POLITECNICO DI TORINO
Repository ISTITUZIONALE

Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis

Original
Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis / Muslim, Fahad Bin; Ma, Liang; Roozmeh, Mehdi; Lavagno, Luciano. - In: IEEE ACCESS. - ISSN 2169-3536. - ELETTRONICO. - 5:(2017), pp. 2747-2762. [10.1109/ACCESS.2017.2671881]

Availability:
This version is available at: 11583/2669854 since: 2018-04-05T21:06:12Z

Publisher:
IEEE

Published
DOI:10.1109/ACCESS.2017.2671881

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

17 April 2024

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 25, 2017, accepted February 6, 2017, date of publication February 20, 2017, date of current version March 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2671881

Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis

FAHAD BIN MUSLIM, (Student Member, IEEE), LIANG MA, (Student Member, IEEE), MEHDI ROOZMEH, AND LUCIANO LAVAGNO, (Senior Member, IEEE)
Politecnico di Torino, 10129 Turin, Italy

Corresponding author: F. B. Muslim (fahad.muslim@polito.it)

This work was supported in part by Xilinx Inc., and in part by the European Commission through the ECOSCALE Project under Grant H2020-ICT-671632.

ABSTRACT FPGA-based accelerators have recently evolved as strong competitors to the traditional GPU-based accelerators in modern high-performance computing systems. They offer both high computational capabilities and considerably lower energy consumption. High-level synthesis (HLS) can be used to overcome the main hurdle in the mainstream usage of the FPGA-based accelerators, i.e., the complexity of their design flow. HLS enables the designers to program an FPGA directly by using high-level languages, e.g., C, C++, SystemC, and OpenCL. This paper presents an HLS-based FPGA implementation of several algorithms from a variety of application domains. A performance comparison in terms of execution time, energy, and power consumption with some high-end GPUs is performed as well. The algorithms have been modeled in OpenCL for both GPU and FPGA implementation. We conclude that FPGAs are much more energy-efficient than GPUs in all the test cases that we considered. Moreover, FPGAs can sometimes be faster than GPUs by using an FPGA-specific OpenCL programming style and utilizing a variety of appropriate HLS directives.

INDEX TERMS High-level synthesis (HLS), FPGA, GPU, OpenCL, low-power low-energy computations, parallel computing.

I. INTRODUCTION

Modern electronic devices like smart phones are required to perform a variety of tasks ranging from simpler text messaging to more computationally intensive multimedia operations. This has resulted in the development of heterogeneous system architectures in modern system-on-chip (SoC) designs. Such systems mitigate the issues encountered by multi-core scaling (using several homogeneous cores), stemming mainly from the so called *memory wall* and *Von Neumann bottleneck* [1], [2]. Designs with such heterogeneous architectures essentially consist of a combination of multi-core processors and a variety of hardware accelerators to speed up the execution of computationally intensive tasks [3].

Graphical processing units (GPUs) offer higher floating point throughput, a favorable architecture for data parallelism and higher memory bandwidth than processors. These properties make them good candidates to be used as accelerators in modern high performance computing (HPC) systems [4]. The HPC systems using GPU-based accelerators however, are inefficient in terms of power consumption [5].

Modern field programmable gate array (FPGA) devices in comparison to GPUs can provide reasonable processing speed while consuming only a fraction of their operating power [6]. Moreover in comparison to multi-core CPUs, there is a continuously widening performance gap favoring FPGAs from one generation to the next, especially with regards to HPC or data center applications. The enhanced performance combined with a superior power efficiency results in increased performance-to-power-efficiency of FPGAs in comparison to both GPUs and CPUs [7]. These capabilities of FPGAs have been acknowledged by several big data companies such as Microsoft and Baidu, as is evident from their decision to use FPGAs rather than GPUs as accelerators in their data servers [8], [9].

One of the main hurdles in the utilization of FPGAs for acceleration, however, is the complexity of programming them. FPGAs are generally programmed using one of the hardware description languages (HDL) such as Verilog or VHDL used by hardware designers. This limitation however can be tackled by a technique called high-level

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

synthesis (HLS). HLS enables designers to program an FPGA using high-level languages e.g C, C++, SystemC or OpenCL. This in turn reduces both verification and design time in comparison to HDL design.

This work is devoted to a detailed analysis, by using a number of algorithms from a variety of application domains, of the design complexity, performance and energy-per-computation trade-offs that can be achieved by accelerating massively parallel applications on GPUs and FPGAs.

In this work, acceleration is done at a coarse-grained level. The algorithm is partitioned into functions which must be accelerated, called *kernels*, and a code that manages their input/output data and coordinates their execution, called *host code*. A kernel in this work is written in the Open Computing Language (OpenCL), and is implemented onto Xilinx FPGA devices by using the SDAccel™ tool chain from Xilinx. This includes both a compiler for host code and the Vivado HLS high-level synthesis tool. It also uses logic and physical design tools from the Vivado Design Suite [10], [11] for the kernels.

During the work, we noticed that optimizing OpenCL code for FPGA implementation by using SDAccel requires a comparable amount of effort to optimizing it for GPU implementation [12]. The main difference between optimizing for a GPU and for an FPGA lies in the different architectures of the targeted hardware platforms. In case of a GPU, the programmer tries to achieve the best mapping of an application onto the fixed hardware architecture. For an FPGA on the other hand, the task is to guide the compiler to generate optimized compute and memory architectures for each kernel in the application [12]. This implies that the usual problem of writing OpenCL code, namely the fact that it is very difficult to optimize it for different GPU architectures, is exacerbated even further when targeting it for FPGA implementation because the architecture must be adapted to the code and vice-versa. However, as we will argue in this paper, the advantages in terms of energy consumption per kernel execution, and sometimes in terms of performance, more than justify the additional effort.

OpenCL is a parallel programming language which is built upon C/C++ and hence can be ported very easily from C/C++ [13]. It exposes the architectural features of the GPU, namely the distinction between global memory implemented in external DRAM, local memory implemented in on-chip SRAM, and private register files, in an intuitive form, that is easy to exploit even by software programmers who are used to the ''flat'' memory models used by languages like C or Java.

OpenCL has been developed by the Khronos group to create applications to be executed on heterogeneous platforms. This portability of OpenCL gives it an edge over the very similar Compute Unified Device Architecture (CUDA) programming framework, which can only be used to program NVIDIA GPUs. Unfortunately, as mentioned above, OpenCL does not offer automatic performance portability across multiple devices. This implies that even though a specific OpenCL code can be executed on multiple devices, its performance would differ from one device to another especially due to the increasing architectural complexity of both GPUs and FPGAs. The Xilinx OpenCL high-level synthesis tool, namely Vivado HLS, has been used in this work to optimize the code for execution on the Xilinx FPGAs. It requires significant manual annotations of the code with both standard and Xilinx-specific OpenCL attributes in order to achieve a good level of performance.

Modern high-performance GPUs have very high-bandwidth DRAM interfaces, which typically outperform those of FPGAs implemented using comparable technologies. Moreover, external DRAM access consumes a significant amount of power. This means that the data organization for FPGA-bound implementations must be carefully optimized to use on-chip resources, even for kernel-to-kernel communication. For this reason, SDAccel offers an on-chip global memory implementation option that is quite advantageous for kernels that can stream data among them using very efficient on-chip buffers. In general, a performance advantage over GPUs can be achieved only for computations that require little overall external DRAM access. In any case, FPGA implementations require much less energy per computation than GPUs because the control structure is hardwired, and thus does not require to fetch and decode instructions. Moreover, the on-chip memory architecture can be tailored much more specifically to the application at hand, by using just a few high-level synthesis directives, thus leading to dramatically reduced multiplexing energy costs.

Note that, although the synthesis engine can process both OpenCL and C/C++ kernels, the coding requirements are slightly different for the two languages. On the one hand, kernels written in OpenCL do not require any additional annotations to be implemented on the FPGA, and attributes are used only to further optimize cost and performance. On the other hand, kernels written in C/C++ require HLS-specific pragmas to be used to define the HW protocol used for the memory interfaces for input/output function arguments, in addition to pragmas used for optimization. We used C/C++ for one of the algorithms, namely Monte Carlo simulation, in which multiple executions of the kernel code do not need to share data and which requires efficient implementation of trigonometric functions that are not yet available for OpenCL models.

A. MOTIVATION

This section of the paper describes the main motivation behind carrying out this research by estimating the annual savings in energy costs that we can obtain by utilizing FPGAs rather than GPUs as accelerators. We consider Algorithm 2 from the K-nearest neighbor (KNN) algorithm as an example to do this analysis. This implementation is explained later in section IV-B1.

The GPU in this case roughly consumes about 100W, while the FPGA implementation that we consider in this section consumes about 3W. We estimate that the DRAM (e.g. 16GB)

could bring the FPGA power consumption to about 10W. At about 10 cents per KWh, this means a saving of about 100$ per year in energy costs alone, ignoring cooling costs [14]. While at the current FPGA costs, this would not make them directly economical alternatives, things would change once they are produced in the same massive amounts as GPUs.

B. CONTRIBUTION

This research aims to investigate, with an extensive set of case studies drawn from a variety of application areas, the issues encountered when implementing and optimizing an application, originally written in non-hardware specific OpenCL or C/C++, on Xilinx FPGAs. It is also aimed at emphasizing the different compilation flows for code to be executed on a GPU and an FPGA, stemming from their different architectures. While GPUs due to their fixed architecture support just-in-time compilation, FPGAs have a much more flexible architecture. On one side they allow a much broader design space exploration, by exploiting various macro- and micro-architectural optimization options. On the other side, they require a much longer synthesis, place and route time than allowed by just-in-time compilation. The OpenCL standard thus allows, and the SDAccel tool exploits, offline compilation of the OpenCL code to be executed on an FPGA.

Finally, it is shown by comparing the performance of the algorithms on various platforms that the code executing efficiently on an FPGA is significantly different in several key aspects from the one leading to the best implementation on a GPU, even though the target platforms offer a similar memory architecture. This difference in performance is because of (1) their different DRAM memory access bandwidths, (2) the use of pipeline rather than SIMD parallelism, thus alleviating the thread divergence problem, as well as (3) the optimizations offered by the HLS tools that are available only for FPGA implementation.

II. RELATED WORK

This section summarizes past work on the use of FPGA-based accelerators in modern HPC systems, with a specific focus on our selected design test cases.

The authors of [13] describe the performance comparison of a complex computer vision algorithm for linear structure detection by considering a CPU, a GPU and an FPGA. They show that the Xilinx Spartan LX150 FPGA beats both the AMD Radeon HD6870 GPU and Intel Core i7 CPU in terms of power and speed. They have ported the code from CPU to GPU by using OpenCL and have used VHDL code to program the FPGA. The code translation effort has also been analyzed and obviously found to be higher for the RTL based FPGA design than for the GPU implementation through OpenCL. In our case however, we have used HLS for FPGA implementation directly from the OpenCL code executing on the GPU. This greatly reduces the design effort while enabling us to generate several hardware implementations from a single OpenCL code by providing various directives to the HLS tool.

A very thorough comparison of several hardware accelerators by considering multiple implementations of the quantum Monte Carlo application has been performed in [4]. A range of programming languages e.g. CUDA, Brook+, OpenCL, C++ and VHDL have been considered on a variety of implementation platforms e.g. Intel multi-core processors, multiple GPUs from NVIDIA and Radeon and a Virtex 4 LX160 FPGA from Xilinx. NVIDIA GPUs in combination with CUDA have been found to give the best performance in this study while the FPGA performance was the worst for a considerable number of computations. The main reason for this, as indicated by the authors is using an old FPGA against very powerful GPUs and CPUs. Here again, the authors mention the issue of programming FPGAs via VHDL, which took them almost a year.

Several techniques to accelerate the KNN algorithm were surveyed in [15], where the author advocated a parallel implementation due to its high computational costs and the opportunity to parallelize large portions of the algorithm. A brute force implementation of the KNN algorithm utilizing the CUDA language and the CUBLAS library was accelerated on a GPU-based hardware platform in [16] and [17]. The results were compared to an extremely optimized C++ library implementation and a huge speed up was observed.

An OpenCL implementation of the KNN algorithm on an FPGA-based heterogeneous system architecture was presented in [18]. The OpenCL compiler from Altera was used in that case for the FPGA implementation. Several different hardware platforms were used to perform experiments. They included an Intel Core i7-3770 processor, an AMD Radeon HD7950 graphics card and a Stratix IV 4SGX530 FPGA from Altera. The authors found their FPGA-based implementation to be more power/energy efficient as compared to both CPU- and GPU-based implementations. The GPU however managed to beat the FPGA in terms of execution time, most likely because of its comparatively higher global memory access bandwidth.

The option pricing problem is widely studied due to both its practical applicability in the financial market and the opportunity to solve it via stochastic Monte Carlo simulations. This method is computationally intensive and easy to parallelize. For example, FPGA-based acceleration was found to be 30X faster than the corresponding CPU-based implementation in [19]. A 4X reduction of the overall design effort by using a high-level language description was also demonstrated. The analysis of another pricing option (the European barrier option) was implemented on several different hardware platforms in [5] and the FPGA-based implementation was found to offer a better balance between performance and energy as compared to the corresponding GPU-based implementation. In [20], the study of the Black-Scholes model and Heston models was applied to European vanilla option. The authors found the FPGA-based accelerators to outperform the GPU-based ones in terms of both power consumption and execution time.

Two different hardware implementations of the bitonic sorting network were presented in [21]. The high performance design in that case utilized a single memory port using a streaming permutation network (SPA), thus resulting in a memory and energy optimized implementation on a Xilinx Virtex-7 platform. A significant performance improvement was also achieved in [22] by proper pipelining of different stages of the sorting network. In [23], the Bitonic sort algorithm was compiled for a GPU-based hardware platform by using CUDA, where optimizations were done mainly to reduce the number of global memory accesses and the number of kernel launches.

FPGAs are hence considered as a viable option as an accelerator instead of GPUs especially when energy-per-operation is the main concern. Researchers at Baidu are thus considering FPGAs for accelerating their deep learning models for image search [8]. Microsoft's Bing search engine also uses Altera FPGAs as accelerators in combination with traditional microprocessors from Intel [24]. Foreseeing the possibility of a greater usage of FPGAs in data centers in the near future, Christoforos Kachris states in a recent issue of EETimes that "*In a sign of the times, four out of the five keynote presentations at FPL 2016, a major FPGA conference in Europe, were given by large companies such as IBM, Intel and Microsoft focused on the efficient deployment and use of FPGAs in data centers.*" [25]. Keeping in view this market trend and the general perception of complexity of FPGA programming, two of the major FPGA manufacturers Altera and Xilinx have recently introduced tools to enable the designers to program their respective FPGAs directly using C, C++, SystemC and OpenCL code [10], [26]. There is hence a considerable interest on this topic in the design community. This provided us with a motivation to perform an extensive study in this regard.

III. METHODOLOGY

A. FPGA IMPLEMENTATION

This section of the paper gives a brief overview of the OpenCL programming framework, including its platform and memory model, followed by a detailed description of the design flow starting from the OpenCL code and terminating with final FPGA implementation.

1) OVERVIEW OF OpenCL

As stated before, OpenCL is a parallel programming language, supported by a compilation framework and libraries, for programming multi-core and heterogeneous compute platforms [27], [28]. OpenCL offers functional portability thus enabling code execution on various supported devices, requiring only minimal modifications to the host code. The programming language is based on C99 and supports both data-parallel and task-parallel programming models [29].

The OpenCL platform model includes both a (possibly multi-core) CPU, called *host*, and massively parallel accelerators, called *devices*, that can take the form of GPUs and FPGAs. The host is responsible for setting up the environment to enable the OpenCL *kernels* to execute on one or more devices. In terms of OpenCL, a *device* represents any supported hardware platform that can be used to accelerate the compute intensive portions of an application i.e. the kernels. An OpenCL device consists of compute units (CU), each further divided into processing elements (PE) as shown in Fig. 1.

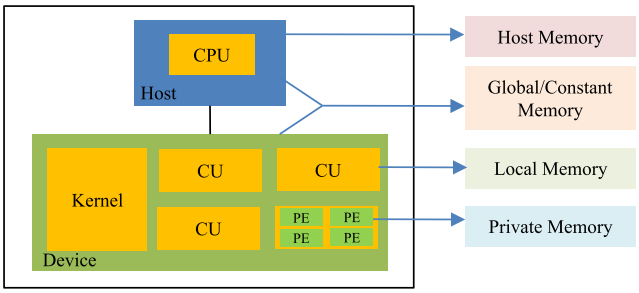

FIGURE 1. OpenCL platform and memory model.

Several concurrent executions of the kernel body (called *work-items*) take place on multiple processing elements. The work-items are further grouped into *work-groups*, which are being executed by compute units. The memory is broadly divided into host (i.e. CPU) memory and device (i.e. GPU or FPGA) memory. The device memory is further divided into private memory (specific to each work-item), local memory (shared by all the work-items in a work-group) and a global/constant memory (shared by all the work-groups). Access to global memory is the slowest (since it typically resides in external DRAM) while private memory is the fastest (since it is typically allocated to register files), while local memory often resides in on-chip SRAM. Global memory however, is the largest in size while private memory is the smallest. The OpenCL memory model is also shown in Fig. 1.

The work-items that compose an OpenCL kernel can be executed in an out-of order manner, in order to ensure high performance on a variety of platforms with different numbers of CUs [30]. Thus, the OpenCL standard uses a three-level synchronization and collaboration model. The execution order of different kernels is completely determined by the host code, either by calling them sequentially, or by using synchronization callbacks that notify the host code when a given kernel has completed execution. The execution of different work-groups within a kernel is completely unsynchronized, thus they must read and write different areas of global memory, and they cannot cooperate in any manner. Finally, the programmer can use explicit *barriers* to ensure local and global memory consistency for work-items within a work-group. A barrier represents a checkpoint within a work-group. All the work-items belonging to that work-group must reach it before any of them can proceed beyond it [10].

2) FPGA DESIGN FLOW

Fig. 2 shows the complete flow of operations performed by the SDAccel tool from Xilinx to take OpenCL code and

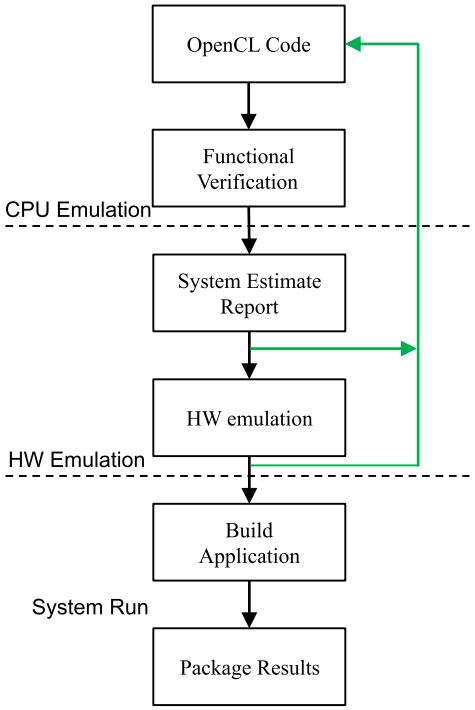

FIGURE 2. SDAccel based FPGA design methodology flow.

implement it on an FPGA. The very first step in the flow is to take the OpenCL code and verify its functional correctness by performing a software (SW) based simulation called CPU emulation. This is the fastest part of the design flow and enables a fast verification of the design functionality. Both the host and kernel code in CPU emulation are run on the x86 based processor [10].

This step typically requires one to add a testbench to the host code, which provides the inputs to and checks the outputs from the algorithm. The testbench generates or reads from files the stimuli to drive the inputs of the design under test (DUT), and monitors its outputs, thereby validating the functionality of the DUT. Once the functionality is verified, the performance of each individual kernel i.e. the performance of an individual compute unit and its resource usage are estimated. This gives an early idea of the final performance gains, by taking into account the target hardware device as well as the generated compute units for the execution of the application.

SDAccel generates hardware from an OpenCL kernel at the level of work-group. Multiple compute units can be instantiated and driven by the host code, in order to improve performance. Each work group can execute its work items either in pipelined fashion, or by unrolling them, or both. Typically pipelining provides the best cost/performance trade-off, but sometimes either the loops over the work items, or some of the loops nested within them, can be further unrolled to improve performance.

Then RTL simulation, also called hardware (HW) emulation in the context of this flow, is used to verify that the compute units created for all the kernels are functioning correctly and to analyze their overall performance. While CPU emulation takes care of the functional correctness of the application, its performance is verified by HW emulation. The logic implementation for each compute unit needs to be generated by SDAccel before HW emulation and hence this step takes more time as compared to the CPU emulation. It runs Vivado HLS under the hood to generate the custom logic for the application, thereby attempting to maximize performance while minimizing resources at the same time. Vivado™ Integrated Design Environment (IDE) is used thereafter in the *build system* step to connect the generated custom units to the infrastructure IPs provided by the target hardware, such as the processor interface used to pass arguments to the kernel, to start it, and to wait for its completion, as well as the DDR DRAM interface [10]. The final step is to package the generated system for deployment on FPGA-based boards.

3) FPGA-SPECIFIC OpenCL OPTIMIZATIONS

During the FPGA implementation flow, the parallelism offered by the FPGA can be exploited by using several HLS optimization directives offered by SDAccel. The (*reqd_work_group_size*) attribute defined in the OpenCL standard is strongly recommended in case of an FPGA-based implementation. This attribute is meant to specify the size of the problem space corresponding to a single execution of the kernel compute unit. This attribute specifies the work-item loop iteration count which in turn can be used by the HLS tool to optimize performance during the generation of custom logic for the kernel. Moreover, the memory access throughput can be improved by utilizing vector data types (when required) instead of the C structs [10].

The SDAccel-based flow can pipeline both the work-item loops as well as any explicit loops in the kernel. Pipelining is better than loop unrolling for loops accessing global memory, as in our case, because of their ability of better matching the limited number of global memory ports available. The limited number of global memory ports may result in data access conflicts, thereby limiting the performance gains obtained by loop unrolling, and serializing potentially parallel loop iterations [31]. SDAccel supports unrolling both explicit loops and implicit work-item loops in case of OpenCL.

As mentioned before, the performance efficiency on FPGAs is reduced in case of applications dominated by DRAM accesses. This is because of the significantly higher global memory access bandwidth of modern GPUs in comparison to that of current FPGAs. On the other hand, the internal bandwidth to move the operands and results of application-specific ALUs is in the order of terabytes/sec (TB/s), which makes FPGAs a suitable platform for high performance computing. SDAccel offers several options in order to minimize the number of off-chip memory accesses and implement more efficient interfaces to DDR in order to exploit the inherent parallelism offered by FPGAs. Some of these are described here.

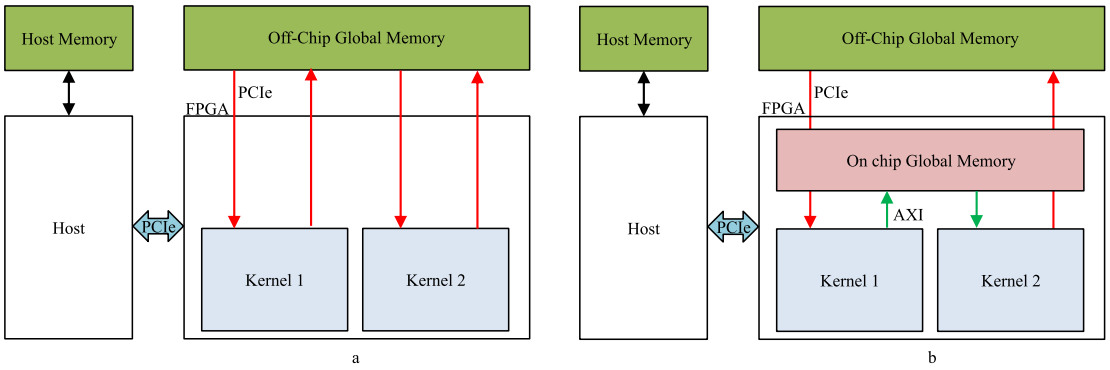

FIGURE 3. (a) Traditional global memory buffer vs (b) On-chip global memory buffer.

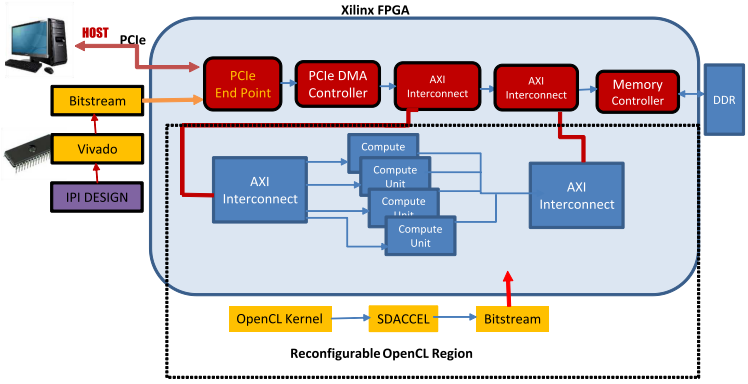

FIGURE 4. Illustration of OpenCL region with multiple compute units.

- Performing global to local memory copy and vice-versa in bursts. Large bursts improve efficiency, as the memory access overhead is shared across large amounts of data being transferred [12].
- Using a dedicated memory port for each global array, thus leading to reduced memory access conflicts.
- Using "on-chip global memory" for inter-kernel communication and avoiding excessive transfers to DDR, as shown in Fig. 3 and used in section V-A. This option automatically maps the global memory buffers used merely for inter-kernel communication, to the on-chip block RAMs. It should be noted that otherwise the global memory buffers are mapped to external slower DRAMs.
- Using on-chip pipes (FIFOs) that allow data streaming between two kernels.

SDAccel creates a customized area on the FPGA called OpenCL region (OCL region) which allows exploiting the OpenCL parallel model by implementing multiple compute units of the same kernel with multiple work groups, as shown in Fig. 4. SDAccel enables FPGA designers to take advantage of multiple work groups by instantiating them separately on the FPGA fabric and executing them in parallel. This improves overall performance by increasing bandwidth utilization and enhances parallelism on a coarse-grained level.

B. POWER ANALYSIS

The power consumption of an FPGA is estimated by using the power analysis features of Vivado™. Vivado allows power analysis through all the stages of FPGA design starting from logic synthesis to placement and routing. The power estimation in this work is performed post routing, which is the most accurate as it is based on the exact logic and routing resources read from the already implemented design database [32]. The complete power analysis flow utilizing Vivado power analysis features is shown in Fig. 5. This work uses the vector-based approach to estimate the power in case of an FPGA, where the switching activities are captured using a Switching Activity Interchange format (SAIF) file which then is used to obtain more accurate power reports.

GPU power estimation on the other hand was performed by using the NVIDIA System Management Interface (nvidia-smi). This is a command line utility that exploits the NVIDIA Management Library (NVML) to profile NVIDIA GPUs [33].

IV. EXPERIMENTAL SETUP AND DESIGN TEST CASES

Several different algorithms are considered for implementation on heterogeneous platforms consisting of a multi-core processor and different accelerators (e.g. FPGAs or GPUs) as depicted in Fig. 6.

TABLE 1. Target platforms comparison.

Device	Global Memory Size	Bandwidth (GB/s)	Bus Interface	min t_{clk}	Datasheet Power (W)	Idle Power (W)
GTX960	2GB GDDR5	112.0	PCIe 3.0 x16	0.85ns	120	8
K4200	4GB GDDR5	172.8	PCIe 2.0 x16	1.27ns	108	13
FPGA	Two 8GB SODIMMs	21.3	PCIe 3.0 x8	*	-	-

* See the other tables for clock frequency and the actual power consumption of each test case.

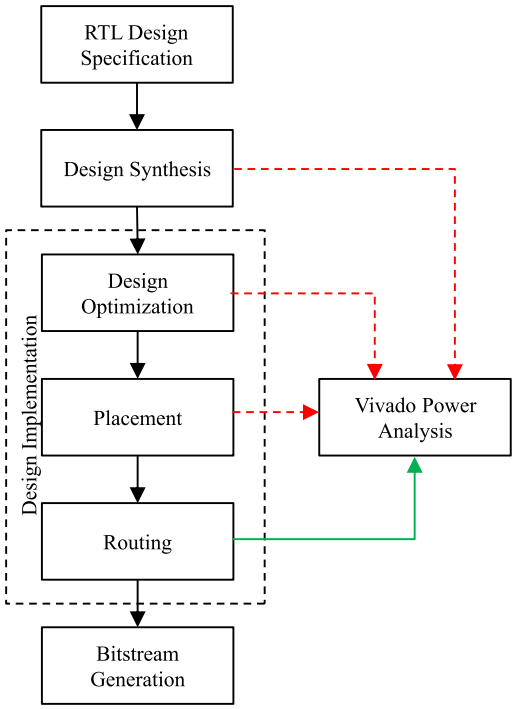

FIGURE 5. Power estimation and analysis flow.

A. EXPERIMENTAL SETUP

The experimental setup considers three target devices to run our applications. Two of them are GPUs while one of them is an FPGA.

1) GPUs

A GPU consists of several parallel processing elements as shown in Fig. 6. These are responsible for executing the kernel functionality concurrently in the form of work-items. The most important properties affecting the performance of a GPU are the number of cores, the clock frequency and the memory access bandwidth. All the kernels targeting GPUs in this work are written in OpenCL, while the host code uses the OpenCL C/C++ API developed by the Khronos group. The two GPUs used in this research work are the GTX960 and Quadro K4200, both by NVIDIA. These are listed in Table 1. The minimum clock period t_{clk} for the two GPUs is given in the table as well.

2) FPGA

For FPGAs to be used as accelerators, their interface protocol needs to be given special consideration. The design flow in our case, as mentioned before, uses HLS to directly program the FPGA from high-level code. Fortunately, the SDAccel and Vivado HLS tools take care of the SW and HW side of the interface protocol respectively. The FPGA board considered in this work is an Alpha data ADM-PCIE-7V3 with a Virtex-7 690t FPGA, whose specifications are also listed in Table 1. The clock period for the FPGA, and hence its power consumption, depends on the design. In all the test cases that we considered, we instantiated the maximum number of work-groups that can fit on the FPGA while keeping the design routable.

B. DESIGN TEST CASES

A number of commonly used algorithms from a variety of application domains are considered for acceleration on the target platforms. In order to compare the performance of the OpenCL compiler with that of the CUDA compiler for the GPUs, we also converted one of our kernels, namely the K-Nearest Neighbor algorithm, to CUDA. We did not observe any significant difference in performance on the GPU platforms. Thus we did all the remaining design experiments using OpenCL, since that language is more portable across platforms. The various test cases are described here briefly.

1) K-NEAREST NEIGHBOR ALGORITHM

K-Nearest Neighbor (KNN) is an important classification algorithm used in a variety of applications such as pattern recognition, computer vision and machine learning. It is used to find the k nearest neighbors of a given query point among a set of reference data points. Typically, training datasets are quite large, hence resulting in a large computation cost for this algorithm [16]. Fortunately, the algorithm is highly parallelizable and hence can be accelerated considerably by exploiting the parallel architectures of GPUs or FPGAs. The algorithm includes the following steps:

1) For given number n of points in the reference data set R and a given query point q, find the n distances between the query point and each point in the reference data set. The distance in our case is the squared Euclidean distance.
2) Sort the n distances while preserving the corresponding indices of the points in the reference data set R.
3) Return the k points in the reference data set R corresponding to the smallest k distances obtained after the sorting step.

The baseline code in this work is based on the parallel implementations from [34], [35] which contain highly optimized code for execution on GPU-based platforms. Two different implementations of the algorithm are considered

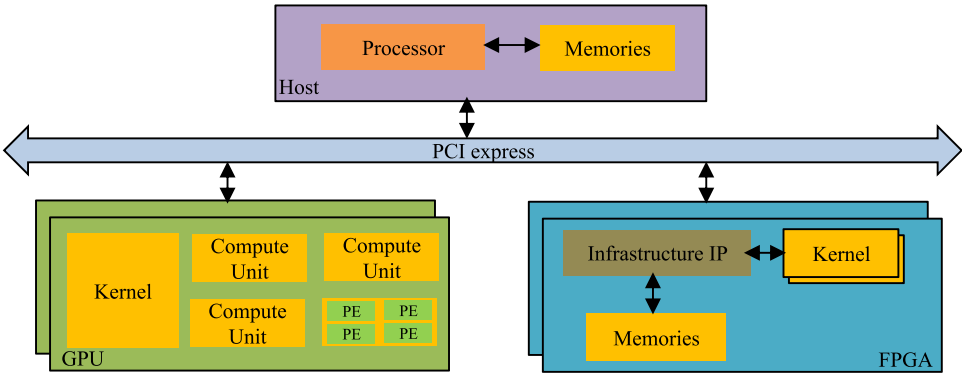

FIGURE 6. A typical heterogeneous system architecture.

and their execution speed, energy and power are compared on GPUs and an FPGA. The implementations mainly differ based on whether the neighbor's estimation is done on the CPU or on the accelerator. In the case where it is done on the CPU, the CPU execution time is also measured while calculating the execution time of the algorithm. The two implementations are presented here.

- Algorithm 1
  This implementation performs a parallel execution of the distance calculation task of the algorithm on the device (GPU/FPGA), while the nearest neighbors estimation takes place on the host. The distance calculation task is the most easily parallelizable part of the algorithm, as separate independent points are read from the reference data set for calculating distances. The implementation uses global memory only, hence performance mainly depends on the accelerator global memory bandwidth. It is illustrated in Algorithm 1.
- Algorithm 2
  This implementation performs both the distance calculation as well as the nearest neighbors estimation task on the device in two separate kernels namely "DISTANCE CALCULATION" and "NEIGHBOR ESTIMATION" respectively. It uses an optimization option offered by SDAccel called "On-chip global memories" to map the global memory buffers used merely for inter-kernel communication, to the on-chip block RAMs. This optimization is depicted in Fig. 3. The pseudo-code for this implementation is given in Algorithm 2.
  It is efficient when k is much smaller than the number of points in R, because it can use a very simple and fast streaming algorithm to find the k smallest elements in a large vector of distances.

Algorithm 1 Distance Calculation on Device and Neighbor Estimation on Host

Input: A query point q and R, a set of reference points;
Output: Indices of the k reference points with the smallest distance from q;

```
Begin
On device:
function DISTANCE CALCULATION
for each reference point r ∈ R do
    compute the distances between q and all points r ∈ R;
end
end function
On host:
function NEIGHBOR ESTIMATION
sort the distance vector;
print the indices in R of the k smallest elements of the sorted distance vector;
end function
End
```

2) MONTE CARLO METHODS FOR FINANCIAL MODELS

The Monte Carlo method (MC) is widely used to solve complicated mathematical and physical problems such as stochastic differential equations or multiple-particle models. Typically, it involves plenty of random number generations and intensive parallel computations. These two features of the MC method make it suitable for acceleration by GPUs or FPGAs.

In this paper, the MC method was applied to two famous financial pricing models, namely the Black-Scholes model and the Heston model. Both models consider a risk-free asset with a fixed interest rate and a risky asset (also known as a stock) with a price that is subjected to geometric Brownian motion as shown in (1). In the Black-Scholes model, the volatility of the stock price is a constant value, while in the Heston model, the volatility is also modeled by a stochastic differential equation as shown in (2).

$$dS = rSdt + \sqrt{V}Sdz_1 \tag{1}$$

$$dV = \kappa(\theta - V)dt + \sigma_v\sqrt{V}d(\rho z_1 + \sqrt{1-\rho^2}z_2) \tag{2}$$

In (1), (2), z_1, z_2 are two Wiener processes, ρ is the correlation factor between them and $\sqrt{V}$ is the volatility of the stock price. In (2), θ is the long-run mean variance, κ is the speed

Algorithm 2 KNN on Device Using Multiple Kernels

Input: A query point q and R, a set of N reference points;
Output: k smallest distances with their indices in a single work-group;

```
Begin
On device:
declare global distance array "dist" for inter-kernel communication;
function KERNEL1: DISTANCE CALCULATION
for each reference point r ∈ R do
    compute all distances between q and all points r ∈ R and save in "dist";
end
end function
function KERNEL2: NEIGHBOR ESTIMATION
declare a local variable "dmin1";
for i = 0 to k − 1 do
    declare and initialize variables "dmin" and "location";
    if (i == 0);
    then
        initialize "dmin1" to 0;
    end
    for j = 0 to N − 1 do
        if (dist[j] < dmin and dist[j] > dmin1);
        then
            dmin = dist[j];
            location = j;
        end
        if (j == N − 1) ;
        then
            dmin1 = dmin;
            print index in R corresponding to the neighbor i.e. "location";
            print distance corresponding to the location i.e. "dmin";
        end
    end
end
end function
End
```

of mean reversion (the rate at which V reverts to θ) and σ_v is the standard deviation of the volatility V.

For a short time $\Delta t << 1$, V can be assumed constant. Thus the numerical solution for (1) is achieved by applying Itô's Lemma, as shown in (3), which demonstrates the relation between the stock price at time $(t + \Delta t)$ and t. (4) is the numerical solution for (2). It is obtained by applying Euler discretization [36] with full truncation scheme [37] avoiding negative values.

$$S_{t+\Delta t} = S_t e^{(r-\frac{1}{2}V_t^+)\Delta t+\epsilon_1\sqrt{V_t^+}\sqrt{\Delta t}} \tag{3}$$

$$V_{t+\Delta t} = V_t^+ + \kappa(\theta - V_t^+)\Delta t + \epsilon_0\sigma_v\sqrt{V_t^+}\sqrt{\Delta t} \tag{4}$$

$$\epsilon_0 = (\rho\epsilon_1 + \sqrt{1-\rho^2}\epsilon_2) \tag{5}$$

where $\epsilon_1, \epsilon_2 \sim N(0, 1)$ and $V_t^+ = max(V_t, 0)$.

Traders in the derivative market calculate the payoff price according to their contracts and the behavior of the risky assets at the end of a preset time. The mechanisms that traders use to estimate the payoff prices are called options. The options implemented for solving the pricing problem in this article include vanilla options (such as the European Vanilla option) and exotic options (such as the Asian option and the Barrier option).

In order to apply the numerical method, the preset time of an option has been uniformly partitioned into M steps. The stock price (and stock volatility) at each time step is estimated by (3) (and (4)). The procedure that calculates all the prices along the time partitions is called a "path simulation". Thousands of path simulations are performed in order to estimate the probabilistic distribution of the payoff price of the given assets and options. In the following, the total number of the path simulations is denoted by N.

The total computation time T_s for sequential execution is proportional to N and M, as shown in (6).

$$T_s = t_c \cdot N \cdot M \tag{6}$$

where t_c is the average execution time to finish the computations of one iteration and can be used as an overall measure of performance. A large amount of high-quality independent random numbers according to the standard normal distribution are generated by the Mersenne-Twister (MT) algorithm and the Box-Muller (BM) transformation, as in the previous studies.

- Algorithm 3
  This algorithm represents the kernel code for both the GPU- and FPGA-based accelerators. [1] Optimizations have been applied in order to accelerate the algorithms on a GPU- or on a FPGA-based hardware. For instance, the outer-most loop in this algorithm could be partially unrolled due to the independence of each path simulation. The unroll factor is denoted by N_u and the host-kernel architecture is shown in Fig. 7. As far as the path simulator is concerned, Fig. 8 presents the detailed dataflow for the Heston model applied to the European barrier options. As can be seen in Fig. 8, the stock price and volatility at each time partition are evaluated from their initial values. Each evaluation involves two independent random numbers and a barrier-checking phase.
  On FPGA, the kernel can be further accelerated by pipelining the inner-most loop as shown in Fig. 9. Since the dependency between V_{m+1}^n and V_m^n and the dependency between S_{m+1}^n and S_m^n, the initiation interval (II) depends the longest latency of these two computations.

[1] As mentioned above, the algorithm was modeled in c++ for FPGA implementation in order to circumvent some temporary inefficiencies of the OpenCL floating point libraries for trigonometric functions in SDAccel.

Algorithm 3 Payoff Price Estimation, Heston Model

Input: Parameters of a stock and an option such as initial stock price, strike price, etc.
Output: Payoff price P_{payoff};

Begin
On device:
function PAYOFF CALCULATION
initialization
for n^{th} *path* $\in N$ *independent paths* **do**
  for $m^{th} \in M$ *partitions along time* **do**
    generate $\epsilon^1_{t_m}$ and $\epsilon^2_{t_m}$ by MT algorithm and BM transformation;
    compute S_{t_m} by (3);
    compute V_{t_m} by (4);
    S = option(S, S_{t_m}); //function depends on the kind of the option implemented;
  end
  $P_n = p(P_n, P_{strike})$; //payoff price for n^{th} path
end
$P_{payoff} = mean(P_n)$; //average payoff price over N paths
end function
End

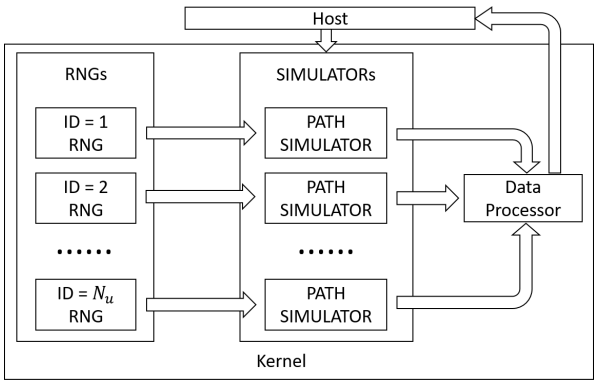

FIGURE 7. Diagram for host-kernel data flow.

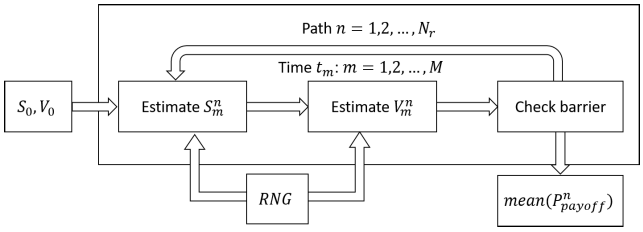

FIGURE 8. Diagram for path simulator.

By swapping the order of the nested loops, the dependency is removed, due to the independence of each path, and better performance can be achieved, as shown in Fig. 10 while more memories are required to store the temporary data. The optimization for the random number generation is done by partitioning the state arrays

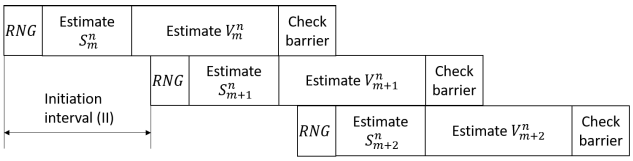

FIGURE 9. Pipeline of the inner-most loop of kernel.

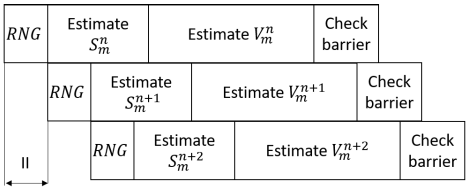

FIGURE 10. Optimized pipelining with small initiation interval.

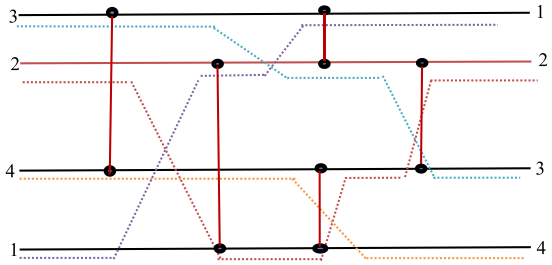

FIGURE 11. Illustration of a simple sorting network.

in order to increase the parallelization of the process as discussed in [20]. We eventually achieved an initiation interval of 2 for the flattened loop. So the time t_c can be roughly estimated by (7) and (8) for the two models respectively [20].

$$t_c^{B.Scholes} = \frac{t_{clk}}{N_u} \tag{7}$$

$$t_c^{Heston} = \frac{2t_{clk}}{N_u} \tag{8}$$

where t_{clk} is the clock period of the FPGA. Then the energy consumption for each time step could be estimated by $E_c = P_d t_c$, where P_d is the power of the given device.

3) BITONIC SORTING ALGORITHM

Sorting is a fundamental operation that is widely used in the field of computer science, including high performance data center applications. Among many sorting algorithms that have been devised, bitonic sorting is one of the fastest known sorting networks. In general, the term "sorting network" identifies a sorting algorithm where the sequence of comparisons is not data-dependent, thus making it suitable for parallel hardware implementation. A simple example of sorting network is depicted in Fig. 11, with five comparators and four inputs. The comparators in a layer can work concurrently.

The depth and number of comparators are key parameters to evaluate the performance of a sorting network. The depth

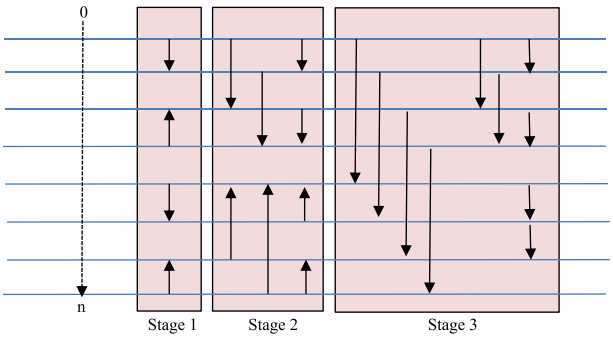

FIGURE 12. Bitonic sort network with eight inputs (N=8). It operates in 3 stages, it has a depth of 6 (steps) and employs 24 comparators.

of a sorting network is the maximum number of comparators along any path. If all the comparisons in each layer are done in parallel, the depth of the network is proportional to the total execution time. The bitonic sort network shown in Fig. 12, is one of the fastest comparison sorting networks, where the depth and number of comparators are mathematically represented by (9) and (10) respectively.

$$D(N) = \frac{log_2 N.(log_2 N + 1)}{2} \tag{9}$$

$$C(N) = \frac{N.log_2 N.(log_2 N + 1)}{4} \tag{10}$$

Bitonic sorting is a recursive divide-and-conquer algorithm that is based on the notion of bitonic sequence, i.e. a sequence of N elements in which the first K elements are sorted in ascending order, and the last $(N - K)$ elements are sorted in descending order (i.e. the $K - th$ element acts as a divider between two sub-lists, each sorted in a different direction), or some circular shift of such an order.

Bitonic sorting first divides the input into pairs of keys and sorts them into a set of bitonic sequences. It then repeatedly merges and sorts pairs of adjacent bitonic sequences, until the entire sequence is sorted [23].

- Algorithm 4 & 5:
  Algorithm 4, executed on the host i.e. a CPU, iterates the execution of three kernels described in Algorithm 5 to complete bitonic sorting in three phases. In the first phase, the algorithm partially sorts an arbitrary input array to obtain a set of bitonic sequences. In the second and third stages respectively, the algorithm merges bitonic sequences repeatedly until a fully sorted array is produced. All the comparisons (i.e. all the work items) in each kernel execution can be performed in parallel and independent from each other. This makes the whole algorithm suitable for parallel implementation.

Algorithm 4 Host Code for Bitonic Sorting Execution

Input: A vector of N keys to be sorted;
Output: A sorted vector;
Begin
On host:
SORTLOCAL(Input, Output);
for $size = 4 * Work_Group_Size$ *to* N **do**
  multiply *size* by 2;
  for $stride = \frac{size}{2}$ *to* $stride > 0$ **do**
    divide *stride* by 2;
    if $stride >= 2 * Work_Group_Size$ **then**
      MERGE LOCAL(Input, Output, size, stride);
    end
    else
    MERGE GLOBAL(Input, Output, size, stride);
    end
  end
end
End

V. RESULTS

A. K-NEAREST NEIGHBOR ALGORITHM

The experiments with the KNN algorithm are based on the data set from [38]. The data set contains data from ''Unisys corporation'' consisting of the floating point locations (latitudes and longitudes) of a series of hurricanes. It is used by the KNN algorithm to find the k nearest hurricanes to a specified query point. The value of k is usually very small in comparison to the number of points n in the reference data set. It has been set to 5 in all our experiments. The number of points in the reference data set is about 0.3 million.

The FPGA implementation of the KNN algorithm has been optimized by using several optimizations offered by SDAccel and described in section III-A3. The (*reqd_work_group_size*) attribute has been used in both implementations to specify the number of work-items in a work-group. 2-element vector data types were used in both cases (instead of C structs) to read the 2-dimensional data points, thereby improving the memory access throughput. Another optimization common to both implementations is the use of burst transfers to read data from the off-chip global memory. Moreover, loop pipelining has been used both for the explicit as well as work-item loops to improve throughput.

The GPU versus FPGA results in terms of execution time, energy and power consumption for Algorithm 1 are presented in Table 2. The resource utilization for the FPGA is also presented. This implementation uses the accelerators only for distance calculation between the query point and all the points of the reference data set. The nearest neighbor identification on the other hand is done on the CPU, hence the CPU time is added to the table as a part of the total KNN execution time.

As evident from Table 2, both GPUs in Algorithm 1 perform faster than the FPGA due to their comparatively higher DRAM access bandwidth. The FPGA however consumes considerably smaller energy and power in comparison. As mentioned earlier, power analysis in case of FPGA is done by using simulation vectors in Vivado, while in case of the GPU, it is based on the results obtained by utilizing the NVIDIA system management interface utility.

Algorithm 5 Bitonic Sorting Kernels

```
On device:
Begin
function KERNEL1: SORT LOCAL(Input, Output)
for local_id = 0 to Work_Group_Size − 1 do
    copy a block of data from global to local memory
    with the size of work_group;
    for size = 2 to size < Work_Group_Size do
        multiply size by 2;
        for stride = size/2 to stride > 0 do
            divide stride by 2;
            perform comparison on each pair and swap
            them if they are not sorted;
        end
    end
    for stride = Work_Group_Size to stride > 0 do
        divide stride by 2;
        pos = 2 ∗ local_id − (local_id&(stride − 1));
        compare and sort each pair;
    end
    write back sorted array to global memory with the
    size of work_group;
end
function KERNEL2: MERGE LOCAL(Input, Output,
size, stride)
for local_id = 0 to Work_Group_Size − 1 do
    read one pair in each work item;
    perform comparison on each pair and swap them if
    they are not sorted;
    write back sorted pair to the global memory;
end
function KERNEL3: MERGE GLOBAL(Input, Output,
size, stride)
declare and initialize a private variable global_stride;
for local_id = 0 to Work_Group_Size − 1 do
    copy a block of data from global to local memory
    with the size of work_group;
    for stride = global_stride to stride > 0 do
        divide stride by 2;
        perform comparison on each pair and swap them
        if they are not sorted;
    end
end
End
```

The performance comparison for Algorithm 2 is shown in Table 3. This implementation uses the ''on-chip global memories'' optimization option offered by SDAccel to map the global memory buffers used for communication between multiple kernels to the block RAMs and to stream communication between them if the kernel code structure allows it. A global memory buffer called ''dist'', as shown in Algorithm 2, has been used for inter-kernel communication. This is an automatic optimization offered by SDAccel, if it detects a global memory buffer which does not need to be exposed to the host.

TABLE 2. Performance and energy analysis of KNN Algorithm 1.

Params/Devices	FPGA	GTX960	K4200
Device time	1.24ms ($t_{clk} = 4.17ns$)	0.04ms	0.05ms
CPU time	3.0ms	3.0ms	3.0ms
Total time	4.24ms	3.04ms	3.05ms
Device power	0.346W	30W	40W
Energy	0.43mJ	1.2mJ	2mJ
Utilization	BRAMs = 0	NA	
	DSPs = 12 (0.33%)		
	FFs = 3109 (0.36%)		
	LUTs = 2006 (0.46%)		

TABLE 3. Performance and energy analysis of KNN Algorithm 2.

Params/Devices	FPGA	GTX960	K4200
Device time	1.23ms ($t_{clk} = 4.17ns$)	0.93s	3.11s
Device power	2.56W	90W	60W
Energy	0.003J	84J	187J
Utilization	BRAMs = 512 (34.83%)	NA	
	DSPs = 12 (0.33%)		
	FFs = 23892 (2.78%)		
	LUTs = 11838 (2.76%)		

The FPGA implementation in this case is considerably faster than both GPUs. The main reason is the fact that the two kernels must use slow DRAM on the GPU to communicate, while on the FPGA they can use the faster on-chip streaming SRAM. This of course also reduces dramatically power and energy consumption with respect to the GPUs. The FPGA-based implementation of NEIGHBOR ESTIMATION is also faster than the GPU-based one because pipeline-based parallelism can efficiently handle conditionals. On the other hand, conditionals that cannot be converted to predicated form cause the so-called ''thread divergence'' problem on GPUs. This is because on a Single Instruction Multiple Data processor those work-items for which a condition is false must stall while those where the condition is true are executed, and vice-versa.

B. MONTE CARLO METHODS FOR FINANCIAL MODELS

This section presents the results of implementing two financial models applied to the European vanilla option, the European Barrier option and the Asian option, and then comparing them across platforms in terms of performance and energy consumption. Single precision floating point variables are used as the basic data type in these experiments in order to guarantee a fair comparison between the two platforms. In the future, fixed-point variables may also be used to further accelerate the algorithms on the FPGA.

1) BLACK-SCHOLES MODEL

I European vanilla option

For the Black-Scholes model, the European vanilla option is simple enough not to require to partition the preset time. So a large amount of paths can be simulated in a reasonable time. t_c and E_c are demonstrated in

TABLE 4. Performance and energy analysis of black scholes european vanilla option.

Params/Devices	FPGA	GTX960	K4200
Device time t_c	0.0788ns ($t_{clk} = 5ns$)	0.164ns	0.203ns
Device power P_d	21.2W	98W	105W
Energy E_c	1.67nJ	14.76nJ	18.68nJ
Utilization	BRAMs = 388 (26%)	NA	
	DSPs = 3072 (85%)		
	FFs = 321392 (37%)		
	LUTs = 317520 (73%)		

TABLE 5. Performance and energy analysis of black scholes asian option.

Params/Devices	FPGA	GTX960	K4200
Device time t_c	0.0815ns ($t_{clk} = 5ns$)	0.168ns	0.302ns
Device power P_d	24.064W	98W	105W
Energy E_c	1.96nJ	15.12nJ	27.78nJ
Utilization	BRAMs = 272 (19%)	NA	
	DSPs = 2176 (60%)		
	FFs = 277824 (32%)		
	LUTs = 257024 (59%)		

Table 4. Moreover, the table also shows the resource utilization information on the Virtex-7-series FPGA. Apparently, the Virtex-7-series FPGA has the best performance and least energy consumption among the three accelerators. Instead, GPU K4200 is the slowest and most power-consuming platform. The Virtex-7-series FPGA is 2X faster than K4200 for this algorithm. In addition, the device power of this FPGA is only 11% of the GTX960.

II Asian option

For the Black-Scholes model applied to the Asian option, the preset time is partitioned into 1024 segments. Table 5 presents the performance, energy consumption and resource utilization on the three accelerators. Also in this case, the FPGA is the best choice from the aspects of both power and performance. Virtex-7 is at least 2X faster than the GTX960 and consumes only 13% of the energy. Compared to the European vanilla option, performance is gained by a faster clock, which on the other hand consumes more power.

2) HESTON MODEL

I European vanilla option Table 6 shows the details on performance, power, resource utilization, etc. of the Heston model applied to the European vanilla options. Compared to the Black Scholes model, more time and energy are required to do the same amount of path simulations, due to the higher complexity. The GTX960 has better performance but worse energy profile than the K4200 in this case. The Virtex-7 is 4X faster than the GTX960 and uses 7% of the GPU energy for this algorithm.

II European barrier option

As can be seen from Table 7, the Virtex-7 FPGA again beats both GPUs in terms of performance and power.

TABLE 6. Performance and energy analysis of heston european vanilla option.

Params/Devices	FPGA	GTX960	K4200
Device time t_c	0.157ns ($t_{clk} = 5ns$)	0.604ns	0.663ns
Device power P_d	21.2W	88W	103W
Energy E_c	3.33nJ	48.32nJ	59.67nJ
Utilization	BRAMs = 112 (8%)	NA	
	DSPs = 2112 (59%)		
	FFs = 360544 (42%)		
	LUTs = 338464 (78%)		

TABLE 7. Performance and energy analysis of heston european barrier option.

Params/Devices	FPGA	GTX960	K4200
Device time t_c	0.158ns ($t_{clk} = 5ns$)	0.813ns	0.894ns
Device power P_d	31.17W	88W	103W
Energy E_c	4.917nJ	65.04nJ	80.46nJ
Utilization	BRAMs = 352 (24%)	NA	
	DSPs =3456 (96%)		
	FFs = 429184 (50%)		
	LUTs = 386016 (89%)		

TABLE 8. Performance and energy analysis of bitonic sort without synthesis directives.

Params/Devices	FPGA	GTX960	K4200
Device time	152ms ($t_{clk} = 5ns$)	16ms	25ms
Device power	5W	30W	40W
Energy	760mJ	480mJ	1000mJ
Utilization	BRAMs = 6 (0.4%)	NA	
	DSPs = 15 (0.42%)		
	FFs = 32758 (3.78%)		
	LUTs = 23523 (5.43%)		

TABLE 9. Performance and energy analysis of bitonic sort with FPGA-specific synthesis directives.

Params/Devices	FPGA	GTX960	K4200
Device time	17ms ($t_{clk} = 5ns$)	16ms	25ms
Device power	16W	30W	40W
Energy	272mJ	480mJ	1000mJ
Utilization	BRAMs = 10 (0.68%)	NA	
	DSPs = 60 (1.67%)		
	FFs = 108178 (13%)		
	LUTs = 111477 (26%)		

Compared to the GTX960, the FPGA has 5X performance and consumes only 8% of the energy for the same amount of workload.

C. BITONIC SORTING ALGORITHM

The baseline source code was provided by NVIDIA and it was optimized to be executed on GPU-based platforms. However, SDAccel has been utilized to optimize the source code further for implementation on a Xilinx FPGA.

Performance of the sorting algorithm is mainly limited by the off-chip memory accesses on the FPGA fabric, which demands a dedicated memory architecture to generate high performance RTL from OpenCL source code. On the other hand, GPUs with comparatively higher memory access

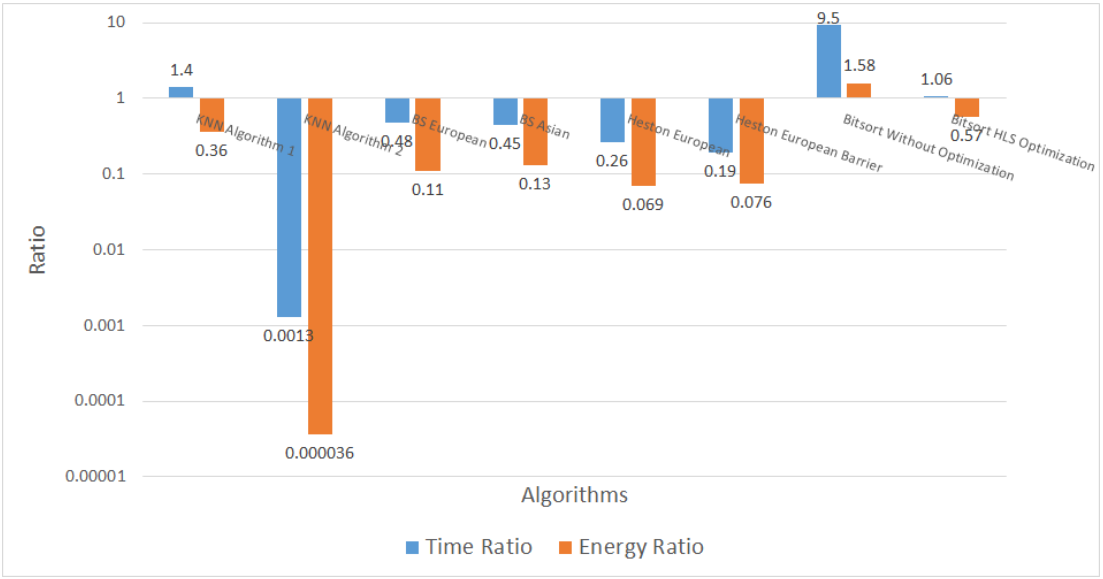

FIGURE 13. Execution time and energy-per-computation ratios between FPGA and GPU.

TABLE 10. Summary of results.

Test cases	Intensiveness	Best case FPGA		Best case GPU		Time ratio	Energy ratio
		Time	Energy	Time	Energy		
KNN Algorithm 1	Memory-access	4.24ms	0.43mJ	3.04ms	1.2mJ	1.4	0.36
KNN Algorithm 2		**1.23ms**	**0.003J**	0.93s	84J	0.0013	3.57E-05
Black Scholes Model European Option	Floating-point computation	**0.0788ns**	**1.67nJ**	0.164ns	14.76nJ	0.48	0.11
Black Scholes Model Asian Option		**0.0815ns**	**1.96nJ**	0.168ns	15.12nJ	0.45	0.13
Heston Model European Option		**0.157ns**	**3.33nJ**	0.604ns	48.32nJ	0.26	0.069
Heston Model European Barrier Option		0.158ns	4.917nJ	0.813ns	65.04nJ	0.19	0.076
Bitonic Sort without HLS Optimizations	Memory-access	152ms	760mJ	16ms	480mJ	9.5	1.58
Bitonic Sort with Optimizations		17ms	**272mJ**	**16ms**	480mJ	1.06	0.57

bandwidth can yield efficient performance at the cost of higher energy consumption, especially for such global memory access intensive algorithms. Several optimizations offered by SDAccel to reduce global memory accesses have been described in section III-A3 and utilized to get the desired performance in this case.

Table 8 presents the performance, resource and energy consumption of the FPGA-based implementation obtained with the default synthesis strategy used by Vivado HLS i.e. prioritizing resource minimization. This implementation however, suffers from performance problems. The performance is improved by utilizing the performance optimizations offered by SDAccel discussed before, i.e. by using:

1) On-chip global memories between the first and second kernel, because the communication between them follows a streaming pattern (data are consumed in the same order as they are produced).
2) Burst accesses to global memory whenever a kernel accesses sequentially adjacent locations.
3) Multiple compute units.

Besides these techniques, which mainly target bandwidth utilization improvement and memory access efficiency, some complementary optimizations were made as well. Local arrays were partitioned, loops were unrolled and work-items were fully pipelined, thus resulting in high performance RTL at the cost of higher resource utilization. Performance comparison of the optimized bitonic-sort algorithm in terms of execution time and power/energy consumption on the various hardware platforms is presented in Table 9. The execution time on the FPGA in this implementation has decreased significantly as compared to the FPGA implementation of Table 8, but the power/energy consumption increases due to increased resource utilization. With regard to the targeted GPUs, this implementation is as performance efficient as the best performing GPU i.e. GTX960 and beats the K4200 GPU. Moreover, this FPGA implementation is also more energy-efficient as compared to its GPU counterparts.

D. SUMMARY

Table 10 summarizes the best execution time and energy consumption for the GPU and FPGA implementations of our test cases. It shows that while performance is comparable, energy-per-operation is almost always significantly in favor of the FPGA. Pareto-optimal implementations (in terms of

execution time and energy consumption) of each algorithm are identified as well in this table by using a bold font. The results are also depicted graphically in Fig. 13, where all the bars below identity depict the test cases with FPGA outperforming the GPU in terms of execution time and energy-per-computation, while those above identity show cases in which the GPU outperforms the FPGA. It should be obvious that FPGA-specific optimizations of the OpenCL code, which was originally optimized for the GPUs, deliver significantly better results than out-of-the-box code.

VI. CONCLUSIONS

This paper performs an extensive analysis of the prospect of using high-level synthesis for implementing FPGA-based accelerators in modern HPC systems. Several algorithms from a diverse range of application domains were selected to perform this analysis. A performance comparison with a couple of high-end GPUs was performed as well. All but one of the test cases i.e. Monte Carlo simulation, were described in OpenCL for FPGA implementation. The Monte Carlo simulation kernel was written in C/C++ since its multiple executions did not require sharing data between them. It was concluded that FPGAs (mainly due to their hardwired control structure) offer greater energy efficiency in comparison to GPUs. Moreover, we were able to obtain more efficient implementations in terms of execution time as well, by performing some FPGA-specific optimizations. These mainly reduce the accesses to global memory, since its bandwidth for modern GPUs is considerably higher than for FPGAs. Moreover, algorithms with a large number of branching operations e.g. the one represented by the ''NEIGHBOR ESTIMATION'' kernel shown in Algorithm 2, can adversely effect the performance of GPUs due to thread divergence but can still be pipelined on FPGAs, thus leading to higher execution speeds.

Several options offered by the SDAccel tool from Xilinx were utilized to optimize the application code for FPGA implementation. They mainly included pipelining work-items and using on-chip global memory buffers for inter-kernel communications rather than using the traditional slower off-chip DRAM-based global memory buffers. Burst memory accesses were used for accessing the off-chip global memory resulting in higher efficiency, since the access overhead is shared between larger amounts of data being transferred. Concurrency on the FPGA was exploited further by splitting the overall kernel computations into smaller chunks and executing them in parallel using multiple compute units. All these optimizations were complemented by the conventional HLS-based datapath optimization options e.g. pipelining and unrolling both the explicit and the implicit loops in the kernels (i.e. the loops over work-items). We intend to exploit the findings from this research activity in the future to enhance the level of automation in existing HLS tools to obtain high performance energy-efficient acceleration of kernels written in non hardware-specific OpenCL by using FPGA-based platforms.

REFERENCES

[1] J. Teubner and L. Woods, ''Data processing on FPGAs,'' *Synth. Lectures Data Manage.*, vol. 5, no. 2, pp. 1–118, 2013.

[2] K. Pereira, P. Athanas, H. Lin, and W. Feng, ''Spectral method characterization on FPGA and GPU accelerators,'' in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Nov./Dec. 2011, pp. 487–492.

[3] M. Horowitz, ''1.1 computing's energy problem (and what we can do about it),'' in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[4] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, ''Comparing hardware accelerators in scientific applications: A case study,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 58–68, Jan. 2011.

[5] C. de Schryver *et al.*, ''An energy efficient FPGA accelerator for Monte Carlo option pricing with the Heston model,'' in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Nov./Dec. 2011, pp. 468–474.

[6] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, ''Accelerating deep convolutional neural networks using specialized hardware,'' Microsoft Res., Cambridge, MA, USA, White Paper 11, 2015, vol. 2.

[7] P. Sundararajan, ''High Performance Computing Using FPGAs,'' Xilinx White Paper: FPGAs, 2010, pp. 1–15.

[8] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, ''SDA: Software-defined accelerator for large-scale DNN systems,'' in *Proc. Hot Chips*, vol. 26. Aug. 2014, pp. 1–23.

[9] (2015). *Microsoft Extends FPGA Reach From Bing to Deep Learning*, accessed on Jul. 12, 2016. [Online]. Available: http://www.nextplatform.com/2015/08/27/microsoft-extends-fpga-reach-from-bing-to-deep-learning/

[10] *SDAccel Development Environment User Guide*, Xilinx, San Jose, CA, USA, v2015.1.

[11] *Vivado Design Suite User Guide: High-Level Synthesis*, Xilinx, San Jose, CA, USA, v2015.1.

[12] *SDAccel Development Environment Methodology Guide: Performance Optimization*, Xilinx, San Jose, CA, USA, v1.0.

[13] L. Struyf, S. De Beugher, D. H. Van Uytsel, F. Kanters, and T. Goedemé, ''The battle of the giants: a case study of GPU vs FPGA optimisation for real-time image processing,'' in *Proc. PECCS*, vol. 1, 2014, pp. 112–119.

[14] K. Ebrahimi, G. F. Jones, and A. S. Fleischer, ''A review of data center cooling technology, operating conditions and the corresponding low-grade waste heat recovery opportunities,'' *Renew. Sustain. Energy Rev.*, vol. 31, pp. 622–638, Mar. 2014.

[15] B. Aydin, ''Parallel algorithms on nearest neighbor search,'' Dept. Comput. Sci., Georgia State Univ., Atlanta, GA, USA, vol. V, no. N, Art. no. A, 2014.

[16] V. Garcia, É. Debreuve, F. Nielsen, and M. Barlaud, ''K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching,'' in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2010, pp. 3757–3760. [Online]. Available: http://dx.doi.org/10.1109/ICIP.2010.5654017

[17] V. Garcia, E. Debreuve, and M. Barlaud, ''Fast k nearest neighbor search using GPU,'' in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2008, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/CVPRW.2008.4563100

[18] Y. Pu, J. Peng, L. Huang, and J. Chen, ''An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL,'' in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2015, pp. 167–170. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2015.7

[19] S. Weston, J.-T. Marin, J. Spooner, O. Pell, and O. Mencer, ''Accelerating the computation of portfolios of tranched credit derivatives,'' in *Proc. IEEE Workshop High Perform. Comput. Finance (WHPCF)*, Nov. 2010, pp. 1–8.

[20] L. Ma, F. B. Muslim, and L. Lavagno, ''High performance and low power Monte Carlo methods to option pricing models via high level design and synthesis,'' in *Proc. Eur. Modelling Symp. (EMS)*, Pisa, Italy, Nov. 2016, pp. 1–6.

[21] R. Chen, S. Siriyal, and V. Prasanna, ''Energy and memory efficient mapping of bitonic sorting on FPGA,'' in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2015, pp. 240–249.

[22] R. Mueller, J. Teubner, and G. Alonso, ''Sorting networks on FPGAs,'' *VLDB J.–Int. J. Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, Feb. 2012.

[23] Q. Mu, L. Cui, and Y. Song. (2015). ''The implementation and optimization of Bitonic sort algorithm based on CUDA.'' [Online]. Available: https://arxiv.org/abs/1506.01446

[24] (2015). *Microsoft Knows Exactly Where Intel's Future Is*, accessed on Jul. 12, 2016. [Online]. Available: http://www.wired.com/2015/06/microsoft-knows-exactly-intels-future/
[25] (2016). *Data Centers Dominate FPGA Event*, accessed on Dec. 12, 2016. [Online]. Available: http://www.eetimes.com/author.asp?section_id=36&doc_id=1330431
[26] D. Singh, "Implementing FPGA design with the OpenCL standard," Altera, San Jose, CA, USA, White Paper, 2011.
[27] L. Howes and A. Munshi, "The OpenCL specification," Tech. Rep., 2015.
[28] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, nos. 1–3, pp. 66–73, 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69
[29] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, no. 1, pp. 1–33, Jan. 2010.
[30] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing with OpenCL 2.0*. San Mateo, CA, USA: Morgan Kaufmann, 2015.
[31] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in HLS," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2015, pp. 163–168. [Online]. Available: http://dx.doi.org/10.7873/DATE.2015.0798
[32] *Vivado Design Suite User Guide: Power Analysis and Optimization*, Xilinx, San Jose, CA, USA, v2015.3.
[33] L. Sánchez, J. Ranilla, and A. Cocaña-Fernández, "EECluster: An energy-efficient tool for managing HPC clusters," *Ann. Multicore GPU Program.*, vol. 2, no. 1, pp. 15–24, 2015.
[34] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797
[35] (2012). *Rodinia/Opencl/NN*, accessed on Jul. 12, 2016. [Online]. Available: https://github.com/kkushagra/rodinia/tree/master/opencl/nn
[36] M. Broadie and Ö. Kaya, "Exact simulation of stochastic volatility and other affine jump diffusion processes," *Oper. Res.*, vol. 54, no. 2, pp. 217–231, Mar./Apr. 2006.
[37] T. Odelman, "Efficient Mont Carlo simulation with stochastic volatility," School Comput., Roy. Inst. Technol., Stockholm, Sweden, Tech. Rep., 2009.
[38] (2012). *2012 Hurricane/Tropical Data for Atlantic*, accessed on Jul. 12, 2016. [Online]. Available: http://weather.unisys.com/hurricane/atlantic/2012/index.php

FAHAD BIN MUSLIM received the M.S. degree in communication engineering from Chalmers University of Technology, Göteborg, Sweden, in 2010. He is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications, Politecnico di Torino, Italy, under the supervision of Prof. L. Lavagno. His research interests include electronic design automation with special emphasis on low-energy high-performance computing based on high-level synthesis.

LIANG MA received the M.S. degree (Hons.) from Politecnico di Torino, Italy, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. L. Lavagno. His research interests focus on electronic system level design, low-power, and high-performance computing, and high-level synthesis.

MEHDI ROOZMEH received the master's degree in electronics engineering from Politecnico di Torino, Italy, in 2014. He is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications, Politecnico di Torino, under the supervision of Prof. L. Lavagno. His main research interest is in high-level synthesis that targets Xilinx FPGAs, focusing on high-performance applications.

LUCIANO LAVAGNO received the Ph.D. degree in EECS from the University of California at Berkeley, Berkeley, CA, USA, in 1992. He was the architect of the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect of the Cadence CtoSilicon high-level synthesis tool. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He co-authored four books and more than 200 scientific papers. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.