2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines

Shortening design time through multiplatform simulations with a portable OpenCL golden-model: the LDPC decoder case

G. Falcao, M. Owaida*, D. Novo†, M. Purnaprajna†, N. Bellas*, C. D. Antonopoulos*, G. Karakonstantis†, A. Burg† and P. Ienne†

Instituto de Telecomunicações, Department of Electrical and Computer Engineering, University of Coimbra, Portugal
**Department of Computer and Communications Engineering, University of Thessaly Volos, Greece*
†École Polytechnique Fédérale de Lausanne, Switzerland

***Abstract*—Hardware designers and engineers typically need to explore a multi-parametric design space in order to find the best configuration for their designs using simulations that can take weeks to months to complete. For example, designers of special purpose chips need to explore parameters such as the optimal bitwidth and data representation. This is the case for the development of complex algorithms such as Low-Density Parity-Check (LDPC) decoders used in modern communication systems. Currently, high-performance computing offers a wide set of acceleration options, that range from multicore CPUs to graphics processing units (GPUs) and FPGAs. Depending on the simulation requirements, the ideal architecture to use can vary. In this paper we propose a new design flow based on OpenCL, a unified multiplatform programming model, which accelerates LDPC decoding simulations, thereby significantly reducing architectural exploration and design time. OpenCL-based parallel kernels are 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 for mapping the simulations into FPGAs. To the best of our knowledge, this is the first time that a single, unmodified OpenCL code is used to target those three different platforms. We show that, depending on the design parameters to be explored in the simulation, on the dimension and phase of the design, the GPU or the FPGA may suit different purposes more conveniently, providing different acceleration factors. For example, although simulations can typically execute more than 3× faster on FPGAs than on GPUs, the overhead of circuit synthesis often outweighs the benefits of FPGA-accelerated execution.**

***Keywords*-design space exploration; simulation tools; parallel computing; FPGAs; GPUs; LDPC decoding;**

I. Introduction

Modern communication systems rely on a concatenation of many complex signal processing tasks and blocks that must be optimized carefully to balance the complexity-performance tradeoff that governs the system design and implementation process. Hence, wireless system design traditionally relies heavily on thousands of Monte Carlo simulations to properly capture the performance under variable channel and working conditions. Extensive design space exploration typically requires repeating such simulations for various algorithms, design variables and hardware architectures for each block of the system. Consequently, the design time increases rapidly which is incompatible with the tight product deadlines, forcing hardware designers to accept pragmatic, but potentially highly suboptimal solutions that do not provide optimal performance, cost or energy-efficiency. Therefore, there is a strong need for an exploration toolset that can accelerate the simulation and exploration time to enable a better design space exploration for such systems.

Over the last years various simulation platforms (CPUs, FPGAs and GPUs) have been used to approach this objective [1], [2]. Each of these platforms have different capabilities: they either provide a relatively simple programming model to enable rapid design space exploration [1], but with limited speedup or they accelerate the simulation time significantly with often considerable effort required for rapid-prototyping and mapping [2]. To bridge this gap, there is a need for a unified programming model that allows exploring the capabilities of the various platforms. It should help designers to take optimal decisions with rapid turn-around time early in the design flow to be confirmed and refined by more in-depth, but also more time consuming evaluations later in the design process [3].

Toward this goal, this paper studies the use of different simulation environments. In our exploration we use the example of the Forward Error Correction (FEC) subsystem which is one of the most computationally intensive and widely researched system components. By using the decoder for Low-Density Parity-Check (LDPC) codes [4] as case study, we exploit different parallel computing platforms (CPUs, FPGAs and GPUs) for simulating different combinations of parameters such as bitwidth, number of iterations, data structures and algorithmic variations that are critical in the design space exploration and performance analysis of this subsystem. Similar principles can be applied in the design of other hardware systems that naturally are not limited to telecommunications.

Unfortunately, each one of these platforms is naturally supported by distinct programming models which requires different skills from system-designers only for the purpose of setting up simulations. In this context, this paper utilizes OpenCL [5], that has emerged as an open computing language supported by some of the most important computer manufacturers. OpenCL allows developing parallel kernels

978-0-7695-4699-5/12 $26.00 © 2012 IEEE
DOI 10.1109/FCCM.2012.46

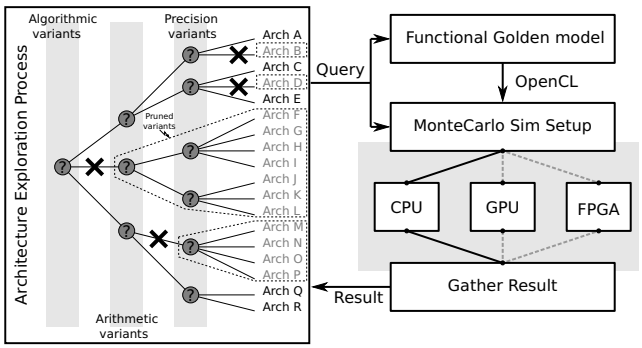

Figure 1: Proposed flow to shorten VLSI design time through multiplatform simulation with a portable OpenCL golden-model.

that are portable across several multicore platforms and that permit achieving good cross-platform performance levels [5]. Recent extensions to OpenCL also allow to transparently mapping algorithms to FPGAs [6]. This motivates the choice made in the paper to adopt such common and generic programming model to support simulation setups for a variety of parallel computing platforms.

The main contributions of the paper can be summarized as follows: We propose *i)* a multiplatform framework for accelerating telecommunication system's simulations and help the hardware designer making decisions earlier in the design cycle. We show that *ii)* OpenCL can be used as a common programming model for developing parallel kernels and we originally propose to execute a single kernel on multicore CPUs, GPUs and FPGAs without code readjustment or hand tuning across different parallel platforms. In the paper we *iii)* assess compilation/synthesis and execution performances on state-of-the-art parallel computer architectures and *iv)* given the capability to easily retarget simulation code, we compare and quantify the different platforms (in terms of speed ups) providing guidelines for the most adequate choices regarding the different stages of the process design as depicted in Figure 1.

II. Background and motivation

Computationally intensive Monte Carlo simulations require methods for acceleration that can be generic and easy to incorporate. Using application-specific acceleration, such as designing custom circuits (ASICs) or application-specific instruction set processors (ASIPs) is not feasible in this domain because of the efforts 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 a GPU is considerably faster than on an FPGA, where development still requires using Hardware Description Languages (HDLs). In such a scenario, it becomes important to be able of quickly retarget a given application with a single specification language.

OpenCL [5] is a framework for programming heterogeneous systems that may comprise conventional chip multiprocessors such as CPUs, GPUs and various other forms of accelerators such as FPGAs. An OpenCL application consists of a host program and a number of kernel functions. The host part executes on the host processor and submits commands that can refer either to compilation and execution of a kernel function or to manipulation of memory objects, to name a few. 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.

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

III. LDPC decoding: case study on intensive simulation

LDPC codes are used in communication systems such as optical (ITU-T G.709) and satellite communications (DVB-S2) or metropolitan area networks (WiMAX). They are linear block codes (N, K) that allow achieving excellent Bit Error Rates (BER) [4] under different channel working conditions. LDPC codes can be described by a binary **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 system of equations [7] represented by matrix **H** as illustrated in Figure 2. In the design of such systems the objective is that performance meets specific BER and throughput for various channel conditions as specified by the target standard.

A. Belief propagation, message-passing and the MSA

The Min-sum algorithm (MSA) consists of a simplification of the Sum-Product algorithm (SPA) [7] and it is depicted in Algorithm 1. Lp_n designates the *a priori* LLR of BN_n received from the channel 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 (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 (3) and (4) and the iterative procedure is stopped if the decoded word

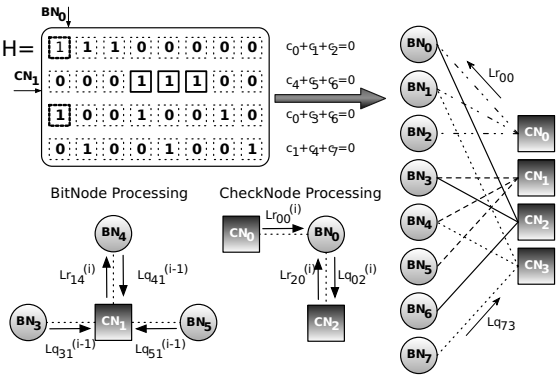

Figure 2: Tanner graph

$\hat{\mathbf{c}}$ 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.

B. Defining fundamental simulation metrics

In order to decide the optimal configuration parameters that lead to best area, performance and energy trade offs, several Monte Carlo simulations are typically used (the work herein proposed only analyzes the LDPC decoder). The configuration metrics to manipuate are briefly mentioned below:

1) LDPC code – H matrix: The data structures that define the LDPC code are imported from an $\mathbf{H}$ 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.

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

3) Number of iterations: Another metric commonly tested in this kind of applications is the number of iterations performed. Here simulation time increases linearly with this parameter. This metric has direct application in the simulation of BER curves, which are fundamental to prove that the design is compliant with the LDPC code requirements defined either by a standard or the client.

4) Bitwidth: Bitwidth definitions are among the most important parameters to decide on the design of a chip because they impact the width of the datapath and the dimension of memory blocks and usually have a correspondence with area and power consumption. On the other hand they should also provide enough BER performance. When performing simulations, normally designers dedicate great importance to this fixed-point optimization phase. In the present case, bitwidth usually ranges from 5- to 8-bit.

Algorithm 1 Min-sum algorithm

1: {Initialization} $Lq_{nm}^{(0)} = Lp_n$;
2: **while** ($\mathbf{H}\hat{\mathbf{c}}^T \neq \mathbf{0} \wedge \mathrm{i} < \mathrm{I}$) {c – decoded word; I – max. # of iterations.}

do

3: {For all node pairs (BN_n, CN_m), where $\mathbf{H_{mn}} = \mathbf{1}$ **do**:}
4: {Compute the LLR of messages sent from CN_m to BN_n:}
(CN Processing)

$$Lr_{mn}^{(i)} = \prod_{\mathrm{n'} \in \mathcal{N}(\mathrm{m}) \backslash \mathrm{n}} sign\left(Lq_{n'm}^{(i-1)}\right) \min_{\mathrm{n'} \in \mathcal{N}(\mathrm{m}) \backslash \mathrm{n}} \left|Lq_{n'm}^{(i-1)}\right| \quad (1)$$

{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)

$$Lq_{nm}^{(i)} = Lp_n + \sum_{m' \in \mathcal{M}(n) \backslash m} Lr_{m'n}^{(i)} \quad (2)$$

{where $\mathcal{M}(n)\backslash m$ represents connect. to BN_n excluding CN_m.}
3. 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:} $\forall n$,

$$\hat{\mathbf{c}}_n^{(i)} = (LQ_n^{(i)} > 0 \quad ? \quad 0 \quad : \quad 1) \quad (4)$$

7: **end while**

IV. Unified high-performance accelerators

For a case study, we developed a single OpenCL representation that can be executed, unmodified, on three distinct platforms: CPUs, GPUs and FPGAs. In other words, an OpenCL software developer or domain expert is able to quickly develop, map, evaluate and optimize an application without specific knowledge of the underlying architecture.

A. Multicore CPUs and GPUs

Once the computational resources are known, the LDPC decoder kernel's workload is partitioned into work-groups, each one of them launching a certain number of work-items in parallel. Both are determined at runtime and are a function of the platform context query supported by OpenCL. The kernel is compiled according to this information and launched for execution in the OpenCL device. Regarding the granularity-level of parallelism adopted for the LDPC decoder, each work-item processes the complete update of a single node of the Tanner graph. Finer- or coarser-granularity levels can penalize throughput performance. Finer-grain activity performs redundant memory accesses, while coarser-grain levels of parallelism do not allow to fully exploit the resources of multicore systems that have a high number of compute units (e.g., GPUs) for processing small to medium sized data sets.

B. FPGAs

1) SOpenCL background: We used the SOpenCL tool [6] to automatically generate hardware accelerators starting from an OpenCL LDPC decoder code. SOpenCL allows to quickly explore different architectural scenarios and evaluate the quality of the design in terms of computational bandwidth, clock frequency and size.

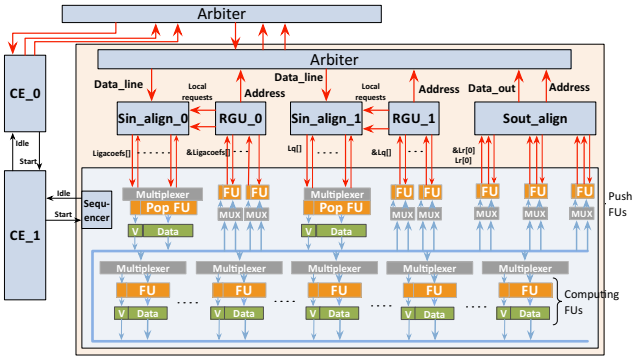

Figure 3: Automatically generated hardware accelerator for the CN Kernel. The datapath at the bottom of the block diagram is used to execute kernel computations and generate addresses for Request Generation Units (RGUs). RGUs are used to coalesce incoming address requests and to interface to the memory system of the FPGA.

2) SOpenCL Front End: The SOpenCL front end adjusts parallelism granularity of the OpenCL kernel to better match the hardware capabilities of the FPGA. OpenCL kernel code specifies computation at a work-item granularity. A straightforward approach would map a work-item to an invocation of the hardware accelerator. This approach is suboptimal for FPGAs which incur heavy overhead to initiate thousands of work-items of fine granularity.

Therefore, SOpenCL applies a series of source-to-source transformations that collectively aim at coarsening the granularity of a kernel function at a work-group level.

3) SOpenCL Back End: After the front-end OpenCL to C transformation, the back-end flow generates the synthesizable HDL of LDPC decoder accelerators. The functionality of the LLVM compiler infrastructure [8] supports *bitwidth optimization* [9], *predication* and *modulo scheduling* [10] as separate compilation passes. Then, the compiler back-end generates the final hardware modules of the LDPC decoder application-specific architecture.

Bitwidth optimization is used to minimize the number of bits required to represent each operand [9]. Experimental evaluation on LDPC decoding kernels shows significant area and performance improvement due to bitwidth optimizations (see section V-B).

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

Swing Modulo Scheduling (SMS) [10] is used to generate a schedule for the kernel. The scheduler identifies an iterative pattern of instructions and their assignment to 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), which is the constant interval between launches of successive work-items. Lower values of Initiation Interval correspond to higher throughput since more work-items are initiated and, therefore, more results are produced per cycle. That makes Initiation Interval the main factor affecting computational bandwidth in modulo scheduled loop code.

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, denoting FPGA device characteristics.

4) Accelerator architecture: Given the modulo-scheduled loop kernels, the compiler backend generates modular Verilog for the steady state body of the kernels as depicted in Figure 3 for the CN kernel.

The input stream Alignment Unit, *Sin_Align*, retrieves incoming data, and presents them in-order to the data path. The output stream Alignment Unit aligns the output data tokens coming from the data path in a FIFO of data-lines of *bus-width* bytes. 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.

Besides generating memory addresses for I/O, the data path executes the computational path of the algorithm. The reconfigurable parameters of the data path are the type and bitwidth of functional units (ALUs for arithmetic and logical instructions, shifters, etc.), the custom operation performed within a generic functional unit (e.g., only addition or subtraction for an ALU), the number and size of registers in the queues between functional units, and the bandwidth to and from the streaming unit.

Finally, Control Elements (CEs) are used to control and execute code of outer loops in a multilevel loop nest. CEs have a simpler, less optimized architecture, since outer loop code does not execute as frequently as inner loop code.

5) Memory System: It is crucial that the memory subsystem provides the accelerator with the necessary bandwidth to keep the data path from stalling. For $II = 1$, the CN accelerator requires 120 bytes for 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. In this case, instructions from 106 contiguous loop iterations are executed concurrently in the data path, requiring 392 adders, 210 shifters, 369 logic units, and 434 comparators, as well as 994 1-bit logic units for predicate manipulation.

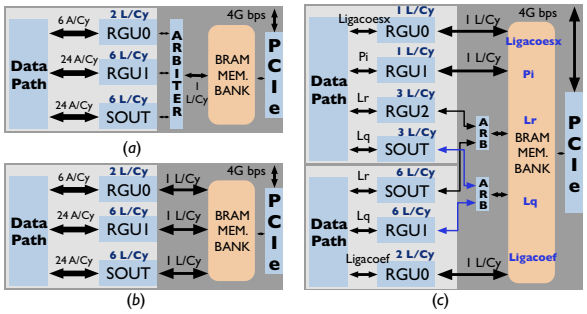

Figure 4: System level block diagram of LDPC accelerators generated by SOpenCL; with a) CN (or BN) kernel communicating through a single port to BRAM b) through three ports to the BRAMs one for each I/O stream; and (c) both kernels are instantiated and interconnected.

SOpenCL allowed to investigate the use of two different memory systems in Figure 4, where a PCIe interface is used for data transfers between the host and on-chip SRAMs. 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 4a) the memory bank is configured as a unified single port memory system, while Figure 4b) shows the memory bank configured as a distributed memory system. Figure 4c) 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 4 shows the throughput required by the data path for $II = 1$, and 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 14 cycles for each computation/address generation cycle. In the distributed memory system, 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 should provide a wider data bus, 96 bytes to L_r and L_q streams and 64 bytes to $ligacoef$ stream.

V. Experimental Evaluation

In this section, we evaluate the performance of LDPC decoders on the three platforms described in section IV. It should be emphasized that a single OpenCL code was used. No platform-specific source-level optimizations were performed to the OpenCL code, such as manual vectorization, or explicit data prefetching, which provides a more fair comparison for the various platforms.

A. Methodology

The OpenCL LDPC decoder kernel was executed on an AMD Phenom X4 945 QuadCore CPU system running at 3 GHz, with 4 GB of DDR3. The CPU executable has been generated with g++ 4.4.

We have also executed the OpenCL LDPC decoding application on an ATI Radeon HD 5870 GPU running at 1.2 GHz, with 3 GB DDR5. This GPU, which follows the Evergreen GPU architecture and is equipped with a conventional L1-L2 cache memory hierarchy, has a peak performance of 2720 Single Precision GFLOPS and 1600 usable stream cores [11].

Finally, 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. The first scenario assumes that a new work-item is scheduled in every clock cycle, i.e. initiation interval $II = 1$. In this case, each LLVM instruction is mapped to its own dedicated functional unit. 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.

Custom hardware synthesis benefits from aggressive bitwidth analysis. We experimentally tested three different code versions, assuming input data (codeword elements) represented with 5-, 6- and 8-bit, 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-bit is emulated in the source code by explicit masking off extraneous bits.

For the evaluation of the FPGA design we used Xilinx Virtex-6 LX760 and Xilinx ISE 12.4 toolset for synthesis, placement and routing. LX760 contains 118560 slices and each slice includes four LUTs and eight flip-flops.

Two different LDPC codes $(1024, 512)$ and $(8000, 6000)$ are profiled, each running for 10, 20 and 30 iterations. Each iteration calls the CN kernel followed by a call to the BN kernel. Each CN kernel invocation spawns $N - K$ work-items, and each BN kernel invocation spawns N work-items.

B. FPGA Results

Tables I and II detail performance and area results of the two LDPC kernels implemented on a Virtex-6 LX760 FPGA. Area costs are minimized when $II = 1$, which seems counter-intuitive since this configuration requires more resources for each functional unit. The LDPC decoder kernel code consists mainly of simple operations (add, shift, logic) between a variable and a constant. Assigning dedicated functional units for 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

Table I: Comparing CN kernels area for different $II = \{1, 2, 8\}$ architecture configurations using variable bitwidth precision with 5-, 6- and 8-bits and a generic on-the-fly bit precision selection approach.

	CS	Slices	Flip-Flops	LUTs	Freq. (MHz)	Latency (cycles)
II=1	8 (no BW opt.)	12061	42718	39594	100	102
	8	11600	41892	38759	101	102
	6	11647	35948	33914	103	106
	5	10369	33639	32861	107	106
	Generic	24108	101960	80115	91	106
II=2	8 (no BW opt.)	25453	64311	92096	88	103
	8	21424	54872	81526	97	103
	6	23632	61035	78884	95	110
	5	19374	61052	65192	88	110
	Generic	28432	67307	73212	63	110
II=8	8 (no BW opt.)	33213	54749	78266	50	210
	8	27556	57582	58788	53	210
	6	27008	56745	64104	50	231
	5	26894	54868	64083	51	231
	Generic	36954	58121	79682	51	231

Table II: Comparing BN kernels area for different $II = \{1, 2, 8\}$ architecture configurations using variable bitwidth precision with 5-, 6- and 8-bits and a generic on-the-fly bit precision selection approach.

	CS	Slices	Flip-Flops	LUTs	Freq. (MHz)	Latency (cycles)
II=1	8 (no BW opt.)	7681	28026	25823	152	53
	8	6466	19584	18433	163	53
	6	5891	17746	17001	175	57
	5	5515	16132	16509	182	57
	Generic	10572	35865	37056	164	61
II=2	8 (no BW opt.)	7134	24332	23482	153	54
	8	6201	18246	17957	176	54
	6	5996	17663	17385	171	58
	5	5665	17269	17077	166	58
	Generic	8226	27190	27891	164	62
II=8	8 (no BW opt.)	8631	20592	22633	151	109
	8	6747	16791	17983	168	109
	6	7032	17524	18697	163	120
	5	6731	17227	18384	172	120
	Generic	9963	23946	26683	132	127

reduce area. When $II > 1$ this opportunity no longer exists, since 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) require more resources than a bitwidth of 8. Our analysis shows that with larger values, the fewer FUs allocated will nearly have similar sizes to serve a population of instructions with various bitwidths (from 5 to 32-bit). This will reduce the gain from custom bitwidths, because the tool necessarily moves towards a larger, more generic FU size with larger II. Finally, the following set of operations are widely used in the code:

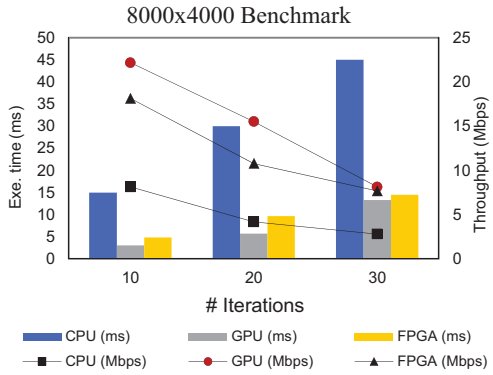

Figure 5: Execution and throughput benchmark results decoding in simultaneous 16 codewords of an (8000, 4000) LDPC code show that the GPU outperforms the FPGA for 30 or less iterations.

```
(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 masking operation for 8-bit because it is not necessary, but those operations remained for 6- and 5-bits kernels. This led to 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 with 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 routability. 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.

C. Crossplatform comparison and discussion

GPUs and FPGAs clearly outperform CPU execution in terms of throughput, as shown in Figure 5 and Figure 6. CPUs remain as the last resource to use in simulations for application-specific designs, when all the others somehow fail to become accessible. Figure 6 shows that for a $(1024, 512)$ code the FPGA is always faster than CPUs and GPUs, but the same does not happen for the larger design, i.e., for code $(8000, 4000)$ shown in Figure 5. In this case, memory accesses cannot sustain peak bandwidth, introducing 5 stall cycles. GPUs are better suited for an early design stage where algorithmic development constantly requires recompilation/resynthesis of the kernel. Figure 7 shows that at this level FPGAs always require much higher synthesis

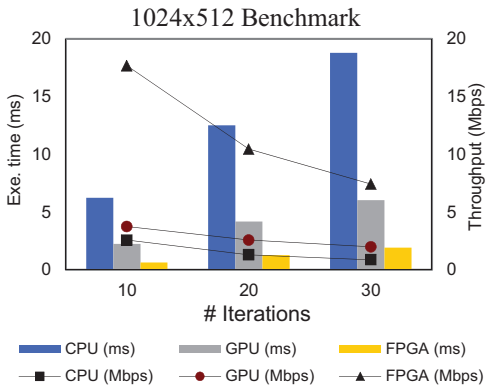

Figure 6: Execution and throughput benchmark results decoding in simultaneous 16 codewords of an (1024, 512) LDPC code show that the FPGA outperforms the GPU.

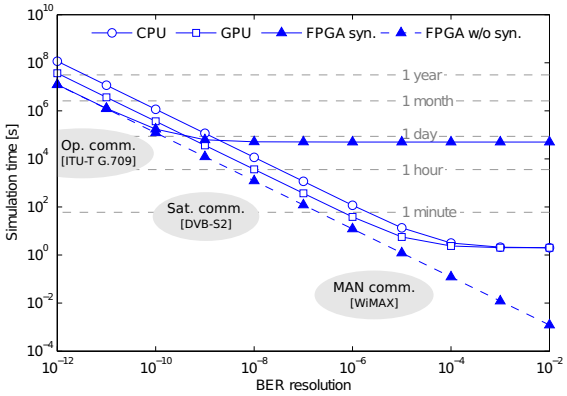

Figure 7: Monte Carlo simulation time as function of a desired error floor for one BER estimate. FPGAs can take 14 hours to synthesize an LDPC decoder design. For this reason, they are better suited for a later stage of the design where the algorithm is stabilized and simulations can run upto 3 times faster than on GPUs.

time compared to compilation time on GPUs. However, at a later phase of the design, when the algorithm is well-defined and stabilized, parameterized FPGA kernels (represented in the *Generic* rows of Tables I and II) can perform better upto a certain dimension of the design. Although they allow more flexibility and may eliminate unnecessary resynthesis iterations, programming Monte Carlo simulations becomes more complex as we add more input parameters to the kernel. Also, parameterized kernels occupy more FPGA resources than fix mode parameters, which in this case were developed for 5-, 6- and 8-bit data representations.

To overcome such penalties imposed by BRAM bandwidth limitations (the FPGA-based approach is bandwidth limited), a possible solution would consist of using boards that supply more than one FPGA or even to adopt FPGA clusters. Naturally, it would also be possible and desirable to use GPU clusters as a solution to improve this type of simulations. The scope of this paper seeks the comparison of both approaches and for making a fair comparison we have decided to compare only one element of each.

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 is given as an input parameter specification, the inspection of Figure 8 shows that a quick simulation at $SNR = -0.5$dB performed on the GPU (where algorithmic changes are recompiled fast) would allow to conclude that 6-bit are not enough to represent data and that at least 8-bit should be considered. After we have a match on the target error floor, we could consider FPGAs to perform the complete BER plots. This approach makes even more sense as extremely time consuming error floors in the order of 10^{-15} are now being adopted by new standards, as it is the case of LDPC codes from the ITU-G.709 [2] standard for optical communications depicted in Figure 7 where each BER plot estimate can take months to compute.

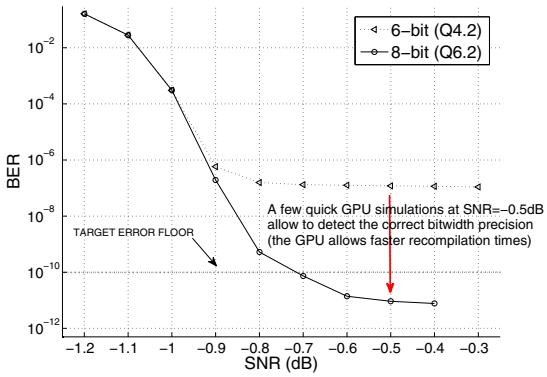

Figure 8: BER curves simulation for 6- and 8-bit variable width precision for a given target error floor.

VI. Related Work

Simulation programs are typically in the software domain. For targeting FPGAs, hardware accelerators need to be extracted from this domain. Methods following this direction have exploited high-level language to hardware, or C-to-HDL translations. The PICO-NPA system translates C functions written as perfectly nested loops into a systolic array of accelerators [12]. The LegUp synthesis tool generates a hybrid architecture comprising a MIPS processor and hardware accelerators to speed up performance critical C code [13]. 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 [14]. OpenRCL does not generate customized hardware accelerators, although each MIPS core

can be configured to match application characteristics. The AutoPilot Compiler [15] 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 [16] is an initiative that retargets CUDA kernels to synthesizable hardware [16] in FPGAs. FCUDA transforms a CUDA kernel into a C function annotated with AutoPilot directives, and then use AutoPilot to generate synthesizable HDL.

Also, recent publications propose using GPUs to perform LDPC decoding [1] or functional programming to target LDPC codes in FPGAs [17], but still none of these approaches provide a unique solution that is suitable to target at the same time CPU, GPU and FPGA architectures. In this paper, our objective is to simplify the exploration of all three target architectures using a single unified programming model that allows extracting the most interesting properties of each. In fact, a single application description proves to be efficient for code modifications, retargetability according to performance, and universal applicability.

VII. Conclusion

In this paper we show that the development of single OpenCL golden-models can generically address different multicore architectures, which is substantially more efficient than the individual programming of CPUs, GPUs and FPGAs. If coordinated appropriately, different phases of the design can exploit more conveniently 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 show that depending on the complexity of the algorithm, the nature of parameters to simulate and phase of the design, GPUs and FPGAs suit different purposes more conveniently, while at the same time they significantly accelerate simulation times compared to traditional methods that use CPUs. In this context OpenCL allows code portability across different multicore platforms at no extra programming effort or particular need of code hand tuning intervention.

This strategy can be extended to other areas of VLSI system design. Although we analyze the particular case of LDPC decoders used in communication systems, similar concerns related with performance, area and energy-efficiency usually hold the attention of hardware designers every time they start a new project.

References

[1] G. Falcao, J. Andrade, V. Silva, and L. Sousa, "GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection," *Electronics Letters*, vol. 47, no. 9, pp. 542–543, April 2011.

[2] B. Smith, A. Farhood, A. Hunt, F. Kschischang, and J. Lodge, "Staircase Codes: FEC for 100 Gb/s OTN," *IEEE/OSA Lightwave Technology*, vol. PP, no. 99, p. 1, 2011.

[3] M. Rupp, A. Burg, and E. Beck, "Rapid prototyping for wireless designs: the five-ones approach," *Signal Processing*, vol. 83, no. 7, pp. 1427–1444, 2003.

[4] R. G. Gallager, "Low-Density Parity-Check Codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[5] K. Group, "OpenCL – The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl/, 2010.

[6] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011.

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

[8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis Transformation." in *International Symposium on Code Generation and Optimization (CGO)*, March 2004, pp. 75–86.

[9] M. Stephenson, J. Babb, and A. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2000.

[10] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero, "Swing Modulo Scheduling: A Lifetime-Sensitive Approach," in *Conference on Parallel Architectures and Compilation Techniques (PACT)*, June 1996, pp. 80–90.

[11] Advanced Micro Device, "Heterogeneous Computing: OpenCL and the ATI Radeon HD 5870 (Evergreen) Architecture," available from http://developer.amd.com.

[12] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. Cronquist, and M. Sivaraman, "PICO: Automatically Designing Custom Computers," *IEEE Computer Magazine*, vol. 35, no. 9, pp. 39–47, 2002.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field programmable gate arrays*, 2011, pp. 33–36.

[14] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: Low-Power High Performance Computing with Reconfigurable Devices," in *Proceedings of the 2010 International Conference on Field Programmable Logic (FPL)*, Sptember 2010, pp. 458–463.

[15] Z. Zhang and et al., *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Nettherlands, 2008, ch. AutoPilot: A Platform-Based ESL Synthesis System.

[16] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-m. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto fpgas," in *Proceedings of the 7th IEEE Symposium on Application Specific Processors*, 2009, pp. 35–42.

[17] A. Gill, T. Bull, D. DePardo, A. Farmer, E. Komp, and E. Perrins, "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*, 2011, pp. 133–140.