Programming Datacenter-Scale Reconfigurable Systems

Stuart Byma
I&C, EPFL

***Abstract*—The growth, complexity and performance requirements of modern datacenter services are outpacing general-purpose server performance. The trend is evident in the end of Dennard scaling, and in the fact that the end of Moore's Law is not far off. Reconfigurable hardware in the form Field Programmable Gate Arrays has recently been shown to be a viable solution to this performance mismatch, using hardware acceleration at scale to boost performance, while maintaining the desirable properties of a datacenter. The barrier to widespread adoption is high however, with FPGAs being difficult to program and use effectively without specialist expertise. In this research proposal, we examine recent work involving FPGAs in the datacenter, as well as current methods to alleviate the programming problem, which include Domain-Specific Languages and High-Level Synthesis. We propose that using these techniques in conjunction with one another may be sufficient to raise the abstraction to an acceptable level for non-hardware experts to develop next-generation, hardware-accelerated datacenter services.**

***Index Terms*—Datacenters, FPGAs, DSLs, HLS**

I. Introduction

The datacenter has emerged as the cornerstone of modern Information Technology (IT) infrastructure. These massive systems are generally based on commodity components and general-purpose CPUs, which keeps costs down while at the same time providing a familiar environment for application developers to work in. The performance increases of general-purpose machines, however, are slowing as we reach fundamental physical limits of the transistors they are built of. Dennard scaling, the concept that power density scales down with transistor size, has already ended, leading to an era of *dark silicon* where the full complement of transistors on an IC cannot be powered simultaneously without risk of catastrophic failure. Moore's Law, the doubling of transistors on chip every two years is also nearing the end as device sizes approach fundamental atomic limits.

Modern datacenter workloads, however, are becoming more complex and computationally demanding much faster than server performance is scaling. This is quickly creating a fundamental mismatch between available resources and demand, a mismatch that will need to be resolved to enable the advanced datacenter services of the future.

Hardware acceleration is one potential solution to this problem, where algorithms or computationally intensive kernels are implemented directly in hardware to provide much higher performance and performance per Watt than general purpose machines. Commonly done in the past using ASICs, silicon fabrication has become far too expensive and risky for most datacenter providers and users. It is also rigid and inflexible, as it is impossible to change a design after fabrication – a fact that is incompatible with the quickly evolving datacenter services we see today. Reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs), can provide a more flexible alternative for hardware acceleration. Using FPGAs in a datacenter is a relatively new concept however, and this proposal will examine some recent work in this area. FPGAs, being low-level hardware devices, also present challenges in programming, especially considering that most datacenter users and developers are not hardware design experts. This proposal will also investigate recent work that can help alleviate this programming problem.

A. Field-Programmable Gate Arrays

An FPGA is a silicon device consisting of a large array of programmable logic blocks that use Look Up Tables to implement digital logic functions. Logic blocks also contain flip-flops and memory elements to create stateful circuits. FPGAs also contain a programmable routing network used to wire logic blocks together to create arbitrary digital circuits,

Proposal submitted to committee: May 11th, 2015; Candidacy exam date: May 19th, 2015; Candidacy exam committee: Prof. Paolo Ienne, Prof. James Larus, Prof. Edouard Bugnion.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

however modern devices are large and require complex CAD tools to map circuits into this fabric. Modern FPGAs also benefit from hardened blocks such as multi-kilobit memories, DSP blocks and communications hardware (e.g. PCI Express). In short, FPGAs provide the benefits of hardware acceleration while remaining reprogrammable.

FPGAs have several traditional application domains. They are commonly found in networking and telecommunication equipment, in signal and image processing applications, as well as in ASIC prototyping systems. Related to datacenter systems, FPGAs are also used in High-Performance Computing systems. FPGAs have not typically been used in modern datacenters as a method of accelerating or enhancing performance. However, recent work has demonstrated that FPGAs can have a large impact on the performance of datacenter services while preserving the desirable characteristics of modern datacenters such as homogeneity and the use of Commercial off the Shelf (COTS) products. This system, called Catapult [1], will be discussed in Section II.

B. The Programming Problem

One of the major, longtime challenges of using FPGAs is the difficulty in programming them – traditional design involves writing Register Transfer Level (RTL) code in Hardware Description Languages (HDL) such as Verilog or VHDL that requires digital design expertise. This problem is especially relevant in the domain of datacenters, as the majority of datacenter services and applications are created by non-hardware experts. There is a clear need for abstractions and systems that can mask the low-level nature of FPGA programming.

1) High-Level Synthesis (HLS): Related to DSLs is the concept of High-Level Synthesis, where high level languages (usually C or C++) are automatically compiled down to HDL or circuit implementations. HLS is generally not domain specific, but does provide abstractions that can make hardware design more accessible to non-hardware experts. To illustrate this, we review in Section III the Autopilot tool [2], an industry-grade HLS tool specifically geared towards HLS for FPGA designs.

2) Domain-Specific Languages (DSLs): A Domain-Specifc Language is a language that is tailored for a specific application domain, containing features and semantics that make it easier to express solutions in that domain. DSLs can provide useful higher level abstractions over traditional general purpose languages, and can also help solve the programming problem by providing abstractions for traditional HDLs. The higher-level nature of DSLs can also constrain the computation model and data flow patterns for a domain, which aids hardware generation by restricting possible circuit architectures to those that fit well with the domain in question. In Section IV we examine a recent, successful DSL for image processing called Halide [3], and also examines how it might bridge the gap between application developers and FPGA hardware.

C. Programming Hardware at Scale

The FPGA programming problem has been acknowledged and studied for quite some time, and numerous tools and systems have emerged in the attempt to solve the problem. These include:

- HLS tools such as Autopilot [2] and LegUp [4], an open source tool
- FPGA system integration and portability tools such as LEAP [5] and CoRAM [6]
- New HDLs that adopt a more functional programming style, examples including BlueSpec Verilog [7] and Chisel [8]
- Overlay architectures that abstract the fine-grain FPGA fabric. Examples are numerous, including all manner of soft processors, Coarse Grain Reconfigurable Arrays (CGRA), and other domain-specific overlays.

While some of these are geared more towards speeding up FPGA development and some, like HLS, attempt to make FPGA programming more software-like, the fact remains that some hardware expertise is generally still required in order to obtain good Quality of Results. The research proposal in Section V of this paper will examine how HLS might be combined with domain-specificity to remove this need for hardware expertise, while also considering the implications of working inside datacenter-scale systems like Catapult, where applications involve not one but many distributed FPGAs.

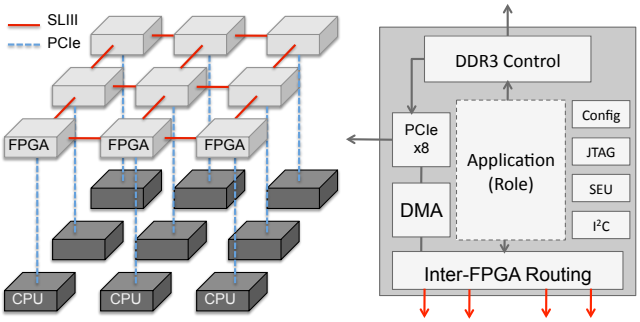

Fig. 1: Catapult system architecture (a 3 x 3 subset) and shell/role architecture, adapted from [1]. Note that the complete torus network configuration is not illustrated here.

II. Catapult

The Catapult system [1] is the result of a recent effort at Microsoft attempting to bridge the gap between server performance and workload needs. Figure 1 shows the architecture of the system. Catapult is a reconfigurable fabric for the datacenter, and consists of "pods" of 48 servers, each containing a custom PCIe daughtercard with a Stratix 5 FPGA. Each card has 8 GB of local DRAM, and FPGAs are connected to one another through dedicated 10 Gb/s links that form a 6x8 two dimensional torus. Catapult was constructed in this way for several reasons. First, it maintains homogeneity – all the servers are the same, which ensures a consistent platform for users and simplifies management. Second, it allows the creation of multi-FPGA systems – the dedicated links of the secondary network allow sufficiently fast data transfer between FPGAs, the alternative of going through PCIe to the host and over the regular datacenter network (Ethernet) being too slow. The architecture also allows flexibility in the

number of FPGAs used in an application, which avoids wasted FPGA fabric. The daughtercards were also designed with the datacenter in mind. They are compact enough to fit inside a dense 1U half-width server, and have a power requirement that can be supported by only the PCIe interface. The increased total cost of ownership per server does not exceed 30%, of which 10% is for power.

A. Shell and Role Architecture

To facilitate design reuse and productivity, FPGA hardware in Catapult is divided into two parts, as shown in Figure 1: a "shell", containing necessary interface and system integration hardware, and the "role", containing application-specific logic. The shell facilitates role access to DRAM, access to the inter-FPGA links with routing, access to the PCIe and DMA, while supporting management functions like reconfiguration and Single Event Upset detection and correction. This design lets the hardware developer focus on implementing their hardware without worrying about the complexities of memory controllers, interfaces, and other device-specific complexities.

B. Software Support

Catapult also has a significant software infrastructure to support it, including two new services: the Mapping Manager and Health Monitor. The Mapping Manager facilitates service start-up by configuring bitstreams (FPGA hardware images) into the devices. The Health Monitor helps to deal with suspected failures. Other major changes to system software were made to:

- **Ensure Correct Operation** – FPGA reconfiguration is a disruptive process. The FPGA can appear as failed on the PCIe bus and can possibly send garbage data on the inter-FPGA links. Disabling non-maskable interrupts for the FPGA PCIe device solves the first problem, while a series of communication management message types serve to solve the second.
- **Detection and Recovery from Failure** – The health monitor is activated when higher level services detect unresponsive servers. Servers are diagnosed, and if healthy, the FPGAs and associated components are checked for errors. If necessary, the Mapping Manager is called upon to reposition roles away from faulty FPGAs in the torus network.
- **Debugging** – To facilitate debugging on such a large scale, the Catapult FPGA hardware includes a lightweight "Flight Data Recorder" that captures runtime data. This data is streamed out of the system, and forms an operation log of the hardware.

C. Bing PageRank Acceleration

To demonstrate Catapult, the authors map the Bing PageRank algorithm into hardware, involving approximately 30,000 lines of C++ converted by hand into Verilog. The hardware is mapped onto seven FPGAs in a macropipeline with one spare – Figure 2 shows a diagram of the application. The pipeline consists of stages to perform Feature Extraction, Free-Form

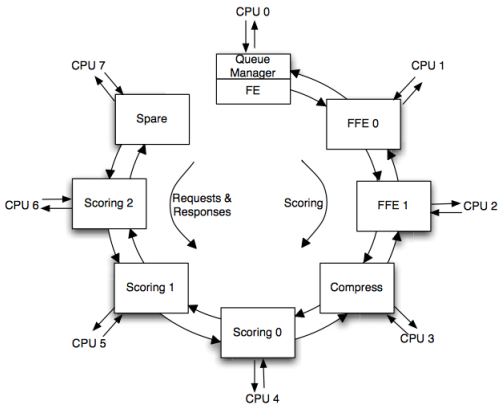

Fig. 2: Bing ranking mapped to eight FPGAs, from [1].

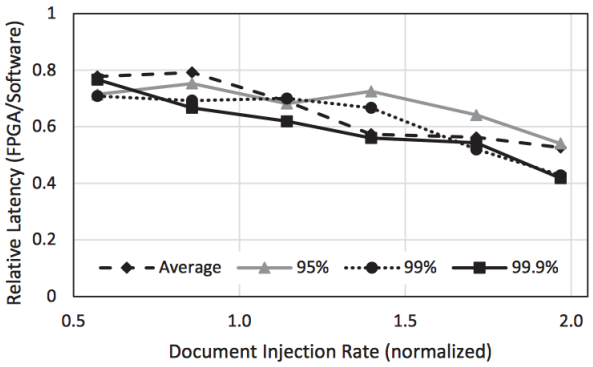

Fig. 3: Relative average and tail latencies, from [1].

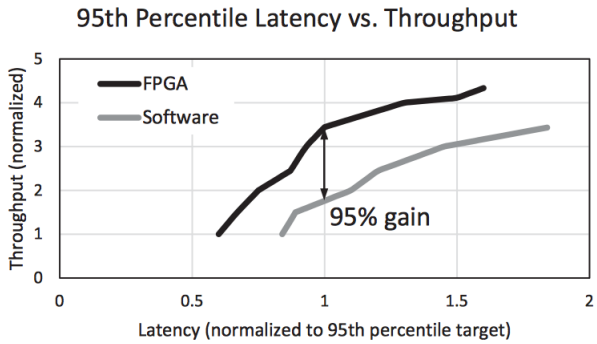

Fig. 4: Throughput relative to software at 95th percentile latency, from [1].

Expression calculations, and finally document scoring via a machine-learned model. Because the software model changed rapidly (on the order of weeks or months), the hardware was designed such that changes in the model could be deployed without redesign of the hardware. All eight CPUs can inject documents to be ranked by the pipeline – injected documents are forwarded via the inter-FPGA links to the pipeline head, and upon completion the result is sent back to the originating server.

Multi-node injection experiments on a 1632 server testbed (with 672 running the ranking service) show that the FPGA-based ranking can significantly reduce latency compared to

pure software (Figure 3). The hardware is shown to reduce worst-case latency by 29% in the 95th percentile distribution, improving at higher injection rates because of software latency variability caused by memory heirarchy contention. For latencies bound at the 95th percentile, the FPGA hardware improves system throughput by 95% (Figure 4).

D. Discussion

The PageRank acceleration makes a strong case for the use of FPGAs in the datacenter. A near doubling in throughput means that the number of servers for a service like Bing can be cut in half, or performance can be nearly doubled with the same number of servers, with total cost of ownership not increasing more than 30%. More importantly, Catapult has shown a commercially-viable method of bridging the increasing performance gap between general purpose servers and large-scale datacenter services. However, the major impediment to widespread adoption is of course the aforementioned programming problem – sections III and IV will look further into this problem.

III. AUTOPILOT HIGH-LEVEL SYNTHESIS

Although Catapult has demonstrated how effective hardware acceleration can be in the datacenter, it required many skilled hardware designers to create applications for it. To gain traction among datacenter service developers, the low-level details of hardware design must be hidden or abstracted away. High-Level Synthesis is one technique that attempts to make this abstraction, automatically compiling software or software-like languages down to HDL. Autopilot [2] is a state of the art HLS tool that we will examine in this section.

A. C-Based HLS

As the authors of Autopilot point out, recent efforts in HLS mostly use C or C++ as input languages. This presents a familiar interface to system designers and allows use of compiler technology developed for software, however C/C++ are inherently sequential languages that lack bit accuracy and timing information that hardware requires. There are also language constructs such as pointers that do not map well to hardware. The common solution to this is to add language extensions and restrict use to a "synthesizable subset", introducing pragmas or directives to inform the compiler about desired hardware architecture. Autopilot follows this methodology, however it goes further than most other tools in supporting large subsets of C, C++ and SystemC. Other tools, such as Cadence C-to-Silicon [9], Mentor Catapult C [10] and Synopsys Synphony C [11], tend to support only one language with a narrow focus, and depend on embedded timing and interface details that complicate design entry.

B. Compilation, Synthesis, Optimization

Autopilot takes a platform approach to HLS, outputting optimized RTL in Verilog, VHDL or SystemC, along with comprehensive simulation frameworks for verification of the generated code. The flow targets Xilinx FPGAs, and includes

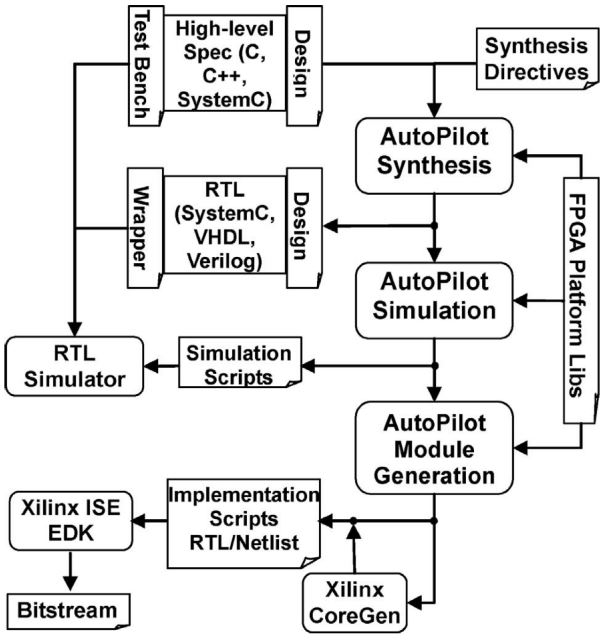

Fig. 5: The Autopilot C to HDL compilation flow, from [2].

capabilities to map hardware constructs onto Xilinx FPGA primitives (DSP multipliers, Block RAMs) to improve performance. Figure 5 shows the Autopilot compilation flow.

Autopilot is tightly integrated with LLVM [12] and uses Clang as a front end to translate C/C++ into LLVM Intermediate Representation (IR). Code optimizations are performed, including constant propagation, dead code elimination, memory dependence analysis, loop transformations, as well as hardware-specific optimizations such as bitwidth analysis.

Autopilot is geared towards FPGA implementation, and makes many assumptions and optimizations specifically for these platforms. Depending on the device, constraints are added depending on how many Block RAMS, DSPs, logic blocks are on said device, and area, delay and power for each of these blocks is characterized and factors into the compilation and optimizations. Certain hardware constructs also have very different costs on FPGAs compared to ASICs – multiplexors are expensive on FPGAs since they are implemented in slow logic blocks, however memory is cheap and generally abundant in the form of Block RAMs. These FPGA-specific features are also factored into compilation.

One of the important problems in the HLS compilation process is scheduling, where operations are scheduled not as processor instructions but as hardware operations that take place within certain clock cycles, subject to their dependencies. Autopilot uses System of Difference Constraints (SDC) scheduling, where operations have a schedule variable to represent the time step at which they occur, with constraints in an integer-difference form. SDC [13] relies on the fact that the resulting constraint matrix is unimodular – a linear program with such a matrix will always have integer solutions, so an optimal schedule can be found in polynomial time. Autopilot expresses data, control and timing constraints in this way. Additionally, Autopilot uses a modified SDC that allows

Metric	RTL Expert	AutoPilot Expert	Difference (%)
Dev. time (man-weeks)	16.5	15	−9
LUTs	32 708	29 060	−11
Registers	44 885	31 000	−31
DSP48s	225	201	−11
18K BRAMs	128	99	−26

Fig. 6: Comparison of hand-coded HDL to Autopilot results, from [2].

some constraints to be violated (called *soft constraints*), which allow the tool to more easily make trade-offs between design goals – this is possible since FPGA component timings are estimated, and a small timing violation will usually not affect correct functionality of a design. Autopilot also applies other optimizations to efficiently map constructs to FPGA primitive blocks, to efficiently share resources, and to partition memory to increase throughput of loops with array accesses.

C. Results

The authors of Autopilot compare a hand-coded RTL design of a sphere decoder to a version created with Autopilot. The RTL version was created by Xilinx, while the HLS version was converted to Autopilot compatible C++ from about 4000 lines of MATLAB code, with the algorithm consisting mainly of linear algebra operations. Autopilot was able to generate final designs consuming less area in less time than hand-coding. Figure 6 compares area and design time of the two methods.

D. Discussion

Autopilot provide several benefits in terms of solving the programming problem for FPGAs. The input as C/C++ can be compiled as either software or hardware, which facilitates algorithmic correctness verification. There is a more familiar interface for designers, and it has been shown that productivity can be enhanced. However, HLS in this form does not give a high enough abstraction – Autopilot code must still be sprinkeled with `#pragma` directives to get comparable performance, and generally needs design iterations that require an understanding of the RTL code that the tool outputs. Such iterations are difficult to automate given the general-purpose nature of the tool, but constraining the application domain may allow higher-level abstractions to be built without sacrificing quality of results.

IV. Halide: Domain-Specific, Functional Image Processing

A DSL is a language with a constrained set of features directed towards a certain type of application. This has two immediate benefits – it is easier for the programmer to express solutions to problems in that domain, and constraints allow the DSL compiler to make assumptions and optimizations that are not necessarily possible in a general purpose language. This can lead to better results and shorter development times. In addition, a DSL can mask underlying complexity from the user, a fact we wish to exploit for hardware compilation. In this section we examine a recent DSL for image processing called Halide [3], by Ragan-Kelley *et al.*.

A. Halide DSL

Image processing code in traditional languages like C++ usually consist of multiple, sometimes deeply nested loops. In optimized forms, code can grow extremely complicated to take advantage of parallelism and locality inherent in the algorithm, data and CPU memory heirarchy – loops are split, parallelized and littered with inline assembly and primitives like SSE or AVX instructions for vectorization. Halide abstracts away these complexities by introducing a functional language for image processing embedded in C++. In Halide, what would be mutable arrays are instead simply functions from coordinates to values. Functions are defined by side-effect free expressions that include arithmetic/logic operations, loads, if-then-else blocks, references and other function calls. Consider the box filter example presented in [3]:

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1, y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

There is no recursion, higher-order functions or data structures. Functions are defined pure and over infinite domains, and boundary conditions are computed on the fly as needed when a `Func` is realized. Halide also provides a way to express some recursive computations like summation through *reductions*, which have an initial value function and a recursive function to redefine values. Reduction order is specified using a bounded *reduction domain*.

B. Pipeline Scheduling and Compilation

Halide programs consist of functions that feed into one another, forming a pipeline stencil computation. However, the functions that define the algorithm do not define when values are computed or where they are stored or cached – these choices greatly affect performance, but not computed results. Ragan-Kelley *et al.* refer to this set of choices as a *schedule*, and it is a trade-off between locality, parallelism and value recomputation.

The simplest schedule is *breadth first*, where a function is fully evaluated before those that depend on it; highly parallel but sacrificing locality between functions. Opposite this, an output value can be computed by computing all intermediate values on the fly, optimizing locality but causing a large amount of redundant work when windows overlap. Other options include interleaving computation over sliding windows and splitting images into tiles that are computed in parallel. Figure 7 from [3] gives a visual overview of the space of scheduling choices using the above box filter as an example.

Compilation begins by lowering Halide functions into loops, proceeding recursively from the output and referring to the schedule for placement of callees. Loops bounds are inferred and injected, also in recursive fashion. Multi-dimensional loads, stores and allocations are flattened into single dimension, and the compiler performs constant loop unrolling and vectorization where possible. Back-end code generation based on LLVM supports x86 and ARM, and supports respective SIMD operations (NEON, SSE, AVX).

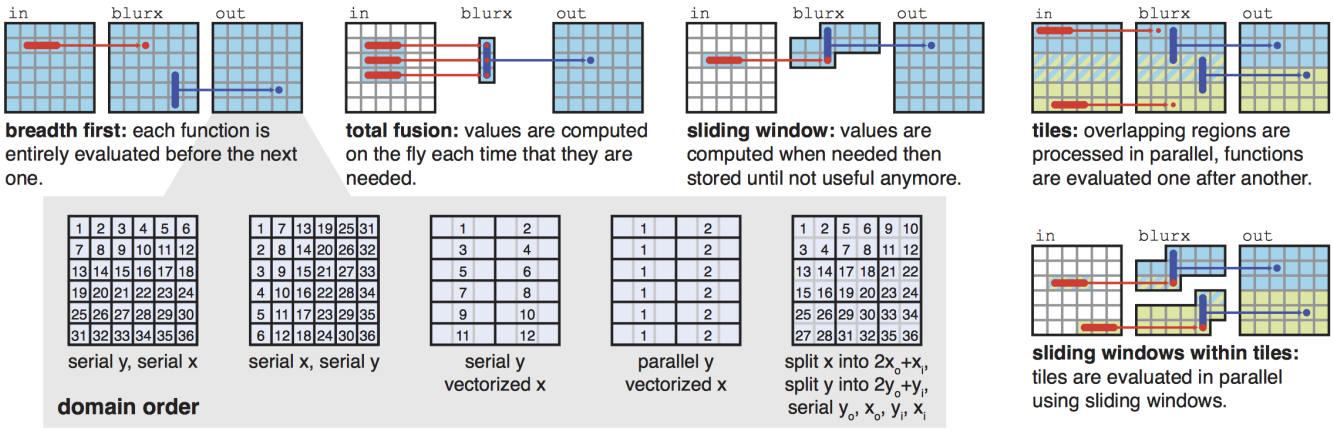

Fig. 7: Different scheduling options for the 3x3 box filter example, from [3].

C. Autotuned Scheduling and Results

The authors of Halide use a stochastic search (specifically a genetic algorithm) to find good schedules for Halide programs. Examples are compared against previously published expert implementations. In most cases, the Halide compiled implementations are faster, while taking few lines of code to express. We consider the local laplacian filter example. The reference is a 262 line Adobe-developed C++ program, which took three months to develop and optimize, employing parallelization and hand-tuned assembly. The Halide version is expressed in 52 lines of code written in one day and is reported to be 1.7x faster than the hand-tuned version, although the stochastic search took about two days to find the schedule that accomplished this.

D. Discussion

The Halide language has experienced success and fairly widespread adoption, as it can produce excellent results while providing a simple, functional interface familiar to many programmers. It is an excellent example of the type of abstraction we hope to bring to large-scale hardware acceleration. The user is able to express their solution and reason about it effectively, while the low-level mechanics of compilation, vectorization, parallelization and optimization are masked. While Halide makes use of existing software compilers, mapping high-level code to HDL requires more specialized tools such as Autopilot as discussed above. Combining technologies like Halide and Autopilot may lead to effective solutions to the FPGA programming problem, as we propose in the following section.

V. Research Proposal

As discussed in this proposal and shown by the Catapult system, FPGAs can provide a way to bridge the increasing gap between server performance increase and datacenter service requirements. To gain wider adoption as a solution, however, FPGA development must be made more palatable to those creating datacenter services. DSLs and HLS are not complete solutions in themselves, and we propose to combine technologies like these together to create a more complete solution.

A. Domain-Specific, Compile-to-HDL Languages

Domain-Specificity will provide ease of design entry and accessibility to non-hardware developers. At the same time, DS will allow specific optimizations to be integrated into the DSL compiler that will help generate higher performance circuits. We plan to leverage existing HLS tools to provide a backend for the front-facing DSL that can generate HDL, such as Autopilot or LegUp. Both of these tools use LLVM as a base compiler infrastructure, which, having a widely used and well-documented IR, will facilitate integration with existing or custom DSLs. In related work, George *et al.* take a similar approach, compiling Scala-embedded DSLs down to hardware by generating HLS-compatible C code [14], focusing on common parallel computation patterns. HIPA^{cc} [15] also takes a similar approach for image processing. Other examples of DSL to HDL compilers include MATLAB's HDLCoder [16] and SPIRAL [17] for DSP applications.

In general we hope to close the semantic gap between the languages programmers are used to, and the languages that describe hardware circuits. One problem lies in the fact that most modern high-level languages assume a standard Von Neumann execution model, and the languages and their semantics heavily reflect this. Variables correspond to registers or storage, control flow maps straightforwardly into branch instructions, and other expressions generally conform to referencing memory and executing arithmetic operations in sequential fashion. Implementing an algorithm in hardware assumes *no* fixed execution model (i.e. an overall circuit architecture and memory architecture), which is why HLS with a C/C++ frontend is so difficult and requires special `#pragmas` and multiple iterations to obtain good results.

We propose to leverage the high-level nature of DSLs to constrain the set of reasonable architectures ("execution models") for a given domain. Figure 8 illustrates the concept. Template architectures can be predefined by hardware experts, or possibly auto-generated by the DSL compiler through domain-constrained program analysis. Underlying HLS compilation and scheduling can be modified or directed to target whatever architecture was selected or generated. The result can then

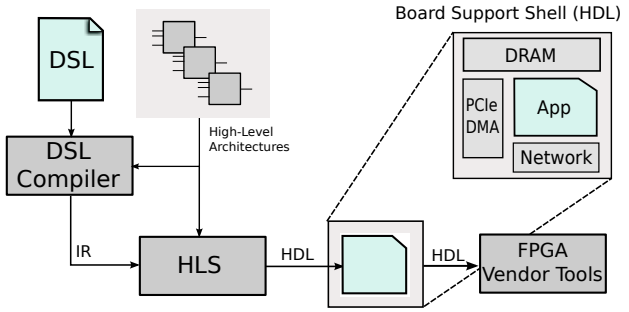

Fig. 8: Proposed flow to leverage domain-specificity in hardware compilation. A DSL compiler lowers a program to a form suitable for HLS that targets a domain-specific high-level architecture. A Board Support Shell provides a fixed interface to common subsystems for the compiled circuit.

be placed in a shell framework similar to that developed for Catapult, and handed off to FPGA vendor tools for synthesis, placement and routing on the target FPGA. The shell is built to support specific FPGA devices and boards (the Board Support Shell in Figure 8), and would ideally export vendor-agnostic interfaces to layers above, increasing portability of the compiler toolchain and user hardware designs. The shell would also have supporting software and drivers that allow testing and execution. Thus datacenter service developers will have access to effective hardware acceleration with acceptable results without having to be digital design experts.

1) Preliminary Work: As a first step, we have used the LegUp HLS tool to provide back-end HDL compilation for Halide programs. A new LLVM bit code generator was written, based on the generators for x86 and ARM architectures, which generates LLVM bitcode that conforms to the constraints imposed by LegUp. All images must be embedded as LegUp does not support dynamic memory allocation. The final output image must also be the same size as the input, as the output buffer size is inferred from the input, so that all loops conditions can be governed by constants. This allows us to leverage the pipelining feature supported by Legup.

The tool is funcional but so far does not provide satisfactory results. This can be attributed mostly to LegUp's relatively fixed, Von Neumann-like architecture. A microprocessor system is assumed, which restricts the memory architecture to a bus system, while accelerator local memory is limited to dual port RAMs. There is no facility for streaming data or implementation of line buffers or similar structures, which are crucial in high-performance image processing hardware. This experience confirms to us that domain-specific high-level architectures are necessary when using general-purpose HLS tools as back-end HDL compilers.

B. Multi-FPGA Systems

As the Catapult experiments have shown, datacenter-scale services are large and complex, and will likely require multiple FPGAs worth of reconfigurable fabric to implement. We additionally propose to generalize the backend hardware system generation to support multi-FPGA systems, as we see in Catapult. This will allow simple implementation of large-scale hardware acceleration that requires more than one device. Other problems may arise through this work that we may also explore, such as how to schedule and deploy multi-FPGA circuits on a shared fabric, and how to partition the compiled circuit among several devices. These may leverage existing work in compute workload scheduling and circuit netlist partitioning.

VI. Conclusion

The increasing gap between server processor performance and datacenter workloads is a problem that will need to be addressed to enable the complex, demanding workloads of the future. FPGAs have been shown to be an effective platform for compute acceleration in the datacenter, but to enable widespread adoption and utilization, low-level hardware design must be abstracted to a programming model more palatable to datacenter service developers. We have proposed using Domain Specificity to constrain execution models or high-level architectures of a given domain, and to then use this information to guide HLS compilation, providing a relatively high-performance hardware system that can readily be deployed on an FPGA board. The user writes their solution in a high-level specification (DSL) and is completely shielded from the intricacies of hardware design.

These ideas and the presented compiler toolchain concept leave many research questions open, such as:

1) High-level architectures can be predefined. Is it possible to automate generation or customization of these by analyzing the input DSL program? To what extent? How does the domain affect this?
2) What is the trade-off between domain constraint and the extent of general-purpose HLS use? That is, can one increase constraints and do less with a given DSL, but get a higher performance circuit?
3) How and when in the compile flow should a design be partitioned to fit in and be deployed on a multi-FPGA fabric like Catapult? What are the trade-offs involved when performing this?

As we build a toolchain that realizes the ideas we have presented in this proposal, we will begin to answer some of these questions, and in the process, begin to provide solutions that will enable new, demanding datacenter services.

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *ISCA 2014*. IEEE, 2014, pp. 13–24.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[3] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," *ACM SIGPLAN*, vol. 48, no. 6, pp. 519–530, 2013.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[5] A. Parashar, M. Adler, K. Fleming, M. Pellauer, and J. Emer, "LEAP: A Virtual Platform Architecture for FPGAs," in *1st Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2010)*, 2010.

[6] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: An in-fabric memory architecture for fpga-based computing," in *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 97–106.

[7] R. Nikhil, "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *Formal Methods and Models for Co-Design, 2004*. IEEE, 2004, pp. 69–70.

[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[9] C-to-Silicon Compiler. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx

[10] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Interactive C Synthesis," in *High-Level Synthesis*. Springer, 2008, pp. 29–52.

[11] Synphony C Compiler. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx

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

[13] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation," in *Proceedings of the 43rd Annual Design Automation Conference*. ACM, 2006, pp. 433–438.

[14] 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*. IEEE, 2014, pp. 1–8.

[15] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2014, pp. 17:1–17:10.

[16] HDL Coder. [Online]. Available: http://www.mathworks.com/products/hdl-coder/index.html

[17] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 2, 2012.