PDF Download
2656207.pdf
22 February 2026
Total Citations: 11
Total Downloads: 365

Published: 17 February 2015
Accepted: 01 September 2014
Revised: 01 January 2014
Received: 01 April 2013

Citation in BibTeX format

RESEARCH-ARTICLE

Enhancing Design Space Exploration by Extending CPU/GPU Specifications onto FPGAs

MUHSEN OWAIDA, University of Lausanne, Lausanne, VD, Switzerland

GABRIEL FALCÃO, University of Coimbra, Coimbra, Portugal

JOÃO MARIA ANDRADE, University of Coimbra, Coimbra, Portugal

CHRISTOS D. ANTONOPOULOS, University of Thessaly, Volos, Thessaly, Greece

NIKOLAOS E BELLAS, University of Thessaly, Volos, Thessaly, Greece

MADHURA PURNAPRAJNA, EPFL, Lausanne, Switzerland

View all

Enhancing Design Space Exploration by Extending CPU/GPU Specifications onto FPGAs

MUHSEN OWAIDA, University of Thessaly
GABRIELFALCAO and JOAO ANDRADE, University of Coimbra
CHRISTOS ANTONOPOULOS and NIKOLAOS BELLAS, University of Thessaly
MADHURA PURNAPRAJNA, DAVID NOVO, GEORGIOS KARAKONSTANTIS, ANDREAS BURG, and PAOLO IENNE, EPFL

The design cycle for complex special-purpose computing systems is extremely costly and time-consuming. It involves a multiparametric design space exploration for optimization, followed by design verification. Designers of special purpose VLSI implementations often need to explore parameters, such as optimal bitwidth and data representation, through time-consuming Monte Carlo simulations. A prominent example of this simulation-based exploration process is the design of decoders for error correcting systems, such as the Low-Density Parity-Check (LDPC) codes adopted by modern communication standards, which involves thousands of Monte Carlo runs for each design point. Currently, high-performance computing offers a wide set of acceleration options that range from multicore CPUs to Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). The exploitation of diverse target architectures is typically associated with developing multiple code versions, often using distinct programming paradigms. In this context, we evaluate the concept of retargeting a single OpenCL program to multiple platforms, thereby significantly reducing design time. A single OpenCL-based parallel kernel is used without modifications or code tuning on multicore CPUs, GPUs, and FPGAs. We use SOpenCL (Silicon to OpenCL), a tool that automatically converts OpenCL kernels to RTL in order to introduce FPGAs as a potential platform to efficiently execute simulations coded in OpenCL. We use LDPC decoding simulations as a case study. Experimental results were obtained by testing a variety of regular and irregular LDPC codes that range from short/medium (e.g., 8,000 bit) to long length (e.g., 64,800 bit) DVB-S2 codes. We observe that, depending on the design parameters to be simulated, on the dimension and phase of the design, the GPU or FPGA may suit different purposes more conveniently, thus providing different acceleration factors over conventional multicore CPUs.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms: Design, Algorithms, Performance

This work was partially supported by the Portuguese Fundação para a Ciência e Tecnologia (FCT) under the FCT projects PEst-OE/EEI/LA0008/2013, UID/EEA/50008/2013 and by grant SFRH/BD/78238/2011. It was also supported by the EC Marie Curie International Reintegration Grant (IRG) 223819.
Authors' addresses: M. Owaida, M. Purnaprajna, D. Novo, and P. Ienne, Processor Architecture Laboratory (LAP), Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland; emails: {mohsen.ewaida, madhura.purnaprajna, david.novobruna, paolo.ienne}@epfl.ch; G. Falcao and J. Andrade, Instituto de Telecomunicações, Department of Electrical and Computer Engineering, Faculty of Science and Technology, University of Coimbra, 3030-290 Coimbra, Portugal; emails: gabriel.falcao@uc.pt, jandrade@co.it.pt; C. Antonopoulos and N. Bellas, Electrical and Computer Engineering Department, University of Thessaly, Greece; emails: {cda, nbellas}@uth.gr; G. Karakonstantis, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, United Kingdom; email: g.karakonstantis@qub.ac.uk; A. Burg, Telecommunication Circuit Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland; email: andreas.burg@epfl.ch.

DOI: http://dx.doi.org/10.1145/2656207

Additional Key Words and Phrases: Design space exploration, simulation tools, parallel computing, FPGAs, GPUs, OpenCL

ACM Reference Format:
Muhsen Owaida, Gabriel Falcao, Joao Andrade, Christos Antonopoulos, Nikolaos Bellas, Madhura Purnaprajna, David Novo, Georgios Karakonstantis, Andreas Burg, and Paolo Ienne. 2015. Enhancing design space exploration by extending CPU/GPU specifications onto FPGAs. ACM Trans. Embedd. Comput. Syst. 14, 2, Article 33 (February 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2656207

1. INTRODUCTION

The design of special-purpose computing systems involves balancing a diverse set of performance criteria such as throughput, power consumption, and resource efficiency. To achieve a balanced design, the evaluation of these characteristics often requires compute-intensive Monte Carlo simulations for design-space exploration because the complex and unpredictable input or environment of many real-world systems defies a straightforward closed-form performance analysis even for simple individual algorithms. Such simulations are often extremely time-consuming. For example, communication systems consist of multiple, complex signal processing tasks and blocks that must be optimized carefully through numerous Monte Carlo simulations to balance the complexity-performance tradeoff that governs the system design and implementation process. Due to the short design cycle of modern consumer electronics products, there is a strong need to provide tools and methodologies to accelerate such simulations. However, at the same time, acceleration must not come at the expense of additional design effort to develop dedicated simulation models for each block under consideration. Ideally, a single model lends itself to various stages in the design process where rapid simulation turn-around times are beneficial at early stages, with their results being confirmed and refined by more in-depth, yet more time-consuming simulation runs during later stages [Rupp et al. 2003].

Simulation systems have typically been processor-based. Algorithm programming and mapping on such systems have been simplified over the years; however, the proliferation of multicores introduced new complexity because algorithms had to be rewritten to take into account new concerns such as data dependencies, load balancing, and synchronization. However, conventional multicores offer rather limited opportunities for parallelism exploitation. In the manycore era, Graphics Processing Unit (GPU) based acceleration is gaining popularity in simulation platforms [Falcao et al. 2011; Falcao et al. 2012], mostly due to the high number of computational cores that these unconventional parallel architectures offer. Unfortunately, programming these systems involves, beyond explicitly identifying parallelism, expressing it using appropriate programming models and languages that are often architecture-specific. The Compute Unified Device Architecture (CUDA) [NVIDIA 2007] programming model is a typical example for NVidia GPUs.

On the other side, Field Programmable Gate Arrays (FPGAs) are also popular as computation accelerators [Smith et al. 2011] because of the extremely parallelism-aware exploitation opportunities they offer at an arbitrarily fine granularity and the possibility to adapt the whole architecture according to algorithmic needs. However, application mapping on FPGAs can be a laborious and time-consuming task. Moreover, it involves a completely new paradigm shift: moving from sequential programming to parallel HDL-based designs. Consequently, although FPGAs and GPUs appear as promising platforms for simulation-based explorations, the requirement of remapping and recoding the simulation on each of these platforms limits their potential.

As a step toward the goal of efficient multiplatform simulations, this work proposes and evaluates a unified framework that runs on top of different computational

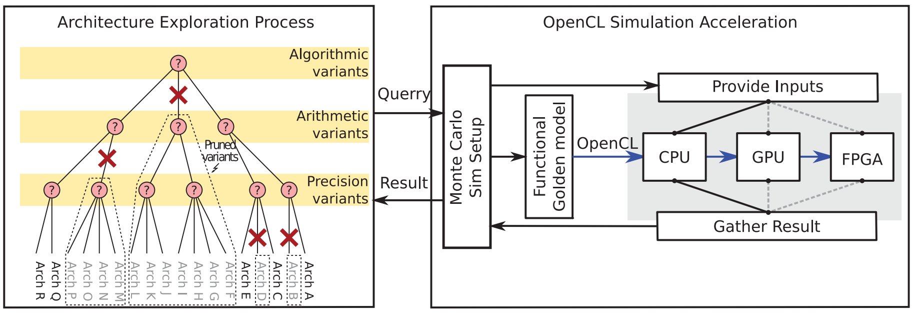

Fig. 1. Proposed flow to shorten VLSI design time through multiplatform simulation with a portable OpenCL golden-model.

substrates. The envisioned environment is depicted in Figure 1. We exploit the Open Computing Language (OpenCL) [Khronos 2010] programming model for expressing simulations that are expected to be executed on different platforms. OpenCL emerged as an open computing programming model supported by some of the most important computer manufacturers such as Apple, Intel, AMD, ATI, NVIDIA, and others. It allows developing parallel kernels that are portable across several multicore and manycore platforms. Beyond code-portability, OpenCL promises a good level of performance portability as well.

In our experimental exploration, we use the example of the Forward Error Correction (FEC) module—one of the most computationally intensive and widely researched components of communication systems. As a case study, we evaluate the decoder for regular and irregular, short-length and very-long-length Low-Density Parity-Check (LDPC) codes [Gallager 1962]. Although limited to a single type of algorithm, we believe that the LDPC case has characteristics of a rather tough example of large classes of modern DSP applications and can be considered a worst-case scenario for functional and performance portability on different substrates using a single codebase. The complicated computation pattern between different modules of LDPC decoders (depicted in Figure 2) represents a particular challenge for accelerators: In GPUs, it leads to predictable yet irregular communication between different kernels, whereas in FPGAs it causes considerable routing overhead.

We exploit different parallel computing platforms (CPUs, FPGAs, and GPUs) for simulating system behavior. We experiment with different configurations for parameters that are critical for system performance, such as bitwidth, number of iterations, and algorithmic variations. We target not only short and regular codes but also the more demanding irregular and long-length LDPC code scenarios that are adopted in real communication systems. However, similar principles can be naturally applied to the design of other hardware systems not related to telecommunications.

The contributions of this article can be summarized as follows: (i) We show quantitatively that, through OpenCL, two acceleration platforms (GPUs and FPGAs) are equally competitive to execute simulations of very complex coding algorithms in view of their later hardware implementation. In particular, we show that FPGAs can be a prime computing platform without any ad hoc source code twisting. (ii) Triggered by the scale of the experiments we conduct, we describe improvements to the FPGA compilation process that significantly improve the results and are key in achieving the above-mentioned goal. For example, we discuss grammar-driven instruction clustering, a technique necessary to enable the mapping of complex OpenCL kernels, such as those used in Digital Video Broadcasting–Satellite 2 (DVB-S2) LDPC decoding, on FPGAs.

Given the capability to easily execute simulation code on different architectures, we compare and quantify the relative performance of those platforms.

2. BACKGROUND AND MOTIVATION

2.1. Motivation

Computationally intensive Monte Carlo simulations require methods for acceleration that should be generic and easy to use, without the need for rewriting of early-stage simulation code. Using application-specific acceleration, such as designing Application-specific Integrated Circuits (ASICs) or Application-Specific Instruction Set Processors (ASIPs) is not feasible in this domain because of the effort and cost involved and the time necessary for design and verification. In a simulation environment, various configuration schemes and parameters of the algorithm necessitate modifying the input source code interactively. Mapping such an application on a conventional processor or GPU is considerably faster than on an FPGA, where development still requires using Hardware Description Languages (HDLs), skills usually only accessible to experienced hardware designers. Nevertheless, FPGAs provide the possibility of exploring higher degrees of parallelism and provide application-level customization during device deployment. In such a scenario, it becomes important to be able to quickly retarget a given application with a single specification language.

2.2. The OpenCL Parallel Programming Model

OpenCL [Khronos 2010] is a programming model for heterogeneous systems that may comprise conventional chip multiprocessors such as CPUs, GPUs, and various other forms of accelerators such as DSPs. Recent developments by co-authors of this work have led to the introduction of FPGAs as potential target platforms for OpenCL as well [Owaida et al. 2011b, 2011a].

OpenCL is based on a platform model that comprises a host processor and a number of computing devices. Each device consists of a number of compute units, which is subsequently divided into a number of processing elements. An OpenCL application is based on a host program and a number of kernel functions. The host part executes on the host processor and submits commands that can refer to compilation and execution of a kernel function or to the manipulation of memory objects. A kernel function contains the computational part of an application at a fine-granularity level of parallelism and is executed on the compute devices. The work corresponding to a single invocation of a kernel is called a *work-item* (i.e., the equivalent of a thread). Multiple work-items are organized in work-groups. OpenCL allows for geometrical partitioning of the grid of computations to an N-dimensional space of work-groups, with each work-group being subsequently partitioned to an N-dimensional space of work-items, where $1 \leq N \leq 3$.

A work-item is identified by a tuple of IDs defining its position within the work-group and by the position of the work-group within the computation grid. Based on the ID, a work-item is able to access different data or follow a different path of execution.

A distinct feature of OpenCL is that it facilitates exposing parallelism at a fine level of granularity, thus making it suitable for hardware generation at different levels of granularity. Another favorable characteristic is the explicit yet not overly detailed expression of data movement in the form of buffer transfers between host and compute devices.

2.3. Related Work

The idea of using various platforms as computational substrates to meet computational demands and comparing their performance and efficiency is certainly not a new one. Weber et al. [2011] use CPUs, GPUs, and FPGAs to execute a Quantum Monte Carlo application. They compare the application's performance and programmability on a variety of platforms, including CUDA with Nvidia GPUs, Brook+ with ATI graphics accelerators, OpenCL running on both multicore and graphics processors, C++ running on multicore processors, and a VHDL implementation running on a Xilinx FPGA.Cope et al. [2010] present a systematic approach for the comparison of GPUs and FPGAs. They apply three performance drivers to heuristically identify which accelerator platform is the most appropriate for a given application domain in an implementation-agnostic way. Both studies reach interesting conclusions in terms of the relative performance of CPUs, GPUs, and FPGAs. The use of a different source code base for each platform in Weber et al. [2011] introduces, however, the hazard of potentially unfair platform comparisons because it is not guaranteed that all implementations benefited from the same level of developer expertise and optimization effort.

Simulation frameworks are typically part of the software domain. In order to effectively introduce FPGAs as potential substrates for the execution of simulations, hardware accelerators have to be automatically generated from software descriptions. Methods following this direction have exploited high-level language to hardware translations. The PICO-NPA system translates C functions written as perfectly nested loops into a systolic array of accelerators [Kathail et al. 2002]. The LegUp synthesis tool generates a hybrid architecture comprising a MIPS processor and hardware accelerators to speed up performance-critical C code [Canis et al. 2011]. The hardware accelerator generation utilizes conventional HLS techniques for resources allocation, scheduling, and binding. The OpenRCL platform utilizes OpenCL to schedule fine-grain parallel threads to a large number of MIPS-like cores [Lin et al. 2010]. OpenRCL does not generate customized hardware accelerators, although each MIPS core can be configured to match application characteristics. The AutoPilot Compiler [Zhang et al. 2008] generates RTL descriptions for each function in a C program. Each function is translated into an FPGA core. AutoPilot provides code directives to facilitate hardware generation. However, the specification techniques proposed are not universally applicable to CPUs, GPUs, and FPGAs. In the GPU domain, FCUDA [Papakonstantinou et al. 2009] is a research effort that retargets CUDA kernels to synthesizable hardware in FPGAs. FCUDA transforms a CUDA kernel into a C function annotated with AutoPilot directives and then uses AutoPilot to generate synthesizable HDL. Recent publications propose using GPUs to perform LDPC decoding [Falcao et al. 2011] or functional programming to target LDPC codes in FPGAs [Gill et al. 2011; Smith et al. 2011].

Still, none of these approaches provides a unique solution that is suitable to simultaneously target CPU, GPU, and FPGA architectures. In this article, our objective is to simplify the exploration of all three target architectures for complex designs using a single, unified programming model and—moving a step further—a single source code. Retargeting various architectures using unmodified source code has several advantages: It reduces development effort—especially when targeting hard-to-program architectures like FPGAs—and facilitates future modifications. At the same time, though, since the code is not optimized for any of the underlying architectures, our approach stresses the automatic optimization capabilities of the toolchains used for the generation of the final executables and bitstreams on each platform.

3. LDPC DECODING: CASE STUDY ON INTENSIVE SIMULATION

LDPC codes are linear block codes (N, K) that allow achieving excellent Bit Error Rates (BER) [Gallager 1962] under various channel working conditions, usually measured in terms of Signal-to-Noise Ratios (SNR). LDPC codes can be described by a binary $\mathbf{H}$ matrix with dimension $(N - K) \times N$. Also, they can be represented by a Tanner graph defined by edges connecting two distinct types of nodes, viz. Bit Nodes (BN), with a BN for each one of the N variables of the linear system of equations, and Check Nodes (CN), with a CN for each one of the $(N - K)$ homogeneous independent linear systems of equations [Wicker and Kim 2003], as illustrated in Figure 2.

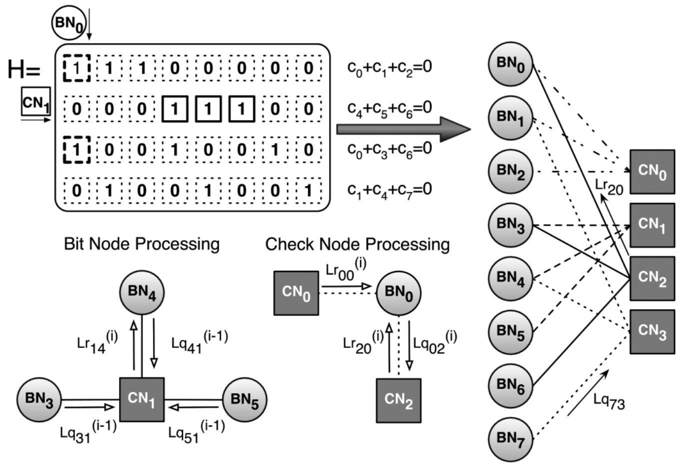

Fig. 2. Message passing illustrating the flow of communications between adjacent bit (BN) and check nodes (CN) of the Tanner graph.

3.1. Belief Propagation and Message-Passing: The MSA

Graphical models, and in particular bipartite Tanner graphs as the one shown in Figure 2, have often been proposed to perform approximate inference calculations [Wicker and Kim 2003]. They are based on iterative message-passing algorithms defined as Belief Propagation (BP), which is also known as the Sum-Product Algorithm (SPA). For certain classes of LDPC codes, in particular for long ones, the SPA requires a high computational effort [Falcao et al. 2011; Smith et al. 2011].

ALGORITHM 1: Min-Sum Algorithm

```
1: {Initialization} $Lq_{nm}^{(0)} = Lp_n$;
2: while ($\mathbf{H}\hat{\mathbf{c}}^T \neq \mathbf{0} \wedge i < I$) {c – decoded word; I – max. # of iterations.}
   do
3:    {For all node pairs ($BN_n, CN_m$), where $\mathbf{H_{mn} = 1}$ do:}
4:    {Compute the $LLR$ of messages sent from $CN_m$ to $BN_n$:}
      (CN Processing)
          $$Lr_{mn}^{(i)} = \prod_{n' \in \mathcal{N}(m) \setminus n} sign\left(Lq_{n'm}^{(i-1)}\right) \min_{n' \in \mathcal{N}(m) \setminus n} \left|Lq_{n'm}^{(i-1)}\right| \quad (1)$$
      {where $\mathcal{N}(m)\backslash n$ represents connect. to $CN_m$ excluding $BN_n$.}
5:    {Compute the $LLR$ of messages sent from $BN_n$ to $CN_m$:}
      (BN Processing)
          $$Lq_{nm}^{(i)} = Lp_n + \sum_{m' \in \mathcal{M}(n) \setminus m} Lr_{m'n}^{(i)} \quad (2)$$
      {where $\mathcal{M}(n)\backslash m$ represents connect. to $BN_n$ excluding $CN_m$.}
      Finally, we compute the a posteriori LLRs:
          $$LQ_n^{(i)} = Lp_n + \sum_{m' \in \mathcal{M}(n)} Lr_{m'n}^{(i)} \quad (3)$$
6:    {Perform hard decoding:}$\forall n$,
          $$\hat{\mathbf{c}}_n^{(i)} = \begin{cases} 0 & \Leftarrow \quad LQ_n^{(i)} > 0 \\ 1 & \Leftarrow \quad LQ_n^{(i)} < 0 \end{cases} \quad (4)$$
7: end while
```

The Min-Sum Algorithm (MSA) consists of a simplification of the SPA [Wicker and Kim 2003], and it is depicted in Algorithm 1. Although its workload is lower than the one required by the SPA, it is still quite significant if the number of nodes is large (in the order of thousands as it occurs in real systems [Falcao et al. 2011]). Lp_n designates the *a priori* Log-Likelihood Ratio (LLR) of BN_n, the logarithm of two complementary probabilities received from the channel [Wicker and Kim 2003], and it initializes Lq_{nm} before proceeding to the iterative body of the algorithm. The MSA is mainly described by two intensive processing blocks, respectively defined by Equations (1) and (2). The former calculates CN processing by producing Lr_{mn} messages that indicate the likelihood of BN_n being 0 or 1. The latter defines BN processing and computes Lq_{nm} messages. Hard decoding decision is performed as shown in Equations (3) and (4), and the iterative procedure is stopped if the decoded word $\hat{\mathbf{c}}$ verifies all parity check equations of the code ($\mathbf{H}\hat{\mathbf{c}}^T = \mathbf{0}$) or a predefined maximum number of iterations I is reached.

3.2. LDPC Codes for the Challenging DVB-S2 Case

Many communication standards already incorporate the use of LDPC codes. Examples are the Digital Video Broadcasting–Satellite 2 (DVB-S2) [EN 302 307 V1. 1.1, European Telecommunications Standards Institute (ETSI) 2005], DVB-C2, DVB-T2, 802.16e (WiMAX), 802.11 (WiFi), CMMB, DTMB, and other emergent standards such as the ITU-T G.709 used in optical communications. Currently, DVB-S2 is one of those standards for which the design of LDPC decoders is particularly challenging. The different weights w_c and w_b of bit and check nodes, the long length N of the words to be decoded, the diversity of codes and rates (21) used, and the fact that this standard supports two different frame length modes increases the development complexity of such systems. The periodic nature of DVB-S2 LDPC codes allows exploiting suitable representations of data structures and parallelize processing for accelerating the intensive computation required. The parity-check matrix $\mathbf{H}$ of DVB-S2 is based on Irregular Repeat and Accumulate (IRA) codes [Eroz et al. 2004; Jin et al. 2000] of type:

$$\mathbf{H}_{(N-K)\times N} = \left[\mathbf{H_1}_{(N-K)\times K} \,|\, \mathbf{H_2}_{(N-K)\times(N-K)}\right]$$
$$= \left[\begin{array}{ccc|cccccc} a_{0,0} & \cdots & a_{0,K-1} & \mathbf{1} & \mathbf{0} & \cdots & \cdots & \cdots & \mathbf{0} \\ a_{1,0} & \cdots & a_{1,K-1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & & & \vdots \\ \vdots & \ddots & \vdots & \mathbf{0} & \mathbf{1} & \mathbf{1} & \ddots & & \vdots \\ a_{N-K-2,0} & \cdots & a_{N-K-2,K-1} & \vdots & & \ddots & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ a_{N-K-1,0} & \cdots & a_{N-K-1,K-1} & \mathbf{0} & \cdots & \cdots & \mathbf{0} & \mathbf{1} & \mathbf{1} \end{array}\right], \quad (5)$$

where $\mathbf{H_1}$ is sparse and has periodic properties, and $\mathbf{H_2}$ has a staircase lower triangular profile, as shown in Equation (1). The periodicity constraints of the pseudo-random generation of $\mathbf{H_1}$ allow a significant reduction on the storage requirements without code performance loss.

The number of edges also changes for each code of the DVB-S2 standard. Messages of type Lr_{mn} and Lq_{nm} circulate in each edge to update the corresponding nodes they are connected to. For example, code with rate = 3/5 is represented by a Tanner graph with a total of 285,120 edges. Since communications occur in both directions (first from CNs to BNs, and then from BNs to CNs), this implies that, for this case, more than 2^{19} messages are exchanged per iteration, which is indicative of the heavy computation and high number of memory accesses required by this family of LDPC decoders.

3.3. Defining Fundamental Simulation Parameters

The optimized implementation of this algorithm depends on many different parameters. Typically, several Monte Carlo simulations are executed in order to decide the optimal configuration for these parameters that leads to a favorable design-point in terms of the area, performance, and energy tradeoffs while at the same time being compliant with system requirements. The configuration metrics to manipulate are briefly mentioned here:

—*LDPC code and* **H** *matrix*. The data structures that define the LDPC code are imported from an **H** matrix in the form depicted in Figure 2. It is of vital importance that the designer can test all the LDPC codes required by the application. LDPC codes can be regular or irregular, and this metric is defined by the number of ones per row and column, which can be constant or variable. In addition to regularity, different code word lengths can be tested.

—*Algorithmic variation*. As mentioned earlier, there can be different variations of the algorithm to test and implement. In the case of LDPC decoding, these consist, among others, of the SPA or MSA. Different algorithms may more appropriately suit different system needs.

—*Number of iterations*. Another metric commonly tested in this kind of applications is the number of iterations performed by the decoder. This metric has direct application in the simulation of BER curves, which are fundamental to prove that the design is compliant with performance and reliability requirements defined either by a standard or client. Simulation time is linearly dependent of the number of iterations.

—*Bitwidth*. Bitwidth definitions are among the most important parameters to decide during the design of a chip because they determine the width of the data path and the size of memory blocks. Moreover, they are usually directly related to the circuit area and its power consumption. On the other hand, the selected bitwidth should be compatible with BER performance requirements and expectations. When performing simulations, designers normally dedicate great attention to this fixed-point optimization parameter. In our study, bitwidth usually ranges from 5 to 8 bits.

4. UNIFIED DEVELOPMENT FOR MULTIPLE HIGH-PERFORMANCE ACCELERATORS

As a case study, we developed a single OpenCL representation of the LDPC simulation that can be executed unmodified on three distinct platforms: CPUs, GPUs, and FPGAs. Our goal is to allow a software developer—or ideally even a domain expert—to rapidly develop a single simulation code, which will then be deployed and executed on any available accelerator architecture. It should be noted that, given the proliferation of parallel platforms and especially multicore and manycore architectures, parallel programming tends to become mainstream. Therefore, programmers often have to develop code using a parallel programming model. OpenCL is one of the most popular ones, given its strong industry support and its promise for functional portability across a multitude of parallel platforms. OpenCL provides a common framework in which the coding constructs are almost the same throughout all supported architectures [Khronos 2010]. In any case, the painstaking part of the development effort is often not the initial implementation but rather the mapping and optimization for each different platform. In this study, we follow a different path: We evaluate the use of a single OpenCL code on all platforms without any further manual mapping and optimization effort.

4.1. Multicore CPUs and GPUs

An OpenCL kernel typically represents the execution of a lightweight thread (work-item) performing computations on a single point of a two-stage 3D decomposition of the problem domain (a 3D organization of work-items within a work-group and a 3D organization of work-groups within a grid).

In the case of LDPC code simulations, the exact parameters of the geometry (number of work-groups and work-items per work-group) are determined at runtime, using the device query functionality offered by OpenCL. Therefore, work partitioning automatically adapts to the capabilities of the underlying architecture. As formalized in Algorithm 1, the Min-sum decoding algorithm is performed in two phases: CN and BN Processing. As the minimum granularity of parallelism is adopted by the LDPC decoder simulator, each work-item is assigned to process a single node of the Tanner graph, represented in Figure 2. Thus, $(N - K)$ work-items are spawned to compute the CN Processing (1) and N for the BN Processing (2). This level of granularity is the minimum level that guarantees an independent execution path for each work-item in the same processing phase. Hard-coding finer or coarser granularity levels could potentially penalize performance. Finer grain activity performs either redundant memory accesses or imposes a non-negligible overhead on the processing since work-items need to exchange data among them. Also, due to the memory hierarchy model defined by the OpenCL standard, this adds constraints to the optimal size of the work-group. Notwithstanding, coarser grain expression of parallelism would not allow the full exploitation of the computational resources available on manycore systems (e.g., GPUs) for processing small to medium-sized datasets because there would not be enough work-items to fully exploit all the computational resources. It should be pointed out, however, that parallelism is not necessarily executed at the granularity at which it is expressed. As explained in Section 4.2.1, the OpenCL compiler and runtime system can collaboratively coarsen the granularity at which parallelism is actually executed in order to limit the parallelism management overhead on certain architectures.

4.2. FPGAs

CPUs and GPUs have been traditional targets for OpenCL codes [Falcao et al. 2012]. In this work, we introduce FPGAs as a new member of the simulation execution ecosystem. We used the Silicon-OpenCL (SOpenCL) tool [Owaida et al. 2011b] to automatically generate hardware accelerators using the unmodified OpenCL LDPC simulator code as input.

SOpenCL allows us to quickly explore different architectural scenarios and evaluate the quality of the design in terms of computational bandwidth, clock frequency, and size. The SOpenCL toolchain consists of two parts: a front end, mainly responsible for adjusting the granularity of parallelism and transforming OpenCL to C, and a back end, which performs a series of transformation and optimization passes that result in the generation of a synthesizable Verilog. All transformations and optimizations have been implemented as separate compiler passes in the context of the LLVM compiler infrastructure [Lattner and Adve 2004].

In the following subsections, we briefly outline the architecture of SOpenCL, focusing particularly on characteristics and optimizations that have produced a profound performance effect.

4.2.1. SOpenCL Front End. SOpenCL adjusts the granularity of parallelism of the OpenCL kernel to better match the hardware capabilities of the FPGA. A straightforward approach would map a work-item to an invocation of the hardware accelerator. This approach is clearly suboptimal for FPGAs because it would incur a heavy overhead to initiate thousands of work-items of fine granularity. Moreover, for most applications, it would be impossible to transfer data to the FPGA at the rate required for such a fine-grain implementation.

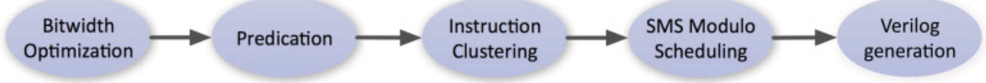

Fig. 3. Sequence of SOpenCL back-end code transformations and optimizations. Of particular interest for LDPC application are *bitwidth optimization* and *instruction clustering*.

Therefore, SOpenCL applies a series of source-to-source transformations that collectively aim at coarsening the granularity of a kernel at a work-group level.[1] The output of the front end is a C function for each kernel corresponding to the invocation of the kernel for a work-group.

4.2.2. SOpenCL Back End. After the front-end OpenCL-to-C transformation, the back-end flow generates the synthesizable HDL of LDPC decoder accelerators. The main transformations shown in Figure 3 include *bitwidth optimization* [Stephenson et al. 2000], *predication*, *instruction clustering,* and *modulo scheduling* [Llosa et al. 1996] and have been implemented as separate passes of the LLVM compiler. As a final step, the compiler back end generates the final hardware modules of the LDPC decoder application-specific architecture.

Bitwidth optimization is an automated method used to minimize the number of bits required to represent each operand [Stephenson et al. 2000]. It uses static code information such as type casts, array bounds, and loop iteration counts to refine variable bitwidths that are unnecessarily long for the purposes of the LDPC OpenCL program. The scope of bitwidth optimization includes integer and fixed-point arithmetic, boolean operations, and bit manipulation operations, all of which are well represented in the LDPC application. In fact, experimental evaluation on LDPC decoding kernels indicates significant area and performance benefits as a result of careful bitwidth optimization (see Section 5.2).

Predication converts control dependencies to data dependences within loops, transforming the loop body into a single basic block. This is a prerequisite to applying modulo scheduling in subsequent steps. LDPC decoder kernels include numerous yet short conditional statements that create hundreds of 1-bit predicate variables.

Instruction clustering automatically generates application-specific macro-instructions from a set of primitive instructions (basic arithmetic and logic operations) [Owaida et al. 2013]. In large-scale data paths, like LDPC, complex interconnection requirements limit resource utilization and often dominate critical path delay. For example, the area cost of a 32-bit adder with a 4-input multiplexer on each input port is dominated by the multiplexers tree (67% of the FPGA slices).

Instruction clustering transforms the basic operations in an application's Data Flow Graph (DFG) into a mixture of "primitive" instructions and application-specific macro-instructions. An application-specific macro-instruction substitutes a set of primitive operations and executes atomically (i.e., it appears as a single operation to the modulo scheduler). Regular computation patterns that appear repetitively in the DFG are strong candidates to be implemented as macro-instructions. The shaded subgraph A in Figure 4 represents a macro-instruction. Figure 4(b) shows the transformed DFG, which has now two primitive instructions and two macro-instructions of type *A*.

The generation of application-specific macro-instructions is a two-step process in SOpenCL. First, we identify candidate instructions to form macro-instructions and then we select a subset of candidate macro-instructions to be implemented as Macro Functional Units (MFUs). During candidate instructions generation, a space

[1]Other granularities can be supported as well.

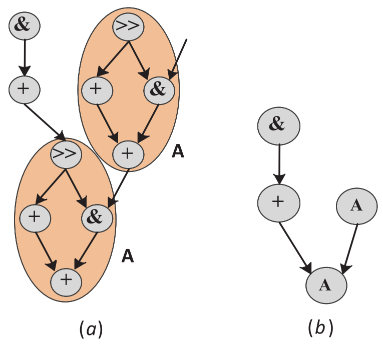

Fig. 4. Instruction clustering: (a) Portion of a program DFG. Shaded areas represent regular pattern candidates to be implemented as macro-instructions. (b) The DFG rewritten using two macro-instructions of type A.

exploration of the application DFG results in the identification of a set of subgraphs, each representing a potential macro-instruction. To identify macro-instructions, we use a grammar-driven approach. Each "primitive" instruction corresponds to a terminal symbol of the grammar. The algorithm tries to hierarchically form grammar rules, which substitute frequent sequences of terminal and nonterminal symbols for new nonterminal symbols. Each rule of the generated grammar corresponds to a candidate macro-instruction.

In the next step, a subset of grammar rules or, equivalently, of candidate macro-instructions is selected for implementation in the form of MFUs based on a fitness metric. The fitness metric targets minimizing the resources used by multiplexers and maximizing the number of DFG nodes covered by the generated macro-instructions. At the same time, the use of macro-instructions significantly reduces routing overhead, thus enabling the placement and routing of designs that would otherwise fail, such as the DVB-S2 decoder.

After identifying the subset of macro-instructions to be implemented as MFUs, the instruction clustering pass optimizes the design of each separate MFU. More specifically, we refrain from adding pipeline registers (full pipelining) between each pair of primitive operations within the MFU. Instead, we approximate an optimal pipelining of the MFU data path by exploiting the specific features of the target FPGA device, thus reducing area and latency in the process. We use an area estimation heuristic—easily calibrated by a standard set of synthetic microbenchmarks on each target FPGA—in order to identify consecutive operations within the MFU that can be implemented on a single Lookup Table (LUT). Pipeline registers are then necessary only between sequences of operations implemented on different LUTs, without any increase in the critical path length.

Swing modulo scheduling (SMS) [Llosa et al. 1996] is used to generate a schedule for the kernel. The scheduler identifies an iterative pattern of instructions and computes a static assignment of those instructions to its Functional Units (FUs), so that each iteration can be initiated before the previous ones terminates. SMS creates software pipelines under the criterion of minimizing the Initiation Interval (II). II is the constant time interval between launches of successive work-items measured in clock ticks. Lower values of II correspond to higher throughput since more work-items are initiated, and, therefore, more results are produced within a given time frame. That makes II the main factor affecting computational bandwidth in modulo-scheduled loop codes.

The inputs to the SMS scheduler are the instructions corresponding to each kernel, as well as an XML-based hardware model description of the target FPGA, which specifies FPGA device characteristics and resources.

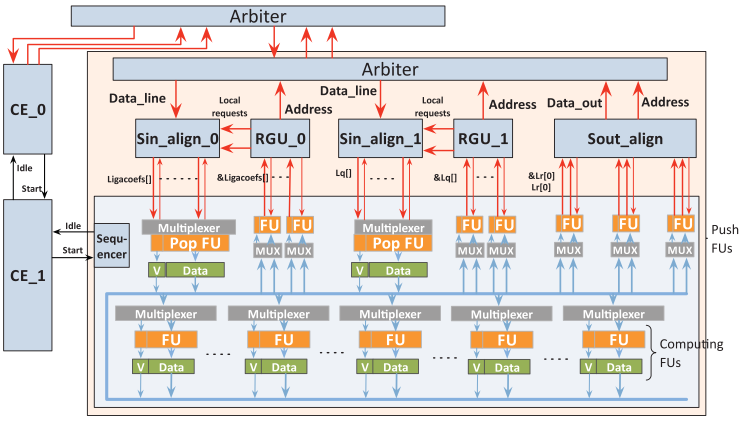

Fig. 5. Automatically generated hardware accelerator for the CN block of the LDPC decoder kernel.

4.3. Accelerator Architecture

Given the modulo-scheduled loop kernels, the compiler back end generates a modular Verilog by instantiating an architectural template according to the requirements of each kernel. Figure 5 depicts the architecture automatically generated for the LDPC CN kernel. The data path at the bottom of the block diagram executes kernel computations and generates addresses for Request Generation Units (RGUs). RGUs are used to coalesce incoming address requests and to interface to the memory system of the FPGA.

The input stream Alignment Unit, *Sin_Align*, retrieves incoming data and presents them to the data path in the order they will be consumed. The output stream Alignment Unit, in turn, aligns the output data tokens coming from the data path in a set of data lines, each having a bitwidth equal to the bitwidth of the data bus and corresponding to a range of subsequent memory addresses. As soon as the FIFO is full or the incoming data token is out of lines, the Alignment Unit issues the write request to the Arbiter. The output Alignment Unit is designed to optimize the use of the bandwidth to memory.

In addition to generating memory addresses for I/O, the data path executes the computationally intensive path of the algorithm, typically one corresponding to the innermost loop of a multilevel loop nest. The reconfigurable parameters of the data path are the type and bitwidth of FUs (ALUs for arithmetic and logical instructions, shifters, etc.), the custom operation performed within a generic FU (e.g., only addition or subtraction for an ALU), the number and size of registers in the queues between FUs, and the bandwidth to and from the streaming unit.

Finally, Control Elements (CEs) are used to control and execute the code of outer loops. CEs have a simpler, less optimized architecture because outer loop code does not execute as frequently nor is it as performance-critical as innermost loop code.

4.4. Memory System

Memory transfers between the host RAM and on-chip SRAM memories (FPGA BRAMs) are routed through an 8x PCIe v2.1 interface, offering 4,000MB/s bandwidth on each direction.

The on-chip memory subsystem needs to provide adequate bandwidth to keep the data path from stalling or, when this is not possible, to minimize it. For example, for $II = 1$, the implementation of the LDPC CN accelerator data path concurrently executes instructions from 106 successive loop iterations, requiring 392 adders, 210 shifters, 369 logic units, and 434 comparators, as well as 994 1-bit logic units for

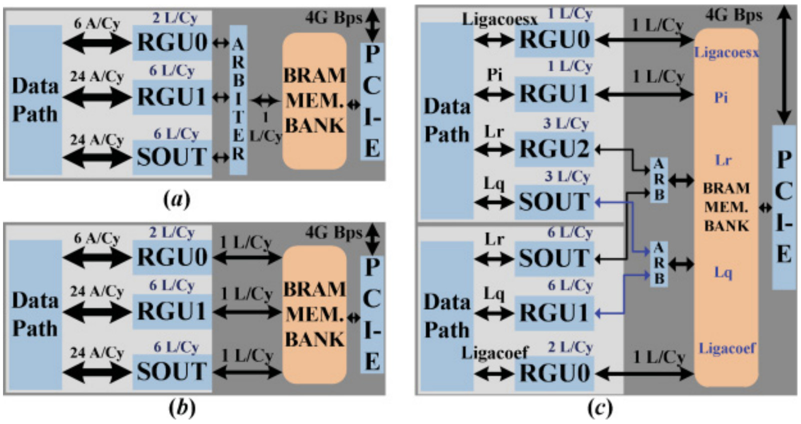

Fig. 6. System level block diagram of LDPC accelerators generated by SOpenCL with (a) CN (or BN) kernel communicating through a single port to BRAM, (b) through three ports to the BRAMs, one for each I/O stream, and (c) both kernels instantiated and interconnected.

predicate manipulation. The data path requires 120 bytes at the input and produces 96 output bytes every clock cycle. Therefore, the memory system should be able to sustain 216 bytes/cycle to avoid stalling the accelerator.

SOpenCL allowed us to evaluate two different memory system designs, depicted in Figure 6. The memory bank is built from FPGA BRAMs, concatenated to provide the total memory space required to store all stream I/O data. In Figure 6(a), the memory bank is configured as a unified single-port memory system, whereas Figure 6(b) shows the memory bank configured as a distributed memory system. Figure 6(c) depicts the two CN and BN kernels instantiated under the latter memory model with an arbiter on each port to orchestrate requests from the two kernels.

Figure 6 shows the throughput required by the data path of LDPC for $II = 1$, as well as the throughput provided by the memory system. The data path will generate in parallel: 6 Addresses/Cycle (A/Cy) for *ligacoesf* stream, $24A/Cy$ for L_q, and $24A/Cy$ for L_r. The RGU and Sout Align modules coalesce these addresses into 2 Lines/Cycle (L/Cy), $6L/Cy$, and $6L/Cy$ respectively, for a 128-bit data bus. The unified memory bank will provide a throughput of one line per cycle (single 128-bit data bus), which leads to stalling the data path for 14 cycles per computation/address generation cycle. In the distributed memory configuration, each RGU and Sout Align module is allocated a dedicated data bus (128-bit) to the memory bank with throughput $1L/Cy$. In this configuration, the stall time is shortened from 14 to 6 cycles. To achieve zero stall cycles, the memory bank would have to provide a wider data bus, 96 bytes to L_r and L_q streams and 64 bytes to *ligacoesf* stream.

It is clear that the distributed memory system configuration is more appropriate for our architecture since it provides higher bandwidth to fulfill the data path requirements. It is also more suitable to the distributed nature of on-chip SRAMs in the FPGA fabric. A unified single-port memory bank would unnecessarily restrict effective bandwidth and severely slow down the accelerator.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of LDPC decoders on the three platforms described in Section 4. It should be emphasized that a single OpenCL code was used. No platform-specific manual optimizations were performed to the OpenCL code at the source code or at any other level. Therefore, this study can serve as a fair comparison between the platforms in terms of the performance offered without platform-specific optimization effort. Moreover, the study evaluates OpenCL as a portable programming model in the heterogeneous architectures ecosystem.

5.1. Methodology

Two different LDPC codes are profiled, namely a regular (8,000, 4,000) LDPC and a more complex, irregular (64,800, 32,400) LDPC code used in DVB-S2 communications, each running for 30 iterations. Each iteration calls the CN kernel, followed by a call to the BN kernel. In the regular case, each CN kernel invocation spawns $N - K$ (4,000) work-items, and each BN kernel invocation spawns N (8,000) work-items, while 32,400 and 64,800 work-items are spawned, respectively, in the irregular case.

The OpenCL regular LDPC and DVB-S2 LDPC decoders were executed on an Intel Xeon E5645 6-core (12-threads) CPU system running at 2.4GHz, with 12MB of L3 cache and 4GB of DDR3. The CPU executable has been generated with g++ 4.4, using the 2012 version of the Intel OpenCL SDK. The decoders were also executed on a AMD/ATI Radeon HD 5870 Evergreen GPU, running at 1.2GHz, with 3GB DDR5, using v 2.8 of the AMD APP SDK.

Finally, we also experimented with an automatic hardware implementation of the decoders on a Xilinx Virtex-6 LX760 FPGA using the SOpenCL toolchain for automatic OpenCL to Verilog conversion and optimization, and the Xilinx ISE 12.4 toolset for synthesis, placement, and routing. LX760 contains $118,560$ slices, and each slice includes four LUTs and eight flip-flops. To evaluate the efficiency of the SOpenCL methodology we used different resource scenarios of hardware availability to guide modulo scheduling of the computational and I/O streaming kernels on an FPGA. The first scenario assumes that a new work-item is scheduled in every clock cycle (i.e., $II = 1$). In this case, each LLVM instruction is mapped to its own dedicated FU. Larger initiation intervals trade off throughput with resource availability and may correspond to platforms in which the memory system cannot sustain peak bandwidth to the accelerators.

5.2. FPGA Experimental Results

On FPGA-specific parts of the experimental evaluation, we focus on the automatic optimization techniques that proved of crucial importance for OpenCL performance portability on FPGAs: bitwidth analysis and instruction clustering. Both optimizations target the reduction of the area and the routing complexity associated with the FPGA designs; however, they have a direct effect on performance as well.

We experimentally evaluated three different regular LDPC code versions, assuming input data (codeword elements) of 5, 6, and 8 bits, and a fourth version in which the size of input data is specified as a runtime input parameter to the OpenCL kernel (*Generic* row in Tables I and II). Note that since OpenCL does not support bit-level specification of variables, any data size less than 8 bits is emulated in the source code by explicit masking-off of extraneous bits. The *Generic* version is particularly useful in the context of explorative simulations since it allows testing decoder performance for different input bitwidths without requiring resynthesizing, replacing, and rerouting the FPGA design.

We first examine the correlation of the area required for the FPGA implementations with respect to the value of the II targeted by the SOpenCL tool. Area costs are minimized when $II = 1$, which seems counterintuitive since this configuration requires a separate FU for each primitive instruction of the data path. The LDPC decoder kernel code consists mainly of simple operations (add, shift, logic) between a variable and a constant. Assigning dedicated FUs to each operation, as is the case when $II = 1$, forces one of the FU inputs to a constant value, thus providing ample opportunities for the synthesis tool to reduce area. When $II > 1$, this opportunity no longer exists since

Table I. Comparing Regular LDPC CN Kernel Implementations for Different $II = \{1, 2, 8\}$ Architecture Configurations

	CS	Slices	Flip-flops	LUTs	Freq. (MHz)	Latency (cycles)	Exec. Time (ms)
II=1	8 (no BW opt.)	12,061	42,718	39,594	100	102	0.48102
	8	11,600	41,892	38,759	101	102	0.476257
	6	11,647	35,948	33,914	103	106	0.467049
	5	10,369	33,639	32,861	107	106	0.449589
	Generic	24,108	101,960	80,115	91	106	0.528637
II=2	8 (no BW opt.)	25,453	64,311	92,096	88	103	0.546625
	8	21,424	54,872	81,526	97	103	0.495907
	6	23,632	61,035	78,884	95	110	0.506421
	5	19,374	61,052	65,192	88	110	0.546705
	Generic	28,432	67,307	73,212	63	110	0.763651
II=8	8 (no BW opt.)	33,213	54,749	78,266	50	210	1.2842
	8	27,556	57,582	58,788	53	210	1.211509
	6	27,008	56,745	64,104	50	231	1.28462
	5	26,894	54,868	64,083	51	231	1.259431
	Generic	36,954	58,121	79,682	51	231	1.259431

The comparison uses variable bitwidth precision with 5, 6, and 8 bits and a Generic on-the-fly bit precision selection approach (for $II = 8$, Generic and 5-bit cases have the same frequency and latency, and this is the reason that execution times are identical for both).

Table II. Comparing Regular LDPC BN Kernel Implementations for Different $II = \{1, 2, 8\}$ Architecture Configurations

	CS	Slices	Flip-flops	LUTs	Freq. (MHz)	Latency (cycles)	Exec. Time (ms)
II=1	8 (no BW opt.)	7,681	28,026	25,823	152	53	0.158243
	8	6,466	19,584	18,433	163	53	0.147564
	6	5,891	17,746	17,001	175	57	0.137469
	5	5,515	16,132	16,509	182	57	0.132181
	Generic	10,572	35,865	37,056	164	61	0.146713
II=2	8 (no BW opt.)	7,134	24,332	23,482	153	54	0.157216
	8	6,201	18,246	17,957	176	54	0.13667
	6	5,996	17,663	17,385	171	58	0.14069
	5	5,665	17,269	17,077	166	58	0.144928
	Generic	8,226	27,190	27,891	164	62	0.14672
II=8	8 (no BW opt.)	8,631	20,592	22,633	151	109	0.212642
	8	6,747	16,791	17,983	168	109	0.191125
	6	7,032	17,524	18,697	163	120	0.197055
	5	6,731	17,227	18,384	172	120	0.186744
	Generic	9,963	23,946	26,683	132	127	0.243386

The comparison uses variable bitwidth precision with 5, 6, and 8 bits and a Generic on-the-fly bit precision selection approach.

each FU input is driven by a multiplexer tree. In fact, configurations with larger II seem to be quite problematic when it comes to routing the design. Larger multiplexer trees cause routing congestion, which is not the case when the ISE placement tool can spread out FUs and make better use of routing channels.

Another interesting observation for $II > 1$ is that shorter bitwidths (5- and 6-bit data representations) in some cases require more resources than a bitwidth of 8. Our analysis shows that with larger bitwidth values, most of the FUs allocated will have appropriate sizes to serve a number of instructions with various bitwidths (from 5- to 32-bit). This will reduce the gain from custom bitwidths because the tool necessarily moves toward a larger, more generic FU size with larger II. Finally, the following set of operations are widely used in the code:

```
(data >> 24) & 255    For 8 bits
(data >> 24) & 63     For 6 bits
(data >> 24) & 31     For 5 bits
```

The LLVM compiler front end was smart enough to eliminate the masking operation for 8-bit because it is not necessary, but those operations remained for 6- and 5-bit kernels. This led to the use of an additional 96 masking operations in kernels with 6 and 5 bits. These additional instructions are significantly more costly with larger II values; they increase the density of the input multiplexer tree and may require more FUs, thus introducing additional input multiplexer trees. In fact, it was more problematic to place and route configurations with smaller bitwidths than 8-bit configurations when II was large.

For $II > 2$, SOpenCL automatically inserts pipeline registers between the multiplexer tree and the FU inputs to reduce the critical path delay and improve routing. This explains why the schedule latency for $II = 8$ is almost twice as large as the latency for smaller II values. In any case, clock frequency was mainly dictated by routing delays in most configurations, especially for the CN kernel. It should be noted that, given the streaming nature of the application, the frequency together with the efficiency of the memory subsystem are the dominant factors affecting performance.

Software developers tend to either pay minimal attention to the size of data types they are using or to be overly conservative. For example, quite often, a programmer will use an integer variable to store a boolean flag. To make the situation worse, many programming languages limit the granularity of standard data types to predefined sizes, not offering the opportunity to define arbitrary bitwidths. Hardware designers, on the other hand, are very sensitive to area limitations and invest significant effort in optimizing their designs to the minimum bitwidth required to support the data flowing at each level of the data path. Therefore, custom hardware synthesis driven by algorithmic descriptions in software can significantly benefit from automatic, aggressive bitwidth analysis and optimization.

The results for the non–bitwidth-optimized configuration are presented in row "*8 (no BW opt.)*" of Tables I and II for each target II. It is clear that bitwidth analysis is particularly effective in reducing area (in terms of the number of slices, flip-flops, and LUTs used) and increasing frequency. This is true even in the case of 8-bit input data. Although this may seem counterintuitive, it can be explained by the fact that the bitwidth analysis algorithm tries to aggressively reduce the bitwidth of internal data lanes and FUs of the data path whenever possible. Moreover, the *Generic* configuration is more area-demanding, which results in lower frequencies of operation (and thus performance) when compared to fixed bitwidth configurations. This is mainly due to the fact that the width of the input data is not known until runtime, thereby limiting the opportunities for savings with compile-time techniques. Second, the support of arbitrary input bitwidths requires additional logic (masking operations with unknown parameters at compile-time). Summarizing, the implementation of generic bitwidth algorithms on FPGAs presents an interesting tradeoff for designers. Data bitwidths can be a runtime parameter for simulations, thus allowing the evaluation of different bitwidths without incurring the overhead of FPGA synthesis, placement, and routing for each bitwidth. On the other hand, generic bitwidth implementations will require more area and result in lower performance.

Instruction clustering is a powerful optimization aiming at reducing area overhead and routing complexity especially in computation-bound designs. The gain of instruction clustering is threefold: area reduction, latency reduction, and frequency

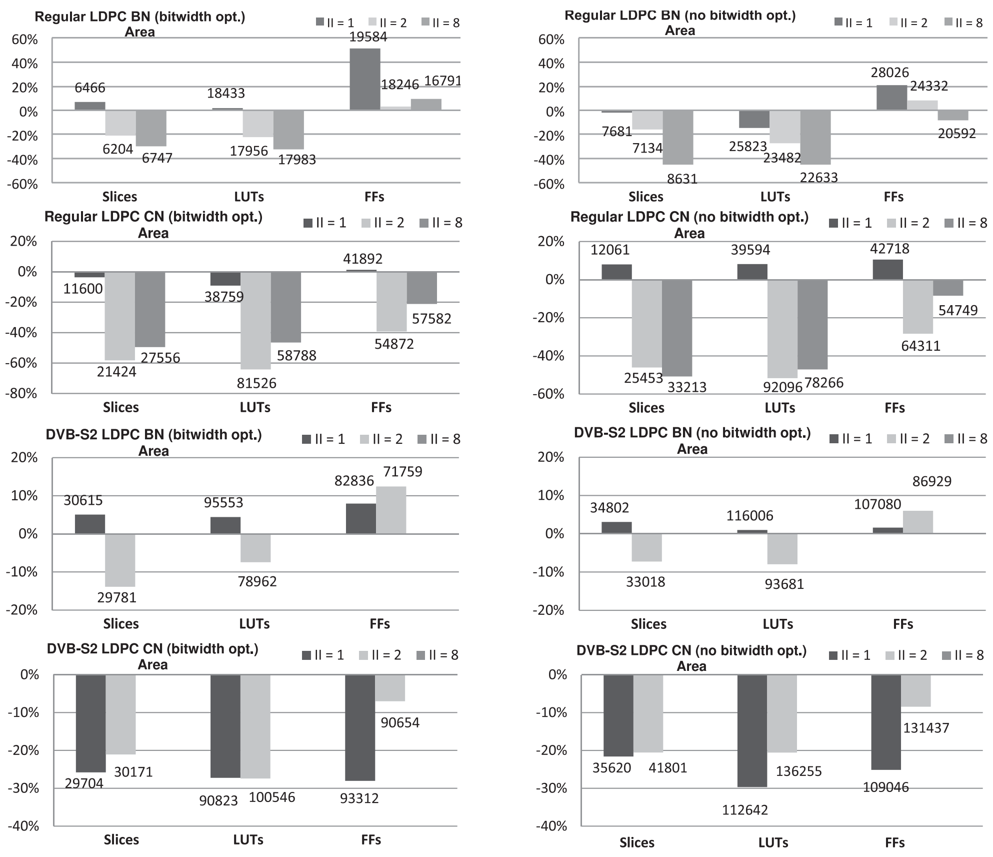

Fig. 7. Percentage of area difference (negative values correspond to reduction, therefore lower is better) achieved by applying the instruction clustering optimization on regular and irregular DVB-S2 LDPC decoder kernels, with respect to the nonoptimized versions of the benchmarks. The numbers on the bars indicate the base area consumed by nonoptimized configurations. On the left column of diagrams, the reader can observe the effect of instruction clustering on bitwidth-optimized kernels. The diagrams on the right column instruction quantify the effect of instruction clustering on non–bitwidth-optimized kernels.

improvement. The area reduction resulted mainly from reduced multiplexer size, which manifests its area as LUTs, especially for $II > 1$.

Figure 7 depicts the percentage of area improvement when the instruction clustering optimization is applied to the 8-bit regular LDPC and irregular DVB-S2 LDPC decoder kernels. The area results of the corresponding versions of the algorithms without applying instruction clustering are used as the base for comparison. In the figure, we can clearly observe that the decrease in area (slices) correlates with the decrease in LUTs, whereas the amount of consumed flip-flops tends to increase in some configurations (more on that shortly) and decrease in others. One can reason on that by analyzing the logic slice architecture of the target FPGA device. In Virtex 6 FPGAs, a logic slice includes four LUTs and eight flip-flops. Therefore, more flip-flops than LUTs can be packed in the same slice; hence, the additional flip-flop overhead in the optimized designs is easily outweighed by the reduction achieved by LUTs.

It appears that instruction clustering optimization sometimes performs poorly at $II = 1$, as the case for the BN kernel indicates. This is expected because, in this case, there are no multiplexers to optimize out. The reductions in consumed area may come mainly from optimized MFU pipelines. In such cases, full logic cells (LUTs) can be optimized out, and the overhead due to the increase of variables lifetime is minimized.

Interesting results can be observed for the DVB-S2 LDPC decoder benchmark. The area results for $II = 8$ are missing from the diagrams because the Xilinx ISE synthesis, placement, and routing tool failed, after 16 hours, to finish successfully without applying the instruction clustering optimization. On the other hand, instruction clustering optimized configurations finished placement and routing successfully within a few hours.

Figure 8 depicts the performance (frequency and latency) effect of the instruction clustering optimization on the 8-bit regular and DVB-S2 irregular LDPC decoder kernels. The instruction clustering optimization succeeded in minimizing routing complexity, thus leading to an increase of frequency in all cases. It produces a compact form of FUs and increases the locality of interconnects between FUs. Compared to nonoptimized configurations, fewer problematically long interconnect segments are generated, hence the interconnect delay is smaller and the frequency is higher.

The data path latency tends to decrease using instruction clustering. The pipelining optimization of MFUs reduces the latency compared to fully pipelined implementations. As a result, the schedule latency of the data path becomes lower. However, the schedule latency effect on the data path throughput is very small since we pipeline the loop iterations that execute on the data path. As a result, as explained earlier, for high loop iteration counts, the II target value and the attained frequency are the main parameters that determine the data path throughput.

As we observed earlier, instruction clustering may lead to an increase in the number of flip-flops. This can be mainly attributed to the fact that instruction clustering tends to increase the lifetimes of program variables. Figure 9 depicts an example of such a situation. Figure 9(a) shows a subgraph of a DFG, which includes a macro-instruction. Note that, given that macro-instructions are treated by the scheduling algorithm as a single instruction, all their operands need to be available before the macro-instruction is scheduled; in Figure 9(b), the macro-instruction is scheduled after all its operands (N0, N1, N2, and N3) are available (all scheduled at T = 0, with latency = 1). For example, N0 requires three registers to delay its value to T = 4 when N7 is scheduled. On the other hand, in Figure 9(c), where macro-instructions are not used and each instruction is scheduled individually, instruction N0 is scheduled just one cycle before its user, thus reducing its lifetime to just one cycle.

Bitwidth analysis and instruction clustering act complementarily toward the goal of reducing the area requirements and increasing the performance of FPGA implementations. When the SOpenCL tool targets $II = 1$, bitwidth analysis is most effective, whereas the effect of instruction clustering is limited or sometimes even results in area overhead. On the contrary, higher target IIs limit the benefits of bitwidth analysis but provide more opportunities for instruction clustering. In any case, Figures 7 and 8 indicate that the activation of bitwidth analysis and instruction clustering is, overall, clearly the configuration of choice, especially when taking into account the difficulties of the synthesis, placement, and routing vendor toolchain with complex designs, such as the DVB-S2 LDPC decoder.

5.3. Cross-Platform Comparison and Discussion

Figure 10 depicts the performance for simulations on the three platforms using the same OpenCL code for all platforms. The simulations decode a stream of 16 8-bit codewords, applying 30 decoder iterations per codeword, on the three simulation platforms. On the FPGA, both bitwidth analysis and instruction clustering have been applied. The execution times are broken down to buffer transfers and computations. Buffer transfers correspond to host-to-device transfers over the PCIe interface for GPUs and FPGAs.

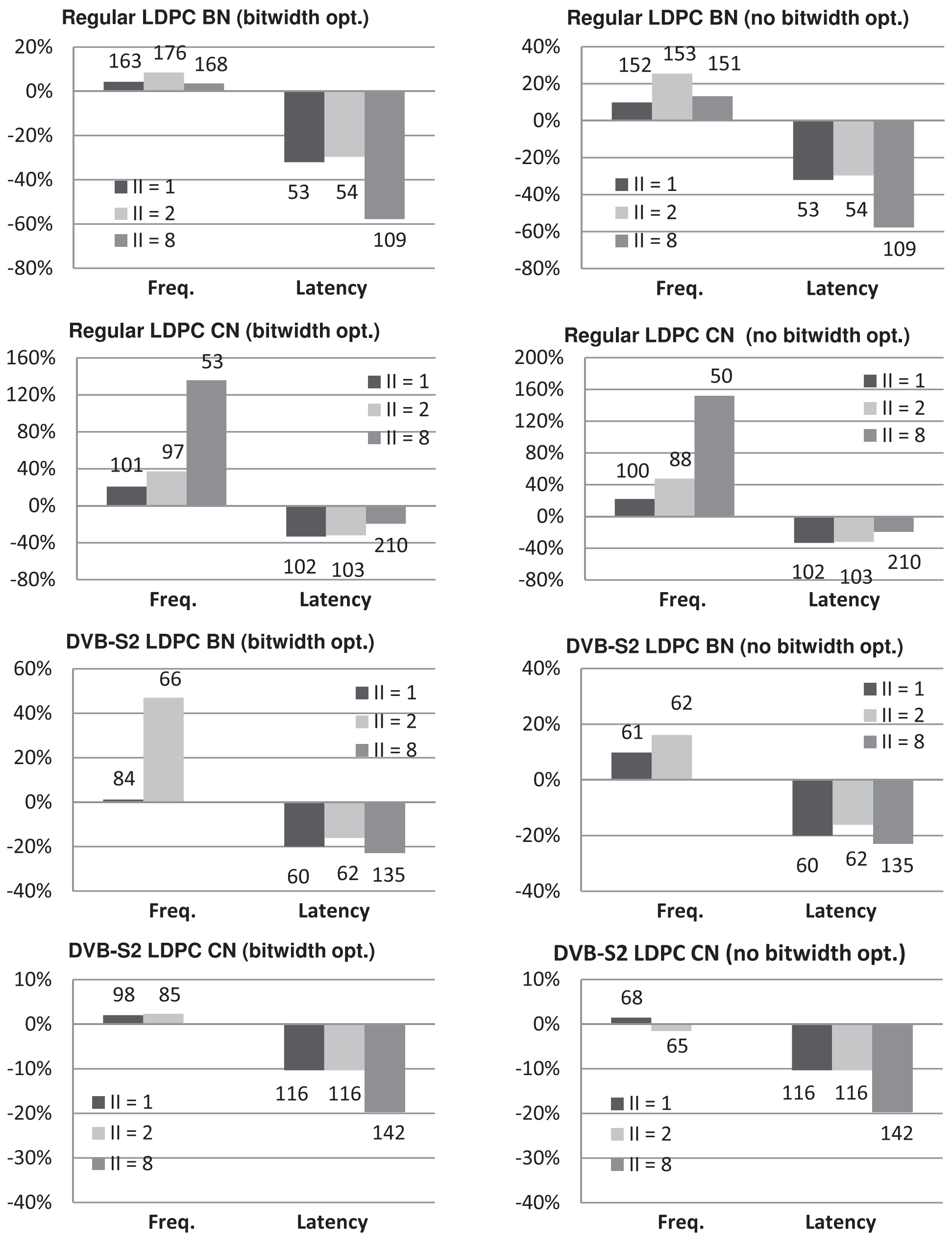

Fig. 8. Percentage of frequency (higher is better) and latency (lower is better) difference for regular and irregular DVB-S2 LDPC decoder kernels achieved by applying the instruction clustering optimization with respect to the nonoptimized versions of the benchmarks. The numbers on the bars indicate the base frequency and latency achieved by nonoptimized configurations. On the left column of diagrams, the reader can observe the effect of instruction clustering on bitwidth-optimized kernels. The diagrams on the right column quantify the effect of instruction clustering on non–bitwidth-optimized kernels.

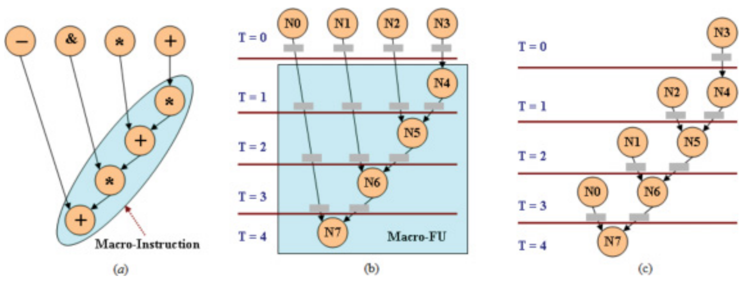

Fig. 9. Instruction clustering effect on variables lifetime. (a) Part of a DFG with one macro-instruction. (b) Scheduler output and FU implementation using instruction clustering. (c) Scheduler output and FU implementation without instruction clustering. The gray rectangles represent registers.

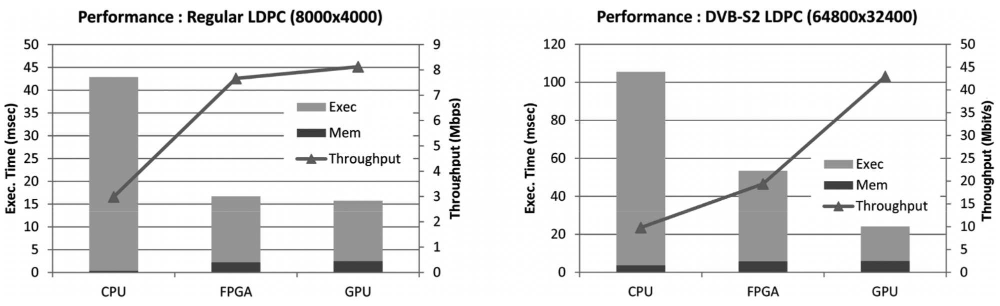

Fig. 10. Execution time (bars) and throughput (lines) on the three target simulation platforms for regular and irregular DVB-S2 LDPC decoders. For the FPGA case, results were obtained for a configuration that uses $II = 1$ and 8 bits with BW optimization.

On the CPUs, they can be reduced to simple memory copies by the vendor OpenCL runtime. It should be pointed out that the execution time of computations also includes the overhead of memory transfers between the device memory (main memory in the case of CPUs) and the execution contexts of each platform. Once again, we used both a regular LDPC code and an irregular DVB-S2 LDPC code as test cases of different nature and complexity. Since increasing the number of iterations resulted in a linear increase of the computation time in all platforms, as expected, we report and discuss the results for 30 decoding iterations. GPUs and FPGAs clearly outperform CPUs in terms of execution and throughput. Therefore, nonaccelerated CPU execution should be considered as the last resource to use in simulations for application-specific designs. GPUs proved to have in all cases the highest performing levels. Markedly, the profiled metrics show that the GPU ALUs show an SIMD packing of 63% and 89% for the CN and BN Processing, respectively, whereas the ratio of ALU to memory instructions is $\sim$30, allowing the overlap of high-latency memory transfers and compute instructions. At the same time, however, FPGAs emerge as competitive programmable accelerator platforms, especially considering that SOpenCL alleviates the development overhead typically associated with FPGA designs. It should be noted that LDPC decoding implementations on FPGAs are bandwidth limited. The memory subsystem cannot sustain the peak bandwidth required by the data path, introducing five stall cycles per access for the regular LDPC case. To overcome such penalties, a possible solution would be to exploit advanced compile-time memory access pattern analysis techniques to facilitate customized, application-specific memory subsystem designs on FPGAs. However, this undertaking is outside the scope of this article.

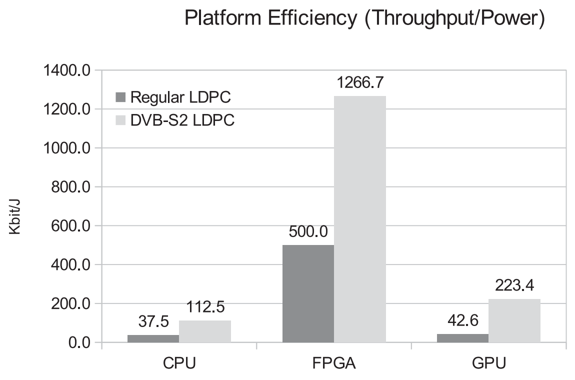

Fig. 11. Throughput versus power consumption (higher is better) for the apparatus defined in Section 5.1 for the CPU, GPU, and FPGA using regular and irregular DVB-S2 LDPC decoders as test cases.

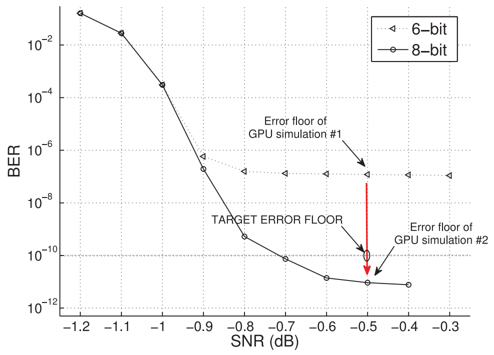

Fig. 12. BER curves simulation for 6- and 8-bit variable width precision for a given target error floor. A few quick GPU simulations at SNR = -0.5dB allow finding the necessary bitwidth precision (the GPU allows faster recompilation times than FPGAs).

Figure 11 outlines the throughput achieved by each of the target platforms considering their maximum power consumption—Thermal Design Power in the case of the CPU and GPU, and upper bound for Virtex6 [Yu and Chakrabarti 2012]. It is clear that FPGAs significantly outperform both CPUs and GPUs when this metric is used for comparison. The throughput performance versus power consumed analysis is an indicator of the capability of each platform to exploit the potential parallelism of the application in a power-efficient manner. Unfortunately, FPGA performance is limited by the low frequencies that FPGA designs typically achieve. For example, the CPU and GPU used in the experimental evaluation are clocked at 2.4 and 1.2GHz, respectively, whereas FPGA designs were limited to frequencies below 200MHz. This is due to a combination of factors beyond the complexity of each individual hardware design: limitations of the current technology at the hardware level and the degree of sophistication of synthesis, placement, and routing tools.

Other optimizations can be exploited together with the right choice of platforms and parameters for different phases of the design. For example, if a BER $< 10^{-10}$ target error floor at SNR $= -0.5$dB is given as an input parameter specification, an inspection of Figure 12 illustrates how a quick simulation performed on the GPU (where algorithmic changes are recompiled faster and do not require resynthesis, as in the FPGA case) would allow concluding that 6 bits are not enough to represent data and that at least 8 bits should be considered. Then, at different phases of the design process, other platforms could be used (FPGAs or GPUs) as accelerators to estimate the complete BER plots. This approach makes even more sense because extremely time-consuming error floors in the order of 10^{-15} are now being adopted by emergent standards, as is the case of LDPC codes from the ITU-G.709 standard for optical communications, in which each BER plot estimate can take months to compute [Smith et al. 2011].

6. CONCLUSION

In this article, we studied the use of a single, generic OpenCL source code in order to execute simulations on heterogeneous systems with diverse architectures. This approach is substantially more efficient than developing the simulation code on each platform, often using a separate programming model. More specifically, not only did we target CPUs and GPUs, but we also introduced FPGAs as a first-class choice for the acceleration of simulations without incurring the development overhead typically associated with FPGA prototyping. We used LDPC decoders as a case study, experimenting with various codes, both regular and irregular.

If coordinated appropriately, different phases of the design can either individually or concurrently exploit the particular features of distinct multicore platforms in order to accelerate the global processing of computationally intensive Monte Carlo simulations for application-specific algorithmic design. We observe that GPUs and FPGAs significantly accelerate simulation times compared to traditional methods that use CPUs. In this context, OpenCL allows code portability and competitive performance across different multicore platforms at no extra programming effort.

This strategy can be applied to other areas of VLSI system design as well. Although we analyzed the particular case of LDPC decoders used in communication systems, similar tradeoffs in balancing performance, area, and energy-efficiency usually attract the attention of hardware designers across application domains.

REFERENCES

A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. 2011. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 33–36.

B. Cope, P. Y. K. Cheung, W. Luk, and L. Howes. 2010. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on Computing* 59, 4 (2010), 433–448.

EN 302 307 V1. 1.1, European Telecommunications Standards Institute (ETSI). 2005. Digital video broadcasting (DVB); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broad-band satellite applications. (2005).

M. Eroz, F. W. Sun, and L. N. Lee. 2004. Dvb-s2 low density parity check codes with near Shannon limit performance. *International Journal of Satellite Communications and Networking* 22 (2004), 269–279.

G. Falcao, J. Andrade, V. Silva, and L. Sousa. 2011. GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection. *Electronics Letters* 47, 9 (April 2011), 542–543.

G. Falcao, V. Silva, L. Sousa, and J. Andrade. 2012. Portable LDPC decoding on multicores using OpenCL. *IEEE Signal Processing Magazine* 29, 4 (2012), 81–109.

R. G. Gallager. 1962. Low-density parity-check codes. *IRE Transactions on Information Theory* 8, 1 (1962), 21–28.

A. Gill, T. Bull, D. DePardo, A. Farmer, E. Komp, and E. Perrins. 2011. Using functional programming to generate an LDPC forward error corrector. In *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. 133–140.

H. Jin, A. Khandekar, and R. McEliece. 2000. Irregular repeat-accumulate codes. In *Proceedings of the 2nd International Symposium on Turbo Codes & Related Topics*.

V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. Cronquist, and M. Sivaraman. 2002. Pico: Automatically designing custom computers. *IEEE Computer Magazine* 35, 9 (2002), 39–47.

Group Khronos. 2010. OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems. Retrieved from http://www.khronos.org/opencl.

C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 75–86.

M. Lin, I. Lebedev, and J. Wawrzynek. 2010. OpenrCL: Low-power high performance computing with reconfigurable devices. In *Proceedings of the 2010 International Conference on Field Programmable Logic (FPL'10)*. 458–463.

J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. 1996. Swing modulo scheduling: A lifetime-sensitive approach. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'96)*. 80–90.

NVIDIA. 2007. CUDA – Compute Unified Device Architecture. Retrieved from http://www.nvidia.com/object/cuda_home_new.html.

Muhsen Owaida, Christos D. Antonopoulos, and Nikolaos Bellas. 2013. *A Grammar Induction Method for Reducing Routing Overhead in Complex FPGA Designs*. Technical Report. Department of Computer and Communication Engineering, University of Thessaly, Greece.

M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. 2011a. Massively parallel programming models used as hardware description language: The OpenCL case. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. 2011b. Synthesis of platform architectures from opencl programs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*.

A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and Wen-mei Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the 7th IEEE Symposium on Application Specific Processors*. 35–42.

Markus Rupp, Andreas Burg, and Eric Beck. 2003. Rapid prototyping for wireless designs: The five-ones approach. *Signal Processing* 83, 7 (2003), 1427–1444.

B. Smith, A. Farhood, A. Hunt, F. Kschischang, and J. Lodge. 2011. Staircase codes: FEC for 100 Gb/s OTN. *IEEE/OSA Lightwave Technology* PP, 99 (2011), 1.

M. Stephenson, J. Babb, and A. Amarasinghe. 2000. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*.

R. Weber, A Gothandaraman, R. J. Hinde, and G. D. Peterson. 2011. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 58–68.

S. B. Wicker and S. Kim. 2003. *Fundamentals of Codes, Graphs, and Iterative Decoding*. Kluwer Academic Publishers.

Chi-Li Yu and C. Chakrabarti. 2012. Transpose-free sar imaging on fpga platform. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'12)*. 762–765. DOI:http://dx.doi.org/10.1109/ISCAS.2012.6272149

Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. 2008. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Netherlands, Chapter AutoPilot: A Platform-Based ESL Synthesis System.

Received April 2013; revised January 2014; accepted September 2014