have been working on optimization directives based on predictive modeling for HPRHC platforms [8]–[13] .

These predictive modeling techniques can estimate some performance metrics, such as power consumption and logic resource utilization [11]. Accurate estimates allow developers to optimize and rewrite their high-level code before compilation. However, the quality of the RTL specifications generated by the OpenCL compilers highly depends on the target platform and the quality of the high-level descriptions [14]. Indeed, the back-ends of the OpenCL compilers use metaheuristic algorithms, which tend to make them sensitive to the OpenCL input code. Thus, FPGA vendors provide and recommend using best practice guides while writing OpenCL code [15]. Even when developers follow the recommended best practices, performance bottlenecks may still occur due to the complexity of OpenCL designs targeting HPRHC platforms. For example, an OpenCL design may contain multiple complex functions and computations, including loading and storing from either on-chip or off-chip memories. Performing these functions can lead to a degradation in performance if they are not implemented judiciously [16]. Another drawback of OpenCL-based FPGA accelerators is that designers using them lose the low-level observability of their designs after compiling to low-level HDL descriptions. Moreover, the size of the design space of an application targeting HPRHC platforms increases with the application complexity, and this increases the probability that the compilers generate nonoptimal designs [17]. As a result, design optimization is challenging mainly because the HDL code produced by OpenCL compilers is mostly unreadable, and this code may contain black-box modules instantiating intellectual property modules offered by FPGAs or third party vendors.

A solution that offers great potential for identifying performance bottlenecks is to include embedded instruments, which can reveal the accurate performance and the internal behavior of the circuits. As explored in this paper, these instruments are specified from OpenCL source code to identify which sections of code or computations cause performance degradation. To improve debugging and performance analysis capabilities, these instruments allow the observation of relevant values and accurate run-time analyses of any data in any section of the generated FPGA circuit. This instrumentation challenge is a known concern with hardware produced by HLS tools [14]. Indeed, timing performance analysis, hardware verification, and debugging capabilities can be performed with an assertion-based approach [18], such as the use ANSI-C assertions, enabling the instrumentation of the HLS-produced circuit [19]. Unfortunately, ANSI-C-like assertions are not supported by OpenCL for FPGA compilers. Moreover, most commercial tools for OpenCL (e.g., The OpenCL Software Development Kit (SDK) from Intel [20]) do not have in-circuit verification and debugging capabilities. There is a need for a computer-aided design approach for OpenCL-based FPGA design flows to enhance the observability inside FPGAs and to diagnose where some undesired behaviors may occur when derived hardware targets run on FPGAs.

This paper addresses the problem of instrumenting OpenCL designs targeting FPGA-based computing systems and the challenge of portability and scalability with regard to such an instrumentation method. It notably presents an instrumentation method and framework that can be used by a software developer and applied to any HPRHC platform using OpenCL. The proposed instrumentation method uses in-FPGA instruments instead of using assertion functions such as ''assert ( )'' from the ''assert.h'' library [21], which is not supported by the OpenCL standard. This instrumentation framework is based on an approach that specifies in-FPGA instruments directly as part of the OpenCL description of the application. These instruments extract cycle-accurate run-time information and signal values from the FPGA. These values may then be used for debugging, and identification of eventual bottlenecks that may cause performance degradation. Thus, the instrumentation framework helps to analyze the behavior of OpenCL designs. This framework is proposed as a complement to the Intel SDK for the OpenCL, thereby demonstrating its low logic area overhead as well as the resulting enhanced observability and diagnosis capabilities. The main contributions of this work are summarized as follows:

1) An instrumentation and performance modeling method that uses inserted instruments to extract the actual performance of OpenCL kernels is proposed. With this method, instruments inserted directly in the OpenCL code are synthesized with the kernel and embedded with its functional modules;
2) A new framework that can be used by a developer, without hardware skills, to identify critical and vulnerable kernels or to record the value over time of any variable inside the OpenCL kernels is designed;
3) An evaluation and a validation of the proposed framework on a set of OpenCL benchmarks are performed. On the reported benchmarks, the average logic resource requirement is between 1.5 and 6 times lower than those reported in best previously published works [22]–[24].

This paper is an extension of the previous work presented in [25], where the focus was a comparative study of two monitor descriptions: one in OpenCL and a second developed in HDL. Compared with [25], this paper performs performance modeling with a framework that supports timing analysis and debugging. A data-tracing technique is added to collect data, to generate a dynamic execution trace and to send it back to a host. Additionally, the framework is applied to a wide variety of benchmarks to study its impact on FPGA resource utilization.

This paper is organized into six sections. The following section reviews the literature on debugging HLS-generated circuits and discusses the specific challenges related to

instrumenting OpenCL-based designs. Section III describes the proposed framework, explains its capabilities, and explains how it can be used for instrumentation purposes. Section IV presents the implementation of this framework with the Intel SDK for OpenCL toolchain, and analyzes the experimental results of the framework when applied to a wide variety of benchmarks. Section V evaluates the framework and compares it to previous works. Finally, Section VI summarizes the main contributions of this work.

II. BACKGROUND AND RELATED WORK

This section provides a brief description of the OpenCL programming model and establishes the importance of in-system instruments for debugging and characterizing HLS-produced circuits for FPGAs. Next, it highlights the instrumentation challenges encountered with OpenCL toolchains compared to those of other HLS tools. Previous works that approached the in-system instrumentation of HLS-based FPGA designs are then discussed.

A. OPENCL PLATFORM DESCRIPTION

OpenCL is a parallel programming language developed by the Khronos Group for heterogeneous programming platforms, which may contain central processing units (CPUs), GPUs, and FPGA devices. Based on [26], the terminology and concepts of OpenCL are summarized here. An OpenCL platform typically consists of one host device (CPU) and one or more accelerator devices. An accelerator device executes kernels written in C99, a C-based language. Each accelerator device (e.g., an FPGA or a GPU) can contain one or more compute units (CUs), and each CU is divided into one or more processing elements (PEs). For example, a CU can infer a vector addition dataflow structure in a PE, as shown in Fig. 1, with three operations: load, store and add. In OpenCL, a work-item corresponds to a single execution of a kernel that is instantiated in an accelerator as a single dataflow structure. Multiple work-items can be grouped into work-groups and executed concurrently on the same device in or out of order. A work-item can be executed by one or more PEs as part of the same work-group. In addition, an OpenCL platform provides an environment for host-accelerator interaction, as well as memory and accelerator management [16].

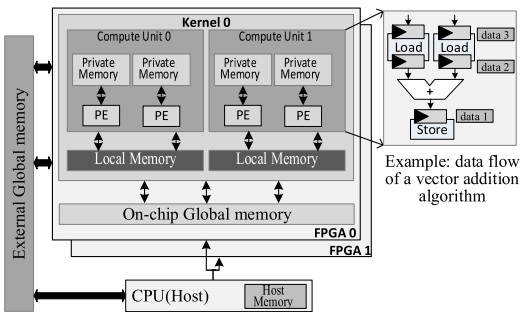


FIGURE 1. An OpenCL platform model with multiple FPGAs [27].

An OpenCL platform may support multiple accelerators executed on OpenCL-compatible devices, which may come from various vendors (e.g., Xilinx, Intel, or NVIDIA). The OpenCL platform model shown in Fig. 1 includes a host (CPU) and more than one FPGA used as accelerators. The rest of the paper assumes that the accelerator is composed of one or more FPGAs (no GPU is considered in this research). An OpenCL design comprises two distinct parts: the host code and the kernel code. The host code, written in C or C++, is responsible for the runtime management of OpenCL kernels. The host code is executed in the CPU and is used to identify the hardware platform, to set up computations, to transfer data for processing, and finally, to program the FPGAs. The compilation of kernel codes by offline compilers generates the FPGA bit-streams. The OpenCL standard [28] includes specifications that organize the kernel description, including the number of work-groups and compute units, barriers, and four memory models. A barrier is an OpenCL function used to ensure the correct execution order of work-items. The memory hierarchy in an OpenCL platform may comprise four different regions (Fig. 1). The first is the global memory that can be accessed from the host or from a device with write and/or read privileges. The second is the local memory, which is accessible by all work-items within a work-group. The third type is private memory, which is accessible only by a work-item; finally, constant memory is read only, and it is accessible from the work-items. The performance of an OpenCL design highly depends on the mapping of OpenCL data into these memory regions and their interactions.

B. LIMITATIONS OF EMBEDDED LOGIC ANALYZERS FOR FPGAS

Unlike the case of CPU-based applications, in-FPGA debugging of HLS-generated circuits is still challenging. In-FPGA visibility is offered by embedded logic analyzer (ELA) tools, such as SignalTap by Intel [29] and ChipScope by Xilinx [31] . These tools provide debugging support to RTL circuits, and they can be used to debug and unveil the internal behavior of circuits expressed at the RTL level while they are running in an FPGA. ELA tools capture and store the values of selected signals in trace buffers, which can be implemented using on-FPGA memories (e.g., BRAMs (Block RAMs)). Indeed, designers must identify the signals that they wish to observe from the RTL specification, which is usually human-readable, and they must link them to ELA tools.

ELA tools are supported in some HLS tools, such as Xilinx SDx and SDAccel [31], but they are almost impossible to use in HLS tools for two main reasons. First, the RTL code and the netlist generated by HLS tools are mostly unreadable due to the generic, structural and nonoptimized nature of the HLS-generated circuit. Second, RTL signal types and HLL data types are incompatible. For instance, a single C variable may correspond to several signals in an HLS-generated circuit. Moreover, different names are assigned by the HLS compilers depending not only on the metaheuristic algorithms

used by the HLS tool but also on the FPGA target. Linking HLL variables, RTL signals, and netlist signals becomes difficult, if not impossible, for the designer. These limitations increase the interest in being able to specify the insertion of instruments directly in the HLL code so that the HLS tools can embed them in the FPGAs without using ELA tools.

C. RELATED WORK ON IN-SYSTEM FPGA INSTRUMENTATION

Typically, embedded instruments record the internal behavior of circuits inside an FPGA [32], [33]. Some instruments can verify that behavior using hardware assertions [34]. This section reviews previous works that approached the instrumentation challenge of HLS-produced circuits. Bussa *et al.* [35] proposed an incremental compilation technique to reduce the time required for debugging and to accelerate the debugging of HLS circuits running on FPGAs. Inserting debugging and trace logic incrementally increases the compilation time and may reduce design productivity. This is significant when each iteration of the compilation process takes several hours. By contrast, our proposed method requires a single compilation of an OpenCL design that leads to a simplified instrumentation flow for developers with limited hardware design skills. In [36], [37], a source transformation approach was proposed to insert instrumentation circuits at the source level. This approach enables debugging by mapping the circuit states to the source code during HLS compilation, and it was applied to the LegUp HLS tool [38]. Unfortunately, this approach requires access to the source code of the HLS compilers, and this is not possible with commercial tools such as Intel or Xilinx compilers.

Monson and Hutchings [22], [39] used a source-to-source compilation approach to enhance observability, focusing on the C code rather than the RTL code generated by HLS tools. The source-to-source approach consists of adding debugging ports that are connected to C expressions in the source code, and these can provide access points in the design. However, these access points need to be connected by an embedded logic analyzer or by modifying the generated RTL code. Moreover, this approach is expensive in terms of logic resources, and it is closely related to HLS tools that have C code as input and is not compatible with existing OpenCL tool-chains.

Pinilla *et al.* [23], [40] described a methodology for debugging C-based HLS designs using a source-level instrumentation approach. First, the C code is structured into an abstract syntax tree (AST), and then the instrument is inserted into a C code by a source-to-source compiler. Finally, Vivado HLS [32] or LegUp generates the RTL description. This approach requires an instrumentation flow with several steps to debug a C-based design. It consumes a costly amount of memory; thus, Pinilla *et al.* applied an optimization technique called array duplicate minimization (ADM) to increase trace memory efficiency. Additionally, their approach requires an additional tool to generate the AST, and then an ADM is used for optimization to decrease the overhead. Finally, [40] and [23] extracted variable content without any timing and ordering information that is essential for debugging the timing issues in high-performance computing applications. Compared to those of previous papers, our proposed framework provides an FPGA-cycle-accurate performance analysis, and it also specifies debugging instruments at the OpenCL level without the need for any other custom tool, thereby reducing the developmental effort.

Eslami *et al.* [41] proposed an instrumentation approach through which the abstraction level to implement the code is elevated using virtual overlays. The overlay is a virtual, reconfigurable architecture that raises the abstraction level by overlaying it on top of the physical FPGA. The virtual overlay architecture for FPGAs is a new software-like design ecosystem that increases the software layer and cannot provide accurate performance characterizations of functions embedded in an FPGA. Additionally, this virtual architecture cannot be associated with or even applied to commercially available OpenCL-for-FPGA design tools. A framework for debugging and monitoring OpenCL-based FPGA designs was proposed in [42]. This framework is limited to capturing events and their sequences based on timestamps. Moreover, this work does not provide sufficient insight into performance to characterize features such as loop performances (initiation interval and latency) and multikernel OpenCL-based kernel performance and to analyze the root causes of their possible performance bottlenecks. Goeders *et al.* [24], [43] proposed an approach to improve trace buffer efficiency that allows for increasing the storage of useful debugging information such as captured signal values. The focus of the authors was to improve trace buffer capability rather than to implement a mechanism for performance monitoring and diagnosis.

As a partial debugging solution, Xilinx SDAccel [31] and Intel SDK for OpenCL tools support the printf function for software or hardware emulation purposes [20]. Software emulation allows for the functional verification of an application on a CPU, while hardware emulation is more accurate as it performs a functional verification using a model of the hardware. These emulations do not allow developers to detect unexpected behaviors exhibited by the circuits while running in FPGAs. Moreover, Intel SDK for OpenCL compiler provides a profiling report that contains general performance such as the logic resources usage and memory bandwidth. However, this performance report does not allow for the identification of a performance bottleneck because no integrated logic analyzers are supported by Intel SDK for the OpenCL toolchain. Thus, to overcome these problems, adding an instrument – including a debugger – could provide interesting information about features such as pipeline stalls. This would allow the acquisition of the precise processing time for each task or function inside OpenCL kernels. A task can be a set of operations, which are functions defined in a section of code inside an OpenCL kernel.

After a deep comprehensive review of the related literature, we found that the features and limitations of the in-FPGA instrumentation of HLS-produced circuits depend on two

main factors. The first factor is the level of abstraction of the circuit description to be instrumented, and the second is the back-end technology of the HLS tool and its supported libraries. Remarkably, some previous works did not take the back-end technology and the specificities of the OpenCL design flow for FPGAs into consideration. Therefore, we address the challenge of extracting the accurate performances of OpenCL-based FPGA designs using a source-level instrumentation method.

In this work, we first propose an FPGA instrumentation method that acts at the HLL level and that can be applied to both open-source and commercial HLS tools. Compared to those of previous works, this proposed method does not require access to the source code of the HLS tool. Second, we propose an RTL instrument that can be inserted in an OpenCL kernel. This instrument assists developers who need to identify possible timing performance bottlenecks for each target task or operation inside the kernels. Unlike in previous works, the proposed RTL instruments capture the variables at high levels associated with their runtime information expressed in terms of FPGA clock cycle accuracy. Additionally, we improve the trace data efficiency by selecting only useful data.

III. IN-SYSTEM INSTRUMENTATION FRAMEWORK OF OPENCL-BASED FPGA DESIGNS

This section describes the proposed instrumentation framework for OpenCL-based FPGA designs. First, the instrumentation method is presented. Then, the hardware and software architectures of the proposed framework are detailed, as well as the instrumentation flow, and the performance model for the OpenCL kernel is finally summarized.

A. PROPOSED INSTRUMENTATION METHOD

The proposed method is based on the OpenCL-Verilog codesign approach and enables instrumenting OpenCL-based FPGA designs using macro assertions inserted in OpenCL kernels. Instrumenting OpenCL kernels is performed in several steps (Fig. 2). In step S1, the instruments are inserted into the OpenCL kernel code (Fig. 2). The new kernel code is then compiled, and the FPGAs are configured using the bitstream. At that point, the resulting circuit can run on the FPGA, from which performances such as specific runtimes of any computations of interest are recorded (S2). That data can be offloaded to a CPU for analysis and to generate dynamic performance metrics (S3). Then, the developer analyzes the extracted performance, and he/she and may decide to optimize some sections of code and can add or remove some instruments (S4).

The instrumentation of kernels consists of instantiating instruments by calling the instrumentation function *instrument* ( ) inside their code, as shown in Fig. 3, in which a loop is monitored inside a vector addition kernel. An array, called TB in Fig. 3, is defined to store the data returned by *instrument* ( ) instances. This array is a trace buffer used to store the data extracted by the instrument. The function *instrument* ( ) is a

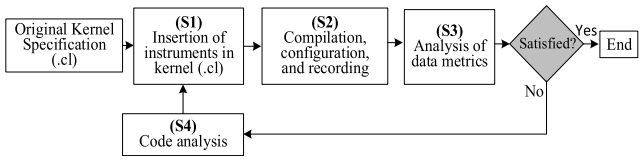


FIGURE 2. Instrumentation steps of OpenCL-based FPGA designs.

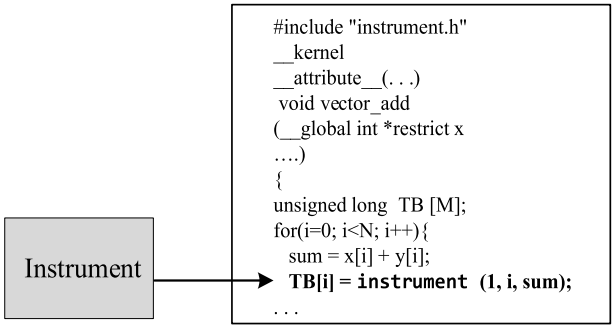


FIGURE 3. Instantiating an instrument at the OpenCL source level.

C-based function that is defined in an OpenCL instrumentation library called ''instrument.h''. The function *instrument* ( ) can be called from any OpenCL kernel and can have one or more inputs as defined in its related library. This function is paired with a precompiled generic instrument described by HDL, and it is called the instrument engine (IE) in the rest of this paper. The main advantage of describing the IE with HDL compared to a high-level description is that its internal architecture can be defined to minimize the logic area overhead in the FPGA. Furthermore, by using the HDL description of the IE, it is easy to abstract and manage not only the clock frequency and reset inside an FPGA but also the FPGA resources or interfaces such as PCI-Express or the cache coherent interconnect for accelerators [44]. This OpenCL source-level instrumentation method makes it easy for a software developer with no hardware skills to perform in-FPGA instrumentation and performance analyses.

The inserted *instrument* ( ) functions in OpenCL kernels memorize the value of a specified variable and its runtime information at FPGA clock cycle accuracy. As shown in Fig. 3, the function *instrument* (1, i, sum) samples the variable ''sum'' associated with its index ''i''. The value 1 is the value of the input ''selector'' of the instrument that enables the instrumentation mode. Fig. 4 shows the conceptual view of an OpenCL kernel with six inserted instruments I1, I2, I3, I4, I5, and I6 and a computation (C) that can be arithmetic, loading, or storage operations. The instrumentation of OpenCL kernels generates a dynamic execution trace that can be analyzed to detect bottlenecks, such as stalls, and to provide runtime information about computations (C1, C2, C3) inside the OpenCL kernel (Fig. 4). For example, C1 can be instrumented either using I1 or I1 and I2 together. Indeed, the difference between the timing values returned by I1 and I2 can determine the latency of C1 for a set of

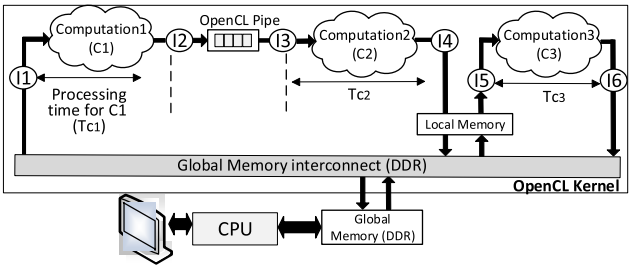


FIGURE 4. Conceptual view of the kernel instrumentation process in an FPGA.

executed work-items. The difference between two successive timing values returned by I2 allows for the determination of the initiation interval (*II*) of C1. The instruments I1 and I6 can be used to determine the kernel performance metrics (processing time (T_{c1}, T_{c2}, T_{c3}), latency and *II*) when executed in an FPGA device.

B. OVERALL FRAMEWORK

The framework is based on a hardware/software codesign approach that comprises two modules: the hardware module, which is designed with Verilog, and the software module that is specified with C++. In this section, the IE architecture and the hardware architecture of the framework are first detailed. Then, the software module of the framework is presented.

1) HARDWARE MODULE OF THE FRAMEWORK

The hardware architecture of the proposed framework combines the circuits generated from an OpenCL specification and the embedded IEs in the FPGA. This hardware architecture shown in Fig. 5 comprises two main modules. The first module is the ''user's circuit'' that corresponds to the original kernel code. The second module includes IEs, which are connected to memories. Each IE communicates with the user's circuit via an embedded interface, such as the Avalon Streaming Interface (ASI) in the Intel Arria 10 FPGA.

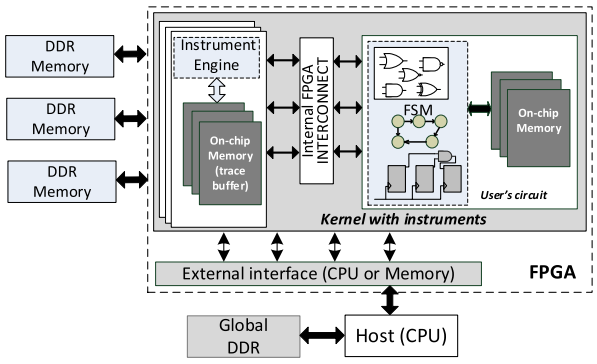


FIGURE 5. System architecture with embedded instruments.

The IE is described using RTL-Verilog and contains three modules (Fig. 6). The first module is a monitor, which is a clock cycle accurate timer that is sampled and buffered with each data value specified by the developer. The second

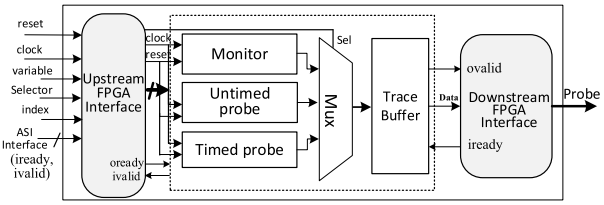


FIGURE 6. Instrument engine architecture.

is an untimed probe that records the value of any variable inside the kernel without any timing information. The third module is a timed probe that provides the variable value associated with its FPGA run-time information. Each module of this instrument can be activated with a selector input that can be specified by the developer during the call of the instrumentation function inside the OpenCL kernel, as shown in Fig. 3. The Verilog description of an IE also contains a trace buffer and an upstream and downstream manager. The upstream and downstream manager allows the instrumented kernel to interact with the IE; the kernel sends and receives data over upstream and downstream FPGA interfaces. Indeed, the upstream and downstream FPGA interface controls the data capture process and enables communication between the kernel and the embedded instrument. These interfaces depend on the specific FPGA technology used. The two most common internal interfaces available are the AXI bus in Xilinx FPGAs [45] and the Avalon Streaming Interface (ASI) [46] in Intel FPGAs. For example, the ASI interacts with the upstream kernel through the inputs of the IE, which are specified in the kernel and the control signals of the ASI.

Each IE records a set of signal values and timing data into a collection of on-chip memories called ''trace buffers''. These trace buffers can be built from either on-FPGA (such as the TB specified in Fig. 3) or off-FPGA memory connected to the FPGA via an external memory interface. The developer can choose between these two types of trace buffers depending on the size of the data set to be extracted. If the data set is relatively small, it is feasible and generally more efficient to store it in on-chip memory than in off-chip memory to reduce latency. Otherwise, some off-chip memory can be used to store data in real time if the data set is too large. Once the data are stored, the host reads all that data and provides the desired information to developers (e.g., timing and debugging data).

2) SOFTWARE MODULE OF THE FRAMEWORK

The software module of this framework contains multiple functions executed by the CPU, including OpenCL platform functions such as defining devices and writing and reading memory buffers. Targeted performance metrics are sampled by instruments from the FPGA and stored in local memory. These data are transferred back to global memories, and then the CPU reads and postprocesses the extracted data to extract the values of variables and the relevant timing performance at FPGA clock cycle accuracy (i.e., the latency and *II*).

The software module includes but is not limited to the following functions: creating buffers, setting up the kernels, transferring data from/to the FPGA, kernel execution on the FPGA, computing the latency, computing the *II* for loops, and executing the checker function. The software module can compute the latency, the *II*, and the number of stalls for computations in real-time. These performance metrics are expressed in FPGA clock cycles as explained in the next subsection (Section III.D). The checker function is a verification function that compares the extracted values of variables with a reference generated by the CPU. The insertion of a checker in the software unit is required for verification purposes and is needed to ensure that the instrumentation framework does not affect the output result. The checker function can be extended to detect hang-up conditions using data returned from the FPGA to the CPU. Hang-up is not included in this framework and may occur not only because of bus congestion but also because of synchronization problems or infinite loops.

C. FRAMEWORK CAPABILITIES

The proposed framework allows developers to implement kernels by offering debugging and timing analysis capabilities at the OpenCL source level: extracting timing performances, analyzing the critical computations and identifying the performance bottlenecks seen in the kernel, such as stalls. The main advantage of the proposed source-level instrumentation framework is that it avoids the difficulty of inserting instruments at the HDL circuit level, which requires in-depth knowledge of hardware design. In fact, in-system debugging consists of inserting an embedded instrument to reveal the behavior of a set of selected OpenCL variables. As shown in Fig. 3, the function *instrument* (1, i, sum) samples the variable ''sum'' associated with its index ''i''. The value 1 is the value of the input ''selector'' of the instrument (as shown in Fig.6), which enables the untimed probe. The monitor is likewise enabled when the input selector is equal to 0. The timed probe performs the timing analysis function. It records the run-time information of the variable to identify possible stalls associated with the variable ''sum''.

The instrument can also be used to provide the runtime information of any operation or computation inside the kernel (e.g., loop, load, or FFT). Furthermore, the software module contains a verification function, which performs a functional verification by comparing the result of the accelerated algorithm that is running on an FPGA to a model of this algorithm that is written in C++ and executed by the CPU. The main advantage of this function is that it ensures that the functionality of an accelerated algorithm remains correct when IEs are present. This software function can then detect unexpected behaviors exhibited by the system that could be caused by IEs. For example, the variable ''sum'' presented in Fig. 3 is compared to the reference value, and this comparison is an addition operation executed on the host (CPU) to verify the integrity of the in-FPGA operation (sum) and then to ensure that the framework does not affect the result during the instrumentation process.

D. PERFORMANCE MODELING AND INSTRUMENTATION

An OpenCL application can be designed using a single-kernel or multikernel structure. The performance of an OpenCL application changes in terms of frequency, bandwidth and logic resource utilization following some changes to the design topology. This section presents performance models of OpenCL using single kernels and multiple kernels together with performance metrics that can be calculated using the proposed framework.

1) SINGLE-KERNEL PERFORMANCE MODELING

A single OpenCL kernel may contain many pipelined computations (C) that can be instrumented to determine different performance metrics (Fig. 7), such as the latency (L) and *II* of various pipelines. These performance metrics are expressed in FPGA clock cycles except for the execution time (T_{sk}), which is expressed in seconds. The inserted instruments extract the run-time of each processed work-item (wi), while the circuit is running on the FPGA. The number of instruments to be inserted depends on which performance metrics need to be characterized. For example, only one instrument is needed to extract the *II* of a given processing pipeline, and a minimum of two instruments are required for measuring the latency of the kernel, which are computed as follows:

$$II_{ij} = t_j(wi(i+1)) - t_j(wi(i)) \quad (1)$$

$$L = t_n(wi(i)) - t_1(wi(i)) \quad (2)$$

where t($wi(i)$) is the measured run-time for the specific work-item $wi(i)$ expressed in FPGA clock cycles. The initiation interval II_{ij} is defined as the difference between the information about two successive run-times of two successive work-items processed by the computation Cj. Instrumenting computations allows for the optimization of FPGA run-times and the detection of pipeline stalls and the causes of these stalls. For example, an efficient and perfect loop implementation in an FPGA has a constant II of 1, meaning that one work-item is processed per FPGA clock cycle. Therefore, inserting instruments allows us to monitor the changes in *II* as the work-items are processed and to detect the occurrence of stalls (number, frequency, profile).

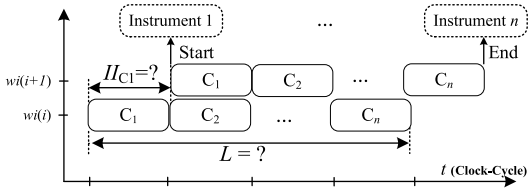


FIGURE 7. Performance metrics of a single kernel.

Alternatively, it is generally difficult to achieve a perfect II in complex computations; thus, it is crucial to insert instruments that can accurately measure the *II* and then detect computations that cause stalls. Therefore, we propose to insert instruments between each computation to allow for the calculation of IIs, to diagnose bottlenecks that may degrade

the performance and to trace stall paths through computations. Fig. 7 shows a single kernel containing n computations. The total number of FPGA-clock cycles required by a kernel to process all work-items is defined by Tsk and given by equation 3:

$$T_{sk} = L + II \times (N_{WI} - 1) + T_{stalls} \tag{3}$$

where N_{WI} denotes the number of work-items. T_{stalls} denotes the overhead due to stalls in clock cycles, and it is difficult to predict because it is highly dependent on hardware architecture and operating systems and very sensitive to data sizes and memory footprints and topologies. T_{stalls} is then extracted and computed by the proposed framework.

The latency (L) is the maximum number of clock cycles needed to produce a kernel output from its inputs, and it is defined as follows:

$$L = \mathrm{Max}(L_{\mathrm{C1}}, L_{\mathrm{C2}}, \ldots, L_{\mathrm{Cn}}) \tag{4}$$

The total processing time (T_{pr}) of an OpenCL application expressed in seconds is calculated by equation 5, where F_{FPGA} denotes the frequency of the kernel.

$$T_{\mathrm{pr}} = \frac{T_{\mathrm{sk}}}{F\mathrm{FPGA}} \tag{5}$$

2) MULTIKERNEL PERFORMANCE MODELING

The main challenge in multikernel OpenCL designs is that bottlenecks may simultaneously occur in several kernels. Moreover, performance degradation may occur due to kernel-to-kernel communication interfaces, which are called pipes or channels in OpenCL. For example, bad settings of pipe depths may significantly degrade the timing performance. Indeed, instruments are inserted at the inputs and outputs of each kernel, as shown in Fig. 8, to dynamically characterize pipes and to track the timing performances of kernels, mainly stalls, during their execution on the FPGA. In addition, these measurements at FPGA clock cycle accuracy can be used to determine the exact depths of OpenCL pipes to avoid stalls and to save FPGA logic resources. The following model is proposed to detect stalls in multikernel data flow-based OpenCL applications. It is assumed that n instruments ($I = \{\mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3, \ldots, \mathrm{I}_n\}$) are inserted in kernels and that a set of m work-items ($WI = \{\mathrm{wi}_1, \mathrm{wi}_2, \mathrm{wi}_3, \ldots, \mathrm{wi}_m\}$) is processed by the kernels.

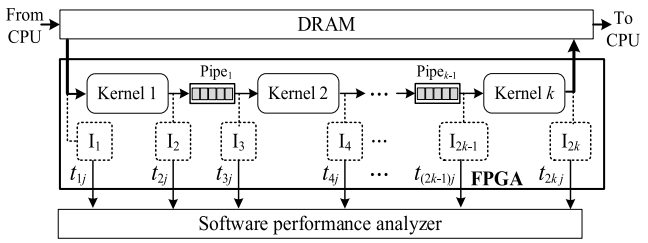


FIGURE 8. Multikernel modeling using the inserted instruments.

The run-time extracted by each instrument for a specific work-item can be represented by a T_{nm} matrix:

$$T_{nm} = \begin{pmatrix} t_{11} & \cdots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{n1} & \cdots & t_{nm} \end{pmatrix} \tag{6}$$

An initiation interval matrix $\Gamma_{n(m-1)}$ can then be computed according to equation (1):

$$\Gamma_{\mathrm{n(m-1)}} = \begin{pmatrix} t_{12} & \cdots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{n2} & \cdots & t_{nm} \end{pmatrix} - T_{n(m-1)} = \begin{pmatrix} II_{11} & \cdots & II_{1(m-1)} \\ \vdots & \ddots & \vdots \\ II_{n1} & \cdots & II_{n(m-1)} \end{pmatrix} \tag{7}$$

A kernel without stalls implies that $II_{nm} = II_{n(m+1)}$ for all values of n and m. A stall is then detected when $II_{nm} \neq II_{n(m+1)}$, and the number of stall cycles in instrument n between work-items $m+1$ and m is computed as $II_{n(m+1)} - II_{nm}$. These differences can be expressed as an II difference matrix ($\Delta_{n(m-2)}$), as in equation 8:

$$\Delta_{n(m-2)} = \begin{pmatrix} II_{12} & \cdots & II_{1(m-1)} \\ \vdots & \ddots & \vdots \\ II_{n2} & \cdots & II_{n(m-1)} \end{pmatrix} - II_{n(m-2)} = \begin{pmatrix} \delta_{11} & \cdots & \delta_{1\mathrm{m}} \\ \vdots & \ddots & \vdots \\ \delta_{\mathrm{n1}} & \cdots & \delta_{\mathrm{nm}} \end{pmatrix} \tag{8}$$

which is a zero matrix in the absence of stalls. A stall exists when one $\delta_{\mathrm{ij}} \neq 0$. If the number of stalls is different from one kernel to another, several kernels may cause stalls simultaneously. As an example based on Fig. 8, if instruments I2, I3, and I4 provide the $\Delta_{\mathrm{n(m-2)}}$ matrix with $\delta_{24} = \delta_{34} = 15$, and $\delta_{44} = 35$, then kernel 1 and kernel 2 have caused different numbers of stalls more or less simultaneously. Finally, each kernel k and a set of work-items m that cause the stalls are detected by the framework for optimization purposes. A latency matrix for the multikernel data-flow architecture, as shown in Fig. 8, can also be computed (equation 9). This matrix provides the latency of the computations between two consecutive instruments for each work-item.

$$L_{(n-1)m} = \begin{pmatrix} t_{21} & \cdots & t_{2m} \\ \vdots & \ddots & \vdots \\ t_{n1} & \cdots & t_{nm} \end{pmatrix} - \begin{pmatrix} t_{11} & \cdots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{(n-1)1} & \cdots & t_{(n-1)m} \end{pmatrix} = \begin{pmatrix} l_{11} & \cdots & l_{1m} \\ \vdots & \ddots & \vdots \\ l_{(n-1)1} & \cdots & l_{(n-1)m} \end{pmatrix} \tag{9}$$

IV. FRAMEWORK IMPLEMENTATION AND RESULTS

The implementation of the proposed framework is presented in this section. The experimental setup used for the development of the framework is first described. The source files required to build the instrumentation library are then detailed. Two use cases, an OpenCL single-kernel design and a multikernel design, are finally used to show the capabilities of the proposed instrumentation methodology. Finally, the logic area overhead of the framework is discussed.

A. EXPERIMENTAL SETUP

Our proposed framework is validated using a heterogeneous platform. This platform includes the Nallatech 510T FPGA board shown in Fig. 9, which features two Arria 10 1150 GX FPGAs and four 4 GB memory banks per FPGA (32 GB total) installed on a Dell machine using a PCIe x8 3.0 interface. The Intel SDK for OpenCL v17.1 is used for the implementation.

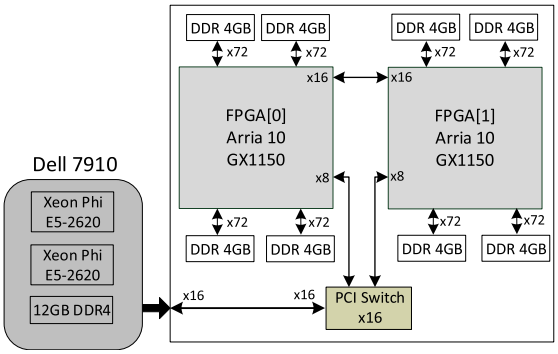


FIGURE 9. Nallatech 510T platform used for the experiments.

B. IMPLEMENTATION OF THE INSTRUMENTATION LIBRARY

Implementing an instrument consists of creating an instrumentation library, which is composed of three files: the header file, a Verilog file, and an eXtensible Markup Language (XML) file, as shown in Listing 1. The Verilog file describes the IE architecture presented in Fig. 6. The output "probe" of the IE, as illustrated in line 8 of Listing 1, returns the sampled 64-bit data defined in lines 31 to 33 according to the selected type of instrument defined by the input selector (sel). The input "sel" is given by the developer at the OpenCL level to select the monitor, the timed probe or the untimed probe, as described in lines 27 to 29 (Listing 1). Lines 19 to 24 implement the synchronization process of the inputs of the instrument with the internal interface of the FPGA, as required by the Avalon Interface for Intel FPGAs.

An XML file is required to describe the IE properties, such as expected the stall-free operations, desired latency, and side effects (lines 39, 41). The stall-free attribute specifies "probe" data to be sent without constant delays. Paired with the expected instrument latency, these data allow for synchronization with different sections of the kernel. The side effects attribute is necessary when the function *instrument* ( ) includes an internal state or communicates with external memory. In addition to properties, in the XML file the developer must specify: the Verilog file of the IE called *instrument.v*; the input and output ports of the IE; and the kernel file called *instrument.cl* that defines the model of the IE used for emulation purposes. These three files are compiled using an OpenCL compiler to generate the instrumentation library (*instrument.aoclib*), as shown in Fig. 10. Finally, the function *instrument* ( ) described in line 1 can be called from any OpenCL kernel to infer an instrument in the FPGA. The main

```
// Header .h file
1 unsigned long instrument (sel,index,var);
// Pseudocode of the Verilog file: instrument.v
2 module instrument
3  (input clock, resetn, ivalid, iready,
4 startofpacket, endofpacket,
5 input [31 : 0] enable, sel,
6 input [15 : 0] index, input_var,
7 output  oready, ovalid,
8 output  [63 : 0] probe);
9 assign probe=data;
10 assign i_inputs = {index,sel,input_var};
// local wires and signals declarations are omitted.
11 always @(posedge clock or negedge resetn)
12 begin
13   if( ~(resetn))
14     counter_time <= 32'h0;
15   else if (counter_time ==32'hFFFFFFFF)
16     counter_time <= 32'h0;
17   else
18     counter_time <=(counter_time + 32'h1);end
19 always @(posedge clock or negedge resetn)
20 begin
21   if( ~(resetn))
22     o_inputs <= WIDTH'h0;  //WIDTH =16 or 32
23   else if (ivalid)
24     o_inputs <= i_inputs; end
25 always @*
26 begin
27  monitor = {counter_time,o_index};
28  untimed_probe = {input_var,o_index};
29  timed_probe={counter_time,o_index,o_input_var};
30  case (o_sel)
31    32'h0 : data = monitor;
32    32'h1 :data = untimed_probe;
33    default : data = timed_probe;
34  endcase end
35 endmodule
//XML file for instrument engine
36 <EFI_SPEC>
37 name="instrument" module="instrument">
38 <ATTRIBUTES>
39   <IS_STALL_FREE value="yes"/>
40   <EXPECTED_LATENCY value="1"/>
41   <HAS_SIDE_EFFECTS value="no"/>
42 <INTERFACE>
43   <AVALON port="list of Avalon signals"/>
44   <INPUT port="list of input signals"/>
45   <OUTPUT port="list of output signals"/>
46 <C_MODEL>
47 <FILE name="instrument.cl"/>
48 <FILE name="instrument.v"/>
```

Listing 1. Subset of the pseudocode for the instrumentation library (Header, Verilog and XML files).

advantage of this structure is that the instrumentation library does not need to be rewritten or recompiled for a new use case. The implementation flow in Fig. 10 uses Intel SDK for OpenCL, which supports the OpenCL 1.0 standard and some of the features of OpenCL 1.1 and 1.2 [47]. The Intel offline compiler compiles the kernel (.cl) to the FPGA image (.aocx). The host code is compiled with a regular C compiler to an executable file (.exe) using GCC under Linux or Microsoft Visual Studio. The instrumentation library is then linked to the OpenCL kernel and compiled with the offline compiler.

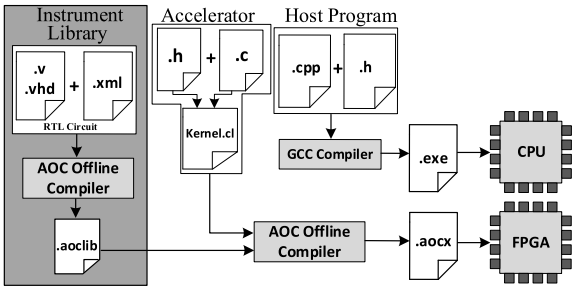


FIGURE 10. Instrument implementation flow using the Intel FPGA SDK for OpenCL.

The interaction between the IE and the OpenCL kernel is made by using an ASI in the FPGA. The ASI is an intellectual property (IP) core supported by the Intel SDK for OpenCL compiler. Its input and output ports (e.g., ivalid, oready) are specified in the Verilog and XML files, as shown in Listing 1 (lines 3, 7 and 43). The ASI receives the inputs as specified in the function *instrument* ( ) inside the kernel. For example, the ASI interacts with the upstream kernel through the ASI control signals (Fig. 6) and the inputs specified in the function *instrument* (2, i, a) (line 11 of Listing 2). The ivalid signal is set to 1 if the input signal accommodates valid data. When the oready signal is equal to 1, this indicates that the IE has valid data to be processed. When ivalid is activated and oready falls to 0, the upstream kernel holds the values during the next clock cycle. The IE communicates with the downstream kernel through the ovalid and iready signals. If the output data signal of the trace buffer contains valid data, the ovalid signal is activated. If the output data are valid, then the iready signal is equal to 1, which means that the kernel possesses valid output data for processing. When ovalid is equal to 1 and iready is equal to 0, the IE holds the valid data until the next clock cycle.

C. INSTRUMENTATION METHODOLOGY AND RESULTS

Two use cases implemented to validate the proposed instrumentation and performance characterization methods are detailed in this section to show how to extract kernel performances using the instrumentation framework introduced in Section III.D. The first use case shows the instrumentation of a single OpenCL kernel to extract its performance metrics. The second use case is a multikernel implementation of the first use case, and it shows the instrumentation methodology of a multikernel OpenCL-based design. Finally, the impact of the inserted instruments on FPGA resources is characterized for both single-kernel and multikernel designs.

1) SINGLE-KERNEL INSTRUMENTATION

An algorithm that contains addition, multiplication, loading and storage operations is described in Listing 2 as an OpenCL single kernel with 4 inserted instruments. I1 to I4 represent the trace buffers that are declared in the kernel as local variables (lines 11, 12, 14 and 16). These local variables

```
1 //kernel.cl
2 #include "instrument.h"
3__attribute__((reqd_work_group_size(WLsize,1,1)))
__kernel
4  void singlekernel (
5   __global const int *restrict x,
6   __global const int *restrict x,
7   __global const int *restrict II1,
// Input & output declarations omitted
8    int N){
// Local variable declaration omitted
9  for(i=0; i<N; i++){
10   a = x[i];b = y[i]; // load operation
11   I1   = instrument (2,i,a);
12   I2   = instrument (2,i,b);
13   add  = a+b; //addition operation
14   I3   = instrument (2,i,add);
15   mul  = add*a; //multiplication operation
16   I4   = instrument (2,i,mul);
17   z[i] = mul; // store operation
// Uncomment for the NDRange model
18   //barrier( CLK_GLOBAL_MEM_FENCE);
19   II1[i] = I1;
// Remaining code omitted } }
```

Listing 2. Pseudocode of a single-kernel example with 4 instruments.

are interpreted by the OpenCL for FPGA compiler as local memories (BRAMs). The width of the trace buffers is mainly defined according to the sampled data (64 bits) extracted by each *instrument* ( ). Since trace buffers are interpreted by the OpenCL for the FPGA compiler as BRAMs, the depth of the trace buffer is managed by the compiler and mainly depends on the FPGA technology used, in which the BRAMs may have different sizes. Each sampled data is then transferred to global memory to provide access to the host, as shown in line 19 of Listing 2. Once the extracted data reside in global memory, the host has access to each data point for postprocessing.

Two execution models are instrumented: the NDRange and single work-item (SWI) execution models. The NDRange model explicitly implements data parallelism while performing computations by dividing the problem into work-groups and work-items. The kernel is interpreted as an SWI when the WLsize parameter (line 3, Listing 2) equals 1 and as an NDRange otherwise. Instruments are inferred as timed probes to compute the performance metrics defined in Section III.D: the latency of the output "mul" and the initiation interval of the loop and its pipeline stalls. Indeed, instruments I1 and I2 probe at the moment when the inputs "a" and "b" are loaded while I3 and I4 allow for computing the latency of both the addition and multiplication operations. Additionally, the kernel presented in Listing 2 is tested with problems of various sizes (numbers of work-items) to reveal its behavior while it is running in the target FPGA. The latency and *II* are then computed according to equations 1 and 2. The *II* is measured by I4 on 50 K work-items (a workload of 200 KB), and it is presented in Fig. 11. The *II* shows that one work-item is processed at each clock cycle in the SWI execution model.

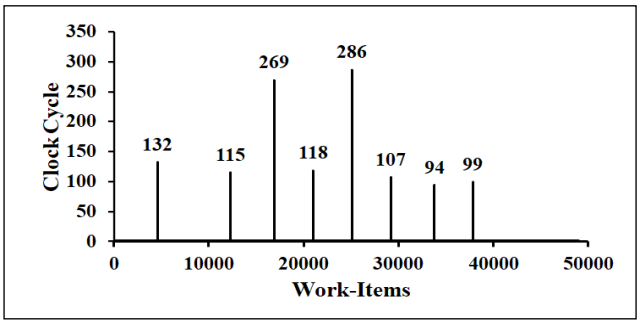


FIGURE 11. Initiation interval measured by I4 in the OpenCL SWI execution model, as described in Listing 2 (WLsize=N=50 K).

However, Fig. 11 also clearly shows eight stalls that range in duration from 94 clock cycles to 286 FPGA clock cycles.

Following the instrumentation methodology for the single-kernel case, a simulation of 50000 work-items is executed, during which work-items are sampled to reveal the behavior of the NDRange execution model (WLsize = 50 K). Fig. 12 shows the II of the NDRange execution model with and without a barrier function (Listing 2), which is used to synchronize operations to local memory. A kernel with a barrier function is interpreted by the OpenCL for the FPGA compiler as an NDRange execution model, and it exhibits a different execution behavior while running in the FPGA (Fig. 12.(b)). Unlike that of the SWI model, the execution of the NDRange model has an unstable throughput of the multiplication operation measured by I4, as shown in Fig. 12 (a), where only the II for the first 100 work-items processed is presented. The II curve starts with 33 clock cycles and jumps to thousands of FPGA clock cycles when in a stalled state. The II of the NDRange model measured by I4 is clearly poor compared to that of the SWI model. As instruments I1 and I2 exhibit the same behavior, as shown by Fig. 12 (a), for load and store operations, this unstable II is then explained by the unsuitable host-FPGA data transfer in the NDRange model, and these stalls are not caused inside the FPGA. The II of the NDRange implementation with a barrier function (Fig. 12(b)) is 50 K clock cycles for most of the 50 K work-items. It also shows several stall-states with durations between 2 (II=50002) and 32 (II=50032) FPGA clock cycles when 200 KB of data are processed (Fig. 12 (b)).

The timing performances are computed as described by equations 3 and 4 for both the SWI and NDRange models without barriers, and they are summarized in Table 1. The framework shows that the SWI execution model requires 21.2 times fewer FPGA clock cycles for execution than the NDRange model does. This overhead is mainly due to the large number of stalls observed in the NDRange implementation that contribute up to 1975 % of the optimal execution time Tsk, which should be 50 K clock cycles for the 50 K work-items. However, these two models maintain a stable minimum latency of 3 clock cycles for the multiplication operation, computed as the difference between I3 and I4

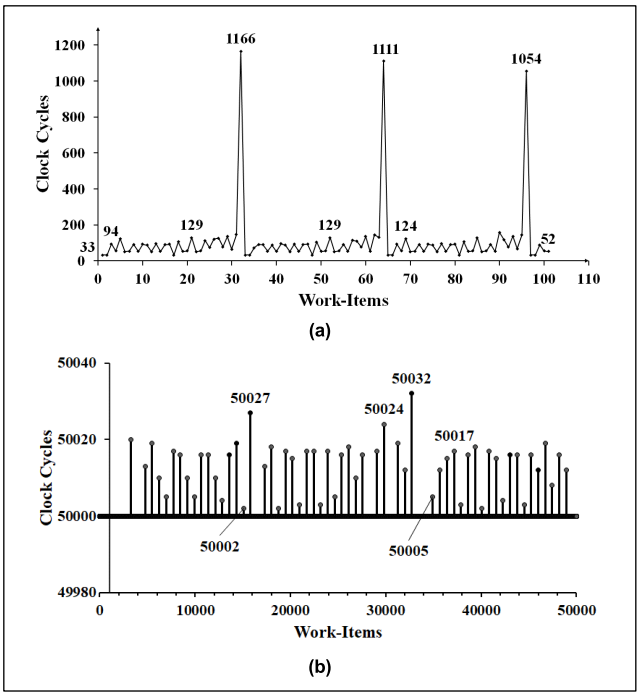


FIGURE 12. Initiation Interval of NDRange execution models described in listing 2 (WLsize=1, N=50 K), which is the same for the four instruments. (a) without a barrier (only the first 110 work-items are presented), (b) with a barrier (50 K work-items).

TABLE 1. Timing performance in terms of FPGA clock-cycles for Listing 2 with 50 K work-items (200 KB workload) for two execution models.

Execution Models	T_{sk}	L_{min}	F_{FPGA} (MHz)	Stalls (%)
SWI (Fig. 11)	51260	3	287	2.1%
NDRange without a barrier (Fig. 12 (a))	1087630	3	259	1975%
Ratio	21.2	1	1.08	914.4

for the same processed work-item. These results help not only to determine the suitable execution model but also to confirm that the choice of an execution model running on FPGAs has a significant impact on the timing performance and must be considered while writing a kernel targeting an FPGA.

The FPGA resource utilizations related to the two execution kernels presented in Listing 2 are summarized in Table 2. The overhead costs of four instantiated instruments measured from the FPGA resource utilization reports are similar for both execution models and remain acceptable. The overhead of these instruments for the NDRange model is 0.72 % greater for the logic unit (LU), 0.61 % greater for flip-flops (FFs) and 1 % greater for ALUTs than the values obtained by the implementation without instruments, and this confirms that the framework has a small impact on FPGA resource utilization. The FPGA operating frequency slightly decreases (from 287 MHz to 280 MHz) with the insertion of four instruments in the SWI execution model. In addition,

TABLE 2. FPGA resource utilization for kernels with 4 inserted instruments.

No. Of instruments	Execution model	FFs	ALUTs	LU (ALMs)	FREQ (MHz)
0	SWI	77490 (5.17%)	40389 (5.39%)	47972 (11%)	287
	NDRange	77827 (5.20%)	40690 (5.43%)	48223 (11.28%)	270
4	SWI	86791 (5.8%)	47957 (6.41%)	(53161) (12%)	280
	NDRange	86961 (5.81%)	48216 (6.44%)	53343 (12%)	258
Overhead of 4 instruments	SWI	0.63%	1.02%	1%	-2.4%
	NDRange	0.61%	1.01%	0.72%	-4.4%

TABLE 3. Kernel execution time (ms) for the SWI kernel (Listing 2).

	Number of instruments per kernel				
Workload	0	2	4	6	8
5 KB	0.24	0.26	0.29	0.33	0.34
40 KB	1.26	1.41	1.59	1.72	1.81
100 KB	3.52	3.59	3.63	3.69	3.72
4 MB	7.61	7.64	7.68	7.71	7.72
40 MB	75.48	75.49	75.50	75.51	75.52

experiments are performed to characterize the impact of the insertion of a different number of instruments on kernel execution times, as summarized in Table 3. The results show that even when the number of work-items increases, the kernel execution time slightly increases (by a maximum of 0.04 ms) when 40 MB of data is processed. Since high-performance applications require a large amount of input data [1], this experiment confirms the efficiency of the proposed framework in terms of its very small impact on the execution time when a heavy workload is processed by an OpenCL application.

2) MULTIKERNEL INSTRUMENTATION

This section presents a use case in which instruments are added to a multikernel-based design. The algorithm presented in Listing 2 is instrumented by dividing the single kernel into three kernels, as shown in Listing 3: the first kernel contains the loading of variables a and b. The second kernel contains the addition and multiplication computations, and the last kernel contains the storage operation. Six instruments are inserted to provide the performance metrics, as explained in section III.D. These instruments monitor the performance of in-FPGA operations and the OpenCL pipe as well.

The run-time is extracted and used to compute the initiation interval and the II difference matrix, as shown in Fig. 13. The instruments allow the detection of stalls either inside kernels or in the pipe between kernels. The second kernel of Listing 3 contains the in-FPGA multiply-add operations, in which stalls are also observed. These stalls can be caused by the

```
__Kernel void load_Data (
//Pipes and Global input/output declaration)
{
// Local variable declaration
   for(i=0; i<N; i++){
     a = x[i];
     b = y[i];
     I1[i] = instrument (2,i,a);
     I2[i] = instrument (2,i,b);
     write_pipe( Load_A, &a );
     write_pipe( Load_B, &b ); } }
__Kernel void inFPGA operations (pipes) {
   for(i=0; i<N; i++)   {
      read_pipe(Load_A,&Ain);
      read_pipe(Load_B,&Bin);
      I3[i] = instrument (2,i,Ain);
      I4[i] = instrument (2,i,Bin);
      Sum   = Ain+Bin;
      Mul   = Bin*Sum;
      I5[i] = instrument (2,i,Mul);
      write_pipe( Mul_OUT, &Mul );   }  }
__Kernel void store_data (pipes, outputs ) {
 for(int i=0; i<N; i++){
  read_pipe(Mul_OUT, &last_val);
  I6[i] = instrument (2,i,last_val);
  z[i]  = last_val;}     }
```

Listing 3. Pseudocode of a multikernel example with six instruments I1-I6.

$$\Gamma_{68}=\begin{pmatrix}1&1&1&1&1&1&1&1\\1&1&1&2&1&1&1&1\\1&1&1&1&1&1&1&1\\1&1&1&2&1&1&1&1\\1&1&1&1&1&1&1&1\\1&1&1&1&1&1&1&1\end{pmatrix}\qquad \Delta_{67}=\begin{pmatrix}0&0&0&0&0&0&0&0\\0&0&1&-1&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{pmatrix}$$

FIGURE 13. Matrices of II (Γ_{68}) and its difference Δ_{67}, computed according to equations (8) and (9), respectively, for the multikernel implementation.

propagation of stalls from the first kernel when the OpenCL pipes (Load_A, Load_B) are empty or when the Mul_OUT pipe is full. In such a case, a new pipe implementation with a larger depth would be required to reduce the occurrences and impacts of stalls and to improve the performance of the model. Fig. 13 shows Γ_{68} and Δ_{67} computed according to equations 7 and 8. The II of the addition and multiplication computations performed by I5 is constant to one clock cycle; this demonstrates a stall-free operation compared to that of the single-kernel implementation.

The latency of the in-FPGA operations is equal to 1 clock cycle, which is ideal. Compared to the unstable *II* curve of the NDRange single-kernel implementation (Fig. 12), this implementation shows a stable *II* of 1 clock cycle, similar to that of the SWI kernel, but with a higher frequency (352 MHz, a 13 % increase). It is also of interest that the framework results in a low logic area. Table 4 summarizes the FPGA resource utilization; the overhead is very small: 1.29 % in ALUTs, 1.34 % in LU utilizations, and 0.73 % in Flip-Flops. The table also reports that the framework causes a 4.54 % reduction in the operation frequency.

TABLE 4. FPGA resource utilization and overhead of multiple kernels with 6 inserted instruments (Listing 3).

No. of instruments /kernel	Flip Flops	ALUTs	LU	FREQ (MHz)
0	80352 (5.37%)	43162 (5.77%)	49604 (11.61%)	352.6
6	91329 (6.1%)	52789 (7.06%)	55322 (12.95%)	336.58
Overhead of 6 instruments	0.73%	1.29%	1.34%	-4.54 %

D. EVALUATION OF FRAMEWORK SCALABILITY

As a final experiment, the impact of inserting instruments on a set of benchmarks is investigated to validate the scalability of the proposed framework. These benchmarks are part of the CHO benchmark suite [48], Intel benchmark [49], and Rodinia benchmark sets [50]. They include the implementation of arithmetic applications such as dfadd, dfmul, dfsin and other algorithms inspired by cryptographic applications such as sha and blowfish. The FPGA resource utilizations for these benchmarks without instruments are presented in Table 5. Two, 10, 20 and 50 instruments are instantiated in each benchmark. The FPGA logic overhead per instrument is summarized in Fig. 14, Fig. 15 and Fig. 16, where the maximum resource utilizations in terms of the ALUTs, FFs and LU per instrument are 0.25 %, 0.22 % and 0.31 %, respectively. These results show that the variation in terms of FPGA resource consumption per instrument is larger when fewer instruments are inserted. However, the overhead converges to a relatively stable percentage of FPGA resource utilization when an important number of instruments is inserted. Notice that the insertion of instruments may also lead to a decrease in the amount of logic resources (FFs or ALUTs) needed for each instrument, as shown by benchmark 9 (−0.01 % for FFs) in Fig. 15. This is possible since the optimization of the OpenCL compiler is applied heuristically, and the instrumentation framework may then lead the OpenCL compiler along new optimization paths, from which the compiler may find a better solution. However, this decrease in the number of FFs corresponds to an increase in the number of ALUTs (0.06 % for ALUTs, as shown in Fig. 14).

TABLE 5. FPGA resource utilization for 17 benchmarks without instruments.

	Benchmark	ALUTs	FFs	LU	Freq MHz
1	mips	43441	82624	50449	252.3
2	kmeans	46492	84110	52123	270.2
3	gsm	52816	88114	54900	115.8
4	adpcm	61975	105273	63742	225.9
5	hotspot3d	67797	104203	69722	205
6	fd3d	79372	125814	74626	206.5
7	fft1D	93141	161411	88694	264.1
8	dfmul	95739	124516	83404	232.8
9	sha	96700	126838	88218	168.4
10	blowfish	109858	172347	92815	135.4
11	cfd	120119	194048	108747	200.8
12	jpeg	142155	222400	118994	171.1
13	srad	143616	223210	122300	220.3
14	dfsin	176717	286011	151017	126.4
15	lud	194698	292677	160469	218.8
16	dfadd	198038	215882	166064	220.7
17	Quicksort	257080	336544	211812	180.4

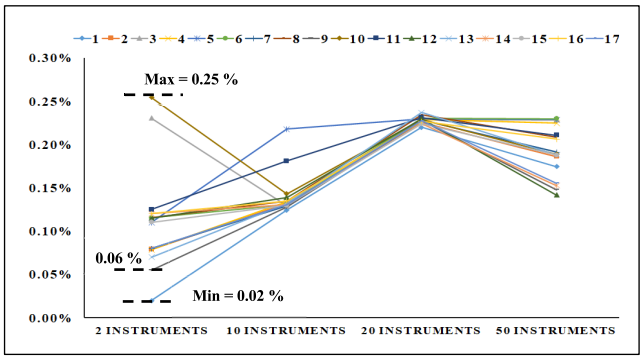


FIGURE 14. ALUT utilization in the FPGA for 17 benchmarks with different numbers of inserted instruments.

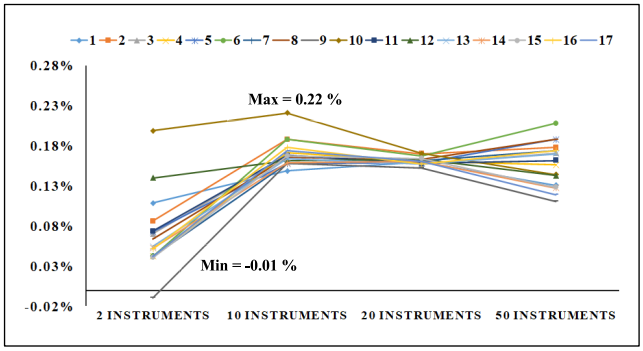


FIGURE 15. FF utilization for 17 benchmarks with different numbers of inserted instruments.

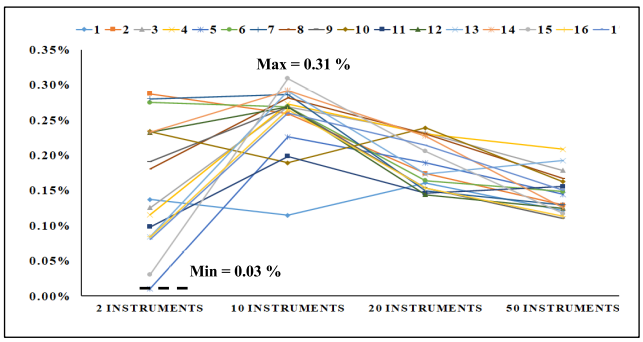


FIGURE 16. LU utilization for 17 benchmarks with different numbers of inserted instruments.

The framework also increases the number of BRAMs needed to record the data and timing information of targeted signals. The maximum reported memory utilization in an Arria 10 is 114 BRAMs when 50 instruments are inserted; this represents 4.2 % utilization of the available number of memory blocks. In fact, the experimental results of the instrumentation of 17 benchmarks show that the average memory utilization per instrument is 1.5, 2.35, 2.45, and 2.22 BRAM blocks when 2, 10, 20, and 50 instruments are inserted, respectively. It is of interest that the inserted instruments do not consume any DSP slices. Finally, these results show that the logic consumption is relatively small, and the framework can be applied to a wide variety of applications.

The operating frequency of the kernels under instrumentation decreases when the number of instruments increases. However, the frequency increases in some benchmarks when embedded instruments are inserted. For example, the frequency of the MIPS benchmark (benchmark 1) improves by 2.39% when 2 instruments are inserted, as shown in Fig. 17. Two reasons explain this improvement. First, the OpenCL-to-FPGA compiler uses a heuristic algorithm for the place and route in the FPGA, and the addition of instrumentation circuits may lead the compiler to find better solutions. Second, the circuit with embedded instruments may be better structured than the circuit without embedded instruments. This may lead to more efficient solutions in the FPGA. Fig. 17 summarizes the impacts of the framework on the operating frequencies of 17 benchmarks, for which the maximum frequency loss is 4.5% (benchmark 5). This result demonstrates that the framework slightly affects the performance of OpenCL kernels.

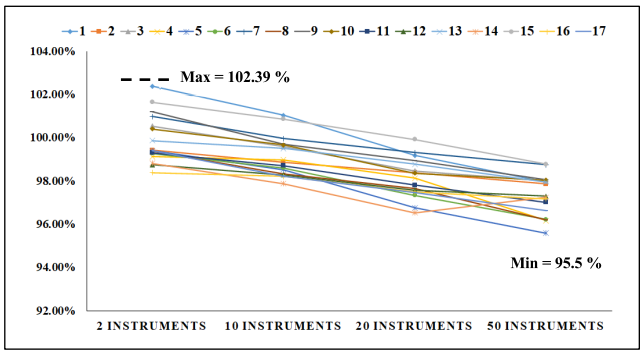


FIGURE 17. Frequency variations for 17 benchmarks with different numbers of inserted instruments.

V. FRAMEWORK COMPARISON

As mentioned in the introduction, several works have reported on the instrumentation of digital circuits for FPGAs. These works focused on the instrumentation of circuits by developing infrastructures from low-level to high-level specifications. Nonetheless, in terms of functionality and compatibility, none of these previous works provide an instrumentation method compatible with OpenCL-based FPGA designs. Moreover, both the Xilinx and Intel for FPGA toolchains do not offer debugging or profiling support for detecting the causes of stalls inside OpenCL kernels including multiple compute units or pipes. The Xilinx SDAccel toolchain [51] generates general profiling reports that contain logic resource utilizations and DDR external memory performance metrics, such as bandwidth, usage, and the occurrence of stalls. The *II* for loops can also be reported when an optimal pipelined kernel is implemented. However, the *II* of a loop is undefined when an OpenCL kernel is implemented as an NDRange. The proposed framework enables performance analyses inside kernels regardless of the contents and the execution model of the kernels, as reported in the use cases presented in Section IV.C.

The framework also offers portability and scalability. Table 6 compares the logic resource overhead per instrument of our framework to those of previous works. The logic area overhead shows that our framework consumes on average 6 times fewer FFs and almost 1.5 times fewer ALUTs than the best previously published work [22] in terms of logic overhead when one instrument is inserted. For example, the average relative Flip-Flop utilization overhead (compared to the original design) in our experiments is 0.10 %, which can be compared to the value of 0.6 % in [22], while the average logic element (LE(LU)) utilization is 2.5% in [24]. To the best of our knowledge, our proposed framework is also the first that can simultaneously support, debugging and timing performance analysis capabilities for OpenCL-based FPGA designs at FPGA clock cycle accuracy, as summarized in Table 7. In addition, this OpenCL-based method can be applied to any OpenCL-based design for a FPGA, and this confirms its portability when using other FPGA vendors, unlike the works reported in [23], [24], [31].

TABLE 6. Comparison of average resource overhead per instrument (%).

	ALUTs		FFs		LE (LU)			
No. of instruments	This work	[22]	This work	[22]	This work	[23]	[24]	[42]
2	0.2	0.3	0.1	0.6	0.14	13.9	2.5	1.09
10	0.13	N/A	0.17	N/A	0.26	N/A	N/A	N/A
20	0.23	N/A	0.16	N/A	0.18	N/A	N/A	N/A
50	0.19	N/A	0.17	N/A	0.15	N/A	N/A	N/A

TABLE 7. Comparison of framework capabilities.

Framework capabilities	This work	[23]	[24]	[31]
Debugging	YES	YES	YES	YES
Timing analysis	YES	NO	NO	NO
Functional verification	YES	NO	NO	NO

VI. CONCLUSION

In this paper, we proposed a framework that offers developer support for enhancing visibility and observability inside

OpenCL-synthesized designs. It does so by adding embedded instruments to OpenCL kernels. These instruments enable the timing performance analyses of OpenCL designs compiled to FPGA targets. This makes our work the first to consider the insertion-based instrumentation restrictions in OpenCL designs compiled for FPGA targets. It also considers the debugging challenges in Intel SDK for OpenCL while overcoming the lack of a supported integrated logic analyzer. To our knowledge, the proposed framework is the first that offers the insertion of embedded instruments described with HDL by specifying them directly into high-level code to extract performances at FPGA clock cycle accuracy. The different use cases presented in this paper demonstrate the accurate measurements of initiation intervals and stalls and confirm that the timing performance highly depends on the OpenCL execution model chosen. The timing performances extracted by the framework enable the developer to choose an implementation and an OpenCL model that help eliminate stalls and achieve better timing performances. Finally, the results also show that the maximum relative FPGA overhead in terms of ALUTs, FFs, and LU cannot exceed 0.25 %, 0.22 %, and 0.31 % per instrument, and the average overhead is 6 times fewer FFs and almost 1.5 times fewer ALUTs than in previous works.

ACKNOWLEDGMENT

The authors specifically thank CMC Microsystems for providing the design tools and FPGA platforms that made this work possible.

REFERENCES

[1] W. Vanderbauwhede and K. Benkrid, *High-Performance Computing Using FPGAs*. Berlin, Germany: Springer, 2013.

[2] D. Singh, "Implementing FPGA design with the OpenCL standard," Intel. (Altera), San Jose, CA, USA, White Paper WP-01173-3.0, Nov. 2013.

[3] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Dordrecht, The Netherlands: Springer, 2008.

[4] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[5] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "OpenCL for FPGAs: Prototyping a compiler," in *Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms (ERSA)*, 2012, p. 1.

[6] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA," in *Proc. IEEE 26th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2015, pp. 189–193.

[7] A. Sanaullah and M. C. Herbordt, "Unlocking performance-programmability by penetrating the intel FPGA OpenCL toolflow," in *Proc. IEEE High Perform. extreme Comput. Conf. (HPEC)*, Sep. 2018, pp. 1–8.

[8] K. O'Neal and P. Brisk, "Predictive modeling for CPU, GPU, and FPGA performance and power consumption: A survey," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2018, pp. 763–768.

[9] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 397–403.

[10] K. O'Neal, M. Liu, H. Tang, A. Kalantar, K. DeRenard, and P. Brisk, "HLSPredict: Cross platform performance prediction for FPGA high-level synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.

[11] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in HPC systems and applications," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 1–38, Oct. 2017.

[12] K. O'Neal, *Performance and Power Prediction of Compute Accelerators Using Machine Learning*. Riverside, CA, USA: UC Riverside, 2018.

[13] D. Holanda Noronha, R. Zhao, J. Goeders, W. Luk, and S. J. E. Wilton, "On-chip FPGA debug instrumentation for machine learning applications," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2019, pp. 110–115.

[14] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019.

[15] *The Intel FPGA SDK for OpenCL Best Practices Guide*, Intel, Santa Clara, CA, USA, May 2018.

[16] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-Based Computing Systems With OpenCL*. New York, NY, USA: Springer, 2018.

[17] B. Carrion Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020.

[18] T. Todman and W. Luk, "In-circuit assertions and exceptions for reconfigurable hardware design," in *Provably Correct Systems*. Cham, Switzerland: Springer, 2017, pp. 265–281.

[19] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis," *Int. J. Reconfigurable Comput.*, vol. 2011, pp. 1–17, Jan. 2011.

[20] *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*, Intel, Santa Clara, CA, USA, May 2018.

[21] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI C specification language," CEA-LIST, Saclay, France, Tech. Rep. v1.2, Oct. 2008.

[22] J. S. Monson and B. Hutchings, "Using source-to-source compilation to instrument circuits for debug with high level synthesis," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2015, pp. 48–55.

[23] J. P. Pinilla, *Source-Level Instrumentation for in-System Debug of High-Level Synthesis Designs for FPGA*. Vancouver, BC, Canada: Univ. British Columbia, 2016.

[24] J. Goeders and S. J. E. Wilton, "Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 127–134.

[25] H. Bensalem, Y. Blaquiere, and Y. Savaria, "Toward in-system monitoring of OpenCL-based designs on FPGA," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.

[26] A. Munshi, *The OpenCL Specification Version: 1.2 Document Revision: 15*. Beaverton, OR, USA: Khronos Group, 2011.

[27] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL Programming Guide*. London, U.K.: Pearson, 2011.

[28] *The OpenCL Specification: Version 1.0*, Khronos Group, Beaverton, OR, USA, 2010.

[29] *Design Debugging Using the SignalTap II Logic Analyzer*, Intel, Santa Clara, CA, USA, 2013.

[30] *ChipScope Pro Software and Cores: User Guide*, Xilinx, San Jose, CA, USA, Oct. 2012.

[31] *SDAccel Environment Debugging Guide*, Xilinx, San Jose, CA, USA, 2018.

[32] Y.-K. Choi and J. Cong, "HLScope: high-level performance debugging for FPGA designs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 125–128.

[33] E. Hung and S. J. E. Wilton, "Speculative debug insertion for FPGAs," in *Proc. 21st Int. Conf. Field Program. Log. Appl.*, Sep. 2011, pp. 524–531.

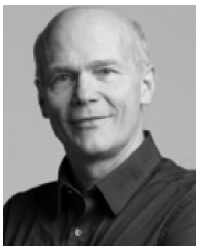
[34] T. Todman, S. Stilkerich, and W. Luk, "In-circuit temporal monitors for runtime verification of reconfigurable designs," in *Proc. 52nd Annu. Design Autom. Conf. - DAC*, 2015, pp. 1–6.

[35] P. K. Bussa, J. Goeders, and S. J. E. Wilton, "Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation techniques," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–4.

[36] N. Calagar, S. D. Brown, and J. H. Anderson, "Source-level debugging for FPGA high-level synthesis," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–8.

[37] J. Goeders and S. J. E. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–8.
[38] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.
[39] J. S. Monson and B. L. Hutchings, "Enhancing debug observability for HLS-based FPGA circuits through source-to-source compilation," *J. Parallel Distrib. Comput.*, vol. 117, pp. 148–160, Jul. 2018.
[40] J. P. Pinilla and S. J. E. Wilton, "Enhanced source-level instrumentation for FPGA in-system debug of high-level synthesis designs," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2016, pp. 109–116.
[41] F. Eslami, E. Hung, and S. J. E. Wilton, "Enabling effective FPGA debug using overlays: Opportunities and challenges," 2016, *arXiv:1606.06457*. [Online]. Available: http://arxiv.org/abs/1606.06457
[42] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-C. Feng, "Developing dynamic profiling and debugging support in OpenCL for FPGAs," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.
[43] J. Goeders and S. J. E. Wilton, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 1, pp. 83–96, Jan. 2017.
[44] CCIX Consortium. (2017). *Cache Coherent Interconnect for Accelerators*. [Online]. Available: http://www.ccixconsortium.com
[45] *AXI Reference Guide*, Xilinx, San Jose, CA, USA, 2012.
[46] *Avalon Interface Specifications*, Intel, Santa Clara, CA, USA, 2018.
[47] S. O. Settle, "High-performance dynamic programming on fpgas with OpenCL," in *IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2013, pp. 1–6.
[48] G. Ndu, J. Navaridas, and M. Luján, "CHO: Towards a benchmark suite for OpenCL FPGA accelerators," in *Proc. 3rd Int. Workshop OpenCL - IWOCL*, 2015, p. 10.
[49] *Intel FPGA SDK for OpenCL Support (Design Examples)*, Intel, Santa Clara, CA, USA, 2019.
[50] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.
[51] *SDAccel Environment Profiling and Optimization Guide*, Xilinx, San Jose, CA, USA, Mar. 2018.

HACHEM BENSALEM (Graduate Student Member, IEEE) received the B.Ing. degree in electrical engineering from the National Engineering School of Monastir, Tunisia, in 2012, and the M.Sc.A. degree in electrical engineering from the National Engineering School of Tunis, in 2015. He is currently pursuing the Ph.D. degree in electrical engineering with the Department of Electrical Engineering, École de Technologie Supérieure (ÉTS), Montreal, QC, Canada. He was a Research Associate with the Electronics and Microelectronics Laboratory, Monastir, Tunisia, in 2013 and 2014. He has also been a Teaching Assistant position with ÉTS since 2017. His general research interests include related to microelectronic circuits and microsystems, in particular, reconfigurable computing and high-performance applications, hardware design and acceleration on ASICs and FPGAs, debugging and instrumentation techniques on FPGAs, digital circuit design techniques, and CAD methods.



YVES BLAQUIÈRE (Member, IEEE) received the B.Ing., M.Sc.A., and Ph.D. degrees in electrical engineering from the École Polytechnique de Montreal, Canada, in 1984, 1986, and 1992, respectively.

From 1987 to 2016, he was a Professor of microelectronic engineering with the University of Quebec in Montreal (UQAM), Montreal, QC, Canada. He has been a Professor with the École de Technologie Supérieure, Montreal, Canada, since 2016. He currently works in the field of electrical/electronic/microelectronic engineering, specifically in ASIC/FPGA design, VLSI/WSI microsystems, high-speed digital circuits, timing tools, architectures, defect tolerance and applications in signal processing, network/high-speed processors, and embedded systems. He has done research and development projects in collaboration with several microelectronic companies, such as Gestion TechnoCap Inc., and DreamWafer Division, Hyperchip Inc. He performed WSI Research with Hyperchip Inc., from 1997 to 2004, including a two-year period where he contributed full time as the Technical Lead Researcher and the Manager of a team of ASIC/FPGA engineers, to deliver a core internet protocol petabit router. He is a member of the Regroupement Stratégique en Microélectronique du Québec and the Ordre des Ingénieurs du Québec (OIQ). He was the Director of the Laboratoire de Recherche de Conception en Microélectronique, UQAM, from 1992 to 1999 and from 2004 to 2008, the Director of the Microelectronic Engineering Program with UQAM from 2004 to 2010, and the Director of Engineering with UQAM from 2011 to 2015.

YVON SAVARIA (Fellow, IEEE) received the B.Ing. and M.Sc.A. degrees in electrical engineering from École Polytechnique Montreal in 1980 and 1982, respectively, and the Ph.D. degree in electrical engineering from McGill University, in 1985.

Since 1985, he has been with Polytechnique Montréal, where he is currently a Professor with the Department of Electrical Engineering. He is also affiliated with the Hangzhou Innovation Institute, Beihang University. He has carried out work in several areas related to microelectronic circuits and microsystems, such as testing, verification, validation, clocking methods, defect, and fault tolerance, effects of radiation on electronics, high-speed interconnects and circuit design techniques, CAD methods, reconfigurable computing and applications of microelectronics to telecommunications, aerospace, image processing, video processing, radar signal processing, and the acceleration of digital signal processing. He is also involved in several projects related to embedded systems in aircraft, radiation effects on electronics, asynchronous circuit design and testing, green IT, wireless sensor networks, virtual networks, software-defined networks, machine learning, computational efficiency, and application-specific architecture design. He holds 16 patents, has published 170 journal articles and 470 conference papers, and was the thesis advisor of 165 graduate students who completed their studies. He was the Program Co-Chairman of NEWCAS'2018. He has been working as a Consultant or was sponsored for carrying out research by Bombardier, CNRC, Design Workshop, DREO, Ericsson, Genesis, Gennum, Huawei, Hyperchip, ISR, Kaloom, LTRIM, Miranda, MiroTech, Nortel, Octasic, PMC-Sierra, Technocap, Thales, Tundra, and Wavelite. He is a member of the Regroupement Stratégique en Microélectronique du Québec (RESMIQ) and of the Ordre des Ingénieurs du Québec (OIQ). He is a member of the CMC Microsystems Board. In 2001, he was awarded a Tier 1 Canada Research Chair on the designs and architectures of advanced microelectronic systems that he held until June 2015.