From Java to FPGA: an Experience with the Intel HARP System

Pedro Caldeira[1], Jeronimo C. Penha[2], Lucas Bragança[2], Ricardo Ferreira[2], José Augusto M. Nacif[2], Renato Ferreira[1] and Fernando M. Q. Pereira[1]

Abstract— **Recent years have seen a surge in the popularity of Field-Programmable Gate Arrays (FPGAs). Programmers can use them to develop high-performance systems that are not only efficient in time, but also in energy. Yet, programming FPGAs remains a difficult task. Even though there exist today OpenCL interfaces to synthesize such hardware, higher-level programming languages, such as Java, C# or Python remain distant from them. In this paper, we describe a compiler, and its supporting runtime environment, that reduces this distance, translating functional code written in Java to the Intel HARP platform. Thus, we bring two contributions. First, the insight that a functional-style library is a good starting point to bridge the gap between high-level programming idioms and FPGAs. Second, the implementation of this system itself, including the compiler, its intermediate representation, and all the runtime support necessary to shield developers from the task of transferring data back and forth between the host CPU and the accelerator. To demonstrate the effectiveness of our system, we have used it to implement different benchmarks, used in image processing and data-mining. For large inputs, we can observe consistent 20x speedups over the Java Virtual Machine across all our benchmarks. Depending on the target function that we compile, this speedup can achieve 280x.**

I. Introduction

The last decade has seen a surge in the popularity of hardware accelerators. This technology stands out today as an effective and useful alternative to preserve the ever-growing performance that the computing industry craves for. In the context of this work, we call an *accelerator* any processor that can be attached to a host CPU to speed up computation which, in principle, that CPU should execute. Typical accelerators include Graphics Processing Units (GPUs) [21], and Field-Programmable Gate Arrays (FPGAs) [18]. A vast number of computing intensive applications have been shown to be amenable to acceleration, in fields as diverse as mathematics [17], biology [8], and physics [13].

Nevertheless, while GPUs are each day more accessible, the use of FPGAs remains restricted to expert programmers [7]. One of the reasons behind this apparent low popularity of FPGAs is the lack of programming support. GPUs can be directly programmed via specific languages such as OpenCL and C for CUDA, or annotation systems such as OpenACC [35] and OpenMP [11]. FPGAs, on the other hand, are still mostly programmed via hardware description languages, such as VHDL or Verilog. Only recently the first attempts to create OpenCL interfaces for FPGAs have reached the mainstream industry [15]. The effort to port higher-level programming languages to the FPGA world seem to collide against the wide semantic wall that separates the efficient hardware from popular programming abstractions such as objects and high-order functions [16]. This difficulty ends up compromising the efficiency of the hardware synthesized from languages such as Java [16].

In this paper, we seek to address such shortcoming – the lack of expressive programming support for FPGAs. To this end, we present a compiler that synthesizes hardware from Java code. However, contrary to previous work that strive to preserve the general object oriented paradigm that permeates this language [16], we translate functional-style code written in Java. To this effect, we chose to set off from the map-reduce pattern [6], a high-level programming abstractions that gives us abundant parallelism. We have designed and implemented a compiler that translates key methods of a functional Java library, ParallelME [1] to the Intel Altera HARP processor. Combinations of methods well-known among functional programmers, such as reduce, map, filter and zip, lets us write non-trivial algorithms used in data-mining or image processing. These algorithms are written in a high-level functional style, and our compiler ensures that they can benefit from all the time and energy efficiency of the FPGA-based accelerator.

Our compiler, plus its companion libraries, are publicly available today. In Section IV we demonstrate that this compiler is effective and useful. We have used this compiler to speedup the execution of four well-known algorithms: KNN, String Hash, image conversion and similarity search with great benefit. For large inputs, we can observe speedups of over 20x with regard to the original program that runs on the standard Java Virtual Machine. For the highly parallel implementation of similarity search, we could observe gains of up to 1000x. To benefit from this performance improvement, developers must write programs in a functional style, as a combination of the four previously cited operations, using the ParallelME library [1]. However, programmers remain unaware of the heterogeneous nature of the underlying hardware architecture: they do not need to insert any primitive to transfer data between CPU and accelerator, for instance. We already provide runtime support to shield developers from such tedious and laborious tasks.

II. Background

A. ParallelMe

ParallelME is a system, formed by a Java Library and a companion compiler, that allows the generation of high-

*This work was supported by CNPq, CAPES and FAPEMIG

[1]UFMG, Campus Pampulha, 31270-901, Belo Horizonte, Brazil {renato,fernando} at dcc.ufmg.br

[2]UFV, Campus Universitário, 36570-900, Viçosa, Brazil {lucas.branganca,ricardo,jnacif} at ufv.br

performance code originally targeting the Android Runtime Environment [1]. ParallelME is composed of two parts: (i) a programming abstraction (user-library), plus (ii) a source-to-source compiler. The programming model fostered in ParallelME was inspired by the Scala collections library [25]. This library consists of a set of different data structures with intrinsic support for parallel computing. The goal of ParallelME is to offer a similar data-structure-oriented library in Java that can be used in the development of dataflow-like parallel programs. An important characteristic of ParallelME is its bias towards functional programming, which follows from our earlier forays in the field [5], [27]. Code is written as the combination of high-order functions such as map, filter and reduce, for instance. A *high-order function* receives other functions as arguments. Developers achieve different behaviors by changing the operators passed to these high-order operations. Programs written in ParallelME can run on the Java Virtual Machine, as ordinary Java code, or can be translated to a parallel backend, such as a GPU or an FPGA. In the rest of this section, we describe the ParallelME system, with a goal of justifying some of the design decisions that we introduce later.

B. User-Library

Programmers interact with our system through a *user library*, that is, a collection of methods which can be combined to build applications. The programming abstraction presented in this work supports three types of operations for performing sub-setting, data-transformation and reduction in images and arrays, covering many of the needs of parallel implementations. These operations are: *filter*, *map* and *reduce*. They follow the form $A \overset{f}{\mapsto} B$, where f represents an operation implemented by the user. The only requirement placed onto this function f is that it must receive a Java collection A and produce an element B. B can be an entirely new collection, an empty set or a singleton. In what follows, we describe the three key operations used in ParallelME.

a) Filter $(A : L, f : L \mapsto \mathit{bool}) \mapsto B : L$*:* This operation provides users with the means to create sub-sets. A filter receives a collection A of type L, plus a predicate $f : L \mapsto \mathit{bool}$, and produces a new collection B of type L. This new collection contains only the elements from A that cause the predicate to be true. *Filter* does not allow side-effects, meaning that the input collection A is considered an immutable data structure. In other words, *filter* causes the creation of a new memory region to hold its output B. The operation is expressed by the following equation, where A is the input collection, f is the user function with a boolean result and B is the resulting collection:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \overset{f}{\mapsto} B = \begin{bmatrix} b^1 \\ \dots \\ b^m \end{bmatrix} \vee B = \emptyset, \quad \text{where} \quad B \subseteq A$$

Example 2.1 (Filter): The abstract type T represents the collection element type, which is also the collection element type associated to the *result* variable. The user code returns a boolean value, meaning that elements that return true in the function call will be inserted into the resulting collection, while those that return false will not.

b) Map $(A : L, f : L \mapsto L') \mapsto B : L'$*:* The *map* operation applies an user function over every element of a given input collection A, returning a new collection B as the result. It is mathematically expressed by the equation below, where A is the input collection, f is the user code and B is the resulting output collection:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \overset{f}{\mapsto} B = \begin{bmatrix} b^1 \\ \dots \\ b^n \end{bmatrix}$$

Like *filter*, a *map* operation does not allow side-effects. Thus, its input A is considered an immutable data structure. However, contrary to *filter*, a *map* outputs a collection B always with the same size as A, albeit with a potentially different type L'. The new data set is formed by elements that result from applying the operator f on each element in A. Notice that the relative order of elements is preserved from input to output. Because *map* does not yield side-effects, changes performed on a given element are discarded after the user function runs.

c) Reduce: $(A : L, e : L, f : (L, e) \mapsto L') \mapsto b : L'$*:* The *reduce* operation is an aggregation function designed to combine in pairs all the elements of an input collection A. *Reduce* returns a single summary value b of type L'. Similar to our previous two operators, *reduce* does not allow side-effects, meaning that the collection A is also considered an immutable data structure. Therefore, changes performed on a given element of A are discarded after the user function runs. The operation is expressed by the following equation, where A is the input collection, f is the user code with an aggregation function and b is the summary value:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \overset{f}{\mapsto} b$$

Example 2.2 (Reduce): The abstract type T represents the collection element type, which is also the summarized value type associated with the *result* variable. The user function, acting as an aggregator, receives a pair of input parameters (*elem1* and *elem2*), performs its operations and returns a single value. Each return value will be used as an input for the next iteration on the collection, forming a new pair of input parameters with an element not yet visited by the reduction iterator.

C. Harp

Multi-core architectures use cache-coherent, low-latency, and high speed fabrics for core communication. In 2011, Intel introduced the Heterogeneous Accelerator Reconfigurable Platform (HARP) [22], to integrate reconfigurable FPGA accelerators to multi-core systems. This new platform uses the Intel QuickPath Interconnect Technology (QPI), to allow all devices to share system memory, intermediated by cache coherence protocol fabrics. Later, in 2015, the first academic version for HARP was available, and in middle 2017 the second version of HARP has been released for the research

community. Finally, in early 2018, Intel announced the first commercial Intel Xeon Scalable processor with integrated Intel Arria 10 FPGA [9].

The HARP design bears several similarities with modern CPU-GPU systems; however, these technologies also present substantial differences. Figure 1(a)-(d) contrasts the HARP and the GPU architectures regarding the CPU data access mechanism. In CPU-GPU systems, the GPU should first transfer the data to the device memory, as we can see in Figure 1(a). On the other hand, the HARP system relies on a data coherence protocol that uses the cache to facilitate the sharing of data between host and device, as shown in Figure 1(b). The accelerator requests data on-the-fly, which is then delivered by the HW/SW Intel APIs. Thus, contrary to ordinary GPU systems, in the HARP data transfer and computation overlap without explicit intervention from developers. Figure 1(c) depicts a GPU implementation by using multiple hardware work queues (Hyper-Q), where $I_1, \ldots, I_n$ are the data sent from host to device, $K_1, \ldots, K_n$ are the acceleration kernel executions, and $O_1, \ldots, O_n$ are the device to host data transfer. To support seamless overlapping, the HARP has a local cache of 64Kb inside the FPGA as shown in Figure 1(d).

Although the Intel APIs help to eliminate the need for creating low level HW/SW data access, several limitations have prevented such tools from becoming widely accepted. First, the accelerator design and the accelerator interface still require expertise in FPGA devices. Even by using high-level languages like Intel's version of OpenCL, the design is highly sensitive to coding style to effectively extract parallelism. Moreover, the Intel "Best Practices Guide" advises in the introduction that programmers should know details of the underlying hardware, and compiler optimizations for FPGAs. One of the goals of this work is exactly to free developers from this subtitles of the hardware. Thus, we bring to developers who are not experts on FPGA design the benefits of this hardware, while letting them code applications in a popular and high-level language such as Java.

D. Accelerator Architecture

In this work, we propose to automatically generate an FPGA-based accelerator unit for the HARP version 2 platform shown in Figure 1(b). If the FPGA has enough resources, we generate the multiple accelerator units (ACC) to achieve higher throughput by exploiting spatial parallelism. Because the HARP provides a 16 GB/s bandwidth through its 512-bit interface, we have created a customized I/O interface to maximize data transfer between CPU and FPGA for the Harp architecture. This interface is implemented as a layer over Intel OPAE (Open Programmable Acceleration Engine). OPAE is an open community effort started by Intel to simplify the integration of FPGA acceleration devices. Therefore, the accelerator unit can be extended to target others FPGA-based cloud platforms.

III. Compiler Design

This paper describes a compiler that translates Java into Verilog, in a four-steps process:

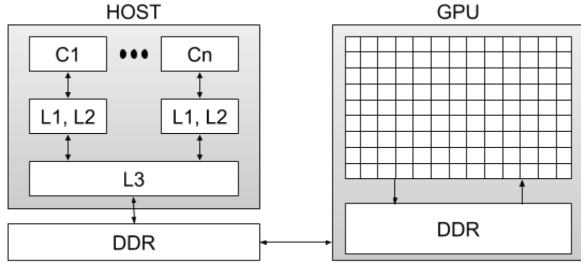
- Front-End: Java code is translated into an intermediate representation. At this level, programs are represented as dataflow graphs, in which vertices are operations, and edges denote dependences between operations;
- Merging: independent graphs representing different functions are merged into super-graphs. The union reduces data transfers between host and device;
- Parallelization: the dataflow super-graph is replicated. Each instance represents code that can run in parallel;
- Back-End: the final graphs are translated into Verilog. Verilog code is imprinted into the HARP FPGA.

Figure 2 shows the several intermediate representations that we use before generating Verilog code. In the rest of this section we describe each one of these representations, and explain how the translation process happens.

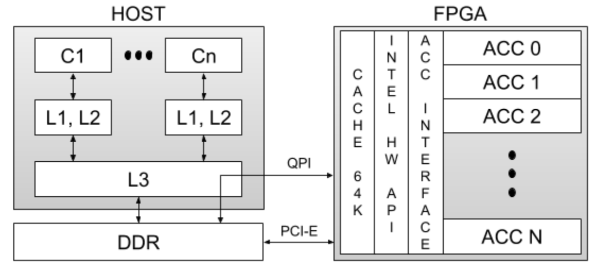
Front-End: the input of our compiler is a Java program; however, only a few parts of this program are translated. As mentioned in Section I, we translate four basic functional operations: map, reduce, filter and zip. These operations must be written as extensions of ParallelME classes. In Figure 2 we use anonymous classes to extend Map and Reduce. The translation of source code does not require parsing; instead, we rely on Java's reflection. Code is translated on-the-fly, that is, while the target program runs on the JVM. The output of this translation is a data-flow graph. Figure 2(b) shows an example.

Merging: the goal of this stage is to unite Dataflow graphs that execute in sequence. This union reduces data transfer between the CPU and the FPGA. Sequences are already discovered during the first phase of our pipeline, e.g., Front-End: if the output of a function f is used as input in a function g, and both are translated, then the joining applies. This optimization replaces the IR graphs produced to f and g into a single graph, which represents both functions. Operationally, joining is a relatively simple operation, which consists in rewiring edges: edges pointing to the output node of f are rewired towards the input node of g. Figure 2 (c) shows the graph that results from merging the two graphs seen in Figure 2 (b).

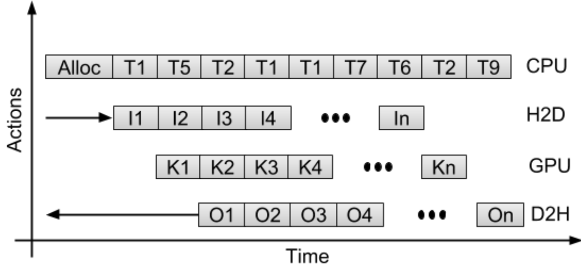
Parallelization: once we have the graphs that result from the merging phase –henceforth called supergraphs– we proceed to replicate them. Replication is the strategy that we use to obtain parallelism. Each one of the operations that we compile: map, reduce, filter and zip, contains one loop that iterates through a data-structure. In principle, we could simply synthesize this loop onto the FPGA. However, this approach would not gives us the parallelism that we need. Therefore, instead of simply implementing the loop directly, we *unroll* it. The unrolling factor is determined by the amount of resorts that we can fit into the FPGA. Typically, the loops that we produce process from 16 to 64 elements of a loop per cycle. Concretely, an unrolled loop consists of a series of similar supergraphs, as Figure 2 (d)



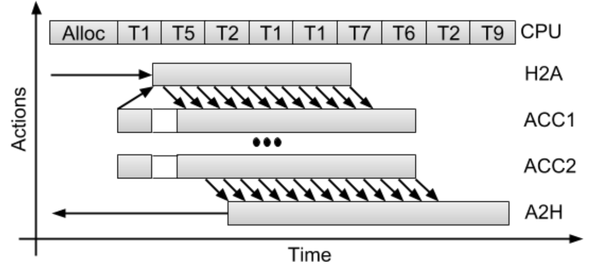
(a) Multicore and GPU architecture;



(b) Multicore and HARP architecture;



(c) GPU with multiple streams;



(d) Accelerator data transfer and execution;

Fig. 1. Multicore, GPU and Harp architectures: asynchronous execution model and data transfer

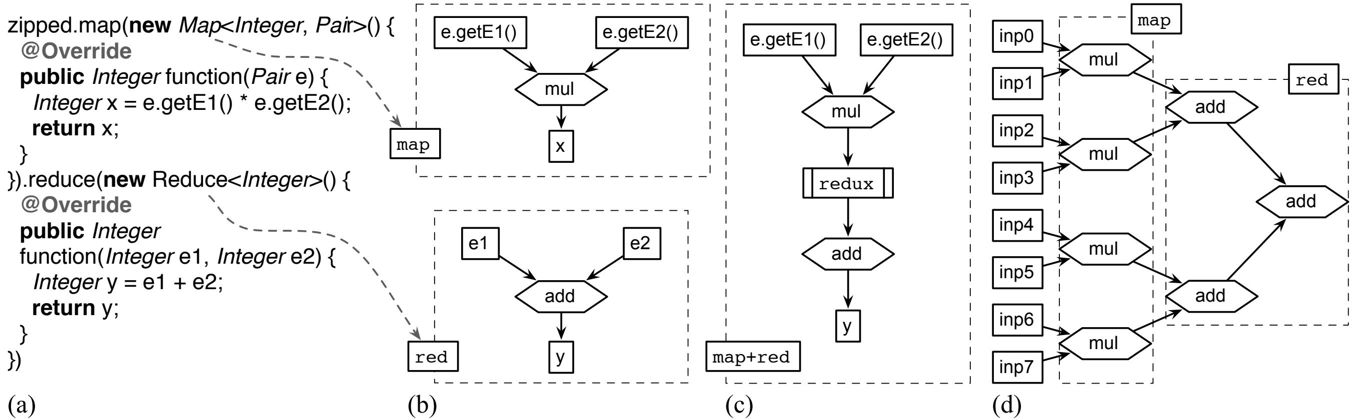


Fig. 2. The inputs of each phase of our compilation pipeline. (a) Java program that uses ParallelME libraries. (b) Dataflow graph produced for each compilable function. (c) Super-graph produced after merging. (d) Replicated dataflow graph.

shows. We also generate an *epilogue* to account for loops whose number of iterations is not a multiple of the unrolling factor. This epilogue consists of a serial loop, which process all the remaining elements.

BACK-END: The final stage of our pipeline consists of hardware generation. To convert a data flow graph onto Verilog, we use Veriloggen [32]. Thus, we do not convert the dataflow seen in Figure 2 directly to Verilog; instead, we produce the input of Veriloggen. This tool receives a description of the hardware, written as a set of Python functions, and produces actual Verilog syntax. This description uses a series of components predefined in a framework developed in a previous work. [23] This tool, Veriloggen, shall read these modules, and will translate them to existing components in Verilog, which we imprint onto the HARP.

IV. EVALUATION

The goal of this section is to answer two research questions (RQs), which we state as follows:

- **RQ1**: is the proposed system expressive enough to let developers write effective and useful programs?
- **RQ2**: what is the speedup that we can expect by translating Java functions into the Intel HARP architecture?

A. RQ1 – Expressivity

It is difficult to measure expressivity: how easy and elegant are the algorithms that we let developers write. To address this research question, even if from a rather subjective point of view, we shall describe four benchmarks, which we have written to demonstrate the effectiveness of our system. These benchmarks are common algorithms, used in data mining

and image processing. Table I summarizes them. In the table, we let **Map**, **Fil**, and **Red** represent the number of map, filter and reduction operations present in each program, respectively. We let **Loc** denote Lines of Code, in Java, and **CLoC** denote the number of Compiled Lines of Code. In other words, **CLoC** represent the number of lines of code that were effectively translated by our compiler into Verilog. Notice that this number is small; however, its size is misleading. A statement such as `list.reduce(0, addition)` encodes, into a single line, relatively complex semantics. We shall describe each individual application in the rest of this section.

TABLE I
THE BENCHMARKS USED IN THIS PAPER.

Benchmark	Map	Fil	Red	LoC	CLoC
SubZip	1			50	8
SimSearch	1	1		81	16
StringHash	1		1	115	15
RGB2YUV	1			101	28

Benchmark 1: SubZip. This benchmark performs pairwise subtraction between two arrays, returning, for each index, the absolute difference of the subtraction of elements in that index. If one array is larger than the other, only the common prefix of the same size is used in the algorithm. Concretely, this operation is a combination of a zip and a map: The former combines elements in the same index, forming pairs, and the latter performs the operation: $f(x_1, x_2) = |x_1 - x_2|$.

Benchmark 2: SimSearch. This similarity search algorithm take an array v of integers, an integer query q and a maximum distance d. From these inputs, the algorithm finds the element in v that is the closest to q, as long as it is less than d. If no element meets this requirement, we return d. The definition of distance can be parameterized; in this paper, we use absolute value. Concretely, our implementation uses a map function to calculate the distance from every element in v to q. After that first phase, a filter function returns only elements that have distance to q under the threshold d. Finally, a reduce operation singles out the minimum element that we are searching.

Benchmark 3: StringHash. This benchmark calculates the hash code for a Java String via a map and a reduce operation. Concretely, we map every character in the string onto its 31st power, and then use a reduction to compute the module of the sum of all these elements:

$$f(s[n]) = s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$$

Benchmark 4: RGB2YUV. This benchmark converts RGB images into the YUV format. Concretely, the benchmark uses only one map operation. However, contrary to the previous algorithms that we use, in this case we use a more complex datatype. Instead of working on arrays of primitive types, we use an array of triples, which are implemented as Java classes. Thus, given an input (R, G, B), our mapping operation applies the following transformation on it:

$$\left(\begin{array}{lcl} Y & = & (9798R + 19235G + 3736B) >> 15 \\ U & = & (21208R + 16941G + 3277B) >> 15 + 128 \\ V & = & (-4784R + 9437G + 4221B) >> 15 + 128 \end{array} \right)$$

d) Discussion: Table I shows the size of each of these benchmarks. They are relatively small, and have been written in a high-level programming language. More importantly, they have been written without any concern about the fact that they would be, ultimately, compiled to a highly parallel computer architecture. As we shall see in Section IV-B, all these benchmarks can run seamlessly onto an ordinary computer, or in the Intel HARP. Thus, not only their semantics is portable across architectures – a fact that is expected, given that these programs are virtualized – but they are also "performance-portable". That is to say: we do not lose performance by translating the programs into the FPGA. On the contrary, we obtain impressive gains, as we soon discuss. Thus, we believe that we can provide a positive answer to our first research question.

B. RQ2 – Performance

Methodology. In this section, we evaluate the performance of the code that we produce, when compared to the original execution of the Java program, in the Java Virtual Machine. Runtime numbers report wall-clock execution. Each sample that we report results from 10 measurements. To reduce the so called warm-up time of the JVM, we discard the first three executions. Warm-up time is the time that the JVM takes to reach a steady state, in which code has been already compiled just-in-time, for instance. Every sample considers the data transfer between CPU memory and FPGA.

Throughput. Table II shows time results for the largest input values available for each one of the four benchmarks that we have. Throughput is measured as the amount of data processed per time unit. It is limited by the quantity of data that we can transfer to the FPGA, and by the amount of operations that we can perform simultaneously in the FPGA. The RGB2YUV benchmark gave us the maximum throughput – a natural fact, given that this benchmark contains the largest number of operations per data. The minimum through comes from SubZip – another natural fact, because this benchmark performs the lowest number of operations per data. Notice that the amount of data transferred differs among benchmarks. For example, RGB2YUV sends to the FPGA an array in which each cell consist of 3×32 bits, and reads back an array of similar size. StringHash, in turn, reads an array of characters, and returns a primitive value formed by 64 bits. In the case of SubZip, the input is twice the size of the output: it reads two arrays, and returns only one. Finally, our implementation of SimSearch has inputs and outputs of the same size. Instead of returning a single element, our implementation of filter uses a flag to indicate, for each element in the array, which cells yield true and false answers.

Speedup. Figures 3 and 4 compare runtimes of our benchmarks with and without acceleration. The goal of these

TABLE II
PERFORMANCE OF BENCHMARKING ON THE HARP2 PLATFORM.

Benchmark	Input GB	Throughput GB/s	Time ms	GOPs
SubZip	4	17.72	340.75	2.93
SimSearch	4	19.06	420.88	11.88
StringHash	4	11.87	337.67	2.96
RGB2YUV	4	19.52	340.75	19.56

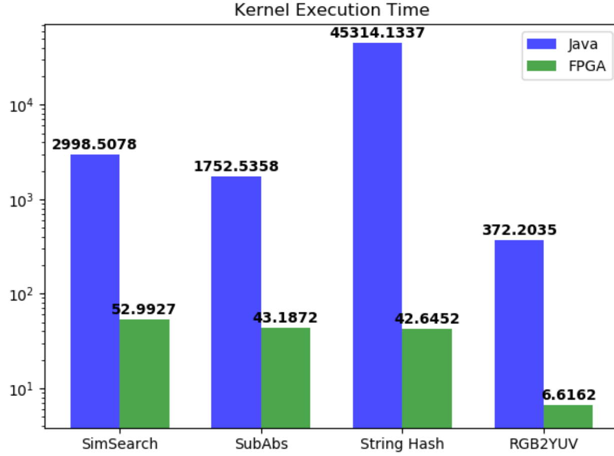


Fig. 3. Kernel Execution time for every benchmark. Bars are using logarithm scale and time are in Milliseconds

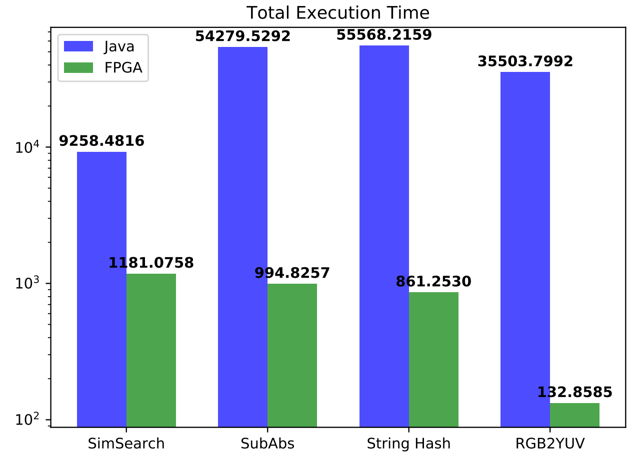


Fig. 4. Total execution time. JVM initialization isn't considered.

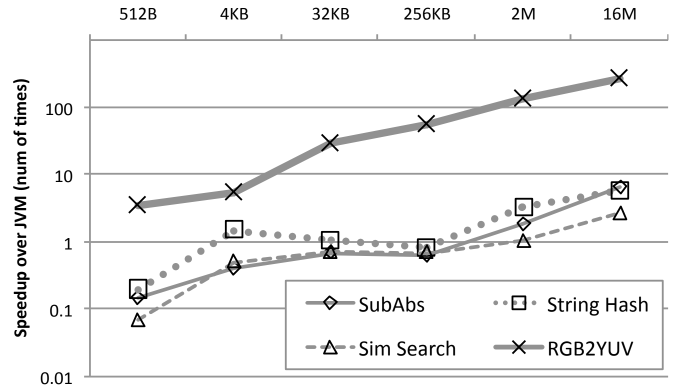


Fig. 5. Relation between speedup and the size of the processed data.

experiments is to show that we can increase the performance of our benchmarks by mapping them onto the Intel HARP. These charts show that our techniques boosts the runtime of all our benchmarks, in some cases, with substantial gains.

Figure 5 shows how our speedup increases, given increasing inputs. The larger the test case, the greater the observed speedup. The most noticeable gains were observed in RGB2YUV. For images with 4M $\times$ 4M pixels, we could observe an speedup of approximately 270 times over the same program running in the JVM without HARP-based acceleration. The fact that we obtain the largest speedup in this benchmark is expected, because it presents the highest amount of data reuse: the same pixel is processed multiple times during image conversion. Another interesting observation concerns the threshold of gains: for very small inputs, the use of the FPGA becomes a liability. In our experiments, input sizes with less than 32,768 elements lead to slowdowns when compiled to the HARP in every benchmark but RGB2YUV. For small inputs, the transfer time does not pay off the performance gained via parallelism.

Resource Usage. Table III shows the resources used by the HARP FPGA, after the synthesis of each benchmark. Every benchmark was synthesized with a clock of 200MHz. Hence, our gains in performance, reported in the previous experiments, come out of parallelism, not from computational power. The number of Adaptive Logic Modules (ALMs), together with the number of Digital Signal Processing (DSP) blocks, give us an idea of the relative complexity of the benchmarks. ALM units implement the basic operations performed by the synthesized hardware; DSPs implement a few batch operations, such as block multiplications and block additions on floating point numbers. The greater the presence of these units, the more complex tends to be the benchmark. The last column of Table III shows the time necessary to synthesize the hardware, in hours. The compilation of each benchmark takes about two hours to finish. This heavy cost in time is paid only once, during the synthesis of the map, reduce and filter operations that constitute a benchmark, and is also present in recent work that compile high-level programming languages to FPGAs [14], [24]. Further invocations of the same benchmark do not incur in such a cost.

TABLE III
RESOURCES USAGE FOR EACH BENCHMARK ON THE HARP2 PLATFORM.

Algorithm	Copies p/ ACC	ALMs	Memory bits	DSP Blocks	Time (h)
SubZip	16	110,405 (26%)	24,344,080 (44%)	0 (0%)	2:08
SimSearch	1	131,326 (31%)	24,344,080 (44%)	0 (0%)	2:02
StringHash	8	112,152 (26%)	24,344,080 (44%)	128 (8%)	1:53
RGB2YUV	15	119,595 (28%)	24,344,080 (44%)	720 (47%)	2:03

e) Discussion: The experiments in this section lets us show that our system is able to deliver very large speedups onto the typical execution of Java programs. Notice that our baseline is the sequential version of each Java program. One could argue that we should compare these programs against parallel implementations. However, we emphasize that in no moment developers had to worry about parallelizing the benchmarks that we have translated to the HARP. That is to say that we are delivering a huge performance boost at no development cost. Such could would have to be paid, for instance, if developers decide to rewrite the applications to use the Java Concurrent Library, for instance.

V. Related Work

In the last decades, several approaches have tried to overcome the FPGA programmability problem. The recent popularization of hardware accelerators have brought rewed importance to this quest, with several contributions been released in the last two years [12], [14], [20], [24], [33]. The work of Prabhakar *et al.* [24], is based on functional languages to express a dataflow representation of applications to be mapped to hardware pipelines. Their starting point is a high-level, domain-specific language embedded in Scala. They target mostly machine learning applications. To enhance programability in this context, Prabhakar *et al.* propose a series of compilation steps to automatically generate the hardware design from an intermediate representation (IR) based on parallel patterns (eg: map, reduce, zip, fold, etc.). These patterns are mapped into hardware via a set of parameterizable hardware templates, which are implemented using a low-level Java-based hardware generation language (HGL) called MaxJ. The experimental results report speedups up to 39.4x on a set of data analytics benchmarks. We could not reproduce these results, because Prabhakar *et al.* do not provide explicit information about execution time and the input data set. For the k-means benchmark, they report a 19.7x speedup relative to the handmade hardware baseline; however, previous work [14] reports a marginal speedup of 1.15x when compared to a 6 core CPU. Koeplinger *et al.* [14] proposed several improvements onto the contributions of Prabhakar *et al.* [24]. Our work differs from Koeplinger's and Prabhakar's in the choice of the language, and in the choice of the back-end. We also believe that we use a different compilation process: whereas Koeplinger and Prabhakar have pre-compiled components for particular functions, we performs actual compilation. In other words, none of our dataflow programs reuse pre-compiled parts.

There exist previous attempt to enhance FPGA programability in languages that do not run on the JVM, such as C and C++. For instance, the work of Wang *et al.* [33] presents a MapReduce framework on FPGAs named Melia, which abstracts FPGAs by using MapReduce interfaces written in C and OpenCL. They propose optimizations to tune a series of parameters which significantly affect the FPGA performance and resource utilization. They have reported speedups ranging from 0.5x to 3.2x compared to a GPU approach [33]. The work of Neshatpour *et al.* [20] presents an end-to-end map-reduce based implementation for a cluster of heterogeneous CPU+FPGA architectures. However, the authors have not experimented with an actual FPGA implementation; instead, they resort to simulation to present results. Kachris *et al.* [12] propose a specialized hardware accelerators by using High-level Synthesis (HLS) tools for the Map tasks as a control-dataflow, and a scratchpad memory reconfigurable accelerator for the Reduce tasks. Programmers must write their applications in a combination of C/C++ code using the Phoenix MapReduce framework. Kachris *et al.* report speedups in the range of 1-5x, compared against an 8-core processor.

In terms of heterogenous CPU-FPGA hardware, several companies have been designing their own platforms. IBM has developed the Coherent Accelerator Processor Interface (CAPI) [31] to integrate the Power8 processors with FPGAs or other accelerators. This interface allows the integration of accelerators to processors using I/O pins and provides an abstraction layer for the accelerators to access the system memory. In 2016, Microsoft released the Configurable Cloud [2], an evolution of the original Catapult project [26]. In this platform, the FPGA accelerator cards are placed between the Network Interface Card (NIC) and the switch, providing low-latency FPGA-to-FPGA communication. There is also a PCIe communication interface between the Xeon processors and the FPGA accelerator cards. Amazon Web Services is also becoming an important player after the release of the EC2 F1 platform, where FPGA-based accelerators are available at processing nodes in the cloud.

Previous work on Intel Harp v1 have demonstrated how to accelerate specific applications [3], [4], [36], [29], [37], [34], [10], [28], [19], [30]. Different from these approaches, we provide a generic Java-based programming interface to the Intel HARP system. We translates Java code directly to both CPU executable binaries and FPGA configuration bitstreams. This combination of CPU-FPGA code does not assume any previous hardware design knowledge from the programmer. Thus, in this paper we claim the design and implementation of the first compiler able to translate programs written in a high-level language to the HARP architecture.

VI. Conclusion

This paper has presented a complete framework that translates Java code into Verilog specifications, and uses the Intel HARP infrastructure to emulate such hardware. The benefit of our framework is programability: developers can write code in a high-level programming language, and profit from all the efficiency of the FPGA. Experiments performed on a number of benchmarks used in different domains show impressive gains. Our work differs from similar techniques not only in our choice of target technologies –Java and HARP– but also in terms of our design decisions: we chose to restrict compilation to a small core of components that could, indeed, benefit from extra parallelism. Our framework is practical and effective; however, much work is still left to be done. In particular, we would like to extend it to other functional operations, such as flat-map and group-by.

References

[1] Guilherme Andrade, Wilson de Carvalho, Renato Utsch, Pedro Caldeira, Alberto Alburquerque, Fabricio Ferracioli, Leonardo Rocha, Michael Frank, Dorgival Guedes, and Renato Ferreira. ParallelME: A parallel mobile engine to explore heterogeneity in mobile computing architectures. In *Euro-Par*, pages 447–459, Berlin, Heidelberg, 2016. Springer.

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[3] M. C. F. Chang, Y. T. Chen, J. Cong, P. T. Huang, C. L. Kuo, and C. H. Yu. The smem seeding acceleration for dna sequence alignment. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 32–39, May 2016.

[4] P. Colangelo, E. Luebbers, R. Huang, M. Margala, and K. Nealis. Application of convolutional neural networks on intel xeon processor with integrated fpga. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[5] Leonardo Luiz Padovani Da Mata, Fernando Magno Quintão Pereira, and Renato Ferreira. Automatic parallelization of canonical loops. *Sci. Comput. Program.*, 78(8):1193–1206, 2013.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[7] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *FPGA*, pages 47–56, New York, NY, USA, 2012. ACM.

[8] Pascale Guerdoux-Jamet and Dominique Lavenier. SAMBA: hardware accelerator for biological sequence comparison. *Computer Applications in the Biosciences*, 13(6):609–615, 1997.

[9] Jennifer Huffstetler. Intel processors and FPGAs: Better together, 2018. `https://itpeernetwork.intel.com/intel-processors-fpga-better-together/`, Last accessed on 2018-05-30.

[10] Z. Istvan, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–211, May 2016.

[11] Julien Jaeger, Patrick Carribault, and Marc Pérache. Fine-grain data management directory for openmp 4.0 and openacc. *Concurr. Comput. : Pract. Exper.*, 27(6):1528–1539, 2015.

[12] Christoforos Kachris, Dionysios Diamantopoulos, Georgios Ch. Sirakoulis, and Dimitrios Soudris. An FPGA-based integrated mapreduce accelerator platform. *Journal of Signal Processing Systems*, 87(3):357–369, Jun 2017.

[13] Gota Kikugawa, Rossen Apostolov, Narutoshi Kamiya, Makoto Taiji, Ryutaro Himeno, Haruki Nakamura, and Yasushige Yonezawa. Application of MDGRAPE-3, a special purpose board for molecular dynamics simulations, to periodic biomolecular systems. *Journal of Computational Chemistry*, 30(1):110–118, 2009.

[14] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127, June 2016.

[15] Andrew C. Ling, Utku Aydonat, Shane O'Connell, Davor Capalija, and Gordon R. Chiu. Creating high performance applications with intel's FPGA OpenCL&Trade; SDK. In *IWOCL*, pages 11:1–11:1, New York, NY, USA, 2017. ACM.

[16] Robert Macketanz and Wolfgang Karl. JVX - A rapid prototyping system based on java and FPGAs. In *FPL*, pages 99–108, 1998.

[17] Svetlin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSCN*, pages 65–68. IEEE, 2007.

[18] Eric Monmasson and Marcian Cirstea. FPGA design methodology for industrial control systems – a review. *Transactions on Industrial Electronics*, 54(4):1824–1842, 2007.

[19] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel HARPv2 xeon+FPGA platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, pages 107–116. ACM, 2018.

[20] Katayoun Neshatpour, Maria Malik, Avesta Sasan, Setareh Rafatirad, Tinoush Mohsenin, Hassan Ghasemzadeh, and Houman Homayoun. Energy-efficient acceleration of mapreduce applications using FPGAs. *Journal of Parallel and Distributed Computing*, 119:1 – 17, 2018.

[21] John Nickolls and William J. Dally. The GPU computing era. *Micro*, 30:56–69, 2010.

[22] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85, Nov 2011.

[23] Jerônimo Penha, Lucas Bragança, Danilo Almeida, José Nacif, and Ricardo Ferreira. Add - uma ferramenta de projeto de aceleradores com dataflow para alto desempenho. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.

[24] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *SIGPLAN Not.*, 51(4):651–665, March 2016.

[25] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Euro-Par*, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, May 2015.

[27] Rodrigo C. Rocha, Luís Fabrício W. Goes, and Fernando Magno Quintao Pereira. Automatic parallelization of recursive functions with rewriting rules. *Science of Computer Programming*, X:1–39, 2018.

[28] T. S. Accelerating k-means clustering on a tightly-coupled processor-fpga heterogeneous system. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 176–181, 2016.

[29] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415. ACM, 2017.

[30] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable window generation for the intel broadwell+arria 10 and high-bandwidth FPGA systems. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, pages 173–182. ACM, 2018.

[31] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan 2015.

[32] Shinya Takamaeda-Yamazaki. Veriloggen: A library for constructing a verilog HDL source code in python. `https://github.com/PyHDI/veriloggen`. Accessed: 2017-07-20.

[33] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A mapreduce framework on OpenCL-based FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, Dec 2016.

[34] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 264–273. ACM, 2016.

[35] Sandra Wienke, Paul L. Springer, Christian Terboven, and Dieter an Mey. OpenACC - first experiences with real-world applications. In *Euro-Par*, pages 859–870, New York, NY, USA, 2012. Springer.

[36] C. Zhang, R. Chen, and V. Prasanna. High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 148–155, May 2016.

[37] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 35–44. ACM, 2017.