acmqueue FPGA Programming for the Masses

The programmability of FPGAs must improve if they are to be part of mainstream computing.

David F. Bacon, Rodric Rabbah, Sunil Shukla, T.J. Watson Research Center

When looking at how hardware influences computing performance, we have GPPs (general-purpose processors) on one end of the spectrum and ASICs (application-specific integrated circuits) on the other. Processors are highly programmable but often inefficient in terms of power and performance. ASICs implement a dedicated and fixed function and provide the best power and performance characteristics, but any functional change requires a complete (and extremely expensive) re-spinning of the circuits.

Fortunately, several architectures exist between these two extremes. PLDs (programmable logic devices) are one such example, providing the best of both worlds. They are closer to the hardware and can be reprogrammed.

The most prominent example of a PLD is an FPGA (field programmable gate array). It consists of LUTs (look-up tables), which are used to implement combinational logic; and FFs (flip-flops), which are used to implement sequential logic. Apart from the homogeneous array of logic cells, an FPGA also contains discrete components such as BRAMs (block RAMs), DSP (digital signal processing) slices, processor cores, and various communication cores (e.g., Ethernet MAC and PCIe).

BRAMs, which are specialized memory structures distributed throughout the FPGA fabric in columns, are of particular importance. Each BRAM can hold up to 36 Kbits of data. BRAMs can be used in various form factors and can be cascaded to form a larger logical memory structure. Because of the distributed organization of BRAMs, they can provide terabytes of bandwidth for memory bandwidth–intensive applications. Xilinx and Altera dominate the PLD market, collectively holding more than an 85 percent share.

FPGAs were long considered low-volume, low-density ASIC replacements. Following Moore's law, however, FPGAs are getting denser and faster. Modern-day FPGAs can have up to 2 million logic cells, 68 Mbits of BRAM, more than 3,000 DSP slices, and up to 96 transceivers for implementing multigigabit communication channels.[23] The latest FPGA families from Xilinx and Altera are more like an SoC (system-on-chip), mixing dual-core ARM processors with programmable logic on the same fabric. Coupled with higher device density and performance, FPGAs are quickly replacing ASICs and ASSPs (application-specific standard products) for implementing fixed function logic. Analysts expect the programmable IC (integrated circuit) market to reach the $10 billion mark by 2016.[20]

The most perplexing fact is that an FPGA running at a clock frequency that is an order of magnitude lower than CPUs and GPUs (graphics processing units) is able to outperform them. In several classes of applications, especially floating-point-based ones, GPU performance is either slightly better than or very close to that of an FPGA. When it comes to power efficiency (performance per watt), however, both CPUs and GPUs lag significantly behind FPGAs, as shown in the available literature comparing the performance of CPUs, GPUs, and FPGAs for different classes of applications.[6,18,19]

The contrast in performance between processors and FPGAs lies in the architecture itself. Processors rely on the Von Neumann paradigm, where an application is compiled and stored in instruction and data memory. They typically work on an instruction and data fetch-decode-execute-store pipeline. This means both instructions and data have to be fetched from an external memory into the processor pipeline. Although caches are used to alleviate the cost of expensive fetch operations from external memory, each cache miss incurs a severe penalty. The bandwidth between processor and memory is often the critical factor in determining the overall performance. The phenomenon is also known as "hitting the memory wall."[21]

FPGAs have programmable logic cells that could be used to implement an arbitrary logic function both spatially and temporally. FPGA designs implement the data and control path, thereby getting rid of the fetch and decode pipeline. The distributed on-chip memory provides much-needed bandwidth to satisfy the demands of concurrent logic. The inherent fine-grained architecture of FPGAs is very well suited for exploiting various forms of parallelism present in the application, ranging from bit-level to task-level parallelism. In addition to the conventional reconfiguration capability where the entire FPGA fabric is programmed with an image before execution, FPGAs are also capable of undergoing partial dynamic reconfiguration. This means part of the FPGA can be loaded with a new image while the rest of the FPGA is functional. This is similar to the concept of paging and virtual memory in the processor taxonomy.

Various kinds of FPGA-based systems are available today. They range from heterogeneous systems targeted at HPC (high-performance computing) that tightly couple FPGAs with conventional CPUs (e.g., Convey Computers), to midrange commercial-off-the-shelf workstations that use PCIe-attached FPGAs, to low-end embedded systems that integrate embedded processors directly into the FPGA fabric or on the same chip.

PROGRAMMING CHALLENGES

Despite the advantages offered by FPGAs and their rapid growth, use of FPGA technology is restricted to a narrow segment of hardware programmers. The larger community of software programmers has stayed away from this technology, largely because of the challenges experienced by beginners trying to learn and use FPGAs.

ABSTRACTION

FPGAs are predominantly programmed using HDLs (hardware description languages) such as Verilog and VHDL. These languages, which date back to the 1980s and have seen few revisions, are very low level in terms of the abstraction offered to the user. A hardware designer thinks about the design in terms of low-level building blocks such as gates, registers, and multiplexors. VHDL and Verilog are well suited for describing a design at that level of abstraction. Writing an application at a behavioral level and leaving its destiny in the hands of a synthesis tool is commonly considered a bad design practice.

This is in complete contrast with the software programming languages, which have evolved over the past 60 years. Programming for CPUs enjoys the benefits of well-established ISAs (instruction set architectures) and advanced compilers that offer a much simpler programming experience. Object-oriented and polymorphic programming concepts, as well as automatic memory management (garbage collection) are no longer seen just as desirable features but as necessities. A higher level of abstraction drastically increases a programmer's productivity and reduces the likelihood of bugs, resulting in a faster time to market.

SYNTHESIZABILITY

Another language-related issue is that only a narrow subset of VHDL and Verilog is synthesizable. To make matters worse, there is no standardization of the features supported by different EDA (electronic design automation) tools. It is largely up to the EDA tool vendors to decide which language features they want to support.

While support for data types such as char, int, long, float, and double is integral to all software languages, VHDL and Verilog have long supported only bit, array of bits, and integer. A recent revision to the VHDL standard (IEEE 1076-2008) introduced fixed-point and floating-point types, but most, if not all, synthesis tools do not support these abstractions yet.

Figure 1 is a C program that converts Celsi us to Fahrenheit. The language comes with standard libraries for a number of useful functions. To implement the same program in VHDL or Verilog for FPGA implementation, the user has to generate a netlist (precompiled code) using FPGA vendor-specific IP (intellectual property) core-generator tools (e.g., Coregen for Xilinx and MegaWizard for Altera). Then the generated netlist is connected in a structural manner. Figure 2 shows the top-level wrapper consisting of structural instantiation of leaf-level modules; namely, a double-precision floating-point multiplier (fp mult) and adder (fp add). (For brevity, the leaf-level modules are not shown here.)

VERIFICATION

Design and verification go hand in hand in the hardware world. Verification is easily the largest chunk of the overall design-cycle time. It requires that a test bench be created either in HDL or by using a high-level language such as C, C++, or SystemC that is connected to the DUT (design under test) using the FLI (foreign language interface) extension provided by VHDL and Verilog. The DUT and test bench are simulated using event-based simulators (e.g., Mentor Graphics ModelSim and Cadence Incisive Unified Simulator). Each signal transaction is recorded and displayed as a waveform. Waveform-based debugging works for small designs but can quickly become tedious as

FIGURE 1

Celsius to Fahrenheit Conversion in C

```
#include <stdio.h>
#include <stdlib.h>

double main (int argc, char* argv[]) {
double celsius = atof(argv[1]);
return (9*celsius/5 + 32);
}
```

design size grows. Also, the simulation speed decreases very quickly as the design size increases. In general, simulations are many orders of magnitude slower than the real designs running in hardware. Often, large-system designers have to turn to prototyping (emulation) using FPGAs to speed up design simulation. This is in complete contrast with the way software verification works. From simple `printf` statements to advanced static and dynamic analysis tools, a software programmer has numerous tools and techniques available to debug and verify programs in a much simpler yet powerful manner.

DESIGN AND TOOL FLOW

Apart from the language-related challenges, programming for FPGAs requires the use of complex and time-consuming EDA tools. Synthesis time—the time taken to generate bitstreams (used to program the FPGA) from source code—can vary anywhere from a few minutes to a few days depending on the code size and complexity, target FPGA, synthesis tool options, and design constraints.

FIGURE 2

Celsius to Fahrenheit Conversion in Verilog

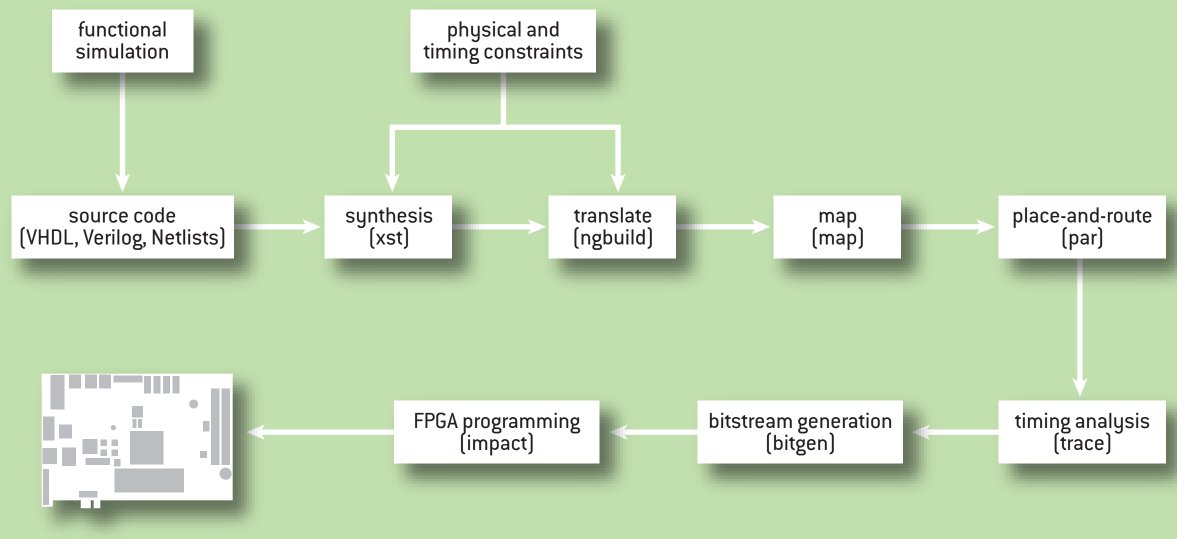
```
module c2f (
input clk,
input rst,
input [63:0] celsius,
input input_valid,
output [63:0] fahrenheit,
output result_valid);

localparam double_9div5 = 64'h3FFCCCCCCCCCCCCD; // 1.8 (=9/5)
localparam double_32 = 64'h4040000000000000; // 32.0
wire [63:0] temp;

// temp = 1.8 * celsius
fp_mult mult_9div5_i (.clk(clk), .rst(rst),
.in1(celsius), .in2(double_9div5), .in_valid(input_valid),
.out(temp), .out_valid(temp_valid));

// fahrenheit = temp + 32
fp_add add_32_i (.clk(clk), .rst(rst),
.in1(temp), .in2(double_32), .in_valid(temp_valid),
.out(fahrenheit), .out_valid(result_valid));

endmodule
```

Figure 3 shows the design flow for Xilinx FPGAs.[22] After functional verification, the source code—consisting of VHDL, Verilog, and netlists—is fed to the synthesis tool. The *synthesis phase* compiles the code and produces a netlist (.ngc). The *translate phase* merges all the synthesized netlists and the physical and timing constraints to produce an NGD (native generic database) file. The *map phase* groups logical symbols from the netlists into physical components such as LUTs, FFs, BRAMs, DSP slices, IOBs (input/output blocks), etc. The output is stored in NCD (native circuit description) format and contains information about switching delays. The map phase results in an error if the design exceeds the available resources or the user-specified timing constraints are violated by the switching delay itself. The *PAR (place-and-route) phase* performs the placement and routing of the mapped symbols on the actual FPGA device. For some FPGA devices, placement is done in the map phase. PAR stores the output in NCD format and notifies the user of timing errors if the total delay (switching plus routing) exceeds the user-specified timing constraints.

Following PAR, timing analysis is performed to generate a detailed timing report consisting of all delays for different paths. Users can refer to the timing analysis report to determine which critical paths fail to meet the timing requirements and then fix them in the source code. Most of the synthesis errors mean the user has to go back to the source code and repeat the whole process. Once the design meets the resource and timing constraints, a bitstream can be generated. The bitgen tool takes in a completely routed design and generates a configuration bitstream that can be used to program the FPGA. Finally, the design can be downloaded to the FPGA using the Impact tool.

The design flow described in figure 3 varies from one FPGA vendor to another. Moreover, to make good use of the synthesis tool, the user must have good knowledge of the target FPGA architecture because several synthesis options are tied to the architecture. The way these options are tuned ultimately decides whether the design will meet resource and timing constraints.

FIGURE 3 **Design Flow for Xilinx FPGAs**


functional simulation
physical and timing constraints
source code (VHDL, Verilog, Netlists)
synthesis (xst)
translate (ngbuild)
map (map)
place-and-route (par)
timing analysis (trace)
bitstream generation (bitgen)
FPGA programming (impact)

Design portability is another big challenge, because porting a design from one FPGA to another often requires re-creation of all the FPGA device-specific IP cores. Migrating designs from one FPGA vendor to another is an even bigger challenge.

HIGH-LEVEL SYNTHESIS

HLS (high-level synthesis) refers to the conversion of an algorithmic description of an application into either a low-level RTL (register-transfer level) description or a digital circuit. Over the past decade or so, several languages and compiler frameworks have been proposed to ease the pain of programming FPGAs. They all aim to raise the level of abstraction at which the user can write a program to compile it down to VHDL or Verilog. Most of these frameworks can be classified into five categories: HDL-like languages; C-based frameworks; CUDA (Compute Unified Device Architecture)/ OpenCL (Open Computing Language)-based frameworks; high-level language–based frameworks; and model-based frameworks.

HDL-LIKE LANGUAGES

Although only minute incremental changes have been made to VHDL and Verilog since they were introduced, SystemVerilog represents a big jump. It has some semantic similarity to Verilog, but it is more dissimilar than similar. SystemVerilog consists of two components: the synthesizable component that extends and adds several features to the Verilog-2005 standard; and the verification component, which heavily uses an object-oriented model and is more akin to object-oriented languages such as Java than it is to Verilog. SystemVerilog substantially improves on the type system and parametrizability of Verilog. Unfortunately, EDA tools offer limited support for the language today: the Xilinx synthesis tool does not support SystemVerilog, and the Altera synthesis tool supports a small subset.

Another notable language in this category is BSV (Bluespec SystemVerilog), which incorporates concepts such as modules, module instances, and module hierarchies just like Verilog and VHDL, and follows the SystemVerilog model of interfaces to communicate between modules. In Bluespec, module behavior is expressed using guarded atomic actions that are basically concurrent FSMs (finite state machines), a concept well known to hardware programmers. A program written in Bluespec is passed on to the Bluespec compiler, which generates an efficient RTL description. A 2011 article in *ACM Queue* provides a comprehensive description of the language and compiler.[14]

C-BASED FRAMEWORKS

Most of the early work in the field of HLS was done on C-based languages such as C, C++, and SystemC. Most of these frameworks restrict the programmer to only a small subset of the parent language. Features such as pointers, recursive functions, and dynamic memory allocations are prohibited. Some of these frameworks require users to annotate the program extensively before compilation. A comprehensive list of C-based frameworks can be found in João M. P. Cardoso and Pedro C. Diniz's book, *Compilation Techniques for Reconfigurable Architectures*.[7]

Interestingly, all the EDA vendors have developed or acquired tool suites to enable high-level synthesis. Their HLS frameworks (e.g., Cadence C-to-Silicon Compiler, Synopsys Synphony C Compiler, Mentor Graphics Catapult C, and Xilinx Vivado), with the exception of the one proposed by Altera, are based on C-like languages. The following section looks at the Xilinx Vivado HLS tool.

Xilinx Vivado (previously AutoPilot). In 2011 Xilinx acquired the AutoPilot framework[5,8] developed by AutoESL and is now offering it as a part of the Vivado Design Suite. It supports compilation from a behavioral description written in C/C++/SystemC to RTL. The behavioral synthesis consists of four phases: compilation and elaboration; advanced code transformation; core behavioral and communication synthesis; and microarchitecture generation.

The behavioral code in C/C++/SystemC is parsed and compiled by a GCC (GNU Compiler Collection)-compatible compiler front end. Several hardware-oriented features are added to the front end, such as bit-width optimization for data types to use exactly the right number of bits required to represent a data type. For SystemC designs, an elaboration phase extracts processes, ports, and other interconnect information, and produces a synthesis data model.

The synthesis data model undergoes several compiler optimizations (e.g., constant propagation, dead-code elimination, and bit-width analysis) in the advanced code transformation phase.

In the core behavioral and communication synthesis phase, Vivado takes into account the user-specified constraints (e.g., frequency, area, throughput, latency, and physical location constraints) and the target FPGA device architecture to do scheduling and resource binding.

In the microarchitecture generation phase, the compiler backend produces VHDL/Verilog RTL together with the synthesis constraints, which could be passed as an input to the synthesis tool. The tool also generates an equivalent description in SystemC, which could be used to do verification and equivalence checking of the generated RTL code.

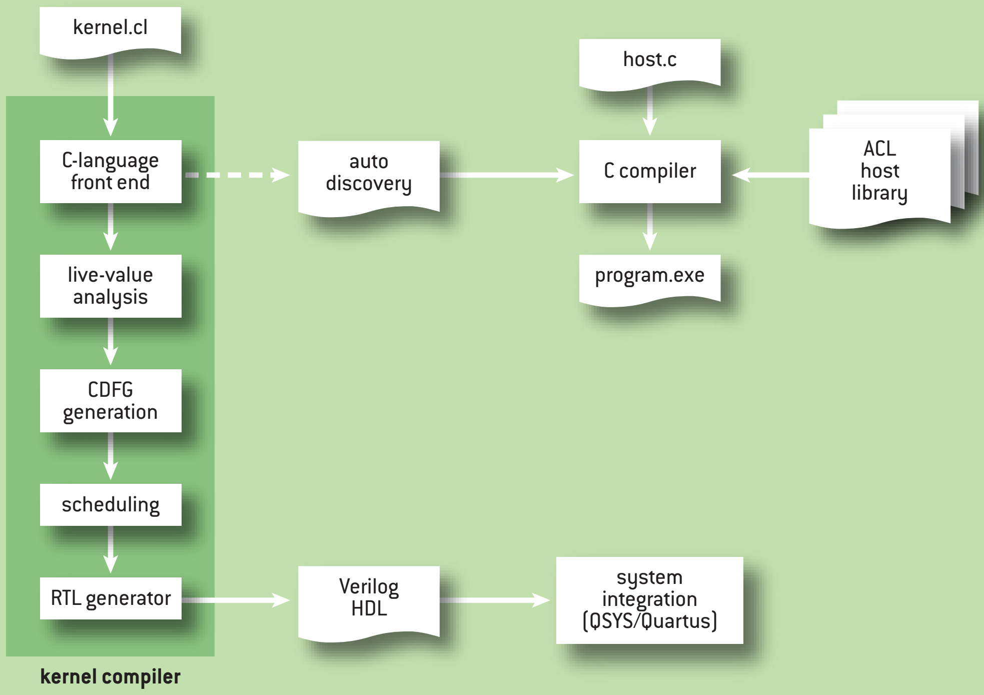
CUDA/OPENCL-BASED FRAMEWORKS

The popularity of GPUs for general-purpose computing soared with the release of the CUDA framework by Nvidia in 2006. This framework consists of language and runtime extensions to the C programming language and driver-level APIs for communication between the host and GPU devices. With CUDA, researchers and developers can exploit massive SIMD (single instruction, multiple data)-style architectural parallelism for running computationally intensive tasks. In 2008, OpenCL, was made public as an open standard for parallel programming on heterogeneous systems. OpenCL supports execution on targets such as CPUs and GPUs. To leverage the popularity of CUDA and OpenCL, a number of compiler frameworks have been proposed to convert CUDA[16] and OpenCL[3,13,15] code to VHDL/Verilog. One of these products is the Altera OpenCL framework.

Altera OpenCL-to-FPGA Framework. Altera proposed a framework for synthesizing bitstreams from OpenCL[3]. Figure 4 shows the Altera OpenCL-to-FPGA framework for compiling an OpenCL program to Verilog for co-execution on a host and Altera FPGA. The framework consists of a kernel compiler, a host library, and a system integration component. The kernel compiler, based on the open source LLVM compiler infrastructure, synthesizes OpenCL kernel code into hardware. The host library provides APIs and bindings for establishing communication between the host part of the application running on the processor and the kernel part of the application running on the FPGA. The system integration component wraps the kernel code with memory controllers and a communication interface (such as PCIe).

The user application consists of a host component written in C/C++ and a set of kernels written in OpenCL. The OpenCL kernels are compiled using the open source LLVM infrastructure. The parser completes the first step in the compilation, producing an IR (intermediate representation) consisting of basic blocks connected by control-flow edges.

FIGURE 4 **Altera OpenCL-to-FPGA Framework**



A live variable analysis determines the input and output of each basic block. A basic block consists of three types of nodes: the merge node is responsible for aggregating data from previous nodes; the operational node represents load, store, and execute instructions; and the branch node selects one thread successor among many basic blocks.

After live variable analysis, the IR is optimized to generate a CDFG (control-data flow graph). RTL for each basic block is generated from the CDFG. An execution schedule is calculated for each node in the CDFG using the SDC (system of difference constraints) scheduling algorithm. The scheduler uses linear equations to schedule instructions while minimizing the cost function.

The hardware logic generated by the kernel compiler for the nodes in the CDFG is added with stall, valid, and data signals that implement the control edges of the CDFG. The resultant Verilog code is wrapped with the communication (e.g., PCIe) and memory infrastructure to interact with the host and the off-chip memory, respectively. A template-based design methodology is used where the application-independent communication and memory infrastructure is locked down as a static part of the system and only the application-dependent kernel part is synthesized. This hierarchical, partition-based approach reduces synthesis time. The design is then synthesized to generate a bitstream.

Altera also provides a host library consisting of APIs to bind the host function calls to the OpenCL kernels implemented in the FPGA. The host-side code written in C/C++ is linked to these libraries, enabling runtime communication and synchronization between the application code running on the host processor and the OpenCL kernels running on the FPGA. The host library also consists of the Auto-Discovery module, which allows the host program to query and detect the types of kernels running on the FPGA.

HIGH-LEVEL LANGUAGE-BASED FRAMEWORKS

Because of the popularity of modern programming languages, several recently proposed frameworks use high-level languages as starting points for generating hardware circuits. The languages in this category are highly abstract, usually object-oriented, and offer high-level features such as polymorphism and automatic memory management. Some of the noteworthy languages and frameworks are Esterel,[23] Kiwi,[12] Chisel,[4] and IBM's Liquid Metal[1].

Liquid Metal. The Liquid Metal project at IBM aims to provide not only high-level synthesis for FPGAs,[1] but also, more broadly, a single unified language for programming heterogeneous architectures. The project offers a language called Lime,[2] which can be used to program hardware (FPGAs), as well as software running on conventional CPUs and more exotic architectures such as GPUs.[10] The Lime language was founded on principles that eschew the complicated program analysis that plagues C-based frameworks, while also offering powerful programming models that espouse the benefits of functional and stream-oriented programming.

Notable features of Lime include the ability to express bit literals and computation at the granularity of individual bits—just as in HDLs—but with the power of high-level abstractions and object-oriented programming. These permit, for example, the definition of generic classes that are parameterized by bit width, polymorphic methods, and overloaded operators.

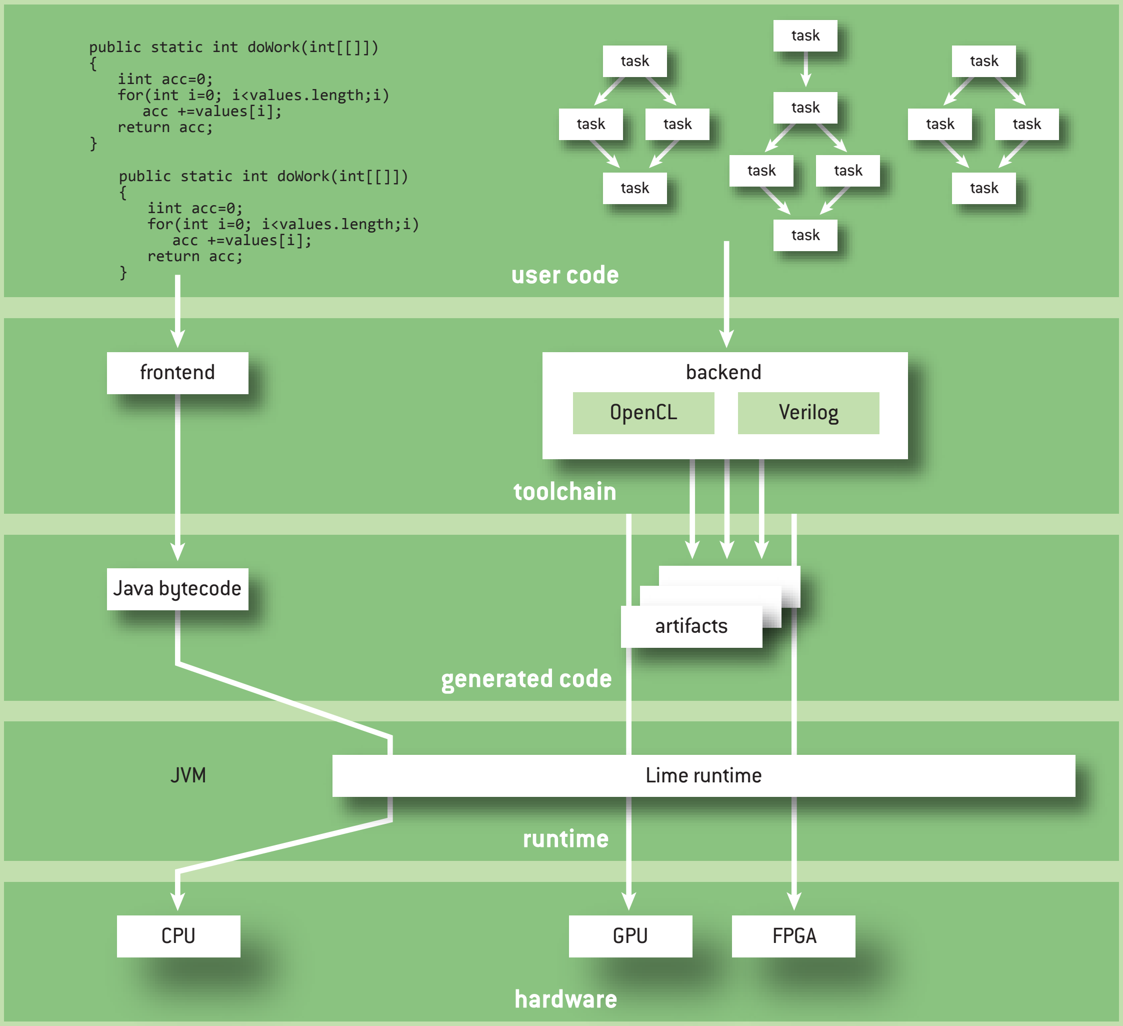
Lime is Java-compatible and hence strongly typed. The language adds features to aid the compiler in deriving important properties for efficient behavioral synthesis into HDL. These include instantiation-based generics, bounded arrays, immutable types, and localized side effects that are guaranteed by the type-checker and exploited by the compiler for the purpose of generating efficient and pipelined circuits.

Another important feature of Lime is a task-based data-flow programming model that allows for program partitioning at task granularity, often resembling a block-level diagram of an architecture or FPGA circuit.

A task in Lime can be strongly isolated such that it cannot access a mutable global program state (because it's prohibitively expensive in an FPGA), and no external references to the task exist, so that its state cannot be mutated except from within the task, if at all. Task graphs result from connecting tasks together (using a first-class connect operator), so that the output of one becomes the input to another.

Figure 5 shows the Liquid Metal compilation and runtime architecture. The front-end compiler first translates the Lime program into an intermediate representation that describes the task graph. Each of the compiler backends is a vertically integrated tool chain that not only compiles the code for the intended architecture, but also generates an executable artifact suitable for the architecture. For example, the Verilog backend, which synthesizes HDL from Lime, will also automatically invoke the EDA tools for the target FPGA to perform logic synthesis and generate a bit file that can be loaded onto the FPGA. Each backend provides a set of exclusion rules that restrict the Lime language to a synthesizable subset. Some notable exclusions are the use of unbounded arrays and non-final classes. Dynamic memory allocation is restricted to final fields of objects, and recursive method calls may

FIGURE 5 **Liquid Metal Compilation and Runtime Architecture**



```
public static int doWork(int[[]])
{
   iint acc=0;
   for(int i=0; i<values.length;i)
      acc +=values[i];
   return acc;
}

   public static int doWork(int[[]])
   {
      iint acc=0;
      for(int i=0; i<values.length;i)
         acc +=values[i];
      return acc;
   }
```

have no more than one argument. These rules are expected to change over time as the compilation technology matures.

Lime code is always executable on at least one architecture: the JVM (Java Virtual Machine). As such, functional verification of Lime programs may be carried out entirely in software using high-level debugging tools, including the Eclipse-based Lime IDE. Tasks that are mapped to the FPGA may co-execute with tasks mapped to other parts of the system (e.g., GPU or JVM), and the communication throughout the system is orchestrated by the Lime runtime.

The language allows programmers to seamlessly integrate native HDL code into their Lime code via the Lime Native Interface. This allows the programmer to implement timing- and performance-critical components of the application in HDL and the rest of the application in Lime in order to meet performance specifications that may not be readily achievable through the current behavioral synthesis technology.

MODEL-BASED FRAMEWORKS

Model-based frameworks provide an abstract way of designing complex control- and signal-processing systems. They improve design quality and accelerate design and verification tasks by using an executable specification. The executable specification facilitates hardware-software partitioning, verification, and rapid design iterations. NI (National Instruments) LabView[9] and Matlab HDL Coder[17] are two Model-based frameworks. They are especially useful for designers with strong domain expertise but little experience with software/hardware programming languages.

NI LabView. NI LabView is a graphical programming and design language. Using functional blocks and interconnects, designers can create a graphical program that looks like their whiteboard designs. LabView provides an integrated environment that consists of a graphical editor for development; debugger; compilation framework; and middleware for execution on diverse targets such as desktop processors, PowerPC, ARM processors, and a number of FPGAs. With LabView, the user can start development and testing on one platform and then incrementally migrate to another platform.

Matlab HDL Coder. Matlab is a high-level language and interactive environment for numerical and scientific computation and system modeling. It provides a number of tools and function-rich libraries that could be used for quickly implementing algorithms and model systems for a wide range of application domains. Simulink, a part of the Matlab product suite, provides a block-diagram-based graphical environment for system simulation and model-based design. Simulink is basically a collection of libraries (toolboxes) for different application domains. HDL Coder is one such toolbox that generates synthesizable VHDL and Verilog code from Matlab functions and Simulink models. Users can model their systems in Simulink using library blocks and Matlab functions and then easily test the models against the functional specifications by creating a test bed using the same modular approach. After verification, users can generate bit-optimized and cycle-accurate synthesizable RTL code.

FUTURE DIRECTIONS

The era of frequency scaling in microprocessor design is largely believed to be over, and programmers are increasingly turning to heterogeneous architectures in search of better performance.

We have already started to see the integration of general-purpose processors and GPUs on the same die. The diversity is expected to increase in the future to include ASSPs, and even programmable logic (e.g., FPGAs). The key is tighter integration of this diverse set of compute elements. The time is not far off when CPUs, GPUs, FPGAs, and other ASSPs will be integrated on the same chip. There are several standards and specifications for co-programming CPUs and GPUs, but FPGAs have been isolated so far, primarily because of the lack of support for device drivers, programming languages, and tools.

There is a serious need to improve the programming aspect of FPGAs if they are to join mainstream heterogeneous computing. Programming technology for FPGAs has lagged behind the advancements in semiconductor technology. As discussed here, HDLs are too low level. Significant design-cycle time must be spent in coding and verification. Design iterations are quite expensive and prohibit thorough design-space exploration.

HLS presents a promising direction, but even after a few decades of research in the area, FPGA programming practices continue to be fragmented and challenging, placing a high engineering burden on researchers and developers. Some problems arise because too many efforts are going on simultaneously without any standardization, yet none provide a comprehensive ecosystem to foster adoption and growth. Many of the ongoing endeavors are centered on C-based languages. Only time will tell if C is the right abstraction for the future, since it was designed for a sequential execution model rooted in Von Neumann architectures and existing design trends suggest increasingly heterogeneous and parallel architectures.

We also need to revisit the approach of compiling a high-level language to VHDL/Verilog and then using vendor-specific synthesis tools to generate bitstreams. If FPGA vendors open up FPGA architecture details, third parties could develop new tools that compile a high-level description directly to a bitstream without going through the intermediate step of generating VHDL/Verilog. This is attractive because current synthesis times are too long to be acceptable in the mainstream.

REFERENCES

1. Auerbach, J., Bacon, D. F., Burcea, I., Cheng, P., Fink, S. J., Rabbah, R., Shukla, S. 2012. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*: 271-276.
2. Auerbach, J., Bacon, D. F., Cheng, P., Rabbah, R. 2010. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*: 89-108.
3. Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D. P. 2012. From OpenCL to high-performance hardware on FPGAs. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*: 531-534.
4. Bachrach, J., Richards, B., Vo, H., Lee, Y., Waterman, A., Avidienis, R., Wawrzynek, J., Asanovic, K. 2012. Chisel: constructing hardware in a Scala-embedded language. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC):* 1212-1221.
5. Berkeley Design Technology. 2010. An independent evaluation of: the AutoESL AutoPilot high-level synthesis tool. Technical Report.
6. Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., Storaasli, O. O. 2010. State-of-the-art in heterogeneous computing. *Scientific Programming* 18(1).
7. Cardoso, J., Diniz, P. 2009. *Compilation techniques for reconfigurable architectures*. Springer.
8. Coussy, P., Morawiec, A. *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.
9. Dase, C., Falcon, J. S., MacCleery, B. 2006. Motorcycle control prototyping using an FPGA-based embedded control system. *IEEE Control Systems* 26(5): 17-21.
10. Dubach, C., Cheng, P., Rabbah, R., Bacon, D. F., Fink, S. J. 2012. Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *33rd SIGPLAN Symposium for Programming Design and Implementation (PLDI)*: 1-12.
11. Edwards, S.A. 2002. High-level synthesis from the synchronous language Esterel. In *IEEE/ACM International Workshop on Logic & Synthesis (IWLS)*: 401-406.
12. Greaves, D., Singh, S. 2010. Designing application-specific circuits with concurrent C# programs. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE).*
13. Jaaskelainen, P. O., de La Lama, C. S., Huerta, P., Takala, J. H. 2010. OpenCL-based design

methodology for application-specific processors. In *Embedded Computer Systems (SAMOS)*: 223-230.
14. Nikhil, R. S. 2011. Abstraction in hardware system design. *ACM Queue* 9(8); http://queue.acm.org/detail.cfm?id=2020861.
15. Owaida, M., Bellas, N., Daloukas, K., Antonopoulos, C. 2012. Synthesis of platform architectures from OpenCL programs. In *FCCM (Field-programmable Custom Computing Machines)*: 186-193.
16. Papakonstantinou, A., Karthik, G., Stratton, J. A., Chen, D., Cong, J., Hwu, W.-M. W. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Application Specific Processors (SASP)*: 35-42.
17. Sharma, S., Chen, W. 2009. Using model-based design to accelerate FPGA development for automotive applications. The MathWorks.
18. Sirowy, S., Forin, A. 2008. Where's the beef? Why FPGAs are so fast. Microsoft Research Technical Report MSR-TR-2008-130.
19. Thomas, D. B., Howes, L., Luk, W. 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGAs)*: 22-24.
20. WinterGreen Research Inc. 2010. Programmable logic IC market shares and forecasts, worldwide, 2010 to 2016. Technical Report.
21. Wulf, W. A., McKee, S. A. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News* 23(1): 20-24.
22. Xilinx. 2012. Command line tools user guide. Technical Report UG628 (14.3).
23. Xilinx. 2012. 7 series FPGAs overview. Technical Report DS180 (1.13).

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

DAVID BACON is a research staff member at the IBM T.J. Watson Research Center. His research interests are in programming language design and implementation, and he is currently working on the Liquid Metal project.

RODRIC RABBAH is a research staff member at the IBM T.J. Watson Research Center. He is interested in programming languages, compilers, and architectures for heterogeneous computing. He has worked in these areas since the founding of the Liquid Metal project at IBM in 2007.

SUNIL SHUKLA is a research staff member at the IBM T.J. Watson Research Center. He likes to work on various aspects, including architecture and programming methodology of accelerator-based computing, especially dynamically reconfigurable accelerators. He is currently working on the Liquid Metal project, which provides a single-language solution to programming heterogeneous architectures (CPU, GPU, FPGA).