Coarse Grained FPGA Overlay for Rapid Just-In-Time Accelerator Compilation

Abhishek Kumar Jain, *Member, IEEE*, Douglas L. Maskell, *Senior Member, IEEE*, and Suhaib A. Fahmy, *Senior Member, IEEE*

Abstract—Coarse-grained FPGA overlays built around the runtime programmable DSP blocks in modern FPGAs can achieve high throughput and improved scalability compared to traditional overlays built without detailed consideration of FPGA architecture. These overlays can be mapped to using higher level compilers, achieving fast compilation, software-like programmability and run-time management, and high-level design abstraction. OpenCL allows programs running on a host computer to launch accelerator kernels which can be compiled at run-time for a specific architecture, thus enabling portability. However, prohibitive hardware compilation times in traditional design flows mean that the tools cannot effectively use just-in-time (JIT) compilation or runtime performance scaling on FPGAs. We present a methodology for runtime compilation of dataflow graphs expressed as OpenCL kernels onto coarse-grained overlays. The methodology benefits from the high level of abstraction afforded by using the OpenCL programming model, while the mapping to the overlay significantly reduces compilation and load times. Key characteristics of this work include highly performant DSP-optimized functional units that scale to large overlays on modern devices and the ability to perform automatic resource-aware kernel replication up to the size of the overlay. We demonstrate place and route times orders of magnitude better than traditional HLS flows, even when running on an embedded processor in the Xilinx Zynq.

Index Terms—Field programmable gate arrays, parallel processing, hardware accelerators

1 Introduction

The need for increased performance while considering energy efficiency has led to the emergence of heterogeneous computing systems that combine traditional CPUs with additional hardware to accelerate computationally intensive tasks [1], [2]. These systems incorporate hardware accelerators like GPUs and FPGAs to scale to large workloads [3]. Cloud computing infrastructure is increasingly heterogeneous, providing access to GPUs and FPGAs for hardware acceleration [4]. Domain specific hardware accelerators, such as for machine learning [2], graph processing [5], or real-time data analytics [6] are also emerging, with FPGAs enabling deployment of such accelerators in a diverse range of application domains.

FPGAs allow modification of computational architectures post-deployment and are commonly used for rapid-prototyping of accelerators to be deployed in heterogeneous computing platforms, allowing developers to customize and update accelerator architectures as algorithms evolve. Examples include Microsoft Catapult [7] that uses FPGAs to accelerate search engine operations and neural network inference [4]. Despite these success stories, FPGA-based hardware accelerators remain difficult to design, maintain, scale, and port to next generation devices using conventional methods. Design productivity remains a major challenge, restricting the effective use of FPGAs to niche disciplines requiring highly skilled hardware design engineers [8].

High level synthesis allows accelerators to be compiled from high level software code in various high level or domain-specific languages. By automating the extraction of parallelism from sequential code, it is possible to simplify the design process significantly [9]. FPGA vendors have recently released OpenCL based tools (Altera OpenCL [10] and Xilinx SDAccel [11]) to bridge the gap between the expressiveness of software programming languages and the parallel capabilities of the FPGA hardware [10].

OpenCL/high level synthesis (HLS) support reduces developer effort, and enables design portability and rapid design space exploration, thus improving productivity, verifiability, and flexibility. However, the FPGA design process remains significantly more complex than that for CPUs or GPUs, with extremely long compilation times for FPGAs (hours rather than seconds) limiting the FPGA to fixed accelerator implementations, similar to using pre-compiled kernel binaries on GPUs. This is a significant obstacle to software developers who are accustomed to rapid compile times, with fast turnaround allowing more efficient testing and tuning of accelerator kernels. It also prevents FPGAs from taking advantage of Just In Time (JIT) compilation and from scaling performance dynamically based on the availability of hardware resources [12]. FPGA design iterations are already extremely slow, and as devices grow in size and

- *Abhishek Kumar Jain is with Xilinx Inc., San Jose, CA 95124 USA. E-mail: abhishek@xilinx.com.*
- *Douglas L. Maskell is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore. E-mail: asdouglas@ntu.edu.sg.*
- *Suhaib A. Fahmy is with the King Abdullah University of Science and Technology, Thuwal 23955, Saudi Arabia. E-mail: suhaib.fahmy@kaust.edu.sa.*

Manuscript received 25 Oct. 2020; revised 26 Aug. 2021; accepted 24 Sept. 2021. Date of publication 30 Sept. 2021; date of current version 28 Oct. 2021.
This work was supported in part by the Ministry of Education (MOE), Singapore, under Grant MOE2017-T2-1-002.
(Corresponding author: Suhaib A. Fahmy.)
Recommended for acceptance by D. Gregg.
Digital Object Identifier no. 10.1109/TPDS.2021.3116859

designs increase in complexity, this problem will worsen, making JIT compilation on FPGA unlikely.

FPGA overlay architectures [13], [14], [15], [16], [17] built around runtime programmable hardware blocks have emerged as one possible solution to this challenge, offering improved design productivity, by virtue of fast compilation, software-like programmability and run-time management, and high-level design abstraction. Runtime programmable hardware blocks may include (soft) processor arrays [18], [19], DMA engines, SIMD/VLIW engines [20], [21], programmable data-flow engines [22], [23], [24], [25], or Network-on-Chip (NoC) nodes [26]. One example is the GRVI Phalanx FPGA overlay [13] which uses the Hoplite NoC IP [26] as a packet-switched network for connecting a large number of clusters of RISC-V soft-processors, local memory, accelerator blocks, and interconnect architecture. Clusters can be customized by using different types and combinations of building blocks (runtime programmable hardware IP) to build domain specific massively parallel accelerator infrastructure.

Programmable data-flow engines [22], [23], [24], [25] are coarse-grained overlay architectures built on top of the FPGA offering a simpler target architecture for the compilation flow, resulting in fast compilation, but at the cost of sometimes significant area and performance overheads. Recent research in this area has demonstrated ways in which more efficient overlays can be built [27], [28]. Being architecture centric, these overlays use the dynamic programmability of DSP blocks to more densely map computations to functional units. These high performance overlays, coupled with high-level design methods, can address both the design and compilation aspects of the design productivity gap.

The proposed methodology benefits from the high level of abstraction afforded by using the OpenCL programming model, while mapping to the overlay significantly reduces compilation and load times. Key characteristics of this work include highly optimized functional units that fully exploit DSP block capabilities while maintaining high throughput and the ability to perform automatic resource-aware performance scaling up to the full size of the overlay.

We envision scenarios where multiple kernels (within an application) are required to be launched one after the other on top of the overlay as shown in Fig. 1. We also assume that the overlay can co-exist with other custom accelerators on the FPGA device; bitstreams of different sized overlays are available in an overlay library and a large overlay can be replaced with a smaller one if custom accelerators require more space on the FPGA device at a given point in time. Since overlay size can vary on the device, we need JIT compilation to dynamically exploit available resources at runtime. Before launching the first kernel of the application, we can JIT compile all the kernels (targetting the available overlay size) during application staging time. This means that only a few Bytes of configuration data need to be managed at run time. Using an overlay means each kernel takes less than a second to compile and application staging time remains an order of magnitude smaller than compilation with traditional flows. The overlay allows kernels to be reconfigured in microseconds (*us*) while traditional partial bitstreams take milliseconds (*ms*) due to their large size.

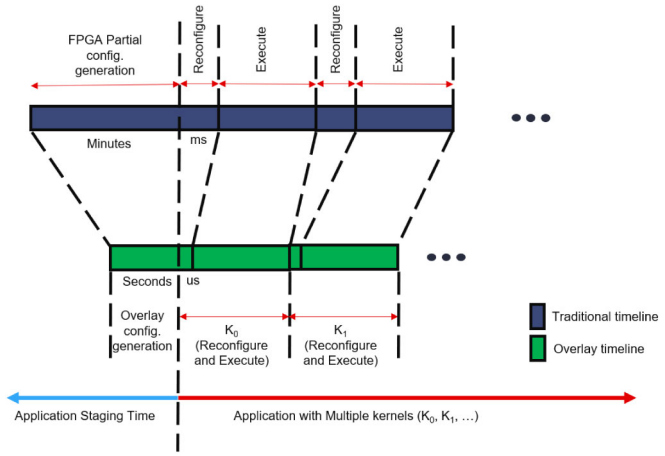


Fig. 1. Timeline showing staging and run time for an application with multiple kernels. Application staging time include the compilation of all the kernels within the application. Application run time include reconfigure and execute for all the kernels.

In this paper, we demonstrate run-time performance scaling using the concept of on-demand resource-aware kernel replication running entirely on a Xilinx Zynq FPGA SoC as a use-case. We extend previous work on DSP block overlays [27], [28] to present a complete architecture and design flow including:

- Demonstration of the performance scalability of these overlay architectures on a variety of FPGA device families.
- A detailed methodology for run-time compilation of dataflow graphs expressed as OpenCL kernels with the ability to automatically scale performance in a resource-aware manner.
- A comparison of place and route (PAR) times for vendor tools targeting traditional fine-grained architectures and the proposed overlay approach for a set of benchmarks, including compilation on an embedded Arm processor in the Xilinx Zynq.

2 Background and Related Work

The increasing design size and fine granularity and heterogeneity of modern FPGA fabrics means compilation times are constantly growing, preventing runtime compilation and tight design iteration. Compilation times for coarse-grained architectures, such as coarse-grained reconfigurable arrays (CGRAs), are significantly lower on account of the higher granularity used. However, CGRAs implemented as ASIC devices [29], [30], [31], [32], [33] have not achieved widespread adoption because functional units (FUs) are often too application specific to be efficient and useful for a wide enough range of applications, while a very general CGRA tends to entail significant area and performance overheads.

A detailed discussion on CGRA-like dataflow accelerators is available in [34]. Such CGRAs implemented on top of commercial FPGAs can be tuned to particular applications, allowing the FUs and interconnect to be adapted to application requirements while still offering rapid compilation. This method of overlaying can be viewed as using an FPGA to emulate a custom CGRA architecture. After refinement, an FPGA-based overlay can be deployed for domain-specific

workload acceleration as demonstrated by Microsoft Brainwave [35]. The benefit of overlays comes at the cost of performance, power, and area overheads over fully custom designs. Reducing these overheads is an active area of research.

Many dataflow CGRA overlay designs proposed in the literature do not consider the low-level FPGA architecture, and as a result, suffer from significant area and performance overheads as shown previously with DySER architecture [36]. FPGA architecture has evolved significantly over time, with the addition of highly optimized hard macros, such as Block RAM and DSP blocks, to enable higher throughput. For example, a key feature of DSP blocks in recent Xilinx families of FPGAs is the ability to dynamically modify functionality on a cycle-by-cycle basis at runtime using dynamic control inputs. This has been used demonstrated in a variety of applications. The iDEA soft processor [19] uses a DSP block to build a full processor execution unit. The Hoplite-DSP NoC [37] embeds NoC router functionality into DSP blocks. Application-specific hardware synthesis [38] exploits the internal structure of the DSP block to significantly improve mapping results compared to Vivado HLS. The DSP Supertile Systolic Array [39] uses DSP blocks to map matrix multiplication and convolution efficiently at near the theoretical maximum frequency achievable.

In our work, we propose exploiting the capabilities of these DSP blocks to design dense high throughput functional units for coarse grained overlays, and present a compiler flow to enable fast mapping of dataflow graphs (expressed as OpenCL kernels) to these overlays. There is significant literature on HLS, including OpenCL to generate application-specific RTL targetting FPGAs [10], [11], [12]. Similarly, several researchers have proposed overlay architectures with methodologies for mapping data flow graphs [22], [23], [24], [25]. JIT compilation of data flow graphs (expressed as OpenCL kernels) was also explored in [40], [41] for overlay architectures. However, performance scaling and resource aware replication of OpenCL kernels based on runtime resource availability has not been explored.

In [40], [41], the authors used overlays with one dedicated functional unit for each OpenCL kernel operation. Five different relatively small overlays (2 floating point and 3 fixed point) were designed, each specialized for a specific set of kernels. A new overlay would be loaded if an unsupported kernel was requested, hence ensuring a large overlay is not underutilized for a small application.

In our work, instead of compiling OpenCL kernels onto relatively small application-specific overlays, we compile replicated instances of kernels onto a large overlay to achieve effective utilization of resources and maximum performance. Major new optimizations in our work include the ability to exploit the OpenCL programming model to perform dynamic performance scaling using kernel replication when additional hardware resources become available. The size of the overlay depends on the available resources after other system logic is mapped. Alternatively fixed funtion accelerators can co-exist with the overlay, with remaining area dedicated to the largest possible overlay. Hence, the overlays we consider can have different sizes and FU types, with this information being exposed by the OpenCL runtime to the compiler, which performs on-demand resource-aware kernel replication to effectively utilize available overlay resources.

In the context of software/hardware systems on hybrid FPGAs, the reconfiguration time required to load a hardware accelerator onto the FPGAis also a significant factor alongside the compilation time [42]. In a cloud or IoT accelerator environment, a large number of different compute kernels need to run on the FPGA. Generation and storage of partial bitstreams for each kernel is one approach to handle that, but overlays allow much faster run-time compilation and reconfiguration, and support for new unknown kernels. Configuration data for the overlay is an order of magnitude smaller than partial bitstreams which would need to be stored on the board before launching the application.

Within a cloud computing setting, where FPGA resources are provisioned to users on-demand, a user can exploit given resources by loading the appropriately sized overlay bitstream and the runtime system can inform the JIT compiler of the available resources. The JIT complier can then perform dynamic kernel replication to efficiently exploit overlay resources. Finally, supporting runtime compilation on an embedded processor allows a lightweight edge accelerator node to compile unknown kernels without the need for a powerful server, enabling the emerging trend of in-network FPGA acceleration [43], [44].

In summary, we require an overlay architecture that achieves high throughput through considered exploitation of FPGA architecture capabilities, that can scale from small to large FPGAs, and a lightweight design flow that can map unknown kernels to this overlay without the need for a powerful server host, exploiting available overlay resources to scale performance beyond a single instance of a kernel.

3 Analysis of Compute Kernels

A key feature of OpenCL that we wish to exploit is the ability to scale the performance of an application by executing multiple replicated copies of the application kernel in hardware. However, mapping multiple instances of a small kernel to an array-based overlay must consider the availability of both compute and I/O resources.

An examination of benchmark kernels for FPGA overlays from the literature [38], [45], [46] shows that most of them are relatively small, limited by the small size of the overlays they targetted. These kernels are taken from benchmark suites like Parboil [47], PolyBench [45], and Polynomial test suite [48]. Table 1 shows the characteristics of these kernels after extracting the data flow graphs (DFGs), including the number of I/O nodes, graph edges, operation nodes, average parallelism, graph depth, and graph width. The graph depth is the critical path length of the graph, while the graph width is the maximum number of nodes that execute concurrently in a timestep, both of which impact the ability to efficiently map a kernel to the overlay. The average parallelism is the ratio of the total number of operations to the graph depth. We observe that for these benchmarks, the average parallelism varies from 1 to 14.4, while the DFGs contain up to 72 operation nodes, 126 edges, and exhibit a depth of up to 13 and a width of up to 36.

An overlay with FUs that properly exploit DSP block capabilities allows us to pack more operations into a single FU by combining simple arithmetic operations into the more complex compound instructions, such as multiply-add, multiply-

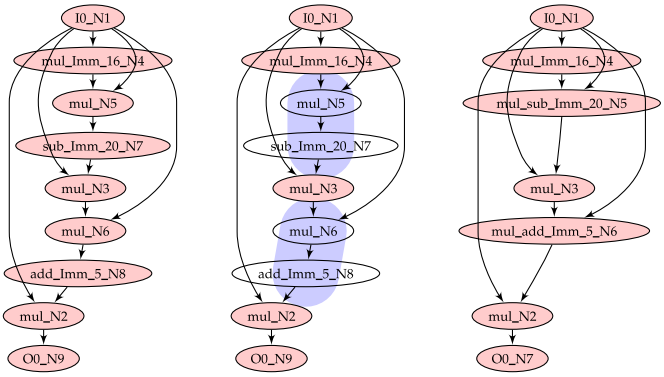
TABLE 1
Characteristics of the Benchmarks Used to Evaluate Our Flow

No.	Benchmark Name	I/O nodes	DFG Characteristics (DSP-Aware) graph edges	op nodes	graph depth	average parallelism	graph width
1.	chebyshev	1/1	12 (10)	7 (5)	7 (5)	1.00 (1.00)	1 (1)
2.	sgfilter	2/1	27 (19)	18 (10)	9 (5)	2.00 (2.00)	4 (3)
3.	mibench	3/1	22 (14)	13 (6)	6 (4)	2.16 (1.50)	3 (3)
4.	qspline	7/1	50 (46)	26 (22)	8 (7)	3.25 (3.14)	7 (7)
5.	poly1	2/1	15 (12)	9 (6)	4 (3)	2.25 (2.00)	4 (4)
6.	poly2	2/1	14 (10)	9 (6)	5 (3)	1.80 (2.00)	3 (3)
7.	poly3	6/1	17 (13)	11 (7)	5 (3)	2.20 (2.30)	4 (4)
8.	poly4	5/1	13 (9)	6 (3)	4 (2)	1.50 (1.50)	2 (2)
9.	poly5	3/1	43 (28)	27 (14)	9 (6)	3.00 (2.30)	6 (6)
10.	poly6	3/1	72 (51)	44 (25)	11 (9)	4.00 (2.77)	11 (10)
11.	poly7	3/1	62 (44)	39 (21)	13 (8)	3.00 (2.62)	10 (7)
12.	poly8	3/1	51 (35)	32 (17)	11 (5)	2.90 (3.40)	8 (8)
13.	fft	6/4	24 (22)	10 (8)	3 (3)	3.33 (2.66)	4 (4)
14.	kmeans	16/1	39 (36)	23 (20)	9 (7)	2.55 (2.85)	8 (8)
15.	mm	16/1	31 (24)	15 (8)	8 (8)	1.88 (1.00)	8 (1)
16.	mri	11/2	24 (20)	11 (7)	6 (5)	1.83 (1.40)	4 (2)
17.	spmv	16/2	30 (24)	14 (8)	4 (4)	3.50 (2.00)	8 (2)
18.	stencil	15/2	30 (24)	14 (8)	5 (3)	2.80 (2.66)	6 (4)
19.	conv	24/8	40 (32)	16 (8)	2 (1)	8.00 (8.00)	8 (8)
20.	radar	10/2	18 (16)	8 (6)	3 (3)	2.66 (2.00)	4 (2)
21.	atax	12/3	123(99)	60(36)	6(6)	12.00(7.20)	27(9)
22.	bicg	15/6	66(54)	30(18)	3(3)	10.00(6.00)	18(6)
23.	trmm	18/9	108(90)	54(36)	4(4)	13.50(9.00)	27(9)
24.	syrk	18/9	126(99)	72(45)	5(4)	14.40(11.25)	36(18)

Numbers in parentheses represent those for the clustered graphs.

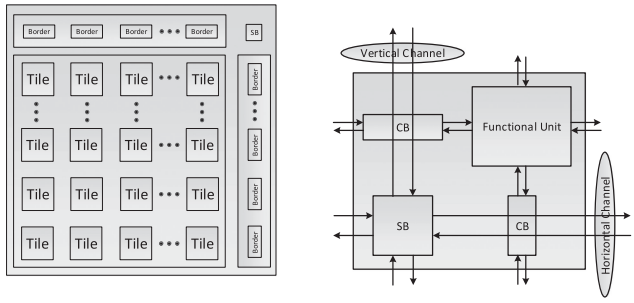
subtract, add-multiply, subtract-multiply, etc., supported by the DSP block [38], as shown in Fig. 2 for the *chebyshev* benchmark. Applying this transformation to all the kernels results in the numbers shown in brackets in Table 1. The *op nodes* column indicates that an overlay with at least 45 DSP blocks is required to support this set of benchmarks, down from the 72 single operation nodes needed if DSP block capabilities were ignored, as is the case in many other overlays, where each FU performs a single operation. This consolidation of arithmetic operations into the FUs also reduces the strain on the routing infrastructure as a number of edges in the original DFG are now absorbed into DSP block internal pipeline paths.

We consider an island-style overlay with a grid of FUs connected by a grid of word-level interconnect, as shown in Fig. 3a. As the size of this overlay increases, the number of I/O interfaces grows linearly while the number of compute tiles grows quadratically. Thus, an $N \times N$ overlay supports N^2 FUs, but as the I/O is determined by the overlay



(a) Input DFG (b) Node-merging (c) Mapped DFG

Fig. 2. DSP48E1 aware DFG generation.



(a) Overlay block diagram. (b) Tile architecture.

Fig. 3. Overlay architecture comprising functional units and routing resources.

perimeter it is proportional to N (e.g., $4N$, $8N$, $12N$ depending on the number of I/O nodes per tile).

It is also possible to make the FUs more dense by including more than one operation nodes. Our DSP block based FUs already include multiple operations, but multiple DSP blocks can also be used in a single FU. In [28] we conducted a thorough evaluation of different ratios of I/O and DSP blocks, and found that four DSP blocks per FU was severely I/O-bound as well as suffering from significant underutilisation of compute nodes. Meanwhile a dual-DSP FU based architecture is well balanced, requiring 100 Slices per DSP block, enabling an overlay with 128 DSP blocks to easily fit onto a Xilinx Zynq 7Z020 device, requiring 12.8K Slices. Further detailed architecture analysis is presented in [28]. In the next section, we describe the detailed architecture of the single-DSP and dual-DSP block based overlay and its associated mapping tool flow.

4 DISO Overlay Architectures

We use the DSP blocks to create a programmable FU in the overlay architecture as it provides an efficient, performant datapath element. Two different FU configurations are considered, resulting in two different overlay architectures. The first, referred to as the DSP Island Style Overlay (DISO) has FUs with a single DSP processing element, while the second, referred to as the Dual DSP Island-Style Overlay (Dual-DISO) has FUs with two DSPs.

The architecture of these overlays have a traditional island-style topology, arranged as a virtual homogeneous two-dimensional array of tiles as shown in Fig. 3a, distributed across the fine grained FPGA fabric. The overlay instantiates the tiles and borders, where each tile consists of virtual word-wide routing resources, comprising one switch box (SB) and two connection boxes (CB), and an FU (as shown in Fig. 3b), and each border has one SB and one CB, forming the boundary at the top and right of the array, as shown in Fig. 3a. This results in an overlay architecture which contains I/O around the periphery which can then be connected to a FIFO or BRAM I/O data port. Hence, an $N \times N$ overlay includes N^2 FUs, $(N+1)^2$ SBs, and $2N^2 + 2N$ CBs.

4.1 Island-Style Interconnect Architecture

Routing resources include switch boxes, connection boxes, and horizontal and vertical channels. Unlike the single-wire tracks in fine-grained FPGA fabrics, the overlay tracks are 16 bits wide to support a 16-bit datapath. Additionally, multiple

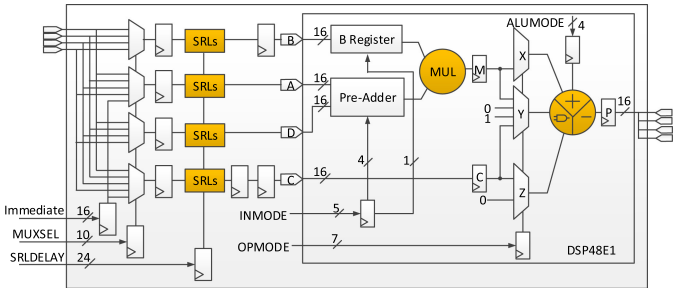


Fig. 4. DISO Functional unit architecture.

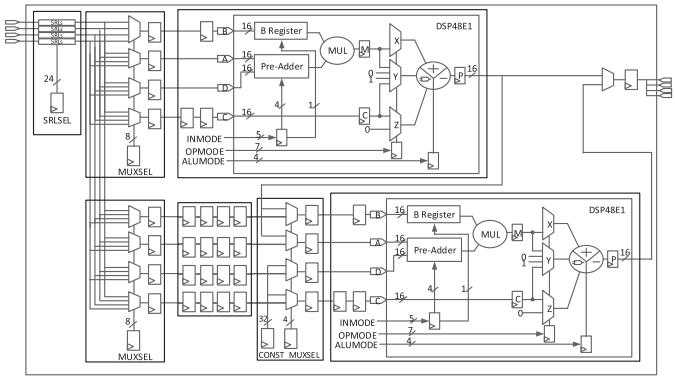


Fig. 5. Dual-DISO functional unit architecture.

tracks can exist in both the horizontal and vertical directions, forming channels within the overlay architecture. The number of tracks in a channel is referred to as the channel width (CW), and as this increases, application routing becomes easier but the area overhead increases. The overlay tile shown in Fig. 3b has two unidirectional tracks in each channel corresponding to a CW = 2.

Switch boxes (SBs) connect tracks to other tracks in intersecting channels, while connection boxes (CBs) connect FU inputs and outputs to routing tracks in adjacent channels. While it is possible to change the flexibility f_s of the SBs depending on the routing requirements of the compute kernels, we choose $f_s = 3$. That is, whenever horizontal and vertical channels intersect, each wire segment can connect to three other wire segments. Multiplexers are used to implement each possible connection in the CBs and SBs, resulting in the routing resources contributing significantly to the overlay area overhead.

4.2 DISO Functional Unit

The DISO FU provides the resources for the mathematical or logical operations of the application and consists of a programmable processing element (PE), MUX based reordering logic and variable length shift register based synchronization logic for balancing pipeline latencies, as shown in Fig. 4. Variable length shift registers are implemented as SLICEM shift register LUTs (SRLs) to achieve maximum performance. The FU has 4 input and 4 output ports logically organized at the 4 cardinal points. The reordering logic is a 4×4 crossbar switch network which allows full connectivity between FU inputs and PE inputs.

Instead of using fine-grained FPGA resources to implement the programmable FU, we use the DSP48E1 primitive as a PE as detailed in [27]. The DSP48E1 primitive has a preadder, a multiplier, an ALU, four input ports for data, and one output port and can be configured to support a number of compound instructions such as multiply-add, subtract-multiply, etc. This functionality is determined by a set of control inputs that are wired to configuration registers. Normally these inputs are fixed during the synthesis stage, but we instantiate the DSP48E1 primitive directly, enabling total control of its configuration, with all three pipeline stages enabled to achieve maximum frequency. These stages, along with the registered outputs of the SRLs result in a total FU latency of 7 clock cycles. Since the DSP48E1 can support three operations, an overlay of size $N \times N$ can support up to $3N^2$ operations. Hence, the peak throughput of an overlay of size $N \times N$ operating at a frequency of F_{max} is equal to $3N^2F_{max}$ ops/s. For CW = 2, an overlay tile consumes 416 LUTs, 390 FFs and 1 DSP block and a border tile consumes 112 LUTs and 76 FFs. More details on the FPGA mapping of DISO overlay is presented in [27].

4.3 Dual-DISO Functional Unit

To improve the compute-to-interconnect resource ratio and hence reduce the number of LUTs per DSP block in the implementation of the overlay, we design a modified version of the DISO FU to include two DSP48E1 blocks [28]. This FU, shown in Fig. 5, has the same 4-input, 4-output structure as DISO, allowing it to connect to any of the four adjacent channels. It also has similar LUT-based SRL32 primitives at each FU input for latency balancing and multiplexer-based reordering logic so that any of the four inputs of the FU can connect to any of the four inputs of a DSP48E1 primitive.

The two DSP blocks are connected in series, with four additional registers added to each input of the second DSP block for pipeline balancing. Lastly, the output from either DSP block can be selected as the FU output. To achieve high frequency, all three pipeline stages in the DSP48E1 primitives are enabled. An $N \times N$ Dual-DISO overlay can theoretically support up to $6N^2$ operations resulting in a peak throughput of $6N^2F_{max}$ ops/s. For CW = 2, an overlay tile consumes 520 LUTs, 625 FFs and 2 DSP blocks while a border tile consumes 112 LUTs and 76 FFs.

We explored how this architecture could scale on different FPGA devices, achieving the results shown in Table 2, all with CW = 2. The Zynq FPGA SoC achieves a respectable frequency of 300 MHz for an 8×8 overlay. The overlay requires 109 configuration bits for the dual-DSP FU and 20 configuration bits for programming the routing network tile. Thus, an 8×8 overlay has a configuration size of 9100 bits (1137 Bytes), and can be configured entirely in 45.5 us, compared to 31.6 ms for the entire Zynq programmable

TABLE 2
Scalability of Dual-DISO Overlay on Different FPGA Architectures

FPGA	Size	DSPs	Freq.	Peak GOPS
Zynq 7Z020	8×8	128	300	115
Virtex 7VX690T	20×20	800	380	912
Alveo XCU280 (3 SLR)	40×40	3200	500	4800
Alveo XCU280 (1 SLR)	22×22	968	650	1887

fabric using the PCAP port, a 1000× improvement in reconfiguration time.

A mid-sized Virtex-7 device (XC7VX690T) can accommodate a 20×20 overlay (using 800 DSP blocks) at a frequency of 380 MHz, offering a peak throughput of 912 GOPS [28]. Being architecture-optimized, the overlay scales well with improvements in the underlying FPGA architecture. On a more modern multi-die SSIT FPGA device (Alveo XCU280) a 40×40 overlay (using 3200 DSP blocks) achieves a frequency of 500 MHz, offering a peak throughput of 4.8 TOPS, despite spanning 3 SLRs that have limited interposer connectivity between them. A 22×22 overlay mapped to a single SLR on the same device achieves a frequency of 650 MHz. For the experiments in Section 6, we consider 8×8 Dual-DISO overlay mapped onto Zynq 7Z020 device.

4.4 Design Challenges and Architectural Optimization

A number of architectural optimizations are performed to improve the performance of the DISO and Dual-DISO overlay architectures.

Frequency and Throughput Optimization: First, both architectures exploit deep pipelining to achieve high frequency, thus maximizing application throughput. To achieve this, we enable all three pipeline stages of the DSP48E1 primitives, add a register at the output of each reordering multiplexer, and register the outputs of the SRLs. As a result, the total latency of the FU is 7 cycles for DISO and 8 or 13 cycles for Dual-DISO. To further increase frequency and eliminate the possibility of combinational loops in the resulting HDL we use a 16-bit register at the output of each MUX in the CB.

Distinguishing PE Inputs: Any of the four inputs of the FU can connect to any of the four inputs of the PE. However, as the input pins of the DSP48E1 block are not logically equivalent, unlike those of FPGA LUTs, we must implement reordering logic for each input pin using a multiplexer. The four outputs of the FU are logically the same single output of the DSP block.

Latency Imbalance at the FU Inputs: The output of an FU can connect to the inputs of multiple other FUs, and multiple FUs may connect to the inputs of a single FU, resulting in different signal propagation latencies due to the deep pipelining in the two overlays. Hence, balancing pipeline latencies at the different FUs is necessary to ensure signal timing is correctly matched. Early work on balancing pipeline latencies was shown in HSRA [49]. Elastic buffers were used in [22] and credit-based flow control was used in [50]. Both of these approaches exhibit stalls when pipeline latencies are imbalanced. In [22], the authors use 64-slot FIFOs to absorb stalls and demonstrate that there would be stalls for less than a 32-slot FIFO. In [23], variable-length shift registers are placed at the input of FUs to avoid stalls. In this design, we use variable-length shift registers implemented using the LUT-based SRL32 primitives. The depth of these variable shift registers is set to introduce the correct amount of delay for each path, and the maximum can be set to 16, 32, or 64 cycles, depending on the flexibility desired. We experimentally determine their optimal maximum depth for our benchmark set. As long as the inputs at any node are not misaligned by more than the depth of the variable shift registers, the place and route algorithms discussed in Section 5 can successfully map, avoiding the need for more complex place and route algorithms supporting pipelined interconnect [51].

5 Compiling Kernels to the Overlay

The design and implementation of the overlay itself requires the conventional hardware design flow using vendor tools, to allow it to achieve maximum frequency and exploit low level architectural features. However, this process is done offline, only once, and so does not impact the mapping of applications on the overlay. Furthermore, the resulting bitstream can be provided to the end user as a package ready for their applications to be mapped onto it. Indeed by ensuring a modular, parameterized design approach, it is possible to build variations of the overlays without considerable re-architecting. Since we propose an overlay built around DSP block optimized FUs, we restrict support to kernels requiring operations supported by the DSP block. DSP blocks inherently support the most important fundamental operations (add, sub, mul, logic, etc.) required in arithmetic computations and hence are capable of supporting a wide range of applications. Our approach is different from some other overlays which require application-specialized FUs, which have the disadvantage of requiring recompilation for new applications though that approach solves the problem of applications that use non-standard operations. For kernels with non-DSP block supported operations, a hybrid approach can be used where some of the FUs can be replaced with partially reconfigurable (PR) regions housing more complex application specific FUs. Exploration of PR regions to replace some FUs is out of the scope of this paper though we plan to work on it in the future.

In OpenCL, parallelism is explicitly specified by the programmer, and the compiler can use system information at runtime to scale the performance of an application by executing multiple replicated copies of the application kernel [12]. That is, while OpenCL supports both online and offline compilation, application kernels in OpenCL are intended to be compiled at run-time [52] so that applications are more portable across platforms. This online compilation of kernels is referred to as just-in-time (JIT) compilation.

The JIT overlay compilation in this paper comprises LLVM Intermediate Representation (IR) generation using Clang for the DFG expressed as an OpenCL kernel, IR optimization using LLVM optimization passes, DFG extraction from the IR, mapping of the DFG nodes to the overlay FUs, FU netlist generation, placement and routing of the FU netlist onto the overlay, latency balancing, and finally, overlay configuration generation. This automated overlay compilation flow is shown in Fig. 6 and is described by demonstrating the step by step process of compiling a simple OpenCL kernel.

5.1 DFG Extraction From a Kernel Description

The flow starts with LLVM extracting a DFG from an OpenCL description. Given a simple OpenCL kernel (as shown in Table 3a), Clang 3.8.0 and the LLVM 3.8.0 disassembler generate the basic block IR, as shown in Table 3b. The *mem2reg* LLVM optimization pass is then used to generate the optimized LLVM IR of the basic block, shown in Table 3c, by removing redundant load and store operations. The Makefile is shown in Table 3d.

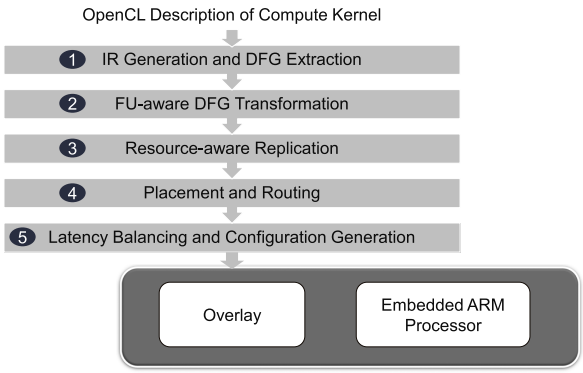


Fig. 6. Automated overlay compilation flow.

A custom IR parser transforms the optimized IR description of the basic block to the DFG description shown in Table 4a. The DFG consists of nodes that represent operations and edges that represent the flow of data between operations. A node executes when all of its inputs are ready, performs its operation, and produces an output, as per the dataflow model of computation. Fig. 7a shows a graphical representation of the DFG described by Table 4a derived from the original OpenCL kernel.

5.2 DFG to FU-Aware DFG Transformation

The DFG description is then parsed and translated into a DSP-aware DFG, as shown in Fig. 7b. This involves merging nodes that can be combined into a single FU, based on the capabilities of the DSP block primitive we have used. For example, we can use a DSP block's multiply-subtract and multiply-add capabilities to collapse the N5–N7 and N6–N8 nodes in Fig. 7a into the N5 and N6 nodes of Fig. 7b, respectively. As a result, the FU aware mapping requires only 5 FUs instead of the 7 that would be required if each node were mapped to a single FU, as is the case for other overlays. We use template matching and node merging algorithms similar to those in [38] and rely on greedy segmentation for clustering. While greedy clustering provides sufficient results in our case, there are opportunities to further improve this by using dynamic programming techniques to achieve optimal clustering, as discussed in DAGON [53].

Next, if compiling for the Dual-DISO overlay, an additional clustering step is applied to support the dual-DSP FU. Two consecutive nodes in the DSP-aware DFG are merged if the fan-in of the resulting node (excluding constants which are instantiated inside the FU) is ≤ 4. In the DFG shown in Fig. 7b, N4 and N5 can be combined together and similarly N3 and N6 can be combined together, resulting in a condensed FU-aware graph as shown in Fig. 7d.

Dual-DSP clustering results in a significant reduction in the number of FUs required compared to an FU with just a single DSP block, also meaning less global routing resources are needed. Our FU-aware DFG transformation currently only supports Xilinx DSP block based FUs, but could be easily modified to support any user-defined FU type.

5.3 Resource-Aware Replication

The FU-aware DFG for the kernel can be optionally replicated a number of times to fit the available resources as exposed by the OpenCL runtime. This enables multiple kernels to run in parallel to better utilize available overlay resources. The replicated DFG is then used to generate the FU netlist. If replication is not required, a replication factor of one is used, meaning a single instance of the kernel is mapped to the overlay.

TABLE 3
Compute Kernel Code Descriptions

(a) OpenCL description of the kernel:

```
__kernel void example_kernel(__global int *A,
    __global int *B)
{
      int idx = get_global_id(0);
      int x = A[idx];
      B[idx] = (x*(x*(16*x*x-20)*x+5)) ;
}
```

(b) Intermediate Representation (IR) of the kernel:

```
%0:
%1 = alloca i32*, align 4
%2 = alloca i32*, align 4
%idx = alloca i32, align 4
%x = alloca i32, align 4
store i32* %A, i32** %1, align 4
store i32* %B, i32** %2, align 4
%3 = call i32 bitcast (i32 (...)*
    @get_global_id to i32 (i32)*)(i32 0)
store i32 %3, i32* %idx, align 4
%4 = load i32* %idx, align 4
%5 = load i32** %1, align 4
%6 = getelementptr inbounds i32* %5, i32 %4
%7 = load i32* %6
store i32 %7, i32* %x, align 4
%8 = load i32* %x, align 4
%9 = load i32* %x, align 4
%10 = load i32* %x, align 4
%11 = mul nsw i32 16, %10
%12 = load i32* %x, align 4
%13 = mul nsw i32 %11, %12
%14 = sub nsw i32 %13, 20
%15 = mul nsw i32 %9, %14
%16 = load i32* %x, align 4
%17 = mul nsw i32 %15, %16
%18 = add nsw i32 %17, 5
%19 = mul nsw i32 %8, %18
%20 = load i32* %idx, align 4
%21 = load i32** %2, align 4
%22 = getelementptr inbounds i32* %21, i32 %20
store i32 %19, i32* %22
ret void
```

(c) Optimized IR of the kernel:

```
%0:
%1 = call i32 bitcast (i32 (...)*
    @get_global_id to i32 (i32)*)(i32 0)
%2 = getelementptr inbounds i32* %A, i32 %1
%3 = load i32* %2
%4 = mul nsw i32 16, %3
%5 = mul nsw i32 %4, %3
%6 = sub nsw i32 %5, 20
%7 = mul nsw i32 %3, %6
%8 = mul nsw i32 %7, %3
%9 = add nsw i32 %8, 5
%10 = mul nsw i32 %3, %9
%11 = getelementptr inbounds i32* %B, i32 %1
store i32 %10, i32* %11
ret void
```

(d) Makefile

```
all:
      clang -emit-llvm -O0 -o IR.bc -c -m32 -x cl test.cl
      llvm-dis IR.bc
      opt -mem2reg < IR.bc > optIR.bc
      llvm-dis optIR.bc
      opt -dot-cfg optIR.bc
      dot -Tpng cfg.example_kernel.dot > optIR.png
```

5.4 Placement and routing of the FU Netlist

We use an adapted version of VPR [54] to map DFG nodes onto the FUs and DFG edges to the overlay routing to connect the mapped FUs. Rather than mapping logic functions to LUTs and single-bit wires to 1-bit channels, we modify VPR to operate at a higher level of abstraction, resulting in fast compilation as the placement and routing problem is much smaller than that of mapping to a fine-grained FPGA. While we rely on simulated annealing for placing DFG nodes onto the coarse-grained overlay, constructive placement strategies have been demonstrated

TABLE 4
Compute Kernel DFG Descriptions

(a) DFG description of the kernel:

```
digraph example_kernel {
N8 [ntype="operation", label="add_Imm_5_N8"];
N9 [ntype="outvar", label="O0_N9"];
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="operation", label="mul_N2"];
N3 [ntype="operation", label="mul_N3"];
N4 [ntype="operation", label="mul_Imm_16_N4"];
N5 [ntype="operation", label="mul_N5"];
N6 [ntype="operation", label="mul_N6"];
N7 [ntype="operation", label="sub_Imm_20_N7"];
N8 -> N2;
N1 -> N5;
N1 -> N6;
N1 -> N2;
N1 -> N3;
N1 -> N4;
N2 -> N9;
N3 -> N6;
N4 -> N5;
N5 -> N7;
N6 -> N8;
N7 -> N3;
}
```

(b) FU-aware DFG description of the kernel:

```
digraph example_kernel {
N7 [ntype="outvar", label="O0_N7"];
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="operation", label="mul_N2"];
N3 [ntype="operation", label="mul_N3"];
N4 [ntype="operation", label="mul_Imm_16_N4"];
N5 [ntype="operation", label="mul_sub_Imm_20_N5"];
N6 [ntype="operation", label="mul_add_Imm_5_N6"];
N1 -> N5;
N1 -> N6;
N1 -> N2;
N1 -> N3;
N1 -> N4;
N2 -> N7;
N3 -> N6;
N4 -> N5;
N5 -> N3;
N6 -> N2;
}
```

TABLE 5
Architecture Description of the Overlay

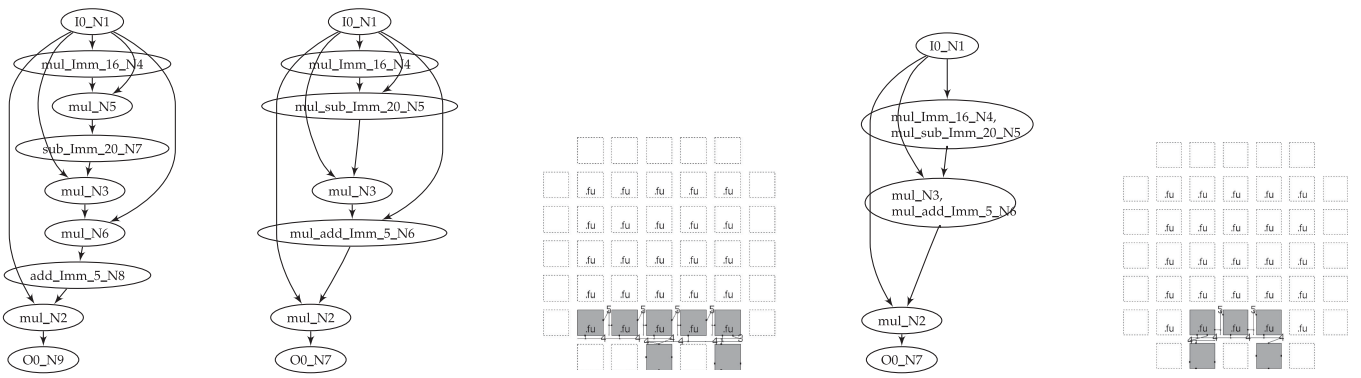
```
<architecture>
        <layout width="5" height="5" />
        <device>
                <sizing />
                <area />
                <chan_width_distr>
                        <io width="1"/>
                        <x distr="uniform" peak="1"/>
                        <y distr="uniform" peak="1"/>
                </chan_width_distr>
                <switch_block type="wilton" fs="3"/>
        </device>
        <switchlist>
                <switch type="mux" name="0" mux_trans_size="10" buf_size="1" />
        </switchlist>
        <segmentlist>
                <segment freq="1" length="1" type="unidir" >
                <mux name="0"/>
                <sb type="pattern">1 1</sb>
                <cb type="pattern">1</cb>
                </segment>
        </segmentlist>
        <typelist>
                <io capacity="1">
                        <fc_in type="frac">1</fc_in>
                        <fc_out type="frac">1</fc_out>
                </io>
                <type name=".fu">
                        <subblocks max_subblocks="1"
                        max_subblock_inputs="4" max_subblock_outputs="4" >
                        </subblocks>
                        <fc_in type="frac">1</fc_in>
                        <fc_out type="frac">1</fc_out>
                        <pinclasses>
                                <class type="in">0 1 2 3 </class>
                                <class type="out">4 5 6 7 </class>
                                <class type="global">8 </class>
                        </pinclasses>
                        <pinlocations>
                                <loc side="left">3 7 8 </loc>
                                <loc side="right">1 5 </loc>
                                <loc side="top">0 4 </loc>
                                <loc side="bottom">2 6 </loc>
                        </pinlocations>
                        <gridlocations>
                                <loc type="fill" />
                        </gridlocations>
                </type>
        </typelist>
</architecture>
```

for fast datapath placement [55] and could be explored in future.

The architecture of the DISO and Dual-DISO overlays consists of a traditional island-style topology, arranged as a virtual homogeneous two-dimensional array of tiles. A VPR 5.0 architecture file is used to describe the overlay architecture, as shown in Table. 5 for a 5×5 overlay. The *layout* field is used to define the width and height of the overlay. The switch box type, flexibility f_s of the switch box, and channel width distribution are specified in the *device* field. The *segmentlist* field specifies that all the segments are unidirectional and span only one block, resulting in one connection box and two switch boxes per FU.

Fig. 7c shows the DFG of Fig. 7b mapped on a 5×5 DISO overlay using the VPR place and route tool. Similarly, Fig. 7e shows the DFG of Fig. 7d mapped on a 5×5 Dual-DISO overlay (with two DSP blocks per FU). At this level of granularity, a netlist can have 100s of edges, making the problem much smaller than that of fine-grained FPGA placement and routing which can involve netlists of millions of edges.

To enable fast resource-aware mapping at runtime, the previous steps (1 and 2 from Fig. 6) can be avoided at runtime by pre-converting the kernels to FU-aware DFGs offline. These FU-aware DFGs can then be used at runtime for



(a) DFG extracted from OpenCL Kernel

(b) FU-aware DFG where FU consists of one DSP block

(c) FU-aware DFG placed and routed on 5x5 DISO overlay

(d) FU-aware DFG where FU consists of two DSP blocks

(e) FU-aware DFG placed and routed on 5x5 Dual-DISO overlay

Fig. 7. Steps of FU aware mapping, placement and routing on DSP block based overlay.

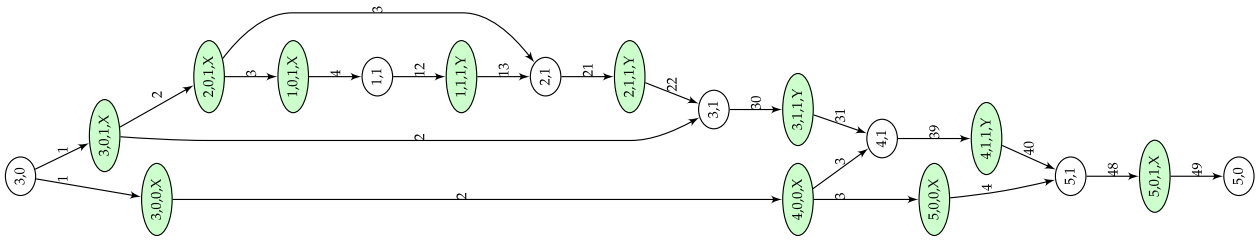


Fig. 8. Overlay resource graph showing timestep labels on edges into nodes.

resource-aware replication (3) and placement and routing (4) on the overlay. We demonstrate this in Section 6, performing runtime placement and routing on the ARM processor of the Xilinx Zynq SoC.

5.5 Latency Balancing

As discussed previously, correct functioning of the mapped compute kernel is ensured only if it is latency balanced, which means that all inputs to an FU arrive at the same cycle. It is clear from Fig. 7b that node inputs can arrive in different cycles. For example, at N6, one input arrives directly from the input node while the other input needs to pass through 3 pipelined DSP blocks associated with nodes N4, N5 and N3. The FUs have delay chains that must be configured to match these latencies. To determine the latency imbalance at each node, we developed a tool that parses VPR output files and generates an overlay resource graph, as shown in Fig. 8 for the mapping of Fig. 7c. An unshaded node in the graph shows the FU information and a shaded node shows the track information used to carry the data from one FU to another. For example, at N5 (placed at (2,1)), the data arrives from N1 (placed at (3,0)) in the 3rd cycle and from N4 (placed at (1,1)) in the 13th cycle. The overlay resource graph is used to generate the configuration of the overlay (including the latency imbalance SRL configuration). This information is loaded onto the overlay at runtime using the OpenCL API.

The tool flow described above supports runtime source-level compilation to the overlay, ending with the accelerator configuration being loaded to the overlay to implement a specific accelerator. However, it is also possible to pre-compile the accelerator configuration, and load a configuration onto the overlay at runtime. However, in this scenario, the exact overlay specification, including size and FU configuration, must be known at compile time, similar to using a kernel accelerator binary on GPU. Finally, it is possible to pre-prepare the FU-aware DFGs which can then be replicated, placed, and routed dynamically at runtime, based on the available resources. The runtime compilation has the added benefit of potential use in multi-tenancy accelerator systems, where available overlay resources can change dynamically at runtime, thereby restricting the choice of how to place and route a new task when requested.

6 Runtime Compilation

To demonstrate the on-demand OpenCL source-level compilation infrastructure, we consider a lightweight heterogeneous computing system based on the Xilinx Zynq XC7Z020, which consists of a dual-core Arm Cortex-A9 processor running at 650 MHz with 512 MB of RAM, and an Artix-7 class FPGA fabric on a single die. The heterogeneous infrastructure, shown in Fig. 9, includes the overlay in the programmable logic region (the FPGA fabric) of the Zynq device. Configuration and overlay management are via an AXI-Lite interface to the ARM processor, while data is streamed from memory via DMA, with BRAM buffers to maintain performance.

The overlay size and FU type are exposed by the OpenCL runtime to the compiler so that it can dynamically replicate a suitable number of kernel copies to fully utilize available overlay resources. We deliberately do not consider a fixed overlay size as other system design requirements may consume significant FPGA fabric resources, requiring a smaller overlay. In that case, the overlay size can be reduced accordingly without requiring any change to the OpenCL source code. For example, in the case where other logic consumes significant resources, leaving only minimal room for a small 2×2 overlay, this information can be exposed by the OpenCL runtime to the compiler which can then choose to map only a single copy of a kernel, as shown in Fig. 10a.

In the case where the other logic is minimal a large 8×8 overlay can be implemented and the compiler can choose to map multiple copies of the kernel, in this case 16 copies of the *Chebyshev* benchmark as shown in Fig. 10g, limited only by the available I/O. Figs. 10b, 10c, 10d, 10e, and 10f show cases in between these two extremes. This dynamic scaling also has a benefit in multi-tenancy applications where a large overlay can be used to support multiple accelerator requests, allocating resources based on the occupancy of the overlay by other users.

Clearly, different sized overlays have different performance characteristics. Fig. 11 shows the effect of performance scaling using kernel replication on overlays having different sizes. The top curve shows the throughput in GOPS for replicated copies of the *Chebyshev* kernel mapped

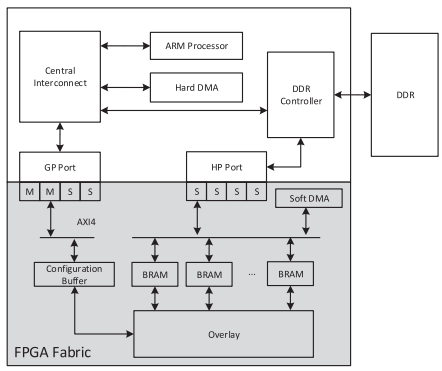


Fig. 9. Overlay infrastructure, implemented on the Xilinx Zynq, with overlay size and FU type exposed by OpenCL runtime.

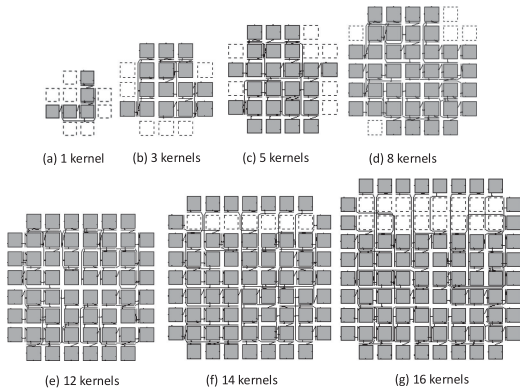


Fig. 10. Performance scaling by the compiler using overlay size information provided by the OpenCL runtime.

to a Dual-DISO overlay. A throughput of approximately 35 GOPS can be achieved using an 8×8 overlay by replicating 16 kernel instances while a single instance of the kernel operates at 2.45 GOPS (as shown by horizontal dashed line). The bottom curve shows the throughput in GOPS for replicated copies of the *Chebyshev* kernel mapped to a DISO overlay. A throughput of ≈ 28 GOPS can be achieved using an 8×8 overlay by replicating 12 kernel instances while a single instance of the kernel achieves a performance of 2.66 GOPS. The main benefit of resource aware replication of kernels at runtime is the possibility of exploiting higher performance on a larger FPGA fabric without changing the application source code.

The time taken to map these different accelerator kernels to the FPGA is important for a runtime compilation flow, as the traditional FPGA flow is too slow to make on-demand kernel replication, exploited with great success by GPUs, feasible. To demonstrate the difference in mapping times for our overlay we compile benchmarks (from Table 1) described in OpenCL onto the Dual DISO overlay and measure the placement and routing time. This is then compared to that of a traditional FPGA implementation. Since overlay size is known only when kernels are launched, we must perform replication and place and route based on the overlay resource availability. Since the FU netlist for a kernel needs only be prepared once offline (using steps 1 and 2 from Fig. 6), we focus on the time to place and route replicated netlists onto the overlay. We consider three different scenarios, with the results shown in Table 6.

Here, the number of replicated copies of the benchmark is shown in brackets after the benchmark name. The first column shows the placement and routing time using Vivado

TABLE 6
Comparison of Place and Route Times for Zynq XC7Z020 (in Seconds)

Target architecture	FPGA Fabric	Overlay	Overlay
Place and Route machine	**x86**	**x86**	**Zynq**
chebyshev(16)	240	0.12	0.90
sgfilter(10)	420	0.16	1.20
mibench(7)	240	0.14	0.80
qspline(3)	240	0.11	0.70
poly1(9)	255	0.10	0.66
poly2(10)	270	0.12	0.97
poly3(3)	240	0.07	0.32
poly4(5)	240	0.08	0.44
poly5(4)	300	0.10	0.66
poly6(2)	240	0.09	0.55
poly7(4)	232	0.14	1.00
poly8(6)	270	0.12	0.90
fft(3)	232	0.09	0.55
kmeans(1)	240	0.06	0.30
mm(1)	240	0.06	0.30
mri(2)	270	0.07	0.32
spmv(1)	255	0.06	0.30
stencil(1)	240	0.06	0.30
conv(1)	255	0.07	0.32
radar(2)	270	0.07	0.32
atax(1)	275	0.09	0.55
bicg(1)	282	0.06	0.30
trmm(1)	268	0.08	0.44
syrk(1)	270	0.11	0.90

2014.2 running on the HP Z420 workstation (x86 machine with an Intel Xeon E5-1650 v2 CPU running at 3.5 GHz with 16 GB of RAM) targeting the Zynq FPGA fabric. This shows the place and route time for HLS generated kernels in the context of generating partial bitstreams. The second column (*Overlay x86*) shows the placement and routing time for the proposed overlay mapping tool on same HP workstation, and represents the situation where an FPGA accelerator card is installed into a workstation. The third column (*Overlay Zynq*) shows the placement and routing time for the proposed overlay mapping tool on the Xilinx Pynq board consisting of a Xilinx Zynq XC7Z020 device with a dual-core Arm Cortex-A9 CPU, running at 650 MHz with 512 MB of RAM. Ubuntu 15.04 runs on the dual-core Arm with the Portable Computing Language (pocl) infrastructure [56] installed.

For the set of benchmarks tested, PAR takes on average 0.1 s, 0.58 s, and 262 s for the overlay on a host workstation, overlay on the Zynq, and for Vivado on the workstation, respectively. This represents a speed-up on the workstation of approximately 2600×, when compiling to the overlay

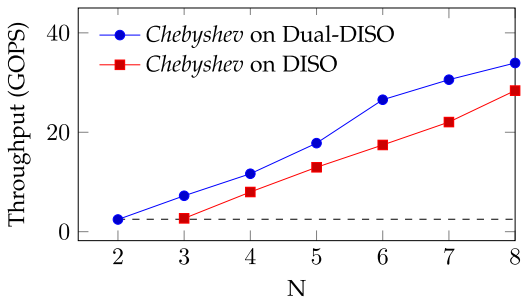


Fig. 11. Performance scaling by replicating the *Chebyshev* kernel on an N×N Dual-DISO and DISO overlay.

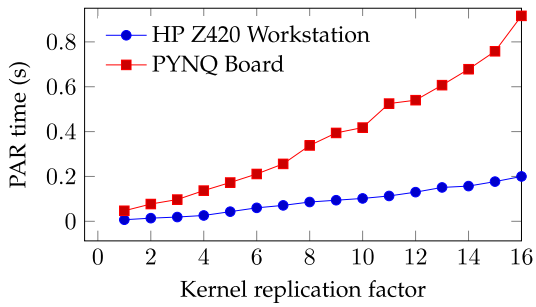


Fig. 12. Effect of *Chebyshev* kernel replication factor on PAR time.

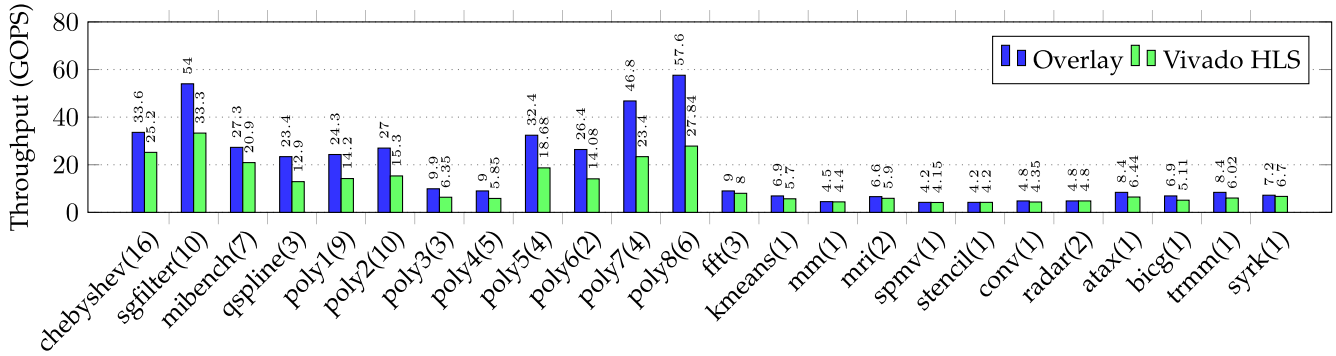


Fig. 13. Performance comparisons of the Dual-DISO overlay and Vivado HLS implementations (number of kernel replications in parentheses).

versus using Vivado. When using the Zynq Arm processor, the placement and routing process targeting the overlay is still over 450× faster than the HLS flow on the workstation. In Fig. 12, the *Chebyshev* kernel is replicated multiple times to show the effect of kernel replication on PAR time. It also shows that the place and route time increases with array size. For larger platforms, such as a Xilinx Alveo U280 with a 40x40 overlay, it would take longer to map kernel copies to the overlay.

As a further comparison, we compare the performance, in terms of throughput (in GOPS, calculated as the product of the DFG compute nodes and the implementation operating frequency), of a conventional hardware implementation with that of the overlay. To achieve this, we generate RTL using Vivado HLS as well as the vendor independent mapping to the overlay using our automated tool flow for the benchmark set. Fig. 13 shows a performance comparison of the overlay implementations (left bar) and Vivado HLS implementations (right bar) in terms of throughput. Many benchmarks can have multiple instances kernel instances mapped to the overlay as shown in brackets. For example, an overlay throughput of 57.6 GOPS is achieved by instantiating 6 instances of the *poly8* benchmark (32 operations per kernel) on the overlay running at 300 MHz. When Vivado is used to map same number of instances of the poly8 benchmark to the FPGA, implementation frequency drops significantly (145 MHz). All the benchmarks run on the overlay at 300 MHz since this is achieved and fixed when mapping the overlay to the FPGA. As we increase the number of kernel instances on top of the overlay, we continue to observe linear speedup with the number of kernels that can fit. On the other hand, Vivado struggles with timing closure as we increase the number of kernel instances.

Our overlay achieves an average throughput improvement of 40% compared to HLS due to the pipelined interconnect structure and fully pipelined DSP block based compute architecture with predictable performance scaling. Fast compilation comes at the cost of interconnect area overhead because LUTs and FFs are used to implement programmable interconnect (word-wide switch boxes and connection boxes).

Next, we evaluate the area overhead of proposed overlay compared to direct FPGA implementation of kernels. The implementations we are comparing use different hardware resource types, making them difficult to compare directly. Instead we normalize the hardware resource utilization using a single "equivalent slices" (e-Slices) metric, where we assume that 1 DSP block is equivalent to 60 slices based on the ratio of slices to DSP blocks on the Zynq 7020 device (which is approximately 60) [57]. Note that this ratio changes from device to device, even within the same family.

For each benchmark in Table 1, we obtain the area in e-Slices, the throughput in MOPS and throughput per unit area in MOPS/eSlice. We observe that the average throughput per unit area for the HLS implementation of the benchmark set is ≈10 MOPS/eSlice. In comparison, the Dual-DISO overlay achieves 2.2 MOPS/eSlice, which is around one fifth of the HLS implementations. However, this 4.5× hardware performance penalty is traded off against a 2600× improvement in place and route time and 1000× improvement in kernel context switch time, making the overlay concept a promising possibility for general purpose on-demand application acceleration. Area overheads associated with overlay interconnect architecture can be reduced by introducing coarse-grained routing primitives in the FPGA fabric such as hardened multi-bit switch boxes and connection boxes [58], [59], [60].

While it is true that an HLS implementation could replicate more copies of a kernel in the same area, we make this comparison to quantify the overhead of the flexibilty afforded by overlays, while also bearing in mind that larger replication factors in HLS implementations would suffer further reductions in frequency, that bring the gap down. As previously demonstrated, the architecture centric approach of the DISO overlay offers a 3.3× improvement in throughput per unit area over a non architecture centric overlay [36], hence narrowing the gap to HLS implementations significantly.

7 Summary and Conclusions

We have presented a coarse grained FPGA accelerator overlay and just-in-time compilation flow for OpenCL kernels. An overlay architecture optimised around the capabilities of modern FPGA DSP blocks significantly improves performance and efficiency compared to previous overlays. The compilation methodology benefits from the high level of abstraction afforded by the OpenCL programming model, while mapping to the overlay offers fast compilation in the order of milliseconds, even on an embedded processor. We demonstrated an end-to-end compilation flow with resource aware mapping of kernels to the overlay. Using a typical workstation, the overlay place and route is ≈2600 times faster than the FPGA place and route using Xilinx Vivado. Furthermore, the overlay can be reconfigured in less than

50 μs using the OpenCL API. Using the pocl infrastructure on a Xilinx Zynq board we demonstrated execution of OpenCL applications, and place and route onto the overlay running entirely on the Zynq in under a second for most applications. We intend to explore the integration of this overlay with partially reconfigurable functional units, and extension of the runtime system to a fully virtualised multi-tenancy accelerator system in future work.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar./Apr. 2008.

[2] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[3] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?," in *Proc. Int. Symp. Microarchit.*, 2010, pp. 225–236.

[4] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *Proc. Int. Symp. Microarchit.*, 2016, pp. 1–13.

[5] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *Proc. High Perform. Extreme Comput. Conf.*, 2016, pp. 1–7.

[6] K. Ganesan *et al.*, "Accelerating Java streams with a data analytics hardware accelerator," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2017, pp. 157–158.

[7] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *Commun. ACM*, vol. 59, no. 11, pp. 114–122, 2016.

[8] G. Stitt, "Are field-programmable gate arrays ready for the mainstream?," *IEEE Micro*, vol. 31, no. 6, pp. 58–63, Nov./Dec. 2011.

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

[10] D. P. Singh, T. S. Czajkowski, and A. Ling, "Harnessing the power of FPGAs using Altera's OpenCL compiler," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2013, pp. 5–6.

[11] G. Guidi, E. Reggiani, L. Di Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, "On how to improve FPGA-based systems design productivity via SDAccel," in *Proc. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 247–252.

[12] S. Gao and J. Chritz, "Characterization of OpenCL on a scalable FPGA architecture," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2014, pp. 1–6.

[13] J. Gray, "GRVI Phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *Proc. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2016, pp. 17–20.

[14] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on FPGAs?," in *Proc. Int. Conf. Pervasive Intell. Comput., 2nd Int. Conf. Big Data Intell. Comput., Cyber Sci. Technol. Congr.*, 2016, pp. 586–593.

[15] C. E. Laforest and J. H. Anderson, "Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, 2017, Art. no. 19.

[16] H. K.-H. So and C. Liu, "FPGA overlays," in *FPGAs for Software Programmers*. Cham, Switzerland: Springer, 2016, pp. 285–305.

[17] C. Y. Lin, Z. Jiang, C. Fu, H. K.-H. So, and H. Yang, "FPGA high-level synthesis versus overlay: Comparisons on computation kernels," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 92–97, 2017.

[18] C. E. LaForest and J. G. Steffan, "Octavo: An FPGA-centric processor family," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2012, pp. 219–228.

[19] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, "The iDEA DSP block-based soft processor for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, pp. 1–23, 2014.

[20] A. Severance and G. G. F. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *Proc. Int. Conf. Hardware/Softw. Codesign Syst. Synth.*, 2013, pp. 1–10.

[21] J. Hoozemans, S. Wong, and Z. Al-Ars, "Using VLIW softcore processors for image processing applications," in *Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Model., Simul.*, 2015, pp. 315–318.

[22] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2013, pp. 1–8.

[23] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant FPGA compilation," *IEEE Embedded Syst. Lett.*, vol. 3, no. 3, pp. 81–84, Sep. 2011.

[24] J. Benson, R. Cofell, C. Frericks, C. H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2012, pp. 1–12.

[25] J. Coole and G. Stitt, "Adjustable-cost overlays for runtime compilation," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 2015, pp. 21–24.

[26] N. Kapre and J. Gray, "Hoplite: Building austere overlay NoCs for FPGAs," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2015, pp. 1–8.

[27] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient Overlay architecture based on DSP blocks," in *Proc. IEEE 23rd Annu. Int. Symp. FPGAs Custom Comput. Mach.*, 2015, pp. 25–28.

[28] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proc. Design, Automat. Test Europe Conf. Exhib.*, 2016, pp. 1628–1633.

[29] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proc. IEEE Symp. Field-Programmable Custom Comput. Mach.*, Apr. 1996, pp. 157–166.

[30] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD – Reconfigurable pipelined datapath," in *Proc. 6th Int. Workshop Field-Programmable Logic Smart Appl., New Paradigms Compilers*, 1996, pp. 126–135.

[31] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May. 2000.

[32] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Jan. 2003, pp. 61–70.

[33] V. Govindaraju *et al.*, "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep./Oct. 2012.

[34] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 416–429.

[35] E. Chung *et al.*, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, Mar./Apr. 2018.

[36] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, "Adapting the DySER architecture with DSP blocks as an overlay for the Xilinx Zynq," *SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 28–33, 2015.

[37] K. H. B. Chethan and N. Kapre, "Hoplite-DSP: Harnessing the Xilinx DSP48 multiplexers to efficiently support NoCs on FPGAs," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2016, pp. 1–10.

[38] B. Ronak and S. A. Fahmy, "Mapping for maximum performance on FPGA DSP blocks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 4, pp. 573–585, Apr. 2016.

[39] E. Wu, X. Zhang, D. Berman, and I. Cho, "A high-throughput reconfigurable processing array for neural networks," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2017, pp. 1–4.

[40] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts," *IEEE Micro*, vol. 34, no. 1, pp. 42–53, Jan./Feb. 2014.

[41] J. Coole and G. Stitt, "OpenCL high-level synthesis for mainstream FPGA acceleration," in *Proc. Workshop SoCs, Heterogeneous Architectures Workloads*, 2014.

[42] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *Proc. IEEE 7th Int. Conf. Cloud Comput. Technol. Sci.*, 2015, pp. 430–435.

[43] R. A. Cooke and S. A. Fahmy, "A model for distributed in-network and near-edge computing with heterogeneous hardware," *Future Gener. Comput. Syst.*, vol. 105, pp. 395–409, 2020.

[44] R. A. Cooke and S. A. Fahmy, "Quantifying the latency benefits of near-edge and in-network FPGA acceleration," in *Proc. Int. Workshop Edge Syst., Analytics Netw.*, 2020, pp. 7–12.
[45] L. N. Pouchet, "Polybench: The polyhedral benchmark suite (2011), version 3.2," 2011. [Online]. Available: https://www.cs.colostate.edu/~pouchet/software/polybench/
[46] C. H. Hoy *et al.*, "Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 203–214.
[47] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center Reliable High-Perform. Comput.*, Univ. Illinois Urbana-Champaign, Champaign, IL, USA, Tech. Rep. IMPACT-12–01, 2012.
[48] D. Bini and B. Mourrain, "Polynomial test suite," *SAGA Project, INRIA Sophia Antipolis, France*, 1998. [Online]. Available: http://www-sop.inria.fr/saga/POL/
[49] W. Tsu *et al.*, "HSRA: High-speed, hierarchical synchronous reconfigurable array," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 1999, pp. 125–134.
[50] Z. Marzec, "Detailed performance evaluation of data-parallel workloads on the DySER prototype system," Graduate Project, University of Wisconsin-Madison, Madison, WI, USA, 2012. [Online]. Available: http://research.cs.wisc.edu/vertical/papers/thesis/marzec-report.pdf
[51] K. Eguro and S. Hauck, "Armada: Timing-driven pipeline-aware routing for FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2006, pp. 169–178.
[52] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, May/Jun. 2010.
[53] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. Des. Automat. Conf.*, 1987, pp. 341–347.
[54] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 1997, pp. 213–222.
[55] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast module mapping and placement for datapaths in FPGAs," in *Proc. 6th Int. Symp. Field Programmable Gate Arrays*, 1998, pp. 123–132.
[56] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *Int. J. Parallel Program.*, vol. 43, no. 5, pp. 752–785, 2015.
[57] X. Inc, "Zynq-7000 SoC data sheet: Overview." [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
[58] A. Ye and J. Rose, "Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2005, pp. 3–13.
[59] A. G. Ye and J. Rose, "Using multi-bit logic blocks and automated packing to improve field-programmable gate array density for implementing datapath circuits," in *Proc. Int. Conf. Field Programmable Technol.*, 2004, pp. 129–136.
[60] M. S. Abdelfattah, A. Bitar, and V. Betz, "Take the highway: Design for embedded NoCs on FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2015, pp. 98–107.

Abhishek Kumar Jain (Member, IEEE) received the bachelor's degree in electronics and communication engineering from the Indian Institute of Information Technology, Allahabad, India, in 2012 and the PhD degree in computer engineering from Nanyang Technological University, Singapore, in 2016. From 2017 to 2018, he was a postdoctoral research staff member with the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA. Since 2018, he has been a senior architect with Xilinx, San Jose, CA, USA. His research interests include computer architecture, reconfigurable computing, multiprocessor system-on-chip, high level synthesis, high-performance accelerators, and domain-specific FPGA overlay architectures.

Douglas L. Maskell (Senior Member, IEEE) received the BE (Hons.), MEngSc, and PhD degrees in electronic and computer engineering from James Cook University, Douglas, QLD, Australia, in 1980, 1984, and 1996, respectively. He is currently an associate professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He has authored or coauthored more than 160 research papers in international journals and conferences. His research interests include embedded systems, reconfigurable computing, intelligent systems, and algorithm acceleration for hybrid high-performance computing systems.

Suhaib A. Fahmy (Senior Member, IEEE) received the MEng degree in information systems engineering and the PhD degree in electrical and electronic engineering from Imperial College London, U.K., in 2003 and 2007, respectively. From 2007 to 2009, he was a research fellow with Trinity College Dublin and a visiting research engineer with Xilinx Research Labs, Dublin. From 2009 to 2015, he was an assistant professor with Nanyang Technological University, Singapore. From 2015 to 2020, he was an associate professor and then a reader of computer engineering with the University of Warwick, U.K. Since 2020 he has been an associate professor of computer science with the King Abdullah University of Science and Technology, Saudi Arabia. His research interests include reconfigurable computing, high-level system design, and acceleration in a networked context. He was the recipient of Best Paper Award at the IEEE Conference on Field Programmable Technology in 2012, IBM Faculty Award in 2013 and 2017, and ACM TODAES Best Paper Award in 2019. He is a senior member of the ACM, and chartered engineer and member of the IET.