Fast and Flexible High-Level Synthesis from OpenCL using Reconfiguration Contexts

James Coole, Greg Stitt
University of Florida
Department of Electrical and Computer Engineering and
NSF Center For High-Performance Reconfigurable Computing
Gainesville, FL, USA
jcoole@ufl.edu, gstitt@ece.ufl.edu

***Abstract*—High-level synthesis from OpenCL has shown significant potential, but current approaches conflict with mainstream OpenCL design methodologies due to 1) orders-of-magnitude longer FPGA compilation times, and 2) limited support for changing or adding kernels after system compilation. In this paper, we introduce a backend synthesis approach for potentially any OpenCL tool, which uses virtual coarse-grained *reconfiguration contexts* to speedup compilation by 4211x at a cost of 1.8x system resource overhead, while also enabling 144x faster reconfiguration to support different kernels and rapid changes to kernels.**

Keywords—OpenCL; FPGA; intermediate fabrics

I. Introduction

High-level synthesis (HLS) from OpenCL is an emerging design strategy for field-programmable gate arrays (FPGAs) that improves productivity while potentially enabling significant advantages over other devices [1][2]. Although previous OpenCL synthesis has shown important technical achievements [3][9][10], several limitations prevent more widespread FPGA usage. One significant limitation is that FPGA compilation commonly ranges from hours to even days [11], which limits productivity and prevents common design methodologies. Lengthy compilation also prevents OpenCL's runtime compilation, which further contributes to niche FPGA usage. Previous OpenCL synthesis also has limited support for the addition of new kernels and changes to existing kernels. Unlike GPUs and multicores, previous FPGA approaches create a fixed number of kernel-specific accelerators [3][9][10], which prevents execution of new and modified kernels without lengthy compilation of accelerator hardware.

Lengthy compilation times have been improved by recent work on *intermediate fabrics (IFs)* [2], which enabled 1000x faster place-and-route compared to device-vendor tools at a cost of 21% of FPGA resources [2][6][11]. IFs achieve fast compilation via virtual coarse-grained resources implemented atop a physical FPGA. When using an IF, synthesis avoids decomposing the application into hundreds of thousands of the FPGA's lookup tables (LUTs), and instead directly maps behavior onto application-specialized resources (e.g., floating-point units, FFT cores).

In this paper, we complement existing OpenCL synthesis by introducing a backend approach, shown in Figure 1, which specializes IFs for OpenCL to enable fast compilation and reconfiguration while improving support for adding and changing kernels. Whereas all previous IFs were manually designed, the main research challenge of integrating IFs with OpenCL is automatically determining an effective fabric architecture for an application or domain. Because it is clearly not possible to create an optimal fabric for all combinations of kernels, we present a context-design heuristic that analyzes kernel requirements from an application (or domain) and clusters them based on similarity

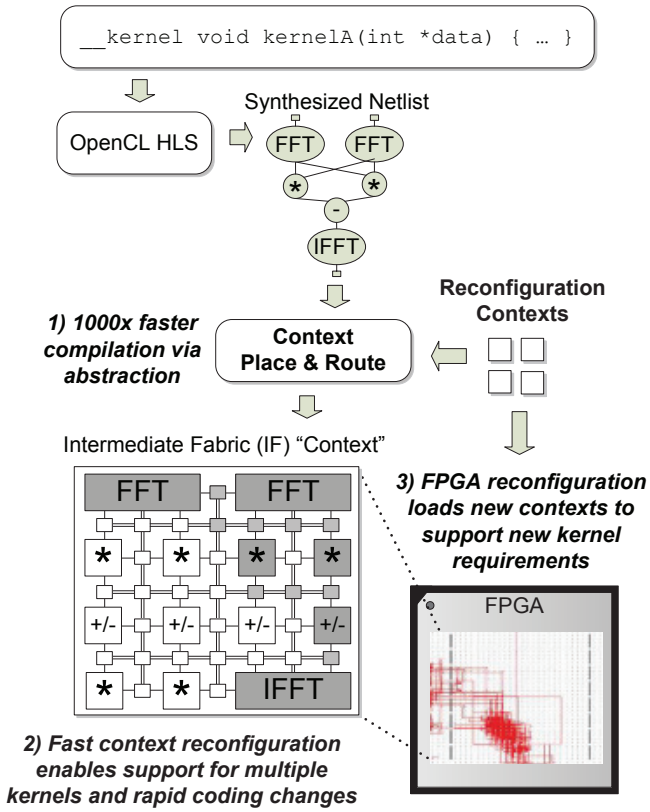


Figure 1: OpenCL synthesis using intermediate fabric reconfiguration contexts to enable fast compilation and improved support for adding/changing kernels.

Digital Object Indentifier 10.1109/MM.2013.108 0272-1732/$26.00 2013 IEEE

This article has been accepted for publication in IEEE Micro but has not yet been fully edited.
Some content may change prior to final publication.

into a set of fabrics referred to as *reconfiguration contexts*. As long as a context supports an application's kernels, the application benefits from orders-of-magnitude faster compilation and reconfiguration between kernel executions. When a context does not support a kernel, our backend reconfigures the FPGA to load a new context provided by an existing bitfile or by synthesizing a new fabric onto the FPGA. The overall goal of the heuristic is to minimize individual context area by minimizing the number of resources required for the context to support all its assigned kernels. This savings in area can help the system fit under area constraints, and may also be used to scale up each context to preemptively increase its flexibility to support similar kernels.

Experimental results show 4211x faster compilation than device-vendor tools. Fixed-point contexts had an average area overhead of 7.8x (5.6x for the entire system), but floating-point context overhead averaged only 1.4x due to the larger cores outweighing fabric interconnect. Area overhead averaged 1.8x for a system of 20 kernels, with negligible average clock overhead. Although this area overhead may be limiting for some use cases, the results are highly pessimistic, and area reduction of IFs remains an active area of complementary research. Furthermore, reconfiguration contexts enable 144x faster reconfiguration than FPGAs, which can further hide overhead when time-multiplexing many kernels.

II. Background: Intermediate Fabrics

Initial IFs [2] used coarse-grained resources with multiplexor-based switch boxes and bi-directional tracks, which showed orders-of-magnitude faster place & route at a cost less than 40% of FPGA resources. A later study [11] evaluated several DSP kernels on a large IF, which showed 1000x speedup in place & route, with the IF using 2.9-4.4x more area than any individual circuit. However, the fabric in that study could be completely reconfigured in just 72 cycles to hide overhead via time multiplexing. A more recent study [6] focused on an optimized virtual interconnect, which reduced area overhead by 50%. Clock overhead varied depending on fabric size, but on average was negligible. These previous studies are complementary to this paper, where we leverage IFs to improve OpenCL synthesis.

III. Reconfiguration Contexts

In this paper, we consider IF implementations of reconfiguration contexts, though other architectures are possible. For example, at one extreme, a context tasked with supporting three kernels could implement three kernel-specific accelerators, similar to existing approaches like Figure 3(a). Although possibly more efficient than an IF, this context architecture provides no support for kernels not known at system generation, which is a limitation of existing OpenCL synthesis [3][9][10]. In addition, kernel-specific accelerators may have limited scalability due to a lack of resource sharing across kernels. IFs are attractive because only common computational resources (e.g., *add, mul, sqrt, rand, FFT*) are fixed after generation, with their interconnection remaining configurable. IF-based contexts provide enough flexibility to support kernels similar to those for which the context was originally designed, as might be seen during iterative system development or creeping system requirements and workloads.

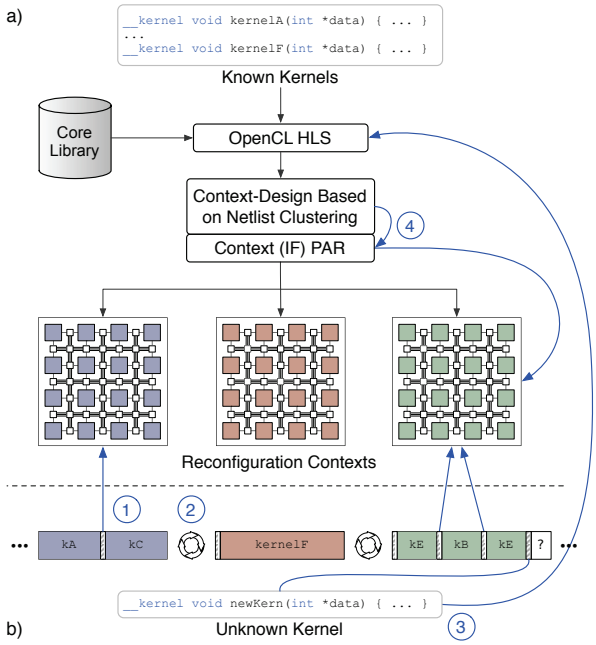


Figure 2: An overview of (a) system generation using OpenCL HLS with a reconfiguration-context backend, and (b) runtime 1) reconfiguration of contexts, 2) loading of different contexts to support all kernels, and 3+4) compilation of new kernels.

A. Overview

Figure 2 gives an overview of reconfiguration contexts. During system generation, Figure 2(a), an OpenCL frontend (potentially any) synthesizes kernels from the system's source to a netlist of coarse-grained cores and control. Our backend then uses a *context-design heuristic* based on netlist clustering to group these netlists into sets with similar resource requirements, and then designs an IF-based context for each set, which device-vendor tools compile to an FPGA bitstream (not shown).

At runtime, Figure 2(b), the system loads a context into the FPGA and then rapidly reconfigures the context to execute different supported kernels (e.g., *kA* and *kB* for the *blue context* in step 1). When the loaded context does not support a kernel, the system reconfigures the FPGA with a new context (e.g., *red context* for *kernelF* in step 2). When a new kernel executes that was unknown during system generation (step 3), the synthesis frontend creates a new circuit, which the backend clusters onto an existing context (e.g., *green context* in step 4), while performing *context place & route (PAR)* to create a context bitfile for the kernel. Note that such runtime compilation is possible due to the rapid compilation times provided by IFs (0.32s per kernel on average). Lengthy FPGA compilation is only required when

Digital Object Indentifier 10.1109/MM.2013.108 0272-1732/$26.00 2013 IEEE

the backend initially creates contexts, or when a new kernel is not supported by current contexts, which gives the worst-case compile time for a new/changed kernel. Therefore, designing appropriate contexts is the most critical part of this tool flow.

B. *Context-design heuristic*

The main challenge of context design is determining how many contexts should be created to support the known kernels, what functionality to include in each context, and how to assign kernels to contexts. Context design can be seen an optimization problem where we'd like to maximize the number of resources reused across kernels in a context while minimizing the area of individual contexts. The resulting savings in context area is then typically used to enable scaling up contexts to increase flexibility to support kernels other than those known at compile time (e.g., new kernels introduced during development or *in vivo*).

Our approach begins with the observation that groups of kernels often exhibit similarity in their mixture of operations—an observation that has motivated application-specialized processors for many domains. Because interconnection remains flexible in IF-based contexts, our context-design heuristic considers this functional similarity over any structural similarity. Note that for other context architectures that want to exploit structural similarity between kernels (e.g., to reduce interconnect area), other approaches to kernel grouping can be used. However, even for these architectures, functional grouping is useful when the cost of functional resources is expected to be greater than the cost of interconnect. Our approach currently ignores functional timing differences, assuming that these differences are minimized through pipelining [2]. However, this information could be considered during grouping when pipelining isn't sufficient, e.g., by using an additional f_{max} dimension in a clustering heuristic.

Contexts that minimally support one member of these groups should support other members of the group, perhaps requiring different numbers of resources, or the addition of several of other resources. Thus, identifying these groups provides a prescription for designing contexts that are compatible with kernels similar to the kernels inside each group. Note that the collection of kernels used for context design can also be augmented with kernels not currently in the program, but which might be considered likely (e.g., based on the design's history) to further guide the flexibility of the contexts designed.

For IF-based contexts, we identify these groups using a clustering heuristic in an n-dimensional feature space defined by the functional composition of the system's kernel netlists. This space includes an element for each core type used in the application, ignoring differences that can be resolved through promotion or runtime configuration (e.g., bit-width or comparator predicate). Because otherwise similar netlists may be of different sizes (e.g., different FIR filters), and because the context may be scaled up to improve flexibility, the relative composition of each kernel is used instead of

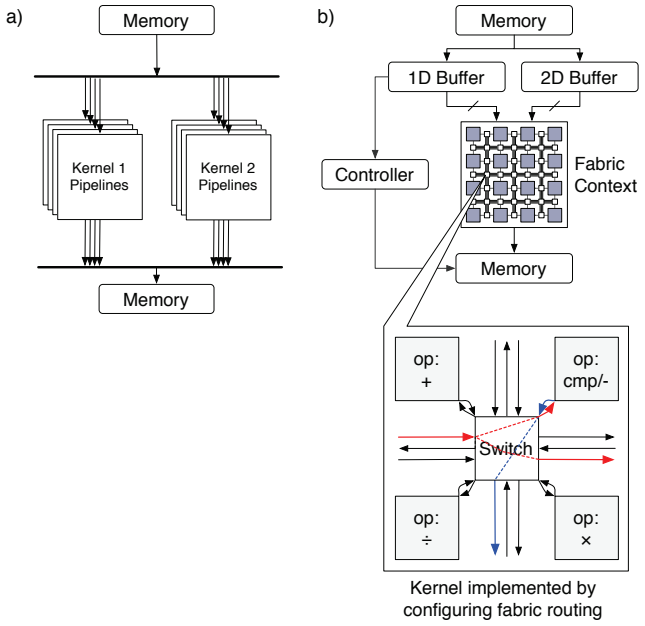


Figure 3: a) Previous OpenCL FPGA implementation with separate datapaths for each kernel. b) Reconfiguration-context architecture, where kernels are implemented as needed via IF configuration.

absolute counts. For example, for an application containing two kernels, *FIR* and *SAD*, clustering would operate on: *SAD*=<0.3, 0.0, 0.7> and *FIR*=<0.5, 0.5, 0.0>, in the space <f_{add}, f_{mul}, $f_{cmp/sub}$>.

We currently use k-means clustering to group netlists in this space, resulting in up to k sets of netlists for which individual contexts will be designed. The heuristic can use the resource requirements of cores to estimate the area required for each cluster, allowing k to be selected subject to device or system-imposed area constraints. The user may also select a value for k to satisfy system goals for flexibility. Fully automated selection of k in these scenarios is left as future work.

We currently assume the application time-multiplexes kernels on each context, with one kernel implemented on a context at a time. Although this limits system parallelism to that achievable by the executing kernel, memory bandwidth frequently imposes this limit on accelerators anyway, especially when they are well pipelined. This assumption allows the resources required by any netlist in a given cluster to be fully shared by other netlists in the same cluster. Thus, the minimum number of each resource type needed for a context is the maximum number of that resource across all netlists in the corresponding cluster. The heuristic designs an IF for each cluster by including at least this count of each resource type, adding interconnect until place & route succeeds for all corresponding netlists.

IV. OpenCL-IF Compiler

Though reconfiguration contexts could potentially be integrated with existing OpenCL synthesis tools (Section

Digital Object Indentifier 10.1109/MM.2013.108 0272-1732/$26.00 2013 IEEE

IV.A), we developed a custom tool to simplify IF integration, which we describe in Section IV.B.

A. *Previous work*

Previous work on OpenCL synthesis has tended towards architectures that instantiate separate accelerator hardware for each kernel in the application, as shown in Figure 3(a). Owaida [9] implemented these accelerators as groups of ALUs between separate memories implementing barrier interfaces. Within a group, these elements were controlled using code slicing, and had arbitrated access to global memory. Altera's OpenCL Compiler [3] implements pipelined accelerators for each kernel, with local memory also implemented in global banks. FCUDA [10], for the related CUDA language, implements accelerator datapaths for kernel functions, using a configurable interconnect to memory to allow efficient routing of results and inputs between kernel stages. All of these approaches require lengthy FPGA compilation and/or reconfiguration for kernel changes.

B. *OpenCL-IF Overview*

Our tool, OpenCL-IF, is based on the Low-Level Virtual Machine (LLVM) [7] and its C frontend. OpenCL-IF first compiles a kernel into LLVM's intermediate representation, substituting custom intrinsics for system functions (e.g., *get_global_id*). The tool uses LLVM's standard optimization passes, including inlining auxiliary functions and unrolling loops, and also performs common hardware-specific optimizations [8]. The tool then creates a control data flow graph (CDFG), simultaneously mapping LLVM instructions to compatible cores provided in a user-specified library. Note that cores may be added to this library to enable limited application- and target-specific optimizations. DFGs are independently scheduled and bound to final resources provided by the context using common approaches [8]. Finally, the resulting kernel netlists are implemented on the context through place & route, yielding a *context bitstream* for each kernel. At runtime, a kernel is executed after configuring its context with this bitstream, which is mutually exclusive per context.

Because the main contribution of this paper is integration of reconfiguration contexts with OpenCL, we omit a complete discussion of all the features of OpenCL-IF. However, one feature that is unique to OpenCL-IF is support for efficient data streaming from external memories. In previous approaches, kernel accelerators comprised of multiple pipelines compete for global memory through arbitration, as shown in Figure 3(a) [9]. Previous work on synthesis from C [5] and custom circuit design has addressed memory bottlenecks common in this approach using specialized buffers [5][12] that detect overlapping memory-access patterns to exploit data reuse. In OpenCL-IF, we adapt those approaches for OpenCL based on the observation that many OpenCL kernels limit accesses for each buffer to some set of constant offsets (a window) relative to their id vector. In this common case, OpenCL-IF schedules work items to enable fully pipelined implementations using 1- or 2-D sliding-window buffers as shown in Figure 4(b). We plan to evaluate the performance impact of this optimization, which is independent of the OpenCL backend presented here, as future work.

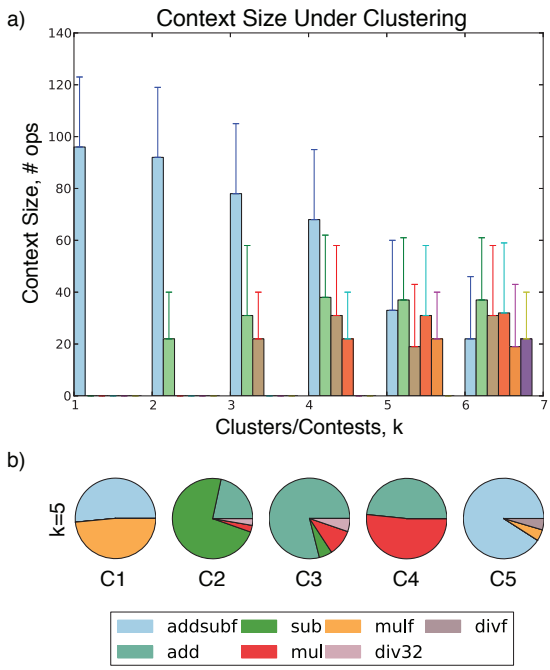


Figure 4: a) Size in # operators (ops) of minimum-sized IF contexts required to implement kernel clusters with different degrees of clustering *k*. b) Resource composition of contexts for *k*=5 based on similarities in clustered netlists.

V. Experiments

In this section, we evaluate reconfiguration contexts over a number of kernels, using the motivating example of a single framework for computer-vision applications that execute multiple image-processing kernels in different combinations at different times (e.g., as stages in larger processing pipelines). In Section V.A, we evaluate the context-design heuristic for this system, providing minimal guidance by using only the framework's known kernels. In Section V.B, we evaluate compilation time, overhead, and reconfiguration time.

The experiments evaluate fixed-point and single-precision floating-point (*FLT*) versions of 10 OpenCL kernels (20 total). *FIR* is a finite impulse-response filter with 16 taps. *Gaussian* performs image blur. *Sobel* performs edge detection. *Bilinear* performs image downscaling. *Threshold* binarizes image pixels in around the local mean. *Mean*, *Max,* and *Min* give the local average, maximum, and minimum image intensity. *Normalize* improves local image dynamic range by scaling. *SAD* performs sum of absolute differences for image recognition.

Digital Object Indentifier 10.1109/MM.2013.108 0272-1732/$26.00 2013 IEEE

Table 1: A comparison of compilation time, area, and clock frequency for OpenCL-IF reconfiguration contexts and direct FPGA implementations for a computer-vision application with *k*=5. Floating-point kernels are shown with an *FLT* suffix.

		Reconfiguration Contexts				Direct FPGA Implementations					Overhead	
Kernel	**OpenCL-IF Time**	**PAR Time**	**Total Time**	**Clock**	**Area**	**XST Time**	**PAR Time**	**Total Time**	**Clock**	**Compilation Speedup**	**Clock**	**Area**
FIR 16 tap	0.114s	0.416s	0.530s			10.8s	85s	96s	275MHz	181x	18.3%	
Gaussian 4x4	0.125s	0.481s	0.606s			10.8s	85s	96s	275MHz	159x	18.3%	
Sobel 3x3	0.130s	0.481s	0.611s	225MHz	13.4%	10.8s	85s	96s	275MHz	157x	18.3%	**14.9x**
Kernel average	**0.123s**	**0.459s**	**0.582s**			**10.8s**	**85s**	**96s**	**275MHz**	**166x**	**18.3%**	
Cluster 1 total	**0.369s**	**1.377s**	**1.746s**			**32.4s**	**255s**	**288s**		**165x**		
Bilinear	0.080s	0.087s	0.167s			9.4s	520s	530s	275MHz	3,170x	6.9%	
Mean 4x4	0.109s	0.189s	0.298s			9.4s	1362s	1371s	320MHz	4,600x	20.0%	
Threshold 4x4	0.117s	0.256s	0.373s	256MHz	10.8%	9.5s	1449s	1458s	280MHz	3,908x	8.7%	**3.9x**
Kernel average	**0.102s**	**0.177s**	**0.279s**			**9.4s**	**1110s**	**1120s**	**292MHz**	**3,893x**	**11.9%**	
Cluster 2 total	**0.306s**	**0.532s**	**0.838s**			**28.2s**	**3331s**	**3359s**		**4,007x**		
Max 4x4	0.178s	0.108s	0.287s			11.3s	88s	99s	229MHz	347x	1.5%	
Min 4x4	0.160s	0.115s	0.275s			10.8s	83s	94s	225MHz	344x	-0.0%	
Normalize 3x3	0.178s	0.127s	0.305s			11.4s	1268s	1279s	228MHz	4,195x	1.1%	
SAD 3x3	0.145s	0.033s	0.178s	225MHz	16.2%	10.6s	81s	92s	263MHz	516x	14.5%	**4.6x**
Kernel average	**0.165s**	**0.096s**	**0.261s**			**11.1s**	**380s**	**391s**	**236MHz**	**1,350x**	**4.3%**	
Cluster 3 total	**0.661s**	**0.383s**	**1.044s**			**44.3s**	**1520s**	**1565s**		**1,499x**		
FIR 16 tap FLT	0.119s	0.260s	0.379s			**9.5s**	**4255s**	4264s	120MHz	11,263x	-63.3%	
Gaussian 4x4 FLT	0.111s	0.297s	0.408s			**9.5s**	**3994s**	4004s	120MHz	9,823x	-63.3%	
Sobel 3x3 FLT	0.116s	0.156s	0.272s	196MHz	40.4%	**9.3s**	**6122s**	6131s	156MHz	22,515x	-25.4%	**1.6x**
Kernel average	**0.115s**	**0.237s**	**0.353s**			**9.4s**	**4790s**	**4800s**	**132MHz**	**14,534x**	**-50.7%**	
Cluster 4 total	**0.346s**	**0.712s**	**1.059s**			**28.3s**	**14371s**	**14400s**		**13,603x**		
Bilinear FLT	0.077s	0.090s	0.167s			**9.9s**	**284s**	294s	185MHz	1,762x	-6.1%	
Mean 4x4 FLT	0.115s	0.183s	0.298s			**9.3s**	**5843s**	5853s	149MHz	19,616x	-31.7%	
Threshold 4x4 FLT	0.114s	0.106s	0.220s			**9.4s**	**203s**	212s	228MHz	965x	14.1%	
Max 4x4 FLT	0.168s	0.080s	0.248s			**10.2s**	**199s**	209s	190MHz	846x	-3.0%	
Min 4x4 FLT	0.150s	0.083s	0.233s	196MHz	48.0%	**10.7s**	**214s**	225s	201MHz	966x	2.7%	**1.3x**
Normalize 3x3 FLT	0.162s	0.096s	0.259s			**10.4s**	**237s**	248s	200MHz	959x	2.4%	
SAD 3x3 FLT	0.173s	0.115s	0.288s			**9.9s**	**284s**	294s	240MHz	1,023x	18.7%	
Kernel average	**0.137s**	**0.108s**	**0.245s**			**10.0s**	**1038s**	**1048s**	**199MHz**	**3,734x**	**-0%**	
Cluster 5 total	**0.959s**	**0.753s**	**1.712s**			**69.7s**	**7265s**	**7335s**		**4,285x**		
Kernel average	**0.132s**	**0.188s**	**0.320s**		**25.8%**	**10.2s**	**1337s**	**1347s**	**222MHz**	**4,366x**	**-2%**	
System total	**2.642s**	**3.758s**	**6.399s**		**128.8%**	**203.0s**	**26742s**	**26947s**		**4,211x**		**1.8x**

All experiments target a Xilinx Virtex 6 XC6VCX130T-1FF1154 and use Xilinx ISE 14.4.

A. Context Design and Clustering

To evaluate the context-design heuristic, we used the heuristic to generate reconfiguration contexts, based on only the 20 known kernels, using different degrees of clustering *k* (e.g., *k=2* uses *two* clusters to support all kernels). Figure 4(a) shows the fabric size (in operators) of resulting contexts, minimally sized to support their assigned kernels. As expected, fewer clusters results in larger contexts, as the different resources used by each kernel limits resource sharing. More clusters result in smaller contexts, as fewer resources are required to support each set of kernels, however this effect is ultimately bounded by the largest netlist in each cluster (here, around *k*=6).

In Figure 4(b), we present a more detailed analysis for *k*=5, with the resulting kernel clustering shown in Table 1. As shown, the resource amounts and types in each context strongly reflect the netlists in each corresponding cluster. For example, cluster *C2* implements *Max*, *Min*, *Threshold*, and *SAD* kernels, which are all dominated by the use of subtractors. The subtractors are used as comparators in all these netlists, and also as differences in *SAD*. Different operators, or pairs, dominate the other contexts.

Figure 4 shows that, for this application, using five clusters provides a significant 60% decrease in largest context size compared to using a single cluster to support all kernels. The tradeoffs represented by different *k* values can be used for different purposes depending on designer intent. The 60% size savings provides significant flexibility to allow implementation of all kernels under area constraints, enabled by implementing each context minimally. However, when flexibility is crucial, the amount of clustering could be increased to provide a better match of the underlying kernels, with the resulting clusters increased in size up to device capacity or an area constraint to provide better support for unknown netlists. Future work includes a detailed analysis of these tradeoffs.

B. Compile Times, Overhead, and Reconfiguration

In Table 1, we compare kernels using reconfiguration contexts in the OpenCL-IF compiler with kernels compiled directly to the FPGA using VHDL generated from OpenCL (i.e., previous approaches). To ensure a fair comparison, the VHDL implementations are pipelined the same (except for IF routing) and use the same core implementations as the context netlists. For direct FPGA compilations, we used standard effort for faster compilation at the expense of some circuit quality. Execution times were compared on a quad-core 2.66 GHz Intel Xeon W3520 workstation with 12GB RAM, running CentOS 6.4 x86 64.

The *OpenCL-IF Time* column gives the time required to synthesize each kernel using the OpenCL-IF compiler,

Digital Object Indentifier 10.1109/MM.2013.108 0272-1732/$26.00 2013 IEEE

including all stages through netlist export. The second group of columns (*Reconfiguration Contexts*) gives the time required to place & route (*PAR Time*) each kernel netlist on its assigned context, the resulting total compilation time, in addition to the clock frequency and the FPGA LUT utilization (*Area*) of the context. The third group of columns (*Direct FPGA Implementations*) gives the time required to implement each circuit directly using ISE, broken into logic synthesis (*XST Time*) and place & route (*PAR Time*), and the frequency of each circuit. The last three columns give the compilation speedup from reconfiguration contexts, along with the resulting clock and area overhead. Note that a performance comparison with hand-optimized VHDL (or different OpenCL tools) is highly dependent on the OpenCL frontend, which is outside the scope of our reconfiguration context analysis.

Table 1 shows that, after context generation, reconfiguration contexts enable compilation of the entire system of kernels in 6.3s, 4211x faster than the ~7.5 hours required by ISE to compile directly to the FPGA. Table 1 also shows that the floating-point kernels experience a greater compilation speedup (6970x vs. 1760x), as more fine-grained device resources are hidden by their contexts. Because individual operators in these contexts are larger, the area used by the fabric's routing resources is also a smaller fraction of total area, decreasing area overhead (avg. 1.4x vs. 7.8x). Each kernel required an average of 0.32s to compile on a reconfiguration context, which also provides an estimate of the average compilation time of new, context-compatible kernels. Clock overhead was negligible on average, with additional pipelining in the fabric's interconnect benefiting some circuits (e.g., Cluster 4). Note that the speedup reported for each kernel is pessimistic, because each direct FPGA circuit was synthesized individually, whereas a real system would include as many circuits as would fit on the FPGA (e.g., [3]), increasing FPGA PAR times dramatically.

For each context, we also define its *area overhead* as the area required by the context's fabric compared to the area of all the corresponding kernels implemented directly on the FPGA. This system as a whole required 1.8x additional area compared to implementing all kernels directly. However, this extra area is not necessarily all overhead due to the significant added flexibility. For example, a new SAD kernel using a 4x2 template would synthesize on Context 3 without requiring modifications to the system's hardware. Under our definition, the addition of this kernel would reduce that context's overhead from 4.6x to ~3.9x. For an application with numerous kernels, reconfiguration contexts may even save area compared to direct FPGA implementations. Thus, this definition of overhead is pessimistic for any system with changing workloads. Furthermore, despite this overhead, the largest context used only 48% of the FPGA, which increases the applicability to many use cases. Though outside the scope of this paper, reconfiguration contexts can also enable runtime synthesis of optimized hardware based on variables known only at runtime (e.g., mask coefficients), which also isn't accounted for in this measure of overhead.

Table 2: A comparison of configuration bitstream sizes and times between reconfiguration contexts and the FPGA.

Context	Context Config Data Size	Context Config Time	FPGA Config Data Size	FPGA Config Time (Best)	Speedup
Cluster 1	578 B	20.6 µs			166x
Cluster 2	429 B	13.4 µs			255x
Cluster 3	683 B	24.2 µs	5.21 MB	3415.6 µs	141x
Cluster 4	964 B	39.4 µs			87x
Cluster 5	1208 B	49.3 µs			69x
Average	772 B	29.4 µs			144x
System total	15580 B				

Table 2 compares the configuration bitstream lengths and times for each context against the bitstream length and best-case reconfiguration time for the FPGA. The table shows that the uncompressed bitstream size of a kernel using this system's contexts is on average 772 bytes, requiring less than 16 KB for the 20 known kernels. Contexts in this system can be reconfigured with a new kernel in 29.4 µs on average (144x faster than FPGA reconfiguration), enabling efficient time-multiplexing of multiple kernels.

VI. Conclusions

In this paper, we introduced a backend approach to complement existing OpenCL synthesis, which uses virtual, coarse-grained *reconfiguration contexts* to enable 4211x faster FPGA compilation compared to device-vendor tools, at a cost of 1.8x area overhead. Furthermore, these contexts can be reconfigured in less than 29 µs to support multiple kernels, while using slower FPGA reconfiguration to load new contexts to support significantly different kernels. To create effective contexts, we introduced a clustering heuristic that groups kernels based on functional similarity and then creates intermediate fabrics to support the requirements of each group. Future work includes evaluating other architectures for reconfiguration contexts, including more specialized (and less flexible) interconnects, strategies for managing multiple contexts through partial reconfiguration, and optimizations enabled by runtime kernel synthesis.

Acknowledgment

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022. The authors gratefully acknowledge vendor equipment and/or tools provided by Xilinx.

References

[1] 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, IEEE Transactions on*, 30(4):473–491, 2011.

[2] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. *CODES+ISSS*, pages 13 –22, 2010.

[3] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From opencl to high-performance hardware on fpgas. *FPL*, pages 531–534, 2012.

[4] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne. Shortening

design time through multiplatform simulations with a portable opencl golden-model: The ldpc decoder case. *FCCM,* pages 224–231, 2012.

[5] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *LCTES*, pages 249–256, 2004.

[6] A. Landy and G. Stitt. A low-overhead interconnect architecture for virtual reconfigurable fabrics. *CASES*, pages 111–120, 2012.

[7] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. *CGO,* pages 75–86, 2004.

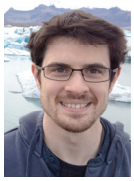
[8] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[9] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos. Synthesis of platform architectures from opencl programs. *FCCM*, pages 186–193, 2011.

[10] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. *SASP*, pages 35–42, 2009.

[11] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *Embedded Systems Letters, IEEE*, 3(3):81–84, 2011.

[12] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. *FCCM*, pages 127–134, 2010.



James Coole is a PhD student at The University of Florida. His research interests include computer architecture and design automation.

Greg Stitt is an Associate Professor in the ECE Department at The University of Florida and is a recipient of the NSF CAREER Award.

Digital Object Indentifier 10.1109/MM.2013.108 0272-1732/$26.00 2013 IEEE