INVITED PAPER

High-Level Language Tools for Reconfigurable Computing

This paper provides a focused survey of five tools to improve productivity in developing code for FPGAs.

By Skyler Windh, Xiaoyin Ma, *Student Member IEEE*, Robert J. Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A. Najjar, *Fellow IEEE*

ABSTRACT | In the past decade or so we have witnessed a steadily increasing interest in FPGAs as hardware accelerators: they provide an excellent mid-point between the reprogrammability of software devices (CPUs, DSPs, and GPUs) and the performance and low energy consumption of ASICs. However, the programmability of FPGA-based accelerators remains one of the biggest obstacles to their wider adoption. Developing FPGA programs requires extensive familiarity with hardware design and experience with a tedious and complex tool chain. For half a century, layers of abstractions have been developed that simplify the software development process: languages, compilers, dynamically linked libraries, operating systems, APIs, etc. Very little, if any, such abstractions exist in the development of FPGA programs. In this paper, we review the history of using FPGAs as hardware accelerators and summarize the challenges facing the raising of the programming abstraction layers. We survey five High-Level Language tools for the development of FPGA programs: Xilinx Vivado, Altera OpenCL, BluespecBSV, ROCCC, and LegUp to provide an overview of their tool flow, the optimizations they provide, and a qualitative analysis of their hardware implementations of high level code.

KEYWORDS | Compiler optimization; high level synthesis; max filter; reconfigurable computing

I. INTRODUCTION

In recent years we have witnessed a tremendous growth in size and speed of FPGAs accompanied by an ever widening spectrum of application domains where they are extensively used. Furthermore, a large number of specialized functional units are being added to their architectures such as DSP units, multi-ported on-chip memories, and CPUs. Modern FPGAs are used as platforms for configurable computing that combine the flexibility and reprogramability of CPUs with the efficiency of ASICs. Commercial as well as research projects using FPGA accelerators on a wide variety of applications have reported speed-up, over both CPUs and GPUs, ranging from one to three orders of magnitude as well as reduced energy consumption per result ranging from one to two orders of magnitude. Application domains have included signal, image and video processing, encryption/decryption, decompression (text, integer data, images, etc.), data bases [1], [2], dense and sparse linear algebra, graph algorithm, data mining, information processing and text analysis, packet processing, intrusion detection, bioinformatics, financial analysis, seismic data analysis, etc.

FPGAs are programmed using Hardware Description Languages (HDLs) such as VHDL, Verilog, SystemC, and SystemVerilog that are used for digital circuit design and implementation. In these languages the circuit to be mapped on an FPGA is designed at a fairly low level: the data paths and state machine controllers are built from the bottom up, timing primitives are used to provide synchronization between signals, the registering of data values is explicitly stated, parallelism is implicit and sequential ordering of events must be explicitly enforced via the state machine. Traditionally trained software developers are not familiar with such programming paradigms. Beyond the program development state, the tool chains are language and vendor specific and consist of a synthesis phase where the HDL code is translated to a netlist, mapping where logic expressions are translated into hardware primitives specific to a device, place and route where hardware logic blocks are placed on the device and wires routed to connect them. This last phase attempts to solve an NP-complete problem using heuristics, such as simulated annealing, and may take hours or days to complete depending

Manuscript received August 21, 2014; revised December 2, 2014; accepted January 20, 2015. Date of current version April 14, 2015. This work has been supported in part by National Science Foundation AwardsCCF-1219180 and IIS-1161997.
S. Windh, R. Halstead, P. Budhkar, Z. Luna, O. Hussaini and **W. A. Najjar** are with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92521 USA (e-mail: najjar@cs.ucr.edu).
X. Ma is with the Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA 92521 USA.

Digital Object Identifier: 10.1109/JPROC.2015.2399275

on the size of the circuit relative to the device size as well as the timing constraints imposed by the user. The steepness of the learning curve for such languages and tools makes their use a daunting and expensive proposition for most projects.

This paper provides a qualitative survey of the currently available tools, both research and commercial ones, for programming FPGAs as hardware accelerators. We start with a historical prospective on the use of FPGA-based hardware accelerators (Section II) showing that the role of FPGAs as accelerators emerged very shorty after their introduction, as glue-logic replacements, in the 1980s. In Section III we discuss the efficiency of the hardware computing model over the stored program model and review the challenges posed by using High-Level Languages (HLLs) as programming tools to generate hardware structures on FPGAs. Related works and five High-Level Synthesis (HLS) tools are described in Section IV, three commercial tools: Xilinx Vivado, Altera OpenCL, Bluespec BSV, and two university research tools: ROCCC and LegUp. We use a simple image filter, dilation, and AES encryption routines to describe the style of programming these tools and explore their capabilities in implementing compiler-based transformations that enhance the throughput of the generated structure (Sections V and VI). Area, performance and power results for both benchmarks are compared in Section VII and, finally, concluding remarks are presented in Section VIII. Note: in this paper we report results and compare tools only to the extent allowed by the terms of the user license agreements.

II. A HISTORICAL PERSPECTIVE

The use of FPGAs as hardware accelerators is not a new concept. Very shortly after the introduction of the first SRAM-based FPGA device (Xilinx, 1985) the PAM (Programmable Active Memory) [3], [4] was conceived and built at the DEC Paris Research Lab (PReL). The PAM P_0 consists of a 5×5 array of Xilinx XC3020 FPGAs (Fig. 1) connected to various memory modules as well as to a host workstation via a VME bus. It had a maximum frequency of 25 MHz, 0.5 MB of RAM, and communicated on a host bus of 8 MB/s. The PAM P_1 was built using slightly newer FPGA, the Xilinx XC3090. It operated with a maximum frequency of 40 MHz, 4 MB of RAM, and used a 100 MB/s host bus. It was described as "universal hardware coprocessor closely coupled to a standard host computer" [5]. It was evaluated using 10 benchmark codes [5] consisting of: long multiplication, RSA cryptography, Ziv-Lempel compression, edit distance calculations, heat and Laplace equations, N-body calculations, binary 2-D convolution, Boltzman machine model, 3-D graphics (including translation, rotation, clipping, and perspective projection) and discrete cosine transform. It is interesting to note that most of these benchmarks are still today subjects of research and development efforts in hardware acceleration. Berlin *et al.* in [5] conclude that PAM delivered a performance comparable to that of ASIC chips or supercomputers, of the time, and was one to two orders of magnitude faster than software. They also state that because of the PAM's large off-chip I/O bandwidth (6.4 Gb/s) it was ideally suited for "on-the-fly data acquisition and filtering," which is exactly the computational model, streaming data, adopted by most of the hardware acceleration projects that rely on FPGA platforms.

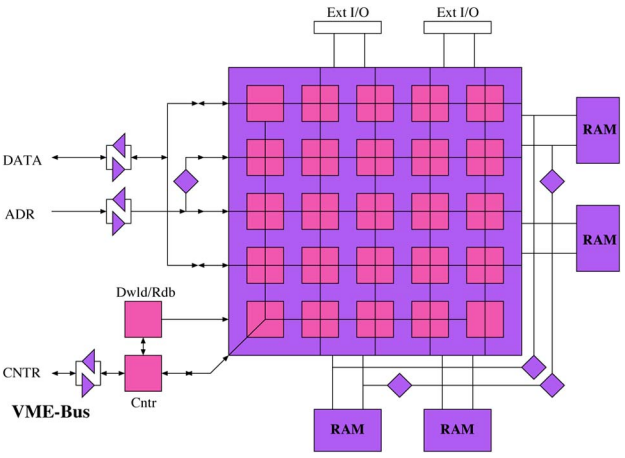


Fig. 1. *Architecture of the DEC PReL PAM P_0.*

This first reconfigurable platform was rapidly followed by the SPLASH 1 (1989) and SPLASH 2 (1992) [6]–[10] projects at the Supercomputer Research Center (Table 1). Each were linear arrays of FPGAs with local memory modules. They were designed for accelerating string-based operations and computations such as edit distance calculations. The SPLASH 2 was reported to achieve four orders of magnitude speedup, over a SUN SPARC 10, on edit distance computation using dynamic programming.

The PAM and SPLASH projects put the foundation of reconfigurable computing by using FPGA-based hardware accelerators. In the past two decades the density and speed of FPGAs have grown tremendously: the density by several orders of magnitude, the clock speed by just over one order of magnitude. Both of these projects could each be easily implemented on single moderately sized modern FPGA device. However, the main challenge to FPGAs as hardware accelerators, namely the abstraction gap between application

Table 1 Architecture Parameters of the SPLASH 1 and SPLASH 2 Accelerators

	SPLASH 1	SPLASH 2
Year	1989	1992
FPGA	XC3090	XC4010
4-LUT/board	10,240	13,600
Max. bandwidth	1 MB/s	100 MB/s
Memory/FPGA	128 KB	512 KB
Interconnect	Linear array	Linear array, broadcast

development and FPGA programming, not only remains unchanged but has probably gotten worse due to increase in complexity of the applications enabled by the larger device sizes. FPGA hardware accelerators are still beyond the reach of traditionally trained application code developers.

III. HARDWARE AND SOFTWARE COMPUTING MODELS

In this section we discuss two issues that define the complexity of compiling HLLs to hardware circuits: 1) the semantic gap between the sequential stored-program execution model implicit in these languages and 2) the effects of abstractions, or lack thereof, on the complexity of the compiler.

A. Efficiency and Universality

The stored program model is a universal computing model: it is equivalent to a Turing machine with the limitations on the size of the tape imposed by the virtual address space. It can therefore be programmed to execute any computable function. Hardware execution, on the other hand, is not universal unless it has an *attached* microprocessor. It is, however, extremely efficient. Consider an image filter applied on a 3×3 pixel window over a frame: the *forall* loop implemented in hardware can be both pipelined (let d be the pipeline depth) and unrolled as to compute multiple windows concurrently, let the unroll factor be k. In the steady state $d \times k$ operations are being executed concurrently producing k output results per cycle. On a CPU, the innermost loop of a typical image filter requires 20–30 machine instructions per loop body including nine load instructions. Assuming an average instruction level parallelism (ILP) of two, each result takes 10–15 machine cycles—which is the ratio of the respective clock speeds of CPUs and GPUs to FPGAs. However, that same loop can be replicated many times on the FPGA achieving a much higher throughput (at least an order of magnitude). Furthermore, the ability to configure local customized storage on the FPGA makes it possible to reduce the number of memory accesses, mostly reads, by reusing already fetched data resulting in a more efficient use of the memory bandwidth and lower energy consumption per task [11]. Hence the higher speedup or throughput observed on a very wide range of applications using FPGA accelerators over multicores (CPUs and GPUs). Further details on CPU efficiency for image filters are discussed in Section V-A.

B. Semantics of the Execution Models

CPUs and GPUs are inherently stored-program (or von Neumann) machines and so are the programming languages used on these. Most of the languages in use today reflect the stored program paradigm. As such they are bound by its sequential consistency, both at the language and machine levels. While CPU and GPU architectures exploit various forms of parallelism, such as instruction, data and thread-level parallelisms, they do so circumventing the *sequential consistency* implied in the source code internally (branch prediction, out-of-order execution, SIMD parallelism, etc.), while preserving the appearance of a sequentially consistent execution externally (reorder buffers, precise interrupts, etc.). The compiling of a HLL code to a CPU or GPU is therefore the translation from one stored program machine model, the HLL, to another, the machine's Instruction Set Architecture (ISA). In the stored program paradigm the compiler can generate a parallel execution only when doing so is provably safe. In other words when the record of execution can be *proved*, by the compiler, to be either identical or equivalent to the sequential execution. For example, in a single level *forall* loop, any interleaving of the iterations produces a correct result. Also, in a single threaded CPU execution the producer/consumer relationship is not a parallel construct since the semantics imply that all the data must be produced before any datum can be consumed. Hence all the data is stored in memory by the producer loop before the consumer loop starts execution.

A digital circuit, on the other hand, is inherently parallel, spatial, with distributed storage and timed behavior. HDLs (e.g., VHDL, Verilog, SystemC, and Bluespec) are arguably the most commonly used parallel languages. In a digital circuit the producer/consumer relation is a parallel structure: the data produced is temporarily stored in a local buffer the size of which is determined by the relative rates of production and consumption. Furthermore, any implementation would be expected to include back pressure and synchronization mechanisms to 1) halt the production before the buffer is full and 2) stall the consumption when the buffer is empty to achieve a correct implementation. Buffering the data is not necessary when compiling individual kernels (e.g., stand-alone filters). However, it becomes a necessity, and often a major challenge, when compiling larger systems. Consider data streaming through a series of filters: buffers and back-pressure are necessary to hold the data between filters. Automatically inferring efficient buffering schemes without user assistance in the forms of pragmas or annotations is a major challenge.

Edwards [12] makes the case that C-based languages are not well suited for HLS. The major challenges described in the paper for C-based languages apply to most HLLs. These challenges are the lack of: 1) timing information in the code, 2) size-based data types (or variable bit length data types), 3) built-in concurrency model(s), 4) local memories separated from the abstraction of one large shared memory. While all these points are valid, the main attraction of C-based languages is familiarity. Most HLS tools using C-based languages provide workarounds for one or more of these obstacles as described in [12].

The abstraction and semantic gaps between the hardware and software computing models are summarized in Table 2. Translating a HLL to a circuit requires a transformation of the sequential to a spatial/parallel, with the

creation of custom sequencing, timed synchronizations, distributed storage, pipelining, etc. The storage in the von Neumann model is abstracted in a single large virtual address space with *uniform* access time (in theory). The spatial model is better served with multiple local small memories. The parallelism in the von Neumann model can be dynamic: threads are created and complete relinquishing resources. In hardware every thread must be provisioned with resources statically. The software model relies on an implicit sequential consistency where all instructions execute in program order and no instruction starts before all the previous instructions have completed execution. The hardware execution is data flow driven.

Raising the abstraction level of FPGA programming to that of CPU or GPU programming is a daunting task that is yet to be fully completed. It is of critical importance in the programming of accelerators as opposed to the high-level design of arbitrary digital circuit, which is the focus of high-level synthesis. Hardware accelerators differ from general purpose logic design in one important way: the starting point of logic design is a device whose behavior is specified by a hardware description code implemented in a HDL such as VHDL, Verilog, SystemC, SystemVerilog, or Bluespec. The starting point of a hardware accelerator is an existing software application a subset of which, being frequently executed, is translated into hardware. That subset is, quasi by definition, a loop nest. Hopefully that loop nest is parallelizable and can therefore exploit the FPGA resources. By focusing on loop nests, the task of compiling HLLs to FPGAs is simplified and opportunities for loop transformations and optimizations abound. The ROCCC compiler takes this approach and is described later in this paper.

C. Related Work

As the number of tools supporting HLS for FPGAs has increased so has the number of surveys comparing and contrasting such tools. However, the rapidly shifting landscape of HLS tools for reconfigurable computing makes most such endeavors obsolete within a few years.

A description of the historical evolution of HLS tools, starting with the pioneering work in the 1970s can be found in [13]. The authors offer an interesting analysis of the reasons behind the successes and failures of the various generations of HLS tools. While the survey is not focused on HLS tools for FPGAs, it does mention several FPGA-specific tools, such as Handel-C, as well as general HLS tools that could be used for FPGAs.

The major research efforts in compiling high-level languages to reconfigurable computing are surveyed in [14]. The paper offers an in-depth analysis of the tools available at that time.

AutoESL is described in [15]. The paper also provides an extensive survey of HLS in general and of tools specifically for FPGA programming.

In [16] the authors reviewed six high level languages/tools based on programming productivity and generated hardware performance (frequency, area). User experience of using the targeted languages is recorded and normalized as a measure of productivity in this study. However, most of the tools evaluated in this work are no longer supported by their developers.

An extensive evaluation of 12 HLS tools in terms of capabilities, usability, and quality of results is presented in [17]. The authors use Sobel edge detection to evaluate the tools along eight specific criteria: documentation, learning curve, ease of implementation, abstraction level, data types, exploration, verification, and quality of the results.

Daoud *et al.* [18] survey past and current HLS tools.

IV. HIGH LEVEL LANGUAGE TOOLS

In this section we provide a brief description of the tools, where they were developed, and highlight some of their optimizations and the user experience of developing with each tool. The following tools can be divided into two classes: commercial software and research projects. We first cover the commercial software, followed by the university research projects.

A. Xilinx Vivado HLS

Vivado High-Level Synthesis is a complete HLS environment from Xilinx. It has been in development for the last several years following Xilinx's acquisition of AutoESL [19]–[21]. Vivado HLS is available as a component of

Table 2 Features and Characteristics of Stored Program and Spatial Computation Models

	Stored Program	Spatial Computing
Storage & data access	Central, large, virtual address space. Multi-level caches	Distributed, small, physical. Streaming data. Limited caching. No virtual memory
Parallelism	Dynamic - separate ILP, DLP, TLP	Static - integrated ILP, DLP, TLP
Sequencing	Central, static, sequentially consistent	Data-flow, asynchronous
Data-Path	Pre-designed, one size fits all. Dynamic data dependencies	Customized, very deep pipelines. No dynamic data dependencies

Xilinx's larger Vivado Design Suite or as a standalone tool. Like most HLS tools, Vivado HLS is mostly oriented towards core generation over full system design. It is possible to create hybrid designs with portions of code running on a soft-core processor communicating with custom hardware accelerators. However, the recommended work-flow [22] requires exporting the IP core from HLS and importing into the full Vivado Design Suite. As a Xilinx tool, there is significant support for different boards of multiple families of Xilinx FPGAs (7-series Virtex, Artix, Zynq, etc.). Depending on requirements, the hardware accelerator can be exported as one of several different Xilinx specific core formats for simple integration into other products, or just the HDL specification.

The Vivado HLS tool is built using LLVM [23], [24] compiler framework. As such it has access to many software optimizations (e.g., loop-unrolling, loop-rotation, dead-code elimination, etc.). However, hardware and software programing paradigms are inherently different so we cannot expect all of LLVM's optimizations to work seamlessly for HLS. Several studies using Vivado HLS to generate FPGA accelerators have been demonstrated, including Dynamic Data Structures[24], Sobel Filter [17], Control Algorithms for Power Converters [25], and real-time embedded system vision [26].

User Experience: Typical design flow (Fig. 2) starts with C code compiled to a pure software implementation and a self-validating testbench to verify correctness. The user must specify the top function in the code that they wish to synthesize to hardware. The GUI provides the user a list of code regions (targeted at loops, function bodies, and other bracketed regions) that can be optimized using synthesis directives to guide the RTL generation.

At the function level, directives include inline, instantiate (local optimization), dataflow (improve concurrency), pipeline (improve throughput), and interface (function defines an interface), among others. At the loop level, dataflow pipelining, and the common optimizations of loop-unrolling, loop-merging, loop-rotation, dead-code elimination, etc., are also available. The interface directive higlights an important aspect of the flexibility in Vivado HLS—the ability to generate user specified I/O protocols. Documentation highlights the convenience of using Xilinx's AXI4 interfaces in terms of compatibility with their IP catalog, however, the tool does not prevent custom protocols, and thus increases portability. The tool also provides the ability to set custom bit-widths for all variables in a design, leading to more efficient use of area.

The directives guiding the optimizations can be defined directly in the source code using *#pragmas*, similar to OpenCL code. They can also be defined separately in a directives.tcl file that the synthesis tools will apply before RTL generation. The second option creates the flexibility of maintaining multiple solutions testing different optimizations using the same source code. As design space exploration is a generally iterative and time consuming process, [27] this can reduce the development time.

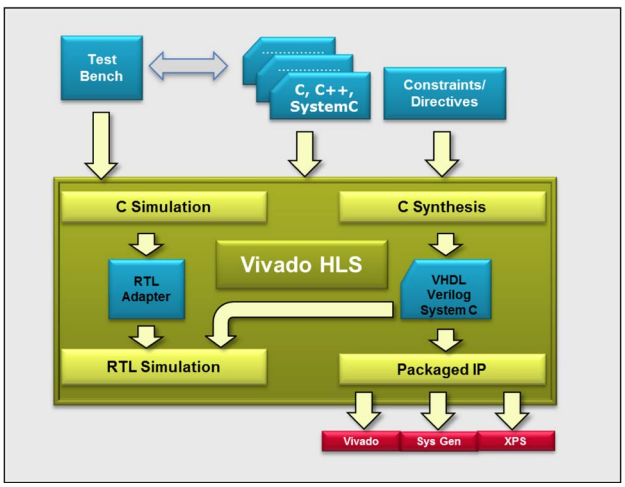


Fig. 2. ***Vivado HLS workflow.***

B. Altera OpenCL

Open Computing Language (OpenCL) is a programming language originally proposed by Apple Inc. and maintained by the Khronos Group [28]. The OpenCL specification provides a framework for programming parallel applications on a wide variety of platforms including CPUs, GPUs, DSPs, and FPGAs [29]. Moreover, OpenCL is a royalty-free, cross-platform, cross-vendor standard that targets supercomputers, embedded systems, and mobile devices. OpenCL allows the programmers to use a single programming language to target a combination of different parallel computing platforms. Parallel computation is achieved through both task-level and data-level parallelism.

The OpenCL framework provides an extension of C (based on C99) with parallel computing capabilities and the OpenCL APIs, which is an open standard for different devices. In the OpenCL programming model, a host is connected to one or more accelerator devices running OpenCL kernels. Device vendors provide OpenCL compilers and runtime libraries necessary to run the kernels. The host program is written in standard C to query, select, and initialize compute devices. Communication between the host program and accelerators is established through a set of abstract OpenCL library routines. Each accelerator device is a collection of compute units with one or more processing elements. Each processing element executes code as SIMD or SPMD.

In the FPGA industry, both Altera and Xilinx have announced support of OpenCL HLS in their FPGA development tools. Altera released an OpenCL SDK in 2013 that supports a subset of the OpenCL 1.0 specifications. Xilinx started to support OpenCL in their Vivado HLS tool in April 2014. In this paper we focus on the Altera OpenCL SDK.

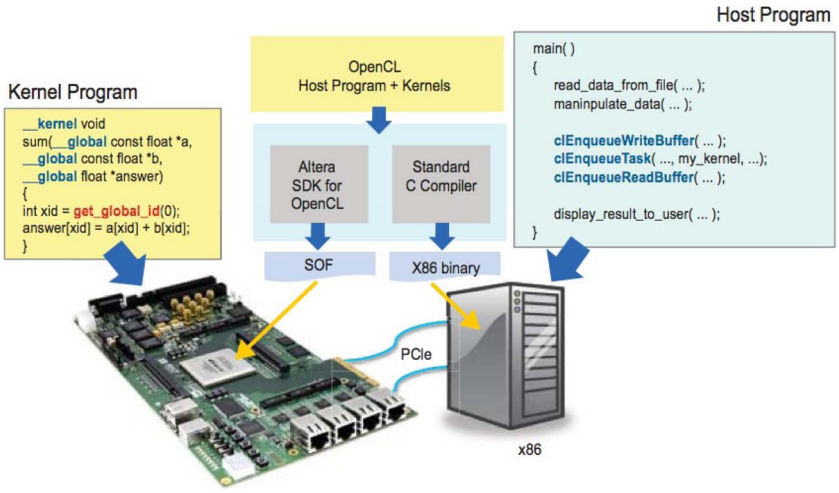


Fig. 3. ***OpenCL system overview, image from [30].***

The Altera OpenCL SDK provides software programmers an environment based on a multi-core programming model that abstracts away the underlying hardware details while maintaining efficient use of FPGA resources. The Altera OpenCL compiler (AOC) is an offline compiler that translates OpenCL to Verilog and runtime libraries for the host application API and hardware abstractions. The OpenCL system overview is shown in Fig. 3. Unlike the OpenCL compiler for CPUs and GPUs, where parallel threads are executed on different cores, AOC transforms kernel functions into deeply pipelined hardware circuits to achieve parallelism. AOC uses a CLANG front-end to parse OpenCL extensions and intrinsics to produce unoptimized LLVM IR [31]. The middle-end performs optimization with about 150 compiler passes such as loop fusion, auto vectorization, and branch elimination. On the back-end, the compiler instantiates Verilog IP and manages control flow circuitry of loops, memory stalls, and branching. Finally the generated kernel is loaded onto an Altera FPGA using an OpenCL compatible hardware image. Various applications using OpenCL to program FPGA accelerators have been demonstrated, such as information filtering [31], Monte Carlo simulation [30], finite difference [32], particle simulations [32], and video compression [33].

User Experience: The AOC is designed for software programmers to construct parallel FPGA applications. It has a similar command line interface to the GCC compiler. All OpenCL codes must be included in a single text file before passing to the compiler. The AOC will generate transformations, create Quartus II project files and perform synthesis, place and route, and bitfile generation for FPGA execution. There are several compiler optimizations that can be applied to OpenCL code: kernel vectorization, static memory coalescing, generating multiple compute units, and loop unrolling. Optimizations, when invoked by the user, are applied automatically by the compiler on the whole code to improve processing efficiency. In addition, the programmer can specify attributes, such as *num_compute_units* and *num_simd_work_items*, in the source code to manually control the degree of kernel vectorization and parallel compute units respectively, as shown in Fig. 4. When setting the *num_simd_work_items* attribute, the data path within a compute unit is replicated to increase throughput and can also lead to memory coalescing. On the other hand, the *num_compute_units* attribute will also duplicate all the control logic which may increase the number of global memory access. Fig. 5 shows the difference between these two optimizations. Both techniques can be combined to further increase the parallelism at a cost of higher memory bandwidth usage and more FPGA resource occupation. Furthermore, AOC allows users to specify the loop unrolling factor by setting the preprocessor directives (#pragma unroll). In Fig. 4, the loop iteration is unrolled 16 times. If the amount of unrolling is not specified, AOC will fully unroll the loop. Moreover, the AOC can perform resource-driven optimizations that

```
__attribute__((num_simd_work_items(1)))
__attribute__((num_compute_units(3)))
__kernel
void SampleKernel(__global int * restrict a_in,
                  __global int * restrict a_out)
{
        int i;
        #pragma unroll 16
        for(i = 0; i < 128; ++i)
                ....
}
```

Fig. 4. ***Sample OpenCL kernel function with programmer set attributes.***

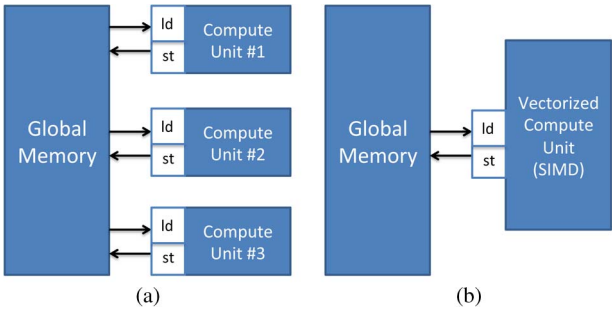


Fig. 5. ***OpenCL compiler optimization techniques to increase parallelism. (a) FPGA kernel with multiple compute units. (b) FPGA kernel of one compute unite with kernel vectorization.***

analyze various combinations of compute unit number, work group size, loop unrolling factor and number of shared resources under the constraints of available hardware resources and memory bandwidth to determine the optimal choice of these values. The resource-driven optimizer is invoked when the programmer sets the *-O3* switch in the compiler.

To test the OpenCL code functionality, AOC provides an OpenCL emulator to simulate the behaviour of the OpenCL kernel program. The emulated kernel is used as a dynamically linked C++ library that can be called from a host program. To compile the OpenCL code for emulation, the *-march=emulator* option should be included in the compilation. Programmers can write a host program to verify if the OpenCL kernel works as designed. In addition, basic C functions like *printf* can be used inside the OpenCL kernel with the emulator to check intermediate values. When the emulator is used, no compiler optimizations can be applied. At present, there is no RTL simulations for hardware programmers to test the generated Verilog kernel.

C. Bluespec System Verilog

Bluespec System Verilog (BSV) [34], [35] is a high level hardware description language built upon the synthesizeable subset of SystemVerilog. BSV's behavioral model is based on *Atomic Rules and Interfaces*. The rules ensure parallelism, which is well suited for complex hardware designs. BSV also provides powerful abstraction mechanisms that were previously believed to be suited only for software applications. BSV derives most of these abstractions from System Verilog, such as module and module hierarchies, loops, and user-defined data types (enum, struct, tagged union). BSV provides architectural transparency—meaning the programmer, not the tool, expresses the architecture of a the design. This transparency places the tool in an area between HLS and HDL; it abstracts away some of the complexities of working at the hardware level, yet it loses some of the automation provided by many HLS tools. BSV has strong type-checking which ensures all objects are compatible and conversion functions are valid [36]. BSV also provides strong static checking by preventing movement of values to/from currently unused modules—ensuring a synchronizing mechanism must be used to cross clock boundaries. All of these features help eliminate timing errors.

Language Semantics: In conventional Verilog, values are typically kept synchronized by using an *always* block. In BSV *Rules* are used to describe how the data is moved from one state to another. The rules are atomic in nature, meaning that the execution of each rule should be considered independently. A Rule is triggered when all of its preconditions are met. Preconditions are boolean expressions, which are purely combinational logic and do not have any side effects. All actions within a rule occur simultaneously, implying all rules should be composed of independent actions. Rules help in achieving higher concurrency and avoiding race conditions. The independent nature of rules allows hardware to execute multiple rules concurrently.

A BSV module's interface consists of methods instead of a ports list. A method is similar in concept to a function, it takes in arguments and returns a result. However, they differ in that a method also carries with it a set of implicit conditions. Each method has an associated *Ready* signal (output port) and an *Enable* signal (input port) if it is an *Action* method. Because BSV does not accept standard C like the other HLS tools presented in this paper, we show a simple BSV code to multiply two integer values for syntax clarification (Fig. 6). The module named *product* provides *Product_Interface*. *Product_Interface* has two methods. The readResult is a *Value Method*, which forms the output port of the module. The second method *setValues* takes three arguments and forms the input port of the module. The attribute *synthesize* tells the BSV compiler to generate a separate hardware implementation (Verilog) of the following module. We then define four registers of type *int* and instantiate them with a BSV defined module *mkReg*. Finally we apply different rules to compute the product.

BSV uses a dynamic scheduler which allows multiple rules to be executed in each clock cycle. The compiler, guided by the scheduler, performs a detailed analysis of all the rules and their interactions with each other and maps the design into the clocked, synchronous hardware. This mapping permits multiple rules to be executed in each clock cycle, however, it may also give rise to a situation where two or more rules conflict (e.g., limited resources). In the absence of user guidance, the compiler arbitrarily chooses which rule to prioritize and issues a warning. As shown in Fig. 7, the BSV compiler makes sure that all the control logic is dictated solely by the applicable rules, thus functional correctness is achieved.

User Experience: Bluespec compilation can be controlled from the Bluespec GUI or from the command line interface. It is very important for the programmer to understand how Bluespec language semantics get converted to

```
interface  Product_Interface ;
  method int readResult ;
  method Action setValues (int p,  int  q,  int  r)  ;
endinterface

(* synthesize *)
module product (Product_Interface) ;

 Reg#(int) x <- mkReg (0) ;
 Reg#(int) y <- mkReg (0) ;
 Reg#(int) z <- mkReg (0) ;
 Reg#(int) result  <- mkRegU ;

 Reg#(Bool) b <- mkReg (False) ;

 rule  toggle  ;
         b <= !b ;
 endrule

 rule  r1  (b)  ;
         result  <= x * y ;
 endrule

 rule  r2  (!b)  ;
         result  <= x * z ;
 endrule

 method readResult = result ;

 method Action setValues (int p,  int  q,  int  r)  ;
         x <= p ;
         y <= q ;
         z <= r ;
 endmethod

endmodule
```

Fig. 6. ***Multiplication example in BSV.***

RTL. For example, to interact with the module we need ports which are formed by Bluespec *interfaces* [17].

The RTL code is generated by scanning each line of the Bluespec code. All the Bluespec design entity names are preserved in the RTL code generated by the compiler. For example, in Bluespec, methods are used to bring data into the module and send out of the module. Methods eventually form port signals of the same name in the RTL, which makes the RTL code more readable and traceable [17].

The benefit of BSC over C-based synthesis tools is that BSC does not invent the architecture for us. With C-based synthesis, the designer writes an (often sequential) algorithm, and the C-based tool figures out an architecture to implement that algorithm. With BSC, the designer chooses the architecture, writes BSV to express the desired hardware implementation, and BSC generates that hardware. However the designer may not have many choices for design exploration since the design entry is at the hardware level.

D. LegUp 3.0 (U. Toronto)

LegUp [37], [38] is an integrated HSL environment that is developed and maintained at the University of Toronto. The project's goal is to take existing C applications in their entirety, compile and run them on an FPGA. This differs from most HLS tools that focus on compiling only specific regions of code to hardware. Currently, LegUp targets two Altera platforms—a Cyclone II on the DE2 board, and a Stratix IV on the DE4 board. Design choices like the soft-core processor and the communication bus are tightly

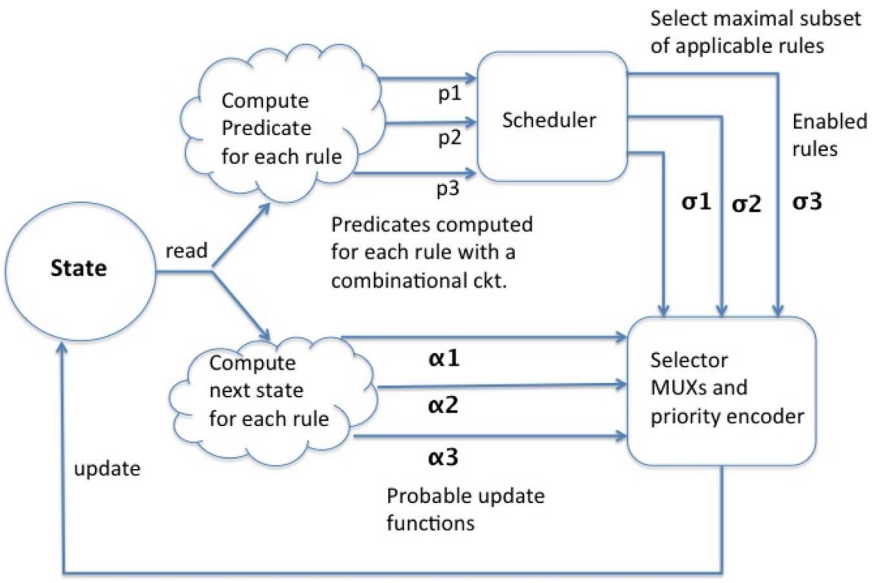


Fig. 7. ***Scheduling of rules in BSV.***

coupled to these specific boards, and porting them to other platforms would be a nontrivial task. However, the hardware accelerator cores themselves have only a few Altera specific components, and can be ported with little effort.

LegUp supports three types of compilation: pure software, pure hardware, and a hybrid approach. In a pure software compilation, LegUp only generates assembly code that can be run on a Tiger MIPS [39] soft-core processor, enabling the FPGA to handle almost all of the C language standard. However, it limits the performance and energy efficiency. In a pure hardware compilation, LegUp generates a custom circuit for the entire application. However, only a subset of the C language is supported for hardware acceleration. For example, dynamic memory and recursive functions do not make sense as a circuit. It should be noted that these limitations are ubiquitous among all HLS tools. In a hybrid compilation, LegUp generates a custom circuit for only part of the application, and assembly code is generated for the rest. LegUp keeps any hardware calls from software transparent to the user by automatically inserting them into the assembly code. Support exists for both blocking and nonblocking hardware calls. Blocking calls improve the energy efficiency, but nonblocking calls will improve the performance.

Fig. 8 shows the complete hybrid architecture that LegUp generates. It has a single Tiger MIPS processor and can have multiple hardware accelerators depending on the application. All memory data is stored in an off-chip memory, but an on-chip cache is used to improve performance. All communication is carried across Altera's Avalon bus [40]. Targeting a single type of architecture has its tradeoffs. A shared global memory space prevents costly data offloading, and assuming only one type of bus simplifies the communication protocols. However, it results in very specialized hardware accelerators. Their performance is limited by the Avalon bus's bandwidth, and cannot be extended to higher bandwidth architectures. LegUp is currently a very specific tool, but extensions could make it more general.

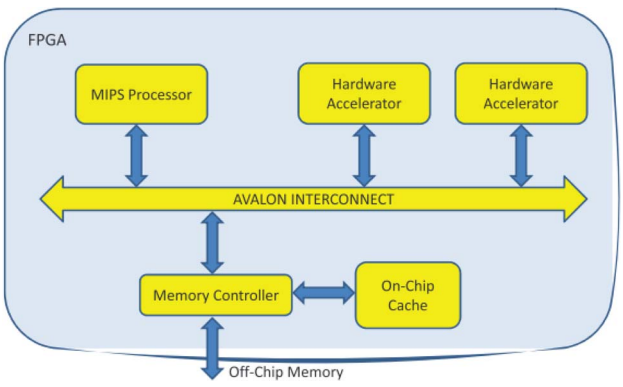


Fig. 8. ***LegUp 3.0 target architecture.***

Typical design flow for an application starts with the C code compiled into a pure software implementation. LegUp provides a built in profiler to help identify computation intensive code regions that are strong candidates for hardware acceleration. This stage is not automated: the user must mark functions for hardware compilation. LegUp then generates the necessary accelerators in Verilog and the application is recompiled to insert the necessary hardware calls. The entire design can be simulated to verify correctness and then synthesized for the target FPGA. These steps are repeated iteratively until the designer is satisfied with the performance.

Similar to several other tools, LegUp is based on the LLVM compiler framework. The impact of various LLVM optimizations on the performance of the generated hardware structures is explored in [41]. Extra passes are added to LLVM for HLS and work in three phases: allocation, scheduling, and binding. The allocation stage determines the available hardware based on the target architecture and manages the application's constraints like clock speed and power consumption. Scheduling orders the operations. Currently, only as-soon-as-possible scheduling is supported, but because of LLVM's modular design this pass can be easily changed. Binding reduces resource utilization for complex instructions (i.e., multiply or divide) by multiplexing the data path through a single component. A weighted bipartite matching heuristic is used to handle the binding problem.

User Experience: Being a research tool, LegUp is not primarily concerned with usability. Installation can be difficult because it requires specific versions of standard tools. The LegUp website [42] does offer an Ubuntu virtual machine, for VirtualBox, with the tools already setup.

The user interfaces with the compiler through command line, but comprehensive makefiles and many example are provided to help get new users started. Most operations can be handled through a provided makefile, from compiling and simulating to automatic project creation and synthesis. Optimizations are applied in two locations: For hardware specific optimizations (e.g., pipelining) the user needs to create a .tcl script. Common software optimizations (e.g., loop-unrolling) can be passed directly to the LLVM compiler through the makefile.

E. ROCCC 2.0 (UC Riverside)

The Riverside Optimizing Compiler for Configurable Computing (ROCCC) [43] is a C to VHDL compiler built using SUIF [44] and LLVM [45]. ROCCC was initially developed at the University of California Riverside. ROCCC 2.0 was developed by Jacquard Computing Inc. with funding from the AFRL under an SBIR contract. ROCCC 2.0 is freely available on GitHub. The SUIF toolset is used to implement the high-level transformations, such as loop and array transformations, and generates an intermediate representation, CIRRF (Compiler Intermediate

Representation for Reconfigurable Fabrics) [46], a readable text file. The CIRRF code is then passed to LLVM for further low-level optimizations and VHDL generation. Hardware specific optimizations, such as pipelining, expression tree balancing etc., are implemented in the second pass of LLVM. The ROCCC design puts a special emphasis on user driven transformations and optimizations whose objective is to maximize throughput by exploiting the inherent parallelism and reducing the area footprint of the generated code [47]. Another objective is reducing the number of memory accesses per output result by automatically reusing already fetched data [48].

The designs in ROCCC are divided between two abstractions: systems and modules. Systems perform the main computations that are being accelerated for the application. They are based around *for-loops* that allow them to process streams of data and access memory through arrays. The system generated code is the only code that can read and write external data. Modules are designed to implement a specific task and are used as function calls from systems or other modules. Both modules and systems have a variable number of parameters that represent I/O ports in the underlying hardware core. These abstractions provide reusability and ease of development for large systems with multiple levels of complexity. ROCCC also allows the addition and use of external cores in projects as if they were another function call.

ROCCC provides a number of high-level optimizations. They include: division elimination, multiply elimination, loop-unrolling, module inlining, redundancy, systolic array generation [49], loop-fusion, and Temporal Common Subexpression Elimination (TCSE) [50]. Additionally, ROCCC provides some low-level optimizations. They include: maximize precision, copy reduction, fanout tree generation, and arithmetic balancing. System code has access to all these optimizations whereas modules do not have the systolic array generation, loop-fusion, and TCSE optimizations. All loops in modules are fully unrolled because modules do not stream data. ROCCC is designed to create platform independent hardware structures. However, by default larger FIFOs use vendor-specific IP cores to improve timing. They are wrapped in an *InferredBRAM* component to allow easy portability. Similar to most C-to-FPGA tools, ROCCC is only able to compile a subset of C that works well with FPGAs. Features like dynamic memory allocation and recursion are not supported by ROCCC.

User Experience: ROCCC development is accomplished through the Eclipse IDE with custom plugins for the GUI. Since the idea behind HLS tools is to make hardware development easier, using Eclipse provides a small learning curve to ROCCC. Short and straightforward tutorials are available online [51]. Those comfortable with the Eclipse IDE will be able to produce systems at the same rate that they would produce a software project. Adding modules and other IP cores to projects is done through their respective menu screens after choosing to import them on the ROCCC drop-down menu. After writing the C code for either a module or system, clicking the build button brings up a menu process. The menu process involves selecting optimizations, setting a pipeline preference, and managing I/O streams. All of which have their benefits to a circuit described in the tutorial. A deep understanding of how the hardware works in order to test it is not required since ROCCC provides a test bench generation process. It involves including .txt files with sequences of inputs and expected outputs. The user can then verify the circuit by reading the console output of the simulator.

Fig. 9. ***Dilation example. (a) Crisscrossed UCR logo; (b) clean UCR logo following dilation.***

V. DILATION KERNEL EXAMPLE

In this section, we use a simple image processing operation, dilation, to demonstrate the use of various HLS tools. Dilation sets the value of the center pixel of a window (in our case 3×3) to the maximum of all values in that window. In the case of a greyscale image, it will make white regions brighter and reduce dark spots as shown in Fig. 9. We start with a black and white image with the white letters "UCR" crisscrossed with black lines. After several passes of a dilation, the cross marks are removed and a clean UCR logo shows through.

Dilation is rather amenable to comparisons because it uses simple, straightforward C code, an efficient circuit is not too challenging to generate, yet compilers optimizations can result in very large performance increases.

The simplest way to implement this filter that follows directly from the sequential description from the C code can be seen in Fig. 12(a). In this circuit, we simply register the nine pixels within the window and then compare pairs of two pixels to find the max of the nine pixels. While simple to generate from the source, it has two major drawbacks—latency and wasted memory requests. Without a mechanism to save values from window to window, we waste 66% effort refetching the same data as seen in Fig. 10.

A slightly improved version can be seen in Fig. 12(b). Balancing the arithmetic between the max operations helps to reduce latency, but does nothing to help with the memory accesses. Unintelligent use of data can cause excessive uses of the memory subsystem, and really hinder performance—especially throughput.

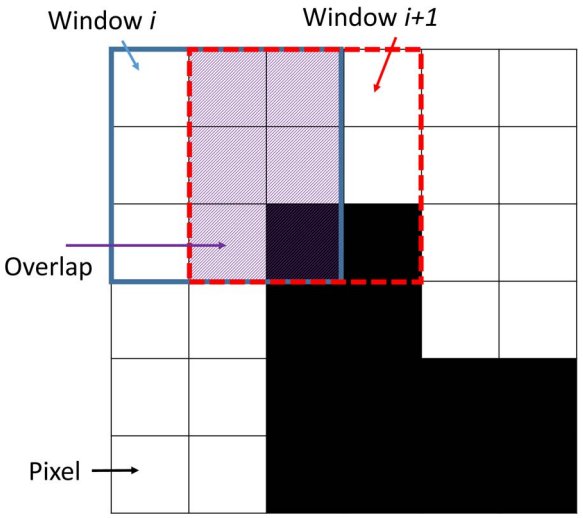


Fig. 10. ***Overlapping region of adjacent Dilation windows can lead to smart optimization or redundant memory requests.***

Fig. 12(c) demonstrates the type of circuit we would like to see produced by the HLS tools. By using temporal common sub-expression elimination, we have data reuse between windows, maximizing the utilization of the memory system. This layout allows for the streaming of three rows of the image from memory, and we take the max of the current column. The max is registered and shifted through a buffer for use in the next cycle. After the initial three cycle latency for the buffer to be filled, this circuit can produce one pixel per cycle. We can further increase performance by duplicating this design on the FPGA by unrolling the inner loop. Each kernel on the device would process overlapped streams of the image, creating data reuse vertically as well as horizontally. This design yields itself to low latency, good data management, and the ability to scale as the resources on the chip allow.

A final possible optimization, represented in Fig. 12(d), uses a line buffer (supported by AOC): an entire row of the image is shifted onto the FPGA and a window of three pixels is used to generate the max. Great performance with this technique is possible as long as there are enough resources on the FPGA to accommodate all the necessary line buffers. It may not be feasible when using large images.

A. CPU Implementation

In the introduction, we proposed an argument about the efficiency of FPGAs in relation to CPUs, and how FPGA codes can achieve their speed-up relative to traditional software. Before presenting how the various HLS tools implement the following benchmarks, we feel it is prudent to have a baseline software implementation (compiled with GCC 4.6.3, -O3) for comparison.

Fig. 11 shows two implementations of the dilation filter: in Fig. 11(a), we have a simple, direct interpretation of dilation [taking care not to do a completely serial implementation like Fig. 12(a)] and in Fig. 11(b) a programmer optimized version to take advantage of TCSE. Since a major focus of this paper is how much the tool can accomplish for

```
//slide window across image
for(i = 0; i < HEIGHT; ++i)
{
  for (j = 0; j < WIDTH; ++j)
  {
    maxRow1 = MAX(img[i  ][j],
                  img[i  ][j+1],
                  img[i  ][j+2]);
    maxRow2 = MAX(img[i+1][j],
                  img[i+1][j+1],
                  img[i+1][j+2]);
    maxRow3 = MAX(img[i+2][j],
                  img[i+2][j+1],
                  img[i+2][j+2]);

    // Max of the three rows
    f_img[i][j] = MAX(maxRow1,
                      maxRow2,
                      maxRow3);
  }
}
```

(a)

```
for(i = 0; i < HEIGHT; ++i)
{
    maxCol1 = MAX(img[i][j],
                  img[i+1][j],
                  img[i+2][j]);
    maxCol2 = MAX(img[i][j+1],
                  img[i+1][j+1],
                  img[i+2][j+1]);

    for (j = 0; j < WIDTH; ++j)
    {
        maxCol3 = MAX(img[i][j+2],
                      img[i+1][j+2],
                      img[i+2][j+2]);

        // Max of all Columns
        f_img[i][j] = MAX(maxCol1,
                          maxCol2,
                          maxCol3);
        //shift maxes for reuse
        maxCol1 = maxCol2;
        maxCol2 = maxCol3;
    }
}
```

(b)

Fig. 11. ***Simple change to access pattern and order of calculations makes a significant difference for the CPU optimization opportunities. (a) Dilation code in C. Same code was used for the ROCCC compilation; (b) optimized dilation code in C. These optimization, along with data reuse, are applied by the ROCCC compiler reducing the number of memory reads (Table 5).***

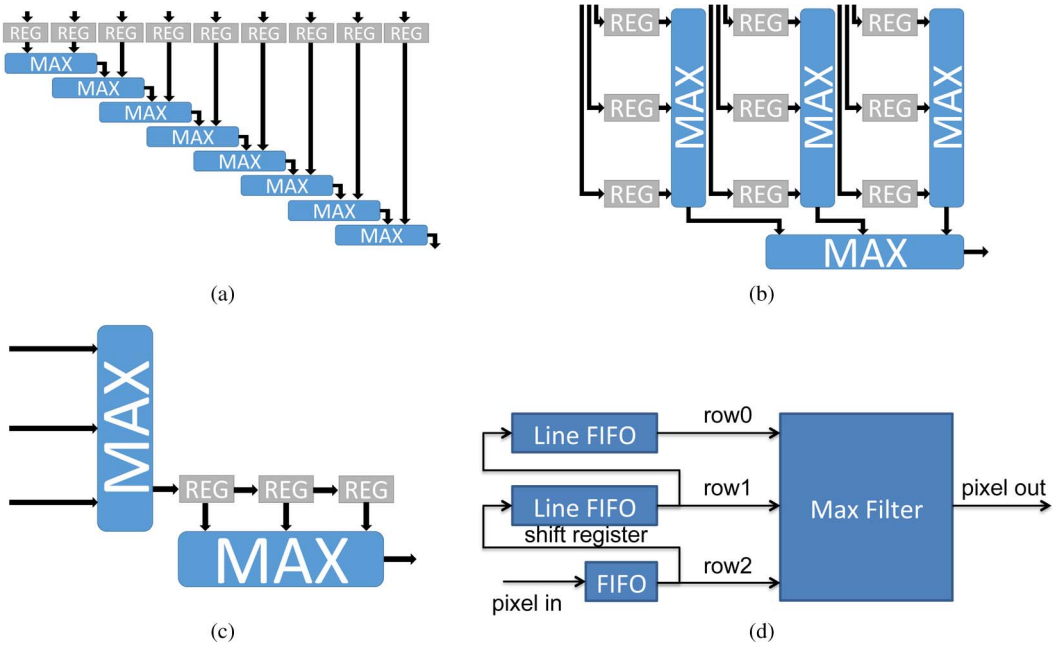


Fig. 12. *Examples of Naïve implementations (top) versus Smart optimizations (bottom) that improve performance. (a) Zero optimization—Lack of tree balancing causes wasted memory access and increased latency. (b) Tree balancing helps latency, but still wastes memory accesses. (c) Example dilation circuit, using TCSE optimization. (d) Example dilation circuit, multi-line buffer.*

the programmer, we started with the direct implementation to determine the level of efficiency GCC could achieve.

In the direct case, the compiled assembly for x86 has 32 instructions for the inner loop, meaning 32 machine instructions executed for every output pixel generated. Another issue with the assembly is the lack of register reuse. Register allocation is a particularly hard problem with compilers, and in this case reusing registers for the next iteration of the loop was not identified—all nine pixels used in a window are loaded with move instructions from memory. Depending on the memory hierarchy and caching structure, this could also be a detriment to performance.

Unrolling only provides a minor benefit to the simple software implementation and yields a total of 60 instructions. In this case, all the compiler can accomplish is duplicating the body of the inner loop once, without any register reuse between the unrolled loop bodies as evidenced by the 18 load requests. Overall, the code executes 30 instructions per output pixel.

In generating the optimized version of the code, we wrote it specifically to force value reuse between iterations. We implemented four different versions, including cache blocked memory accesses to determine the best performing implementation—row based access and non-memory blocking. The dilation filter by default reuses a lot of values that will be brought to cache as it passes over the rows, so blocking the accesses only adds overhead and reduces performance. For value reuse, we simply precompute the two maxes of the initial columns of the current window before iterating over the rows of the image. This minor reorganization simplifies the inner loop to only calculate one new max from the input, write the output, and then shift two of the current maxes to reuse on the following iteration.

The compiler performs more extensive optimizations on this implementation. During initialization in the outer loop, the code executes 46 instructions and stores two results in this stage with 12 loads, for a total of 23 instructions per output pixel. Loop unrolling is able to unroll the inner loop six iterations, for a much better stride over the majority of the data. During this stage, the code does 18 loads and six stores and executes 76 instructions, for 12.67 instructions per output pixel.

With the best compiler optimizations applied to an already hand-optimized version of the dilation code, the peak performance we saw was 13 instructions per output pixel. CPUs run at an order of magnitude faster clock frequencies, but still has to execute at least an order of magnitude more instructions and thus, clock cycles. And that ignores any many-cycle-stalls from memory misses. The key point we want to highlight is that in order to achieve that performance, a programmer had to be knowledgeable about the possible shortcomings of the platform and spend development time altering the source to generate the desired assembly performance—the tools could not achieve that independently. Contrast this to some of the following examples where naïve code and a compiler flag are enough to generate well optimized hardware implementations.

B. Vivado HLS

To implement the dilation project in VivadoHLS, we started with the sample C code provided in the beginning of the section, except for a minor difference in the parameter list. To have VivadoHLS process the input as a stream, and thus pass the input as a pointer, a protocol must be created to interface between the stream and the circuit. Since the information we are interested in is how the tool compiles the kernel and not the data passing, we elected to use an input array of fixed size to avoid the extra overhead.

With that minor change, VivadoHLS was able to compile the given source to hardware. The tool also uses the programmer-provided software test harness to generate a hardware testbench to validate correctness. It is possible to go from C to VHDL without ever looking at a waveform—a benefit to software developers without experience in hardware development.

By default, VivadoHLS does not apply any optimizations. It is easy to get a working RTL solution, but the tool will only do exactly as the programmer directs. In this example, the first solution generated had similar performance characteristics to Fig. 12(a). Adding simple directives to the inner loop like pipelining and unrolling and the resulting RTL resembles Fig. 12(c).

C. OpenCL

We have implemented dilation in OpenCL by using a single loop structure, as in AOC, nested loops should be avoided for performance considerations [52], [53]. Note that, for a single loop structure, the output image has the same size as the input image. Also, the edge values may not be correct since all loop iterations should have the same computation. However, this is a minor issue for image processing as edges are typically not the region of interest. The OpenCL implementation code is shown in Fig. 13. The code was compiled with and without optimizations (-O3 optimizations) however the generated hardware description is not readable and it is not possible to observe the effects of the optimizations on the generated circuits. We did observe significant changes in the resource utilizations and the clock frequency.

```
#define R  1080
#define C  1920
__kernel
void max_filter(__global unsigned * restrict p_i,
                __global unsigned * restrict p_o)
{
        unsigned rows[2*C+3]; // line buffer
        const unsigned iterations = R*C;
        unsigned count = 0, wimMax= 0;
        while (count != iterations )
        {   // infer shift registers as line buffer
                #pragma unroll
                for (unsigned i = C*2 + 2; i > 0; --i)
                {
                        rows[i] = rows[i - 1];
                        rows[0] = p_i[count];
                }
                winMax = 0;
                #pragma unroll
                for (unsigned i = 0; i < 3; ++i)
                {
                        #pragma unroll
                        for (unsigned j = 0; j < 3; ++j)
                        {
                                unsigned p = rows[i*C+j];
                                if (max_pix < p)
                                        winMax = p;
                        }
                }
                p_o[count++] = max_pix;
        }
}
```

Fig. 13. *OpenCL implementation of Dilation using a single loop structure.*

D. Bluespec System Verilog

The Bluespec *dilation* module reads three rows of an image matrix at a time. It forms the pipeline of incoming data and as soon as the window of 3×3 is received, the module outputs the maximum value in a window. The Bluespec module is built similar to Fig. 12(c). The interface to dilation in BSV is declared as depicted in Fig. 14.

The interface has three methods. An *Action* method usually causes some state change. In this example *input_in* is an *Action* method and brings in three inputs *a,b,c* into a module. The *out_data* method is a *Value* method which outputs data from the module. In our case we return 16 bit wide *maximum value of* 3×3 *window* from the module, which the Bluespec compiler names the same as method name, *out_data*. The other two methods act as the control signals to the outside modules. The *done* method terminates the execution of a module and *isPipeflush* method let the other module know that pipeline is being flushed. The dilation module is shown in Fig. 15. The module begins by importing packages and is instatiated with two parameters: height and width of an image. As we can see in Fig. 15, rule *incr* writes to *wCount* and rule *check* reads the value of *wCount*, the BSC attribute *descendency_urgency* guides the compiler to schedule the more urgent rules first in case both of these rules fire in the same cycle. We then apply rules to compute the maximum value of window. Each of the rules have some conditions specified under which they fire. The pseudo-code can be seen in Fig. 15.

```
typedef Int#(16) size;
interface Incomingdata;
 method Action input_in (size a,
                         size b,
                         size c);
 method Bool isPipeFlush();
 method size out_data();
 method Bool done();
endinterface
```

Fig. 14. *BSV interface to dilation module.*

```
import Incomingdata :: *;
import Vector :: *;
module mkTb #(parameter size height,
              parameter size width)
             (Incomingdata);

function size maxOf3(size x,
                     size y,
                     size z);
        let temp = max(x,y);
        return max(temp,z);
endfunction

(* descending_urgency = "check,incr" *)
rule shift_to_stage1( !pipeflush && valid0 );
        stage1[0] <= stage0[0] ;
        stage1[1] <= stage0[1] ;
        stage1[2] <= stage0[2] ;
        valid1 <= valid0;
endrule

rule incr(!pipeflush && valid0 );
    wCount <= wCount + 1;
endrule

rule check( !pipeflush && wCount == width-1 );
    pipeflush <= True;
endrule

rule find_max( valid2 );
 let temp_max0 = maxOf3(stage0[0],
                        stage0[1],
                        stage0[2]);
 let temp_max1 = maxOf3(stage1[0],
                        stage1[1],
                        stage1[2]);
 let temp_max2 = maxOf3(stage2[0],
                        stage2[1],
                        stage2[2]);
 let temp_final =maxOf3(temp_max0,
                        temp_max1,
                        temp_max2);
     max_of_window <= temp_final;
     max_of_all_stage <= True;
endrule

method Action input_in(size a, size b, size c)
              if ( !pipeflush);
    stage0[0] <= a;
    stage0[1] <= b;
    stage0[2] <= c;
    valid0 <= True;
endmethod
endmodule
```

Fig. 15. ***Dilation implementation in BSV.***

As a part of verification, testbenches are manually written in Bluespec. In our case, we instantiate a dilation module in our testbench module. We keep sliding our window one column at a time and as soon we reach the width of an image we slide our window by one row. We repeat the steps until the last window can be formed. In this testbench, the top-level interface is "Empty," which means when we see the synthesized verilog, we can see only clock and reset lines. Bluespec provides a test driver module for modules with "Empty" interfaces which applies reset and then drives the clock indefinitely.

E. LegUp 3.0

As we previously stated LegUp targets a very specific architecture, and this limits any generated kernel's bandwidth to two memory channels. Even though these kernels cannot increase the number of memory channels they can be optimized to better utilize the available bandwidth. Table 5 shows the performance results as we apply loop unrolling to the dilation kernel. An important note we want to point out is that for every test, the total number of write memory accesses is exactly the same because LegUp only duplicates the hardware engines, but does not merge their computations.

However, duplicating the engines does increase the number of available memory requests. LegUp's FSM can then schedule the requests closer together to improve bandwidth utilization. This also improves the number of runtime cycles. Clock frequency is affected as the kernel is unrolled due to larger resource utilization on the FPGA. Overall runtime is also affected as shown in Table 5.

F. ROCCC 2.0

The ROCCC implementation of dilation starts with the C code shown in Fig. 11(a). The same code was used in [43] to demonstrate the TCSE optimization. The sample code uses a modular approach which replaces each section of if-statements with a call to a MAX module. The current version of ROCCC would not unroll either the inner loop or the outer loop more than twice with the modular approach. In order for ROCCC to unroll either loop further, the code had to be written with the max functions inlined by hand.

To take advantage of the loop unrolling, input and output streams were added to the circuit through the ROCCC GUI. Multiple input streams divide up the number of memory accesses, further reducing the execution time. Each extra window gained from loop unrolling shares the bottom two rows with the window above it, meaning that an extra input stream is required for the bottom row of the new window. With no unrolling, three inputs (one input for each row of the 3×3 window) and one output were used. The amount of outputs is equal to the amount of times the loop is unrolled and the number of inputs is two more than the number of times the loop was unrolled. For example, two unrolls uses four inputs and two outputs.

The circuit that ROCCC generates processes input similar to the circuit depicted by Fig. 12(c). The MAX component represents the if-statements sequence inside the nested loop. Every cycle an element from each row of the 3×3 window is pulled in through a FIFO. Once all three elements of the row are in, they get compared and the max from each row goes to the next comparison block to produce the maximum number of the 3×3 window. This specific example is a dilation circuit with three input streams. With only one input stream there would be just one FIFO for all three MAX components. Loop unrolling and adding the appropriate amount of input streams creates multiple compute units.

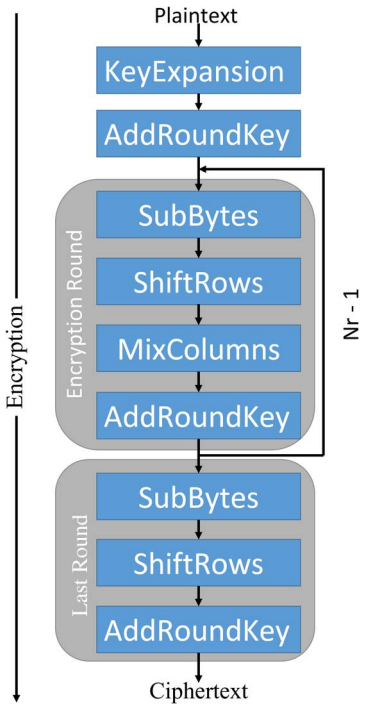


Fig. 16. ***AES flow graph.***

VI. AES ENCRYPTION KERNEL EXAMPLE

The Advanced Encryption Standard (AES) [54] is a symmetric block cipher that can be used to encrypt and decrypt information to protect electronic data. The AES algorithm has become the default choice for various security services in numerous applications [55]. The encryption process converts plaintext into an unintelligible form known as cipher-text, and decryption is the inverse of this process. AES processes the data in 128-bit input blocks using a key size of 128, 192, or 256 bits. With respect to those key sizes, the algorithm executes 10, 12, or 14 iteration rounds of transformations. The five core operations of the algorithm are KeyExpansion, AddRoundKey, SubBytes, ShiftRows, and MixColumns. A visual flow-graph of the algorithm can be seen in Fig. 16. The input block, called the state array, is constructed as a 4×4 matrix of bytes. The state array goes through the appropriate amount of processing rounds where the subBytes, ShiftRows, MixColumns, and AddRoundKey steps are performed on the state array. After the final round, the state array contains the encrypted data.

For this paper, we focused on the KeyExpansion and MixColumns operations of the AES encryption algorithm with a 128-bit key. Both steps of the algorithm are more computationally intensive than the rest, meaning they do more than a single operation on each element of the state array. The KeyExpansion algorithm takes the four 32-bit words of the input key and expands the key into 44 words. The AddRoundKey operation XORs four of the key words at a time, which requires four words for each round and an additional four during the initial round before the main encryption rounds begin. The SubBytes algorithm makes use of a precomputed array of round constants and a substitution box to generate the next four words based on the key words of the previous iteration or the initial four key words. ShiftRows is a transposition step where the last three rows of the state array a rotated cyclically. Finally, the MixColumns algorithm is simply a matrix multiplication of the state array with a predefined array composed of the values 0×01, 0×02, and 0×03. The arithmetic for the matrix multiplication in this step is done in the Galois field $\mathrm{GF}(2^8)$ in which addition becomes XOR and multiplication becomes bit shifting and XORing.

A. Implementing AES

When implementing AES in hardware, the HLS tools divided themselves into two groups: those capable of compiling the entire source into functioning hardware code and those that had to break each sub-component of the algorithm into individual components to be combined by hand.

VivadoHLS and LegUp were able to compile the C source with little or no modification, and provided good design space exploration. LegUp also allowed the programmer to specify how much of the AES code should be executed in hardware or software. To compile with the Altera OpenCL Compiler, we simply adapt the original C code to have one input and one output stream as the plain text and encrypted text respectively, and add appropriate OpenCL grammars (function qualifiers, attributes, *et al.*). Then the code can be directly compiled by AOC and an executable file is generated. However, as noted previously, there are no human-readable Verilog files and corresponding RTL simulation methods provided by AOC, which prevents examining the generated architecture.

ROCCC and Bluespec System Verilog, however, required more direct effort to handle AES. ROCCC is currently unable to compile source with multiple loops at the same nesting level within the same system. Input streams

Table 3 Area Utilization and Timing Results for the Pass-Through Filter

Tool	Unroll #	LUTs	Registers	BRAM	DSP	Clock Freq.
ROCCC	1	243	298	1	0	338 MHz
	8	645	1479	9	0	337 MHz
	16	1008	2746	16	0	260 MHz
	32	1675	5298	30	0	279 MHz
LegUp	8	282	271	0	0	376 MHz

must be accessed within a for-loop and all inputs will be accessed at every iteration of the loop. In ROCCC each system contains only one top-level loop. Multiple systems can be configured in producer/consumer relationships. In order to implement AES in ROCCC, each component of AES had to be compiled as its own system and then combined together. Since BSV is closer to the HDL level than HLS, each component had to be built by hand with significant effort on the programmer.

VII. SYNTHESIS RESULTS

In this section we report on the FPGA area and performance results where applicable. Note that LegUP and ROCCC were designed with different goals in mind. LegUp focuses on compiling a large subset of the C language. ROCCC focuses on streaming applications, which require a smaller subset of C, with an emphasis on extensive user-directed compile-time transformations and optimizations. We first start with a simple pass-through filter to establish the baseline data for both tools. Results are reported for a Xilinx Virtex-7 (XC7VX690T) using ISE 13.4.

A. Pass-Through Filter

We compiled a pass-through Filter using ROCCC and LegUP to obtain a baseline for the resource utilization in each compiler. Using different unrolling factors we report the results in Table 3. Resource utilization is reported as the total occupied slices (LUTs and registers), BRAM, and DSP elements. We also report the clock frequency after placement and routing. However, our designs did not use pin assignments, or did they specify a target clock frequency. Faster results are likely possible with higher effort in the synthesis tools.

Unrolling is handled very differently in ROCCC than in LegUp. Since ROCCC does not support the use of arbitrary aliases (pointers) within loop bodies, ROCCC can detect the independence of loop iterations and generate parallel and separate loop bodies, one for each unrolled iteration. ROCCC does not make any assumptions regarding the interface to the outside world, e.g., memory, therefore unrolling eight folds would require that eight data elements can be fetched each cycle. As expected the resource utilization scales linearly with the unroll factor. Pass-through is a minimalist design, and most of its resources cannot be shared as the design is unrolled.

LegUp supports the whole C language including pointers. The LegUp architecture assumes two memory channels for all memory requests. Unrolling, by hand, can increase the number of parallel instructions within a loop body, which in turn allows for better utilization of the two memory channels. The compiler does not do any code analysis to identify loop-level parallelism and exploit unrolling. In Table 3 we report an unroll factor of eight because it is the smallest value possible in LegUp. Parallelism could easily be added to the LegUp design by modifying the C code.

Table 4 Dilation Area Utilization

LegUp

unroll	LUTs	reg.	FIFO18E1	FIFO36E1	DSP48
1	3142	4883	0	0	4
8	3142	4883	0	0	4
16	3292	4981	0	0	4

ROCCC

unroll	LUTs	reg.	FIFO18E1	FIFO36E1	DSP48
1	1065	1426	4	4	12
8	2903	4092	18	21	33
16	5486	8133	33	38	102

It is worth mentioning that, due to architectural assumptions made by each compiler, LegUp does not utilize BRAMs in its design, while ROCCC utilizes multiple BRAMs. ROCCC generates a general-purpose kernel for any architecture, which includes architectures having high bandwidth and large memory latencies that often support many outstanding requests. If the datapath should stall for some reason, the outstanding requests will be buffered in the BRAMs. In contrast, LegUp assumes a local cache will handle all memory requests. Therefore, the request latencies are always short and the design will have few outstanding requests.

B. Area Utilization

In Table 4 we report the number of LUTs registers, FIFOs and DSPs used by the LegUp and ROCCC implementation of dilation. For BRAMs, Xilinx Virtex-7 uses FIFO36E1 blocks, which are true dual-port 36 Kb BRAMs.

As the loop is unrolled, the area utilized by the ROCCC implementation grows linearly (from 1065 to 10 763 LUTs) while that of LegUp stays relatively constant. The unrolling in LegUp affects mainly the scheduling of by FSM and does not increase the area used by the logic. In fact, in Table 5, the number of memory reads performed by the LegUp code stays constant while the one by the ROCCC code decreases.

Table 5 Dilation Runtime Performance

LegUp

unroll	Freq(MHz)	Memory Reads	Runtime	
			cycles	ms
1	167	20676040	99453175	595.53
8	167	20676040	99453175	595.53
16	167	20676040	61103567	365.89

ROCCC

unroll	Freq(MHz)	Memory Reads	Runtime	
			cycles	ms
1	155	2071440	99453175	13.39
8	147	259200	99453175	1.76
16	145	129600	61103567	0.90

Table 6 AES Encryption Steps—Area, Frequency, and Power

LegUp

AES step	reg.	LUTs	FIFOs	Freq(MHz)	Power (W)
MixColumns	606	536	0	470	21.59
KeyExpansion	784	827	0	320	15.73

ROCCC

AES step	reg.	LUTs	FIFOs	Freq(MHz)	Power (W)
MixColumns	813	450	6	265	13.59
KeyExpansion	2231	1001	9	370	17.67

C. Runtime Performance

Table 5 shows LegUp and ROCCC's runtime performance on the dilation as the kernel is unrolled. How each tool targets streaming applications is immediately evident in how both tools unroll their designs. As the kernel is unrolled, LegUp will duplicate some of the hardware, but the kernel is always limited to two memory channels. The main performance benefits come from the FSM's memory scheduling. Higher unrolling means more memory request are available sooner, which yields better bandwidth utilization. In fact, unrolling less than eight is not possible in LegUp as evidenced in the table. No unrolling unroll and eight unrolls performs the same because there is no difference in the code. Unrolling is more like a software optimization that improves scheduling than pure hardware replication. On the other hand, ROCCC's focus on streaming applications allows it to fully utilize any memory channels it has available. Therefore, unrolling affects the hardware, input channels, and output channels together. Going from no unroll to 16 unrolls shows a 14.9X speedup compared to LegUp's 2.5X speedup. Comparing the 32 unrolled versions of each, where ROCCC can assume 34 streams, ROCCC shows a 536X speedup.

D. AES Performance and Power

Table 6 shows LegUp and ROCCC's performance and power results for the MixColumns and KeyExpansion algorithms in AES. Power estimates were generated using Xilinx's Power Estimator (XPE) version 2014.2 and using the reported clock rate. It is interesting to note that if the power estimation is kept to the default of 250 MHz, both tools achieve about 13 W—meaning very similar designs architecturally. The main difference comes from the different clock frequencies achieved by each tool. ROCCC was able to perform strongest on KeyExpansion since it is purely compute while LegUp performed well on MixColumns, which benefits from the memory scheduling of LegUps FSM.

VIII. CONCLUSION

In this paper we have addressed one of the main obstacle to a wider adoption of FPGA-based reconfigurable hardware accelerators, namely the programmability of FPGA devices. We have shown that, historically, the use of FPGA accelerators has immediately followed their introduction in the mid 1980s. We have discussed the challenges that must be overcome to raise the abstraction level of FPGA programming to levels similar to those in software. We have identified five high-level programming tools currently available, both research and commercially, that attempt to raise the abstraction level and make it easier for traditionally trained software developers to write applications for FPGA accelerators. These tools are Xilinx Vivado HLS, Altera OpenCL Compiler, Bluespec BSV, LegUp (University of Toronto), and ROCCC (University of California Riverside). We describe the user interaction in using these tools and, using image dilation filter and AES encryption algorithm, we report on how the code is expressed and compiled and discuss the resulting utilization. A summary of the main features of these HLS tools is in Table 7. ■

Table 7 Summary of Tools Features

	VivadoHLS	OpenCL	BSV	LegUp 3.0	ROCCC 2.0
Developed By	Xilinx Inc.	Altera	Bluespec	University of Toronto	UC Riverside
Targets	Xilinx FPGAs	Generic(includes GPUs)	Generic	Mostly Generic	Generic
Origin	Commercial	Commercial	Commercial	Academic	Academic
Learning Curve	Small	Moderate	Large	Small	Small
Interface & Optimizations	GUI development. Local, user driven optimizations	Command line interface. Basic global optimizations	GUI development hand optimized.	Command line interface. Basic global optimizations (.tcl scripts necessary for hardware optimizations)	GUI development. Local, user driven optimizations

REFERENCES

[1] R. Moussalli, I. Absalyamov, M. R. Vieira, W. Najjar, and V. J. Tsotras, "High performance FPGA and GPU complex pattern matching over spatio-temporal streams," *GeoInformatica*, pp. 1–30, 2014.

[2] R. Moussalli, M. Salloum, R. Halstead, W. Najjar, and V. J. Tsotras, "A study on parallelizing XML path filtering using accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, p. 93, 2014.

[3] P. Bertin, D. Roncin, and J. Vuillemin, *Introduction to Programmable Active Memories*. Upper Saddle River, NJ: Prentice Hall, 1989, pp. 300–309.

[4] J. Vuillemin, P. Bertin, D. Roncin, M. Sh, H. Touati, A. H., and P. Boucard, "Programmable active memories: Reconfigurable systems come of age," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 4, no. 1, pp. 56–69, Mar. 1996.

[5] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable active memories: A performance assessment," in *Parallel Architectures and Their Efficient Use. Paderborn, Germany, Lecture Notes in Computer Science*. New York, NY, USA: Springer Verlag, 1992, pp. 119–130.

[6] N. K. Ratha, D. T. Jain, and D. T. Rover, "Fingerprint matching on Splash 2," in *Splash 2: FPGAs in a Custom Computing Machine*. Los Alamitos, CA, USA: IEEE CS Press, 1996, pp. 117–140.

[7] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*. Los Alamitos, CA, USA: IEEE CS Press, 1996.

[8] N. K. Ratha, D. T. Jain, and D. T. Rover, "Convolution on Splash 2," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1995, pp. 204–213.

[9] D. Hoang, "Searching genetic databases on Splash 2," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, 1993, pp. 185–192.

[10] D. V. Pryor, M. R. Thistle, and N. Shirazi, "Text searching on Splash 2," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, 1993, pp. 172–178.

[11] X. Ma, W. Najjar, and A. Roy-Chowdhury, "Evaluation and acceleration of high-throughput fixed-point object detection on FPGAs," *IEEE Trans. Circuits Syst. Video Technol.*, to be published.

[12] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design Test Comput.*, vol. 23, no. 5, pp. 375–386, 2006.

[13] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul./Aug. 2009.

[14] J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Comput. Surv.*, vol. 42, pp. 13:1–13:65, Jun. 2010.

[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[16] E. El-Araby, S. G. Merchant, and T. El-Ghazawi, "A framework for evaluating high-level design methodologies for high-performance reconfigurable computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 33–45, Jan. 2011.

[17] W. Meeus, K. V. Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, 2012.

[18] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Adv. Syst. Sci.*, vol. 240, *Adv. Intell. Syst. Comput.*, J. Swiątek, A. Grzech, P. Swiatek, and J. M. Tomczak, Eds. New York, NY, USA: Springer, 2014, pp. 483–492.

[19] T. Feist, "Vivado design suite," 2012, vol. 1, Xilinx, White Paper Ver.

[20] Xilinx Vivado, 2013. [Online]. Available: http://www.xilinx.com/tools/autoesl_instructions.htm

[21] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based ESL synthesis system ," *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds. New York, NY, USA: Springer Verlag, 2008.

[22] S. Neuendorffer, T. Li, and D. Wang, "Accelerating OpenCV applications," Xilinx, Inc.

[23] LLVM. [Online]. Available: http://llvm.org

[24] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *Int. Conf. FPT*, Dec. 2013, pp. 362–365.

[25] D. Navarro, O. Lucia, L. A. Barragan, I. Urriza, and O. Jimenez, "High-level synthesis for accelerating the FPGA implementation of computationally demanding control algorithms for power converters," *IEEE Trans. Ind. Informat.*, vol. 9, pp. 1371–1379, Aug. 2013.

[26] J. Hiraiwa and H. Amano, "An FPGA implementation of reconfigurable real-time vision architecture," in *Adv. Inf. Netw. Appl. Workshops*, Mar. 2013, pp. 150–155.

[27] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Commun. ACM*, vol. 56, pp. 56–63, Apr. 2013.

[28] Khronos. [Online]. Available: https://www.khronos.org/news/press/2008/12

[29] N. Trevett, "OpenCL introduction," in *SIGGRAPH Asia*, 2013.

[30] D. Singh, "Implementing FPGA design with the OpenCL standard," Altera Corp.

[31] D. Chen and D. Singh, "Invited paper: Using OpenCL to evaluate the efficiency of CPUs, GPUs, and FPGAs for information filtering," *Int. Conf. Field Programmable Logic Appl.*, pp. 5–12, Aug. 2012.

[32] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," *Int. Conf. Field Programmable Logic Appl.*, pp. 531–534, 2012.

[33] D. Chen and D. Singh, "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms," in *Design Automat. Conf.*, Jan. 2013, pp. 297–304.

[34] R. S. Nikhil Arvind, "What is Bluespec?" *SIGDA Newslett.*, vol. 39, pp. 1–1, Jan. 2009.

[35] R. Nikhil, "Bluespec SystemVerilog: Efficient, correct RTL from high level specifications," in *Proc. 2nd ACM IEEE Int. Conf. Formal Methods Models for Co-Design*, Jun. 2004, pp. 69–70.

[36] *Bluespec SystemVerilog Reference Guide, rev. 17*, Jan. 2012.

[37] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. D. Brown, and T. Czajkowski, "Legup: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. ACM 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2011, pp. 33–36.

[38] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, p. 24, 2013.

[39] TigerMIPS. [Online]. Available: http://www.program-transformation.org/tiger/mips

[40] Avalon Interface Specifications. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf

[41] A. Canis *et al.*, "From software to accelerators with legup high-level synthesis," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embed. Syst.*, 2013, p. 18.

[42] LegUp. [Online]. Available: http://legup.eecg.utoronto.ca/

[43] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proc. 18th IEEE Annu. Int. Symp. Field-Program. Custom Comp. Mach.*, May 2010, pp. 127–134.

[44] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Notices*, vol. 29, pp. 31–37, Dec. 1994.

[45] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis transformation," presented at the Int. Symp. Code Generat. Optimiza., Palo Alto, CA, Mar. 2004.

[46] Z. Guo, B. Buyukkurt, J. Cortes, A. Mitra, and W. A. Najjar, "A compiler intermediate representation for reconfigurable fabrics," *Int. J. Parallel Program.*, vol. 36, no. 5, pp. 493–520, 2008.

[47] B. Buyukkurt, Z. Guo, and W. A. Najjar, "Impact of high level transformations within the ROCCC framework," *ACM Trans. Architect. Code Optimization*, vol. 7, no. 4, Dec. 2006.

[48] Z. Guo, B. Buyukkurt, and W. A. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *ACM SIGPLAN/SIGBED Conf. Languages, Compilers Tools Embed. Syst.*, New York, NY, USA, Jun. 2004, pp. 249–256.

[49] B. Buyukkurt and W. A. Najjar, "Compiler generated systolic arrays for wavefront algorithm acceleration on FPGAs," in *Int. Conf. Field Program. Logic Appl.*, Heidelberg, Germany, 2008, pp. 655–658.

[50] J. Hammes, A. P. W. Böhm, C. Ross, M. Chawathe, B. A. Draper, R. Rinker, and W. A. Najjar, "Loop fusion and temporal common subexpression elimination in window-based loops," in *Proc. 15th Int. Parallel Distrib. Process. Symp.*, San Francisco, CA, 2001, p. 142.

[51] *Riverside Optimizing Compiler for Configurable Computing*, 2012. [Online]. Available: http://roccc.cs.ucr.edu/

[52] Altera SDK for OpenCL Programming Guide.

[53] Altera SDK for OpenCL Best Practice Guide.

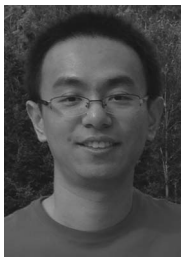
[54] *FIPS PUB 197: Advanced Encryption Standard (AES)*, Nat. Inst. Stand. Technol., Nov. 2001.

[55] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, "Design and implementation of low-area and low-power AES encryption hardware core," *Digital Syst. Design: Archit., Methods Tools*, pp. 577–583, 2006.

ABOUT THE AUTHORS

Skyler Windh is currently a graduate student in the Embedded Systems Laboratory at the Department of Computer Science at University of California, Riverside, CA, USA.

His current research interests involve improving database and data mining operations using hardware accelerators such as FPGAs and GPUS. He has also done some work in routing algorithms for microfluidic labs-on-a-chip.



Xiaoyin Ma (Student Member, IEEE) is currently pursuing the Ph.D. degree at the University of California, Riverside, CA, USA, where he received the Master's degree in electrical engineering in 2012. He received the Bachelor's degree in electronic science and technology from Huazhong University of Science and Technology, Wuhan, China.

His research interests include embedded systems, FPGA-based code acceleration, parallel and high-performance computing, and their applications in computer vision and image/video processing.

Robert J. Halstead is currently pursuing the Ph.D. degree at the University of California, Riverside, CA, USA.

His research interests are in the area of reconfigurable computing, HPC, and database. In particular, he is looking at how hardware multithreaded techniques can be used to accelerate irregular applications on FPGA accelerators. He has also looked at incorporating these techniques into HLS compiler tools to improve hardware development times.

Prerna Budhkar is currently pursuing the Ph.D. degree at the University of California, Riverside, CA, USA.

Her research interest includes FPGA-based code acceleration and its application in database operations and graph processing.

Zabdiel Luna received the B.S. degree in computer engineering from University of California, Riverside, CA, USA, in 2014.

He is now a Software Engineer at Zodiac Aerospace Corporation working on embedded systems.

Omar Hussaini, photograph and biography not available at the time of publication.

Walid A. Najjar (Fellow, IEEE) received the B.E. degree in electrical engineering from the American University of Beirut in 1979, and the M.S. and Ph.D. degrees in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 1985 and 1988, respectively.

He is a Professor in the Department of Computer Science and Engineering at the University of California, Riverside, CA, USA. His areas of research include computer architectures and compilers for parallel and high-performance computing, embedded systems, FPGA-based code acceleration, and reconfigurable computing.

Dr. Najjar is a Fellow of the AAAS.