Survey of Domain-Specific Languages for FPGA Computing

Nachiket Kapre
School of Computer Science and Engineering
Nanyang Technological University
Singapore, 639798
nachiket@ieee.org

Samuel Bayliss
Department of Electrical and Electronic Engineering
Imperial College London
London SW7 2BT
samuel.bayliss@imperial.ac.uk

***Abstract*— High-performance FPGA programming has typically been the exclusive domain of a small band of specialized hardware developers. They are capable of reasoning about implementation concerns at the register-transfer level (RTL) which is analogous to assembly-level programming in software. Sometimes these developers are required to push further down to manage even lower levels of abstraction closer to physical aspects of the design such as detailed layout to meet critical design constraints. In contrast, software programmers have long since moved away from textual assembly-level programming towards relying on graphical integrated development environments (IDEs), high-level compilers, smart static analysis tools and runtime systems that optimize, manage and assist the program development tasks. Domain-specific languages (DSLs) can bridge this productivity gap by providing higher levels of abstraction in environments close to the domain of application expert. DSLs carefully limit the set of programming constructs to minimize programmer mistakes while also enabling a rich set of domain-specific optimizations and program transformations. With a large number of DSLs to choose from, an inexperienced FPGA user may be confused about how to select an appropriate one for the intended domain. In this paper, we review a combination of legacy and state-of-the-art DSLs available for FPGA development and provide a taxonomy and classification to guide selection and correct use of the framework.**

I. Introduction

In operation, all practical computers execute a series of state transitions to realize user intent. Computer programming languages allow users to unambiguously and compactly declare their design intent in a way that, through the intermediary action of a compilation tool, can give rise to the required sequence of state transitions on a computing machine. Executed on a typical processor, each of these software components has sole use of the compute resources of the machine as a temporal sequence, handing off control, in-turn to the next compute unit.

By contrast, programming and configuring *spatial* hardware systems such as FPGAs, which support millions of lightweight concurrent operations at the gate-level, is a harder parallel programming challenge. Computation in such machines is typically realized by programming the contents of lookup tables and the fixed topology of a switching-fabric connecting those lookup-tables with hardware registers. In the common programming abstraction for designing spatial hardware, register-transfer languages, users specify concurrent processes which manipulate subsets of program state as a sequence of state transitions. These processes can communicate a subset of that state through wires, with data exchange synchronized using a common clock signal. Designs produced using this abstraction can be extremely difficult to reason about due to the complexity of interactions possible between these concurrent processes. The design and verification problem requires the programmer to have the correct mental models of absolute timing relationships between millions of concurrent digital components.

To further compound this problem, long compile-flows for FPGA implementations limit the number of compile-test cycles a developer may complete in a working day. The production of a configuration bitstream for a FPGA device requires a series of compilation stages, each of which is a NP-complete optimization problem. By contrast, compilation for processors is comparatively quick; compilers can rapidly synthesize binary executables for a target ISA specification from a high-level description of a computation thread such as those found in C/C++ programs. High-level program descriptions hide low-level ISA details such as specific instruction selection, instruction scheduling, register allocation and memory management within a single abstract datastore.

Yet despite all these significant drawbacks in designer productivity, computation using spatial FPGA hardware remains attractive because it can often deliver significantly better performance, power and area efficiency than computation using processors. The increasing mismatch between the achievable bandwidth and latency of computer memory and the capabilities of the logic devices they are connected to, means that computer processors devote ever larger proportions of their area to preserving the illusion of uniformly fast access to random data in memory.

The goal of this paper is to explore domain-specific languages (DSLs) that map to parallel FPGA-based hardware systems. An ideal DSL allows you to specify *what* you want to compute while relegating *how* you need to implement to automated tools. We consider languages that achieve high-programmer productivity by specializing for specific computing applications as well as languages that achieve high-compute efficiency by closely matching the underlying target machine. Unlike general-purpose parallel programming lan-

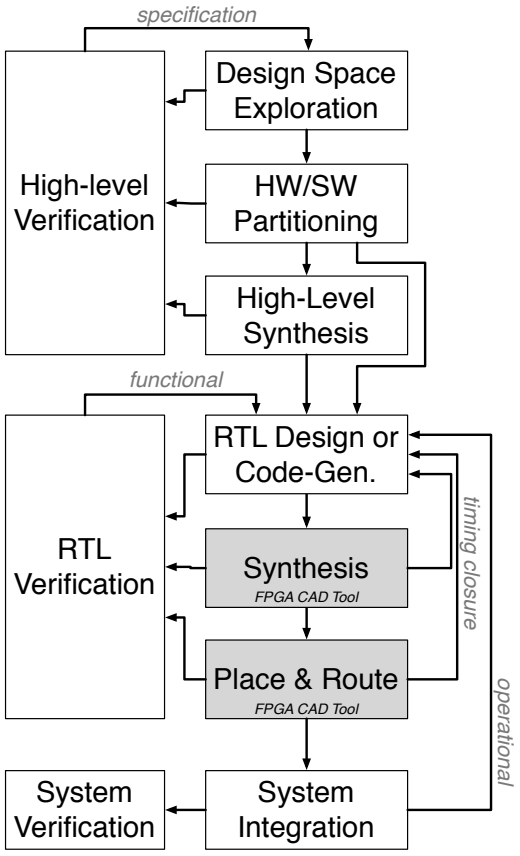

Fig. 1: FPGA Design and Development Flow

guages, domain-specific languages allow the programmer to describe computation in a manner that is *natural to the domain*. Such descriptions hide the scope and kind of parallelism in the computation from the programmer while still enabling the DSL compilers to exploit domain-specific concurrency for spatial acceleration.

In Section II, we review the low-level FPGA design flow and myriad of decisions that must be made by developers to deliver a complete FPGA design. We then highlight the expressiveness gap between Verilog and C-based descriptions through a simple example in Section III. Section IV introduces our high-level taxonomy for classifying the different DSLs for FPGA design. We then describe a selection of prominent DSLs in greater detail in Section V. We evaluate the productivity and solution quality of a range of DSLs in Section VI (for a small illustrative example). Finally, we provides and wrapup with conclusions and concrete recommendations in Section VII.

II. FPGA Design Flow

Industrial surveys [37] have found software programmer productivity to average 10–100 lines of code/developer day while hardware productivity lags behind at roughly 100s LUTs (or 1000s of equivalent gates) per developer day which correspond to dozens of line of code. We also know that there are 10 times as many software programmers as there are hardware programmers. The net result is that the implementation of an algorithm in an FPGA takes more time and is more costly than a software implementation with equivalent functionality. In this section, we briefly describe the steps in a typical FPGA design process and consider how each contributes to the wide productivity gap between software and hardware design flows. Figure 1 illustrates the development stages in producing an FPGA design. The diagram shows an iterative cycle in which several design iterations are required to ensure that the design meets specifications, satisfies functional correctness, meets timing constraints (i.e. to meet the required clock-frequency) while operating correctly in the final deployed system. Briefly, the design stages are:

- **Design Space Exploration**: Preliminary design space exploration is typically conducted with high-level models using high-level programming languages such as C/C++, Java or even Python. Here, we first establish a case for spatial FPGA hardware implementation and bound achievable performance and energy limits. These functional descriptions are not usually synthesizable to high-performance hardware but serve as a rich source of ideas for developing automation and DSLs for hardware generation *e.g.* Simulink flow diagrams for signal-processing computations can first help establish functional correctness and permit algorithm prototyping but then can even be synthesized to FPGA circuits using Xilinx System Generator (subset). Increasingly, high-level synthesis tools such as Vivado HLS can play an important role in this phase where preliminary resource and performance estimates can be extracted even when the design is not fully mature. For accelerator-oriented designs, profiling tools can help identify performance bottlenecks and hotspots.
- **Hardware-Software Partitioning**: Once we identify the scope of the problem, we must pick an architecture organization that will drive the design optimization process. It is not unusual to expect the designer to focus on FPGA hardware implementation of parallelizable bottleneck computations while relegating the control-oriented sequential tasks to a control processor in software. This hardware-software partitioning is performed through a profile-guided manual process [12] or automated tools such as *e.g.* VLIW-SCORE [31]. Lack of automated hardware-software partitioning tools has long been perceived as a key weakness of FPGA-based computing. However, the recent emergence of Xilinx SDSoC environment, and OpenCL-based development flows paves the way for commercial solutions that assist with partitioning or provision of accelerator systems.
- **Hardware/RTL Design**: FPGA hardware is well-suited for fully spatial circuit-style mapping, streaming pipe-and-filter compositions, data-parallel operations and hard-to-parallelize irregular dataflow problems. Good FPGA designs are deeply pipelined circuits coupled to local on-chip memories capable of processing an item of data in a cycle. The design of the datapath requires careful attention to timing, manual packing of logic into register stages and reasoning about dataflow through the datapath circuitry. The

effectiveness of the datapath design is tied closely to the engineering of the memory subsystem. FPGAs permit the design and development of custom caches and managed memory structures that can be tailored to application requirements. While this freedom is great, it is an additional burden on developer to make choices that tune and optimize memory performance. Most DSLs aim to address the design challenge posed in the RTL design stage. By eliminating the strict timing discipline of low-level HDL descriptions, most DSLs capture dataflow parallelism in the purest form. Certain DSLs such as FX-SCORE [36] use C-to-gates compiler in the inner-loop of the compilation phase riding the improvements in HLS toolflows. These C-to-gates HLS toolflows have mastered the art of compiling regular loops to hardware with automatic scheduling and generation of loop-level control flow and handshaking interfaces. Some DSLs even automate the inference of suitable memory interfaces (*e.g.* Maxeler streams), and we also see automation in memory re-ordering and buffering for improving memory subsystem performance.

- **Verification**: Design verification is typically handled through a combination of RTL simulation tools and formal verification tools that can statically prove properties and equivalence of circuit designs. Specialist languages such as Specman E [27] or SystemVerilog [1] are often used to describe valid stimulus and expected results from a design. Random stimulus, generated to meet these constraint is used to exercise an discrete-event simulator such as Modelsim or VCS to produce outputs, which are checked against the expected result. These tools operate in an event-driven fashion by tracking changes to signals at the bit-level requiring large amounts of RAM and exceptionally long runtimes for large industrial circuits. In many cases, it becomes infeasible to completely verify the full system prior to deployment and additional time needs to be set aside during physical system deployment. Mistakes and bugs discovered at this late stage can still be rectified due to the reconfigurable nature of the FPGA fabric but are extremely hard to diagnose and require long CAD times. These long simulation and in-system verification times represent a significant challenge to the classic edit-compile-debug loop in software development.
- **FPGA Execution**: The generation of the physical executable bitstream from RTL proceeding through the various stages of synthesis, technology mapping, placement, routing can require hours-to-days of runtime for large FPGAs. Stability of the brittle FPGA-to-host interface is yet another challenge that further complicates the design flow. The lack of binary compatibility across different device families and generations, and lack of API uniformity across host-side IO interfaces means the complete design flow must be redone each time a new platform is selected. Thus, each new FPGA board and new host system represents a new device driver engineering challenge that is a sink of developer time and effort. OpenCL certified design of next-generation FPGA boards aims to bridge this gap.

III. Motivation

Most hardware developers may have used some subset of Verilog, VHDL, System Verilog, or System C as an input to a hardware synthesis tool. These traditional languages continue to enjoy widespread use even as an intermediate backend to the various DSLs we classify in this paper. Expressiveness in languages like Verilog and VHDL is limited to structural composition of digital primitives or limited behavioral specifications that are inferred during synthesis. For example, the Verilog code below in Figure 2 shows how we may describe a simple polynomial $a{\cdot}x^2{+}b{\cdot}x{+}c$ using Verilog. The complexity of this description should be apparent from the need to specify operation within clock cycles boundaries, bit-level type specification requirements and the need to select an appropriate event to trigger code evaluation. More significantly, the interface does not abstract away the implementation details of this design. The interface does not capture the fact that polynomial results are valid one clock-cycle after the inputs are presented. Therefore when this block is implemented in a larger design, that information needs to be communicated explicitly to ensure that the data-flow pipelines correctly balance the delays of their individual components. The model of parallelism using concurrent assignment and concurrent processes appears powerful but is poorly supported during synthesis and provides confusing and conflicting semantics for simulation and synthesis.

```
module poly(
        input clk, rst,
        input [31:0] x,
        output reg [31:0] y);

        reg[31:0] a=3,b=2,c=1;

        always @ (posedge clk)
        begin
                if(rst)
                        y = 32'b0;
                else
                        y = a*x*x + b*x + c;
        end

endmodule
```

Fig. 2: Verilog code listing

```
void poly(int x, int* y) {
        int a=3,b=2,c=1;
        *y = a*x*x + b*x + c;
}
```

Fig. 3: C code listing

A large range of tools have sought to use traditional imperative-languages with a single thread of control as input languages for hardware synthesis. Tools for high-level synthesis from subsets of imperative languages (such as C or C++)

typically achieve some parallelism in their implementations. Some of these tools allow extraction of parallelism from single-threaded designs using a combination of automated analysis (e.g. loop-analysis in Synopsys Symphony-C), and user-supplied attributes communicated through language extensions (e.g. dataflow-directives in Vivado-HLS [16], or 'par' blocks in Handel-C). Compare the Verilog code block in Figure 2 to the C function shown in Figure 3. The C function is significantly smaller and easier to understand. It can be compiled by any modern HLS tool to generate Verilog behind the scenes for targeting hardware. These languages offer freedom from reasoning about timing and clocking discipline and can perform a limited set of optimization on loops and interface handshake control generation. C-based design tools can allow users to more quickly consider alternative hardware-software partitioning and to use meta-programming through constant propagation to explore a range of different area-performance trade-offs.

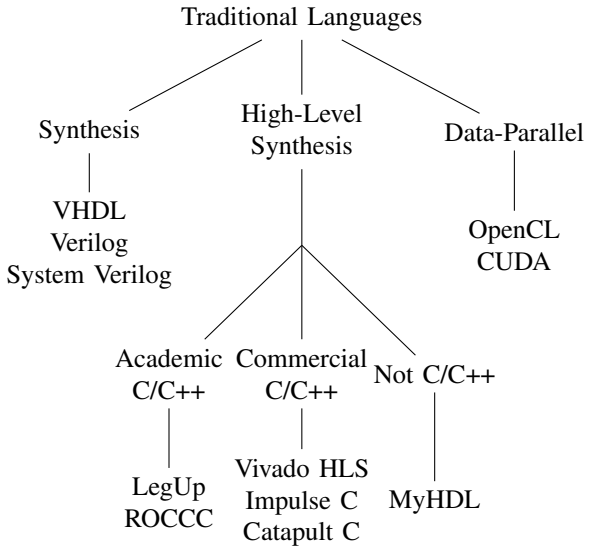

Fig. 4: Traditional Languages

We highlight a simple classification of traditional Verilog-like or C-like languages in Figure 4. It is worth noting that beyond Verilog and C, both Altera and Xilinx provide compilation flows for OpenCL which allow users to specify the behaviour of one thread of execution, and then instantiate that thread many times with different parameters to realize a data-parallel design. While CUDA and OpenCL share syntactical similarity with C/C++, they impose architectural restrictions in terms of the model of parallelism (*i.e.* data-parallelism) and concept of memory hierarchy (*i.e.* co-processor address space). Prospective developers of new FPGA-oriented DSLs are advised to consult Figure 4 to seek existing well-established and popular languages for embedding functionality. In Section IV that follows, we explore the salient features of modern DSLs with an emphasis on design choices that influence FPGA compatibility.

IV. FPGA DSL Classification

A practical survey of different DSLs, even within the specialized field of FPGA computing, brings with it the need to establish a framework within which to discuss different languages. In this section we develop a high-level classification of FPGA DSLs that we can use to discuss specific DSLs in more detail in Section V. To begin this process, we consider existing work which seeks to classify and compare domain specific languages. The problem has been approached in [48] where the author identifies common software design patterns used to create or consume DSLs. The paper describes 'structural' design patterns which utilize DSLs, while 'creational' design patterns which describe common approaches to creating a DSL. In both cases, the focus is on the language and tool implementation, rather than domain specialization.

When considering FPGA-specific DSLs, we must consider various axes along which we can classify languages; considering the level of abstraction of the language, i.e. the distance from its eventual embodiment as an interconnected network of LUTs and registers. At each level of abstraction, there is a need to capture knowledge that is relevant to that specific abstraction level. For this reason, FPGA development flows often involve chains of language processors, each consuming an abstract language and lowering to a more concrete implementation expressed in another language. Classification of domain specific languages along this axis is based on sub-dividing languages according to what domain-specific knowledge the language designer intended to capture. As shown in Figure 5, we classify language in this way, separating them into those that try to capture (1) Application-based knowledge (2) Knowledge about Specific Compute Models and (3) Knowledge specific to FPGA Implementation. These three classes represent a gradual lowering of the abstraction level from high-level application-oriented concepts to low-level bit-manipulations.

The *Application Domain* categorization is the classic interpretation of how and where DSLs are used. These languages are tightly coupled to the computational patterns and terminology of the user application. Such languages are characterized by syntactical familiarity between the keywords and parallel patterns used in the language and well-understood concepts within the target domain. By staying close to the application domain, the primary consumers are domain experts who may not necessarily be hardware architects capable of identifying the best possible implementation for their algorithms. Some applications must model domains that include interactions with people, or interfacing with sensors and actuators in the physical world (IoT platforms). This can bring with them application specific constraints on those interactions, especially latency of response, required data throughput and acceptable jitter of computing hardware. Some application domains must capture these 'real-time' constraints either explicitly, or through an implied relationship of real-time constraints with domain concepts. The language provides an unambiguous method of describing the application without forcing

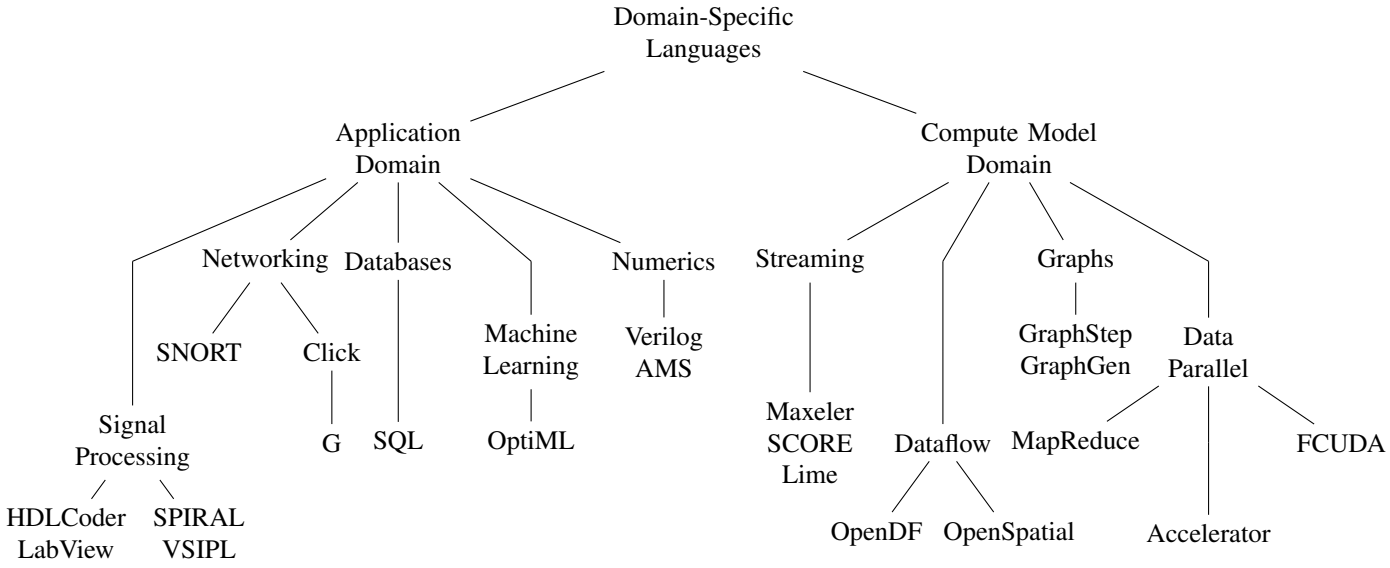

Fig. 5: DSL Taxonomy focussing on the Application and Compute Domains. *Caveat*: Not an exhaustive list, but highlighted to select a few prominent DSLs.

the designer to specialize that application for a particular architecture. An ecosystem of analysis and debugging tools can develop around the application domain and permit the application developer to focus on correctness and validation of the computation that would otherwise be challenging to pursue at the circuit level. Tools which synthesize an application description into spatial hardware can be developed by experts and pursue goals such as portability between different devices, efficient implementation within a target fabric, and performance scaling from small to large spatial fabrics.

The second classification category contains DSLs whose principle purpose is to express a formal compute model. This *Compute-Model Domain* of languages capture computation and communication patterns using formal semantics but do not target a specific application domain. These languages enable language developers and compiler writers to target multiple computational fabrics, ensuring a design expressed in a particular way is portable to FPGA, GPU and multi-core platforms equally well. These models do not exist as mere academic curiosities but are also the foundation of commercial tools. Under this classification, the languages are organized in terms of the model of computations being used which enforce restrictions on the kinds of computations that may be correctly described *e.g.* streaming, dataflow, among others. These frameworks restrict programmer freedom to specific unambiguous compute semantics and regular communication patterns which can be practically targeted for parallel implementation.

The final classification category, the *Design Domain*, shown separately in Figure 6, is our interpretation of the languages that are tightly coupled to existing stages in a classic FPGA design flow itself. The compilation of circuits can be managed effectively through specialized languages devoted to permitting description and automated optimization of functions required to implement a circuit *e.g.* layout, parametric generation. This view of computation is particularly valuable for

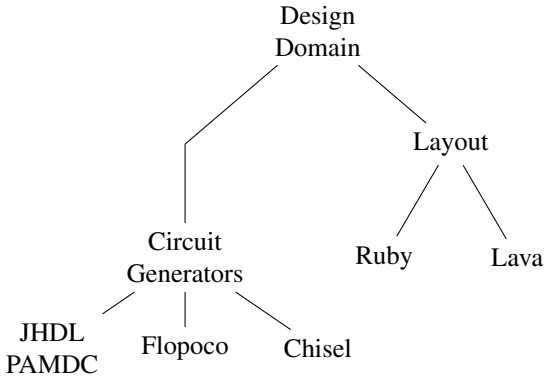

Fig. 6: Taxonomy of Design Domain

system developers that build libraries and tools for particular FPGA platforms as part of a packaged offering to prospective consumers of the FPGA platform. Sometimes more obscure features of FPGA design such as partial reconfiguration may also need support in the language (*e.g.* JBits). This categorization also includes the generator-based model of building circuits where pre-compiled layout recipes are provided for constructing circuits tailored for particular FPGA families. This flow captures and saves circuit designer effort and makes it available to developers and consumers who may not be intimately familiar with the underlying features of the FPGA platform. By eliminating the FPGA CAD process we not only speedup the design mapping process, we also directly generate close-to-optimal solutions inspired from good layout practices.

V. Review of DSLs

In this section we briefly describe the salient features of a few key DSLs available for FPGA programming. The selection of languages presented here are promising and representative candidates and are chosen to highlight the fundamental trends and patterns.

A. Application Domain

MATLAB [28], [24], [25]: MATLAB is used widely in the scientific and engineering community for analysis and simulations that use matrices and vectors. With its extensive support for toolboxes that target various domains, it has adopted a library-view towards domain specialization within the same design environment. The MATLAB HDL Coder toolbox from Mathworks is an extension that automatically generates hardware from simple MATLAB functions. It also provides support for floating-point to fixed-point conversion through automated range and error analysis routines. In Figure 7, we see an example HDL Coder function that looks like ordinary MATLAB function. A key difference is the ability to automatically infer fixed-point precision of the variables through range and error analysis without having to explicitly specify type. Simulink-based inputs allow graphical design entry through drag-and-drop functionality with the same underlying hardware generation capability for signal flow graphs. Pipeline control, clocking and reset, and handshaking are all automatically inferred in the generated HDL code. AccelDSP (no longer supported) and Vivado System Generator (previously called Xilinx System Generator) are alternate MATLAB-based programming frameworks for FPGA design that are somewhat less general than HDL Coder. These are aimed primarily at a DSP-oriented signal processing audience and provides optimized building blocks (graphical and textual) for FPGA substrates. MATLAB is an ideal first DSL that scientists, engineers and non-specialists should choose when designing FPGA accelerators.

```
function [y] = poly (x)
        a=3; b=2; c=1;
        y=a*x*x+b*x+c;
```

Fig. 7: Matlab HDL Coder code listing

LabVIEW [42]: LabVIEW is a graphical design environment for describing circuits for data acquisition, automation and control applications. It ships with deep integration with LabVIEW hardware that includes a compiler, libraries and driver support for some FPGA-based platforms. LabVIEW programs are essentially dataflow graphs that connect sensors inputs and actuator outputs to control elements. The FPGA backend support compilation of signal processing, streaming computations described using dataflow semantics. LabVIEW is a mature FPGA development environment with and emphasis on simplicity of use and out-of-the-box hardware integration. This makes it ideal for consumers who are interested in fast time to solution and are willing to invest in a closed software and hardware environment.

SPIRAL [43]: Instantiation of libraries of components has long-been a feature of FPGA-design flow, both using structural composition (e.g. the creation of FPGA netlists) and automatic inference (e.g. instantiation of Synopsys DesignWare components). SPIRAL provides libraries for various DSP and signal-processing applications such as FFTs, DCTs, Filters, Multipliers, and Sorting Networks through an automated program generation system. The SPIRAL system even offers optimized libraries for multi-core targets in addition to FPGAs. SPIRAL is aimed at developers with prior exposure to the FPGA development flow, but still provides a useful application-oriented perspective in synthesizing signal processing blocks.

VSIPL [29]: VSIPL is an industry standard for supporting vectorized signal and image processing functions on different processing platforms including x86 CPUs, PowerPC, Cell, GPUs and FPGAs. Computations are described as VSIPL functions operating on VSIPL data objects with data communication with the FPGA cards handled transparently behind the scenes. VSIPL hardware support is limited to dated FPGA boards such as the Annapolis Microsystems Wildcard II, and the Cray XD1 and development seems to have ceased beyond 2006. The recent OpenCL and OpenVL standardization efforts by the Khronos consortium have superseded VSIPL in the modern day.

SNORT [39]: A classic example of a DSL aimed at an application-domain is the SNORT rule description language. SNORT is a network intrusion detection and prevention system that performs real-time deep-packet inspection and analysis of network traffic to identify malicious intrusion attempts. SNORT rules are a way of compactly specifying regular expression patterns on the network data to detect intrusions. Originally developed for supporting the software implementation of SNORT, the language was repurposed by the FPGA community without modification to compile hardware-accelerated parallel state-machines to process and analyze a stream of network packets at line rates [26]. A compilation flow for SNORT rules using a Perl-Compatible Regular Expressions (PCRE) to VHDL generation [23] is available online.

Click: Click [32] is a software architecture for describing and configuring network routers and other packet processing hardware. More generally, it is an DSL for networking applications which are built by composing elements through ports. Chimpp [46] is a Click-based development environment that can program packet-processing systems on network-ready FPGA boards such as the NetFPGA card with an emphasis on ease of use rather than hardware optimization and without automated hardware generation of Chimpp elements. Cliff [33] is a Click-derived DSL for automatically generating packet processing hardware but was discovered to be of limited appeal due to poor quality of generated hardware. A recent Xilinx-Bell labs collaboration has produced a language called G [11] that generated high-quality, highly-pipelined hardware capable of supporting modern telecommunication networks bandwidths [2]. While neither Chimpp nor Cliff are presently available online, the Xilinx G language (superseded by the PX language) is supported via NetFPGA [53] through the PacketXpress toolsuite and is meant to be used in collaboration with Xilinx Labs. More recently, Xilinx announced the SDNet framework for generating packet processing hardware while also providing preliminary support for P4 [10].

SQL [40]: SQL is a language used for manipulating and managing relational databases. SQL queries are data retrieval

operations subject to conditional filtering and structured sorting in the simplest case. The Glacier compiler [41] translates SQL queries into streaming circuits. Netezza and Xtreme Data sell FPGA-based SQL accelerators for their commercial database appliances. More recent studies [20] exploit FPGA partial reconfigurability to compose SQL queries on-the-fly as required. Neither of these tools are available online.

OptiML [21],[50]: OptiML is a DSL designed to map parallel machine-learning computations to FPGAs using the streaming compute model. It leverages the Delite framework for embedding the DSL in Scala and target multi-cores, GPUs as well as FPGAs. The FPGA backend exploits domain knowledge to generate and optimize FPGA kernels, implement control and co-ordination logic as well as manage data movement and allocation simultaneously. The language is designed for machine learning experts with little exposure to FPGA programming and are supported on Xilinx VC709 board. The DSL is an active research platform but the latest v0.3.4-alpha release does not have an operational FPGA backend.

Verilog-AMS [30]: Verilog-AMS is a DSL for high-level descriptions of compact device models in SPICE. Verilog-AMS compilers translate this code into optimized implementations for different circuit simulators used in industry with appropriate interface APIs. This same frontend can be also enable generation of parallel FPGA hardware for fast circuit simulation. The ADMS compiler [35] for integration with `ngspice` is available online but the FPGA code generator is not freely available online.

B. Compute-Model Domain

Maxeler [45]: The Maxeler framework is a combination of Java-based DSL frontend, a compiler associated runtime and a tight coupling with their own custom FPGA boards. Programs written in Maxeler Java are essentially RTL representations embedded in Java with certain language features enabling simpler and easier programming. Java can also be used to write functional test benches and perform fast verification without resorting to Modelsim-based flows. OpenSPL [51] is an open specification for spatial dataflow languages based on the Maxeler dataflow. In Figure 8, we again see a simple Java-based representation of the polynomial example. Here, we need to use specifically declare IO ports as well as use special Java types that are tied to the FPGA compilation flow. The actual dataflow description of the polynomial calculation is simple without the need to specify or reason about clock cycles. A companion `Manager` kernel (not shown) specifies the linkages to external memory or PCIe bus for the IOs. The Maxeler framework is ideal for developers with little background in FPGA design and many non-experts have been able to use the Maxeler environment in domains such as scientific computing, data analytics and financial modeling.

Lime [4]: Lime is a multi-platform parallel programming language for heterogeneous architectures. In common with JHDL and OpenSPL, it embeds a throughput-oriented DSL in Java. Lime supports both a stream-based computation model and a dynamic runtime that can compile code as and when it

```
class Poly extends Kernel {
  Poly(KernelParameters parameters) {
    super(parameters);

    DFEVar x = io.input("x", dfeUInt(32));
    int a = 3, b = 2, c 1;
    DFEVar y = a*x*x + b*c + c;
    io.output("y", y, dfeUInt(32));

  }
}
```

Fig. 8: Maxeler code listing

is needed. With the exception of the non-deterministic match operator, Lime has similar semantics to the computation-model-based SCORE framework also discussed next. The Lime Development Kit is available for public use through the Eclipse Java IDE and is relatively new. To setup a complete working environment with real FPGA boards will need interaction and guidance from the IBM support team. As shown in Listing 9, the `poly` function is practically no different from how you may describe it on an FPGA. There is, however, a need to use the `Task` construct to express thread-level concurrency more explicitly.

```
public class Poly {

  public static void main(string[[]] args) {

    int x = 1;
    Task poly_t =  Tasks.single(x)
      => ([ task poly ])
      => task System.out.print(int);

    poly_t.finish();
  }

  static local int poly(int x) {
    int y=0, a=1, b=2, c=3;
    y = (a*x*x) + (b*x) + c;
    return y;
  }

}
```

Fig. 9: IBM Lime code listing

SCORE [13], [36]: SCORE is a high-level system architecture for supporting stream computation on reconfigurable platforms like FPGAs and HSRAs. Computations described in SCORE obey the dynamic dataflow paradigm where variable data-rate operators are supported but execution is completely deterministic. As a consequence, FIFO capacities are unbounded. The paged SCORE model allows circuits to be broken down into virtual pages that are managed by a lightweight runtime. The FPGA backend automatically synthesizes streaming handshake interfaces and buffering FIFOs between streaming operators which are managed using internal state machines. SCORE supports a variety of backends with custom runtimes such as multi-core CPUs, GPUs, Microblaze soft-processors

as well as fully spatial FPGA circuits. In Figure 10, we show the polynomial code example expressed a SCORE operator in TDF (Task Description Format). Apart from clearly indication port directions, we must pack the polynomial evaluation into a state of a state machine. SCORE TDF aims to be an intermediate-level DSL between a higher-level DSL and low-level Verilog. It is freely available online for research use but lacks integration with real FPGA toolflows.

```
poly(input unsigned[32] x,
     output unsigned[32] y)
{
        unsigned[32] a=3,b=2,c=1;

        state always (x):
                y = a*x*x + b*x + c;

}
```

Fig. 10: SCORE code listing

OpenDF [7]: Inspired by the CAL actor language, OpenDF is a dataflow programming framework and toolset for FPGAs. Dataflow languages support description of computation on a graph of dataflow actors with firing rules that trigger only when all inputs are ready. Actors allow strong encapsulation of state and actions, allow concurrent evaluation of ready actors, allow asynchronous scheduling freedom for ordering their evaluation and permit decoupling of scheduling and communication. The OpenDF sourceforge repository hosts the Eclipse IDE frontend, and the VHDL/C code-generation backends but has had no updates since 2012. In Listing 11, we show a representative example where the actor encapsulates the compute expression within an action that is triggered when data is available on the input x. A structural wrapper is required for generation of VHDL.

```
network Top ()  X,  ==>  Out :

entities

poly=Poly();

structure

X--> poly.X;
poly.Out --> Out;

end

actor Poly ()
  int(size=32) X ==>
  int(size=32) Out :
  action [x] ==> [a*x*x+b*x+c] end
end
```

Fig. 11: OpenDF code listing

GraphStep [18], [19]: FPGAs are quite suitable for accelerating parallel, iterative sparse graph computations. In this scenario, we load portions of the graph into on-chip memories and process multiples nodes in parallel while routing edges as point-to-point messages over an FPGA overlay NoC (network-on-chip). Unlike streaming and dataflow techniques, the nodes and edges here are represented in memory and the computations performed on the nodes and edges are implemented in hardware. The GraphStep framework supports Bulk-Synchronous parallel graph workloads with a global barrier separating iterations on the graph. Grapal [18] is an OCaml-based concrete embodiment of GraphStep semantics within a DSL for sparse graph computations. The DSL provides fundamental types for graphs, nodes and edges and imposes a timing discipline on their evaluation. Partitioning and load-balancing of the parallelism is also supported.

GraphGen [44]: GraphGen is a vertex-centric graph computing framework that assembles hardware through a templated specification. Users are still required to write the pipelined RTL computations themselves in a tool or language of their choosing. GraphGen is part of the CoRAM [14], [15] framework that supports the Xilinx ML605 and the Altera DE4 FPGA boards. Code generation is only supported through a web portal that directly generates low-level Verilog for use with the respective FPGA vendor flows. Some experience with the FPGA design flow is expected for final system assembly.

Accelerator [9]: Accelerator is a high-level data-parallel library for multi-core CPUs, GPUs (Microsoft DirectX and CUDA) and FPGAs. Data-parallel operations are available as .NET and C++ functions operating on ParallelArrays objects. The Accelerator runtime captures the expression graph of operations scheduled on these objects and generates code at runtime on demand for evaluation on parallel hardware. Accelerator v2.2 supports VHDL generation of high-level Accelerator constructs. The project has not seen development since 2011 but the Preview SDK is available for download and non-commercial use (as of publication FPGATarget.h is missing from the release). In Listing 12, we show a C# code fragment that computes the polynomial. The computations are performed over a Microsoft.ParallelArrays object and executed using lazy evaluation. VHDL generation is performed only when the result is copied back from the ParallelArray to an regular array $yArr$ (*i.e.* lazy).

C. Design Domain

The following languages are DSLs which fall into the 'Design Domain' classification of our taxonomy because they target particular stages in the hardware design flow. These are not aimed at FPGA programmers but rather towards FPGA experts who want to design or generate FPGA-optimized libraries. Other surveys [52], [38] have similar classifications for DSLs like YACC that are used to generate code for compact grammars.

JHDL [5]: JHDL embeds the specification of a register transfer description language into Java. It is an early example of the use metaprogramming environments, that allowed easy generation and layout of spatial accelerators in hardware supporting partial-reconfiguration and generation of the generic software interfaces needed to communicate with a host system.

```
using PA=Microsoft.ParallelArrays.ParallelArrays;

namespace Poly
{
  class Program
  {
    static void Main(string[] args)
    {
      int N = 1024;
      int a = 3, b = 2, c = 1;

      int[] xArr = new int[N];
      int[] yArr = new int[N];

      FPGATarget t = new FPGATarget();

      PA x = new PA(xArr);

      PA t1 = PA.Multiply(a, x);
      PA t2 = PA.Multiply(t1, x);
      PA t3 = PA.Multiply(b, x);
      PA t4 = PA.Add(t3, t2);
      PA t5 = PA.Add(t4, c);

      yArr = t.ToArray1D(t5);
    }
  }
}
```

Fig. 12: Accelerator C# code listing

The direct low-level generation of circuit form and layout in EDIF form bypasses the synthesis process thereby exposing low-level control of the FPGA fabric directly at a higher level of abstraction. JHDL is unsupported on the latest FPGA platforms and has not seen updates since 2010. The spiritual successors of JHDL today that exploit partial reconfiguration on modern FPGAs are HMFlow [34] and Torc [49] but their state of development is at the mercy of the FPGA vendors and the brittle interfaces they provide. PAMDC [6] is another C++ module generator for algorithmically constructing gate-level FPGA netlists. In Listing 13, we see how the polynomial evaluation is composed structurally using FPGA primitives such as `mult18x18` and `adder`. This allows the programmer to exploit the high-level features in Java at compile-time to construct or generate these structural netlists for EDIF generation in a manner that is easier and more powerful than VHDL generation constructs.

Flopoco [17]: As FPGAs get larger, it becomes possible to accommodate large floating-point compute graphs inside a single chip. When these computations involve exotic elementary functions like square root, exponential and logarithm, it is possible to get highly-pipelined operation through the use of suitable IP cores. Flopoco provides a simple Linux command-line tool for generating a variety of these operators. Flopoco is actively developed and supported on most modern FPGA device families. While not strictly a DSL from the point of view of FPGA users, Flopoco provides a library of state-of-the-art floating-point operators customized across various FPGA families in a parameterizable manner. From the perspective of an FPGA library developer, Flopoco supports a

```
public class Poly extends Logic {

  // Interface
  public static CellInterface[] cif = {
    in("x", 18), out("y", 36),
  };

  // Constructor
  public Poly(Node parent, Wire y, Wire x) {

    // Connect wires
    connect("y", y);
    connect("x", x);

    // Build our logic
    new mult18x18(this, x, x, t1);
    new mult18x18(this, t1, a, t2);
    new mult18x18(this, b, x, t3);
    new adder(this, t2, t3, cin, t4, cout);
    new adder(this, t4, c, cin, y, cout);
  }
}
```

Fig. 13: JHDL code listing

DSL for specifying dataflow graphs of the internal arithmetic computations of a higher-level function we wish to provide to an FPGA user. The developer also provides a separate specification of pipelining constraints along with directives for constraining usage of FPGA-specific DSP blocks. Flopoco then provides a C++ compiler with a VHDL interface to automatically convert the input graph into optimized, retimed and pipelined VHDL. Flopoco also provides a lightweight DSL for generation of floating-point datapaths of arbitrary polynomials based on a Python-like syntax.

Chisel [3]: Chisel is an embedded DSL based on Scala that supports hardware construction using highly parameterized generators, object-orientation, functional programming, parameterized types and inference. Chisel compiles code into fast, cycle-accurate C++ simulation binaries as well as Verilog code for synthesis. It can exploit Scala language features for further layer of additional DSLs if the programmer desires. Designs using Chisel can leverage IDEs, debuggers and other support tools from the wider Scala user community. It is actively developed and supported and has an interest from multiple users. In Figure 14, we show how a Chisel `component` (analogous to a Verilog module) is defined in terms of operations on directional wires in a Chisel io `bundle` (analogous to a combination of C structs and Java interfaces).

Functional Languages [47], [8]: Languages such as Ruby and Lava describe hardware using composition of functional blocks and allow explicit description of the spatial relationship between hardware blocks. They are embedded in Haskell and embody functional programming principles for composition of digital logic. Lava is still supported for VHDL generation of bit-level operations and can be downloaded as a Haskell package `chalmers-lava2000`. A modern, functional approach towards hardware design is explored in Verity [22]. This language and associated compiler supports both higher-order

```
class Poly extends Component {
        val io = new Bundle {
                val a = Bits(32, INPUT)
                val b = Bits(32, INPUT)
                val c = Bits(32, INPUT)
                val x = Bits(32, INPUT)
                val y = Bits(32, OUTPUT)
        }
        io.y := io.a * io.x * io.x +
                io.b * io.x + io.c
}
```

Fig. 14: Chisel listing

programming (passing functions as arguments) and affine recursion generating VHDL descriptions. The foreign-function interfaces and library support in Verity allows for integration of existing RTL models and portable abstractions for low-level board support. Verity is actively developed but is aimed at expert functional programmers to develop libraries or FPGA middleware rather than entire applications.

VI. Evaluating DSLs

We evaluate a subset of the DSLs from our taxonomy by compiling a simple $a \cdot x^2 + b \cdot x + c$ polynomial expression (32b operands) in these different languages. Our goal is to record the development time as well as the conciseness of the code. The assignment was a classroom exercise for NTU Embedded Systems Master's students, class of 2014-15, where each team of 3–4 students was assigned a particular DSL. While we recognize that the different teams operating here have varying capabilities, it nevertheless provides an idea of the range of development times to expect for those with a limited FPGA background. For this small example, we initially noticed that a majority of the time and effort was spent configuring and setting up the tools rather than development. Hence we only considered the time required to program and develop the code. The DSLs were installed and configured on Ubuntu 14.04 64b platform for most cases (except CentOS 5 for Maxeler, and Windows 7 for Microsoft Accelerator). We were able to execute the code on the FPGA in two instances; (1) with the Maxeler framework on the MaxWorkstation platform, and (2) with the Altera OpenCL framework on the Terasic DE5-NET FPGA card. In Table I, we report the results of this exercise by recording programming time, lines of code, resource utilization and clock frequency of the resulting designs. For certain DSLs like the Maxeler Java framework, IBM Lime and Altera OpenCL, the high resource utilization includes the FPGA driver costs that enables immediate evaluation on a physical FPGA device (we report these separately under `baseline` for completeness). Overall, the lines of code for the different DSLs are quite low except in the case of JHDL where a verbose structural description was required. Across all DSLs, we observe that the short DSL code fragments were expanded into RTL codes that were 10–100× longer (ignoring driver code). Most of the DSLs are thus concise and the associated DSL compilers unfold control circuits and other supporting hardware automatically to generate longer high-performance descriptions in RTL. We then mapped the generated RTL blocks to a Xilinx Zedboard FPGA in most cases wherever possible. The Maxeler framework uses a Xilinx Virtex6 SX475T (MaxWorkstation), Altera OpenCL configures a Stratix V 5SGXEA7N2F45C2 FPGA (DE5-NET card) while JHDL only supports Virtex-6 series devices with 18×18 DSPs. We observe differing resource utilizations (again ignoring driver harness infrastructure) and variations in performance mostly attributed to differences in the use of DSP blocks, and the extent of pipelining.

TABLE I: Comparing DSLs with $ax^2 + bx + c$ mapping

DSL	Dev. Time	Lines of Code		Resources			Freq.
		DSL	RTL	LUTs	FFs	DSPs	MHz
Flopoco[1]	30m	2	1702	1679	1288	0	91
Maxeler (baseline)	30m 30m	15	NA[2]	6036 5837	5391 5364	3 0	120
Vivado HLS	1h	4	92	53	71	3	117
Lime (baseline)	2h30m 2h30m	22	111	245 189	284 209	2 1	160
OpenCL[3] (baseline)	2h30m 2h30m	4	1262	3281 3230	4443 4192	2 0	267
Chisel	3h	25	39	129	64	10	66
OpenDF	3h30m	26	689	171	305	9	120
JHDL	4h	40	2529[4]	41	90	3	84
SCORE	4h	7	111	139	245	2	74

[1]Flopoco only provides floating-point support for these expressions [2]MaxCompiler does not produce any intermediate RTL, directly generates executable bitstreams [3]Altera resources measured in LEs instead of LUTs, Altera 18×18 DSPs are also different from Xilinx 25×18 DSPs [4]JHDL directly generates a circuit netlist in EDIF format instead of generating RTL

VII. Conclusions

In this paper, we show how to broadly classify FPGA-based DSLs using three key properties of the hardware mapping (1) application domain, (2) compute model domain, and (3) FPGA design domain. Using these high-level categories, we summarize a set of DSLs for readers interested in exploring the historical roots of FPGA-focused DSL development and the sheer breadth of ideas and domains that have been explored by the FPGA community. We hope this survey helps guide the reader through our curated list of few noteworthy languages from the large space of DSLs developed for various design goals. Based on our survey, we have some recommendations for FPGA users: (1) Novice developers should consider the mature and well-supported Maxeler, LabVIEW DSL frameworks or even non-DSL frameworks such as OpenCL/SDSoC. Developers with some prior FPGA experience may want to consider MATLAB HDL coder as a starting point for optimizations. (2) Academics and researchers with a broader willingness to try nascent frameworks should consider Chisel, CoRAM, GraphGen, OptiML, or Verity for their requirements. (3) FPGA experts interested in building optimized, low-level FPGA libraries for specific application domains should explore SPIRAL, SDNet and Flopoco for their needs.

REFERENCES

[1] Accellera. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification and Verification Language. *IEEE Std. 1800-2012*, 2012.

[2] M. Attig and G. Brebner. 400 Gb/s Programmable Packet Parsing on a Single FPGA. *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 12–23, 2011.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: constructing hardware in a Scala embedded language. In *DAC '12: Proceedings of the 49th Annual Design Automation Conference*. ACM Request Permissions, June 2012.

[4] D. F. Bacon, R. Rabbah, and S. Shukla. FPGA programming for the masses. *Communications of the ACM*, 2013.

[5] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 175–184, 1998.

[6] P. Bertin and H. Touati. PAM programming environments: practice and experience. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 133–138, Apr. 1994.

[7] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF. *ACM SIGARCH Computer Architecture News*, 36(5):29, June 2009.

[8] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. ACM Request Permissions, Jan. 1999.

[9] B. Bond, K. Hammil, L. Litchev, and S. Singh. FPGA circuit synthesis of accelerator data-parallel programs. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 167–170. IEEE, 2010.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[11] G. Brebner. Domain-Specific Programming of Very High Speed Packet Processing. Technical report.

[12] 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. *Transactions on Embedded Computing Systems (TECS*, 13(2), Sept. 2013.

[13] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, 2000.

[14] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *FPGA '11: Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM Request Permissions, Feb. 2011.

[15] E. S. Chung, M. K. Papamichael, G. Weisz, and J. C. Hoe. Cross-platform FPGA accelerator development using CoRAM and CONNECT. In *FPGA '13: Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, Feb. 2013.

[16] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, Apr. 2011.

[17] F. de Dinechin, C. Klein, and B. Pasca. Generating high-performance custom floating-point pipelines. In *International Conference on Field Programmable Logic and Applications*, pages 59–64. IEEE, 2009.

[18] M. deLorimier, N. Kapre, N. Mehta, and A. DeHon. Spatial hardware implementation for sparse graph algorithms in GraphStep. *ACM Transactions on Autonomous and Adaptive Systems*, 6(3):1–20, Sept. 2011.

[19] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*. IEEE, IEEE Computer Society, 2006.

[20] C. Dennl, D. Ziener, and J. Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *FCCM'12 : 20th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 45–52, Apr. 2012.

[21] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from Domain-Specific Languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, 2014.

[22] D. R. Ghica. Geometry of Synthesis: A Structured Approach to VLSI Design. *SIGPLAN Notices*, 42(1):363–375, 2007.

[23] C. L. Hayes and Y. Luo. DPICO: a high speed deep packet inspection engine using compact finite automata. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 195–203. ACM, 2007.

[24] T. Hill. AccelDSP IP Explorer. Technical report, Xilinx Inc., Jan. 2006.

[25] T. Hill. Using MATLAB to Create IP for System Generator for DSP. Technical report, Xilinx Inc., Jan. 2006.

[26] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 111–120, 2002.

[27] S. Iman and S. Joshi. *The e-Hardware Verification Language*. Springer Science & Business Media, May 2004.

[28] M. Inc. MATLAB HDL Coder.

[29] R. Janka, R. Judd, J. Lebak, M. Richards, and D. Campbell. VSIPL: an object-based open standard API for vector, signal, and image processing. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, pages 949–952, 2001.

[30] N. Kapre and A. DeHon. Accelerating SPICE Model-Evaluation using FPGAs. In *FCCM '09: Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 1–8. IEEE Computer Society, Mar. 2009.

[31] N. Kapre and A. DeHon. VLIW-SCORE: Beyond C for Sequential Control of SPICE FPGA Acceleration. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–9, Dec. 2011.

[32] E. Kohler, R. Morris, B. Chen, J. Jannoti, and F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Nov. 2000.

[33] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a domain specific language to a platform FPGA. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 924–927, 2004.

[34] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping. *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 117–124, 2011.

[35] L. Lemaitre, C. McAndrew, and S. Hamm. ADMS-automatic device model synthesizer. In *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*, pages 27–30. IEEE, 2002.

[36] H. Martorell and N. Kapre. FX-SCORE: A Framework for Fixed-Point Compilation of SPICE Device Models using Gappa++. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 77–84, Mar. 2012.

[37] M. Meredith and S. Svoboda. The Next IC Design Methodology Transition Is Long Overdue. *Accellera Systems Initiative*, Feb. 2010.

[38] M. Mernick, J. Heering, and A. Sloane. When and How to Develop Domain-Specific Languages . *ACM Computing Surveys*, 37:1–29, Jan. 2005.

[39] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM Request Permissions, Dec. 2007.

[40] R. Mueller, J. Teubner, and G. Alonso. Streams on wires. *Proceedings of the VLDB Endowment*, 2(1):229–240, Aug. 2009.

[41] R. Müller, J. Teubner, and G. Alonso. Glacier: a query-to-hardware compiler. In *SIGMOD Conference*, pages 1159–1162, 2010.

[42] National Instruments. NI LabVIEW.

[43] G. Nordin, P. A. Milder, J. C. Hoe, and M. Puschel. Automatic generation of customized discrete fourier transform IPs. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*. ACM Request Permissions, June 2005.

[44] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014.

[45] O. Pell and O. Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *SIGARCH Computer Architecture News*, 39(4), Dec. 2011.

[46] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat. Chimpp: A Click-based programming and simulation environment for reconfigurable networking hardware. *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*, pages 1–10, 2010.

[47] M. Sheeran and S. Singh. Ruby as a basis for hardware/software codesign. *Verification of Hardware Software Codesign, IEE Colloquium on*, page 5, 1995.

[48] D. Spinellis. Notable design patterns for domain-specic languages. *Journal of Systems and Software*, 56:91–99, Jan. 2001.

[49] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French. Torc: towards an open-source tool flow. In *FPGA '11: Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM Request Permissions, Feb. 2011.

[50] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, May 2011.

[51] The OpenSPL Consortium. OpenSPL : Revealing the Power of Spatial Computing. Technical report, Dec. 2013.

[52] A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, Feb. 2000.

[53] Xilinx, Inc. G.