PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development

Yuanlong Xiao
University of Pennsylvania
Philadelphia, PA, USA
ylxiao@seas.upenn.edu

Eric Micallef
University of Pennsylvania
Philadelphia, PA, USA
micallef@seas.upenn.edu

Andrew Butt
University of Pennsylvania
Philadelphia, PA, USA
butta@seas.upenn.edu

Matthew Hofmann
University of Pennsylvania
Philadelphia, PA, USA
matth2k@seas.upenn.edu

Marc Alston
University of Pennsylvania
Philadelphia, PA, USA
maa24@seas.upenn.edu

Matthew Goldsmith
University of Pennsylvania
Philadelphia, PA, USA
mwgold@seas.upenn.edu

Andrew Merczynski-Hait
University of Pennsylvania
Philadelphia, PA, USA
andrewme@seas.upenn.edu

André DeHon
University of Pennsylvania
Philadelphia, PA, USA
andre@acm.org

ABSTRACT

FPGA-based accelerators are demonstrating significant absolute performance and energy efficiency compared with general-purpose CPUs. While FPGA computations can now be described in standard, programming languages, like C, development for FPGAs accelerators remains tedious and inaccessible to modern software engineers. Slow compiles (potentially taking tens of hours) inhibit the rapid, incremental refinement of designs that is the hallmark of modern software engineering. To address this issue, we introduce separate compilation and linkage into the FPGA design flow, providing faster design turns more familiar to software development. To realize this flow, we provide abstractions, compiler options, and compiler flow that allow the same C source code to be compiled to processor cores in seconds and to FPGA regions in minutes, providing the missing -O0 and -O1 options familiar in software development. This raises the FPGA programming level and standardizes the programming experience, bringing FPGA-based accelerators into a more familiar software platform ecosystem for software engineers.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**; • **Software and its engineering → Development frameworks and environments**.

KEYWORDS

FPGA, Compilation, Partial Reconfiguration, Data Center, DFX

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507740

ACM Reference Format:
Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. 2022. PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507740

1 INTRODUCTION

At the twilight of Moore's Law, reconfigurable-based accelerators have emerged capable of delivering energy, cost, and performance benefits compared with CPU and ASICs platforms [1, 10, 15, 19, 20, 24, 51, 58, 63]. The boom of data-center-based FPGA deployments [17, 45, 67] further provides more opportunities to diverse FPGA-based applications on image processing [12, 31], machine learning [13, 19], and data analysis [8, 13]. Xilinx now provides Alveo Data-center cards that communicate with a Linux host via PCIe similar to GPUs.

Despite the enhanced programmability of FPGAs, reducing binary compilation time is a key to enabling wide FPGA utilization. Compile time determines the edit-compile-debug cycles that directly impacts developer efficiency. However, FPGA compilation is slow. For modern, data-center-scale FPGA accelerator cards, compile time can run into tens of hours. This is incompatible with modern software engineering approaches that emphasize rapid, incremental refinement of applications [2, 6]. In many ways, programming FPGA computations today is like stepping back to the batch processing era of the 1960s.

FPGA compilation is slow because it is a monolithic compile of the entire accelerator in one piece using super-linear algorithms to find high-quality solutions to hard, spatial optimization problems. While a modern software compiler has distinct optimization levels

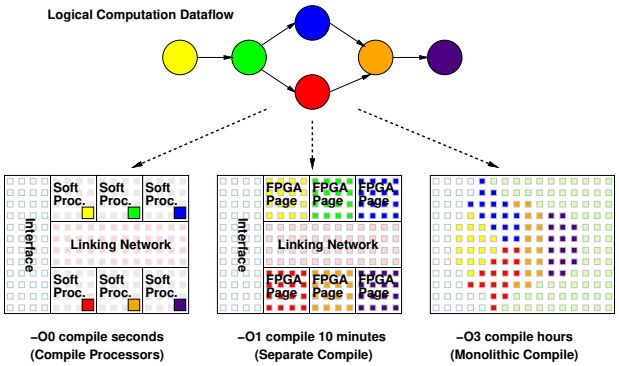


Figure 1: Fast and Separate Compilation Mapping Strategies

(`-O0` to `-O3`) to trade off performance and compile time, FPGA compilers have limited strategies. This greatly degrades development and debugging efficiency in the initial development stage, when optimized implementations are not needed. On almost all aspects, FPGA compilation is different from how we design compilers for processors and how we compile software (Sec. 2.1–2.2).

We argue that FPGA accelerator development can be more approachable for modern software developers by adapting some of the good ideas from processor compilation including supporting (1) separate compilation and linkage for modules with (2) distinct fast and quality compile options for (3) single-source C code that can run on both FPGAs and processor targets (See Fig. 1). This supports faster edit-compile-debug turns (seconds and minutes) and always leaves the developer with a running application that can be tested for validity and measured for performance. We present our framework, **PLD** (**P**artition **L**inking and Loa**D**ing on **P**rogrammable **L**ogic **D**evices).

A streaming dataflow abstraction [9, 16, 22, 29, 30, 47, 55] between concurrent operators and an on-chip network are utilized to support this separate compilation and linking. Separate compilation enables incremental compiles of only the modules that change as well as parallel compilation of independent modules. The on-chip network can link together the separately compiled modules without long FPGA compiles while supporting the dataflow communication abstraction. The use of a standard, dataflow streaming interface to connect independent modules allows tasks to change their implementation and location without programmer intervention and without impacting functionality.

Compiler options that trade off compile time with performance (e.g. `-O0`, `-O1`, `-O3` in traditional compiler) can further support rapid recompilation and refinement (See Fig. 1). Separate compilation of small modules can occur in minutes instead of hours, providing an `-O1 option`. Compilation of the same C source code to processors, perhaps soft-cores in the FPGA fabric, can occur in seconds, providing an `-O0` option.

We make the following contributions:

- Articulate an abstraction for dataflow operators that can migrate between FPGA accelerators and processors (Sec. 3)
- Show how to reduce FPGA compile time to minutes using separate compilation of operators onto sub-regions of the FPGA and how to link the separately compiled modules back together in seconds (Sec. 4).
- Show how the same source can be compiled to processors with compatible streaming interfaces (Sec. 5), allowing the same module C source code to be compiled in seconds and integrated into the computation.
- Provide firmware and an automated tool flow that supports this discipline (Sec. 6), providing a familiar experience for software developers.
- Characterize the compilation-time and performance tradeoff introduced by these disciplines on the Rosetta Benchmark set [74] (Sec. 7).
- Provide an open-source release of our PLD framework (https://github.com/icgrp/pld2022).

2 BACKGROUND

2.1 Processor Compilation

Compilation for processors is typically fast. Historically, compiler writers have actively eschewed non-linear algorithms, to keep compilation fast. Furthermore, uniprocessor compilation has not had to deal with complicated spatial concerns, giving them a simpler problem than the placement-and-routing needed for spatially arranging computations on FPGAs. Additionally, processor compilation supports separate compilation with linking so that it is only necessary to recompile the parts of the code that change. Linking connects the separately compiled components of a program. Finally, processor compilers support varying levels of effort from no optimization (typically optimization level 0 or `-O0`) to level 3 (`-O3`) where more aggressive, and potentially longer running, optimization algorithms are invoked.

2.2 FPGA Compilation

FPGA compilation is slow. Today's data-center-scale FPGAs take hours to map even for small designs (Tab. 2) and can take 10+ hours for large designs [5, 41, 57]. Typical FPGA compilation maps the entire design at once. This is good for quality, allowing the compiler to perform cross-module optimizations and use cross-module constraints to drive efficient physical placements. However, this means the mapping must deal with a large problem—millions of individual elements on modern data-center-scale FPGAs—for any change. Individual elements, most of which are gates producing a single bit, must be placed and routed. The placement and routing problems are all NP-hard problems, typically solved by heuristics, and the good heuristics in use are super-linear, with no guarantees on runtime or quality.

2.3 Partial Reconfiguration and Dynamic Function Exchange

Modern FPGAs support partial reconfiguration, where a portion of the FPGA is reconfigured while the rest of the FPGA continues operation [25, 68]. This speeds bitstream loading since the size of the bitstream, and hence time to load the bitstream, is proportional to the amount of FPGA logic being reconfigured. While a full bitstream may be hundreds of megabytes, a partial bitstream can be tens of kilobytes. Xilinx now calls this Dynamic Function eXchange

(DFX) [68]. This provides a mechanism to decompose the FPGA into partial reconfiguration (PR) regions that are separately compiled and loaded. Vendors tool flow supports the definition of interfaces between PR/DFX regions and the designation of logic to be mapped to a DFX region, but leave it to the FPGA application designer to decide how to use them. Directly dealing with DFX region design and mapping is tedious and demands significant, low-level FPGA expertise that precludes direct use by most software engineers. We build on these capabilities to define FPGA "pages" for separate compilation (Sec. 4) and abstract the low-level details away from software engineers with our toolflow (Sec. 6).

2.4 Vitis OpenCL

Xilinx Vitis provides a standard, OpenCL interface for communicating with the FPGA accelerators and launching computational kernels on the accelerator [66]. Vitis_HLS allows the kernel to be specified in C and allows separate C kernels to be linked together. Vitis supports separate HLS compilation from C to a Register Transfer Level (RTL), like Verilog. However, Vitis does not support a notion of separate compilation at the place-and-route, implementation level and hence does not support linking of separately compiled functions and kernels post place-and-route. As we will see (Tab. 2), place-and-route is often the most time-consuming part of mapping from C to FPGA logic. Our solutions are compatible with the Vitis OpenCL interfaces and HLS, providing separate compilation and linkage as an alternative to the monolithic Vitis implementation flow.

2.5 Data-Center FPGAs

Data-Center cards have high capacity FPGAs and local memory and are designed to operate on the PCI bus [67]. PCI interface functionality exists in the FPGA configurable logic. Since the PCI interface is exposed to the server and Linux OS, it is necessary to keep the PCI interface stable even when reconfiguring the FPGA logic to support a distinct application or, in our case, to support an incrementally compiled updates to the logic. As such, data-center cards are designed with interface "Shells" that remain stable across applications and use partial reconfiguration (DFX) (Sec. 2.3) to load the application logic.

The largest data-center FPGAs from Xilinx use silicon interposer technology where multiple die are stacked on top of a silicon interconnect substrate. Xilinx calls each such die a Super Logic Region (SLR). Latency is higher and bandwidth lower at SLR crossings than within the FPGA fabric logic on a single die. This often requires extra care in communication across the SLR interfaces, including additional pipelining.

3 COMPUTE MODEL

In this section, we describe how PLD utilizes a streaming dataflow [16, 29] model to abstract operator and communication implementations. This model allows the composition of highly parallel computations out of modular computing components or *operators*. Streaming dataflow abstracts how operators are connected, hiding compute and communication details and allowing operators to be implemented on different architectures with different kinds of communication. This supports high-level composition of operators into computational graphs while also supporting diverse implementation architectures for operators and communication. Hiding operator internals, it provides a well-defined meaning to operator interaction while allowing separate compilation and independent refinement of operator specification.

3.1 Dataflow Composition Model

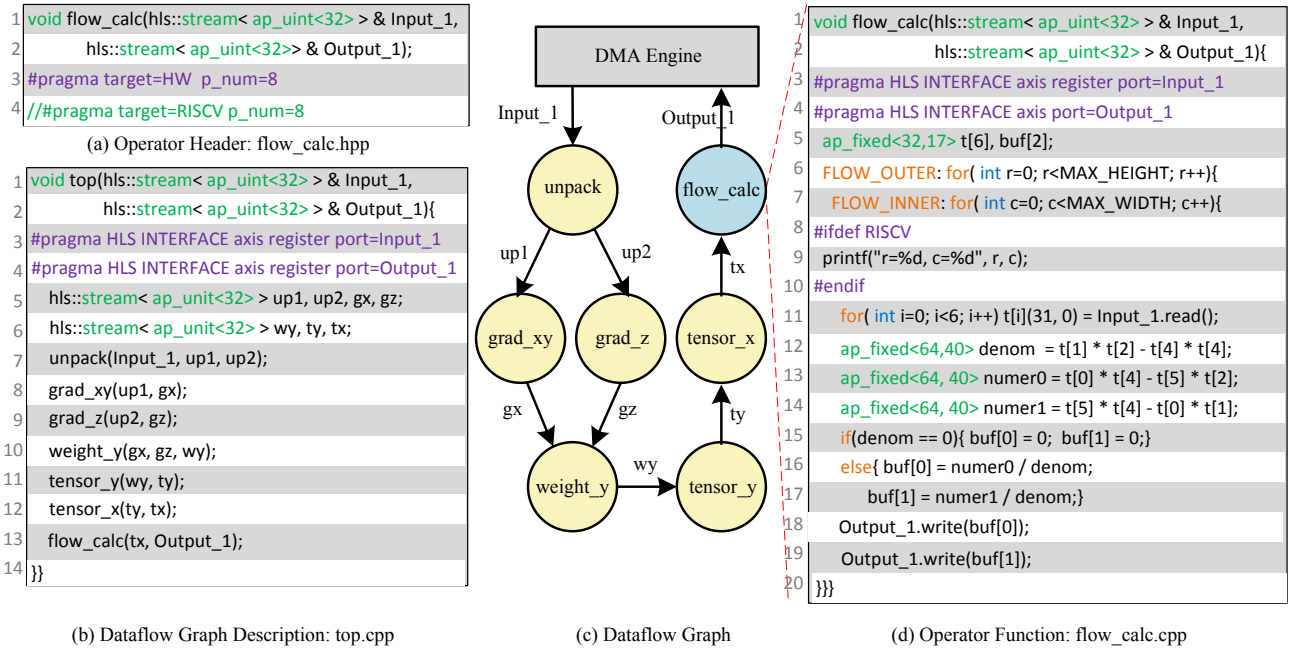
PLD uses the SCORE streaming dataflow computational model [16] based on Kahn Processing Networks [29]. Basic kernel computations are described in C as operators that receive inputs over latency-insensitive streams [7] and produce outputs to latency-insensitive streams. This builds on the extensive use of dataflow streaming communication between concurrent computations in High-Level Synthesis (HLS), including (1) between functions, (2) between loops, and (3) (in Vitis [66]) between host and FPGA. We extend the model and implementation to be agnostic to how an operator or the producer or consumer that it communicates with is mapped.

3.2 Communication Abstraction: Latency Insensitive Links

The key to abstracting operators and communications is to define how data is transferred and synchronized between operators. PLD uses latency insensitive stream links [7] to provide this abstraction across a wide range of implementations. The latency-insensitive stream links act like FIFOs between the source and destination. They include data presence as the producer writes results and the consumer reads them. Reads from empty streams block until data becomes available. This provides integrated synchronization between producer and consumer and makes the communication behavior independent of the timing of the operators or the transport between producer and consumer. Back pressure from the consumer through the FIFO link stalls the producer to prevent data loss. Consequently, if either the producer or consumer run faster or slower from being mapped to FPGA or processor substrates, this doesn't change the functional behavior of the computation. Similarly, different communication timing from various, possibly shared, communication medium does not change functional behavior.

3.3 Application Description

The top-level kernel is a graph of operators connected by latency-insensitive stream links as shown in Fig. 2(c). This graph can be described in C by function composition of operator functions using stream links as arguments (See Fig. 2(b)). We add pragmas that specify where each operator is mapped (See Fig. 2(a)). The code in Fig. 2(b)) works directly with native Vitis_HLS which ignores our added pragmas. This top-level kernel specification can be automatically compiled into a monolithic design with direct stream links for compilation with Vitis_HLS (Sec. 6.3). Alternately, it can be compiled with our separate compilation tools (Sec. 6.2) to generate the linking graph needed to configure the linking network (Sec. 4.3).




```
1 void flow_calc(hls::stream< ap_uint<32> > & Input_1,
2         hls::stream< ap_uint<32>> & Output_1);
3 #pragma target=HW  p_num=8
4 //#pragma target=RISCV p_num=8
```

(a) Operator Header: flow_calc.hpp

```
1  void top(hls::stream< ap_uint<32> > & Input_1,
2           hls::stream< ap_uint<32> > & Output_1){
3  #pragma HLS INTERFACE axis register port=Input_1
4  #pragma HLS INTERFACE axis register port=Output_1
5    hls::stream< ap_unit<32> > up1, up2, gx, gz;
6    hls::stream< ap_unit<32> > wy, ty, tx;
7    unpack(Input_1, up1, up2);
8    grad_xy(up1, gx);
9    grad_z(up2, gz);
10   weight_y(gx, gz, wy);
11   tensor_y(wy, ty);
12   tensor_x(ty, tx);
13   flow_calc(tx, Output_1);
14 }}
```

(b) Dataflow Graph Description: top.cpp

(c) Dataflow Graph

```
1  void flow_calc(hls::stream< ap_uint<32> > & Input_1,
2                 hls::stream< ap_uint<32> > & Output_1){
3  #pragma HLS INTERFACE axis register port=Input_1
4  #pragma HLS INTERFACE axis port=Output_1
5    ap_fixed<32,17> t[6], buf[2];
6    FLOW_OUTER: for( int r=0; r<MAX_HEIGHT; r++){
7      FLOW_INNER: for( int c=0; c<MAX_WIDTH; c++){
8  #ifdef RISCV
9    printf("r=%d, c=%d", r, c);
10 #endif
11     for( int i=0; i<6; i++) t[i](31, 0) = Input_1.read();
12     ap_fixed<64,40> denom  = t[1] * t[2] - t[4] * t[4];
13     ap_fixed<64, 40> numer0 = t[0] * t[4] - t[5] * t[2];
14     ap_fixed<64, 40> numer1 = t[5] * t[4] - t[0] * t[1];
15     if(denom == 0){ buf[0] = 0;  buf[1] = 0;}
16     else{ buf[0] = numer0 / denom;
17         buf[1] = numer1 / denom;}
18     Output_1.write(buf[0]);
19     Output_1.write(buf[1]);
20 }}}
```

(d) Operator Function: flow_calc.cpp

Figure 2: Dataflow Graph and Operators

3.4 Operator Discipline

There are some restrictions for C functions to make good, concurrent dataflow operators for acceleration. Refining computations into this form is one of the demands for rapid, incremental compilation of C. This allows the developer to move one function at a time and to incrementally refine each operator to meet the full set of requirements then tune for performance.

Fig. 2(d) shows a C example for the operator. Operator functions must be refined to use streaming computation and limit themselves to constructs that can be supported by a fixed-hardware accelerator. All function arguments for communicating data should be `hls::streams` (lines 1-2) and the functions should use the associated stream API operations for all communication. Operators must avoid allocation or recursion since these cannot be directly supported by HLS mappings on the FPGA; to exploit processor-only operations when they are on the processor, like `print`, they should be guarded by suitable `ifdef` software macro guards (lines 8-10). Operators must use a standard set of datatypes with compatible implementations for processor and FPGA (e.g., arbitrary precision integer and fixed-point libraries: `ap_int`, `ap_fixed`). Operators should not use global shared memory directly, instead relying on the streaming dataflow setup to stream data in and out of the kernel as needed; this restriction could be removed in future work, but DMA streaming is a powerful technique to achieve high-performance FPGA kernels.

4 PAGE DECOMPOSITION

The basic idea to support separate compilation is to divide the FPGA into a collection of independent regions. We call each atomic region of management a "page", similar to Active Pages [48] by analogy with a virtual memory "page" that is the atomic region of memory management (Sec. 4.1). Each FPGA page holds a contiguous region, typically rectangular, of the FPGA resources include gates (LUTs), memory (BRAM), and coarse-grained arithmetic (DSPs). Unlike Active Pages, which focused on data-parallel computations, PLD supports concurrent execution and communication between pages with heterogeneous logic that is provided by a linking network (Sec. 4.3). The linking network along with some common support logic for clocking and DMA data streaming form a fixed infrastructure context for the pages (See Fig. 3).

4.1 FPGA-Mapped Pages

FPGA compile time is driven by both the size of the logic being mapped and the resources the logic is being mapped onto. If smaller pieces of logic (operators) are mapped onto small FPGAs (small pages), the mapping time is smaller than mapping an entire application (all the operators) onto a large FPGA (an entire data-center scale FPGA). The key to accelerating FPGA mapping time is to divide the FPGA up into many separate and smaller compilation problems that can be solved independently and potentially in parallel. Once the FPGA is divided into pages, an operator can be mapped to a physical FPGA page without concern for what resources other operators use.

Vitis provides an abstract shell[1] option [69] that allows one to create a design file that only describes the necessary interface logic to a DFX region—one of our pages. This is essential to achieving

[1] Abstract shell is not to be confused with the "Shell" logic used to hold the fixed PCI logic in on data-center boards (Sec. 2.5); the abstract shell is purely a mechanism in the CAD flow and is not directly related to the PCI interface shell. Both uses of "shell" are about abstracting out the details of the design that exist beyond the shell interface, but the terms are being used at different levels in the abstraction stack.

fast compilation as it allows the page compilation to ignore the logic on the FPGA beyond the page and its interface with the rest of the FPGA. Without the abstract shell option, Vitis will still load and check all the logic for the linking network and all the pages, slowing down page compilation.

Each page communicates with the linking network through a standard leaf interface. The leaf interface is included in the operator logic mapped to the page and provides the common logic so that the operator can talk to the linking network.

Page sizing is a balance between compilation time, efficiency, and convenience. Small pages allow shorter mapping times. However, small pages pay higher overhead for the leaf interface to the linking network and higher communications overhead. Small pages also put higher demands on the developer (or mapping tools) to divide the logic into small operators that will fit in each page. Pages also suffer fragmentation similar to memory pages. Larger pages may have high internal fragmentation when the operator logic does not fully use the logic capacity allocated to the page, while smaller pages demand the developer and the mapping software keep track of more pages. The efficiency is roughly:

$$Eff. = \frac{\sum(\text{Operator Page Use})}{\sum(\text{Page Size} + \text{Leaf Interface}) + \text{Linking Net}} \quad (1)$$

Our network interfaces run about 500 LUTs and the current linking network needs about 500 LUTs per endpoint. As such, we choose about 18,000-LUTs pages so that we have around 95% efficiency before considering fragmentation.

Page sizing is further complicated by the fact that today's commercial FPGA fabrics are not completely regular. Memory (BRAM) and coarse-grained logic (DSPs) are inserted into the gate-level fabric as heterogeneous columns at irregular intervals. As such, it is impossible to divide the logic up into a grid of equal sized pages. Furthermore, fixed logic in the FPGA (e.g., PCI interface, hardcore processors, and PLLs) also break up the regularity of the logic fabric. As a result, pages on conventional FPGAs are a heterogeneous mix of resources (See Tab. 1).

4.2 Data-Center FPGAs and Cards Abstraction

PLD virtualizes the data-center FPGAs as pages and only provides page resource information to the developers abstracting away tedious physical implementation information from software developers. Since the vendors have already partitioned the data-center FPGA into one static region (for bitstream configuration and communication with the Linux host) and one user DFX region, the traditional partial reconfiguration techniques cannot be leveraged directly. Fortunately, the new hierarchical DFX feature enables sub-DFX region partitions [68]. PLD leverages this new technique to reserve the original user DFX region as a level 1 (L1) DFX region (block ① in Fig. 3) to utilize the static shell (block ③ in Fig. 3). This is compatible with the Vitis OpenCL driver from the vendor. We partition the L1 DFX region into a cluster of level 2 (L2) DFX regions (PLD pages ② in Fig. 3).

4.3 Linking Network

To connect the physical FPGA pages containing separately compiled operators, PLD adopts a linking network. The linking network provides a similar high-level abstraction to software linking and loading, connecting the individual implementations for operators together so they can transfer data. Our linking network provides the physical connectivity implementation for the latency-insensitive dataflow streams. The linking network can be configured to connect operator inputs and outputs, as physically instantiated on pages, together according to the application dataflow graph.

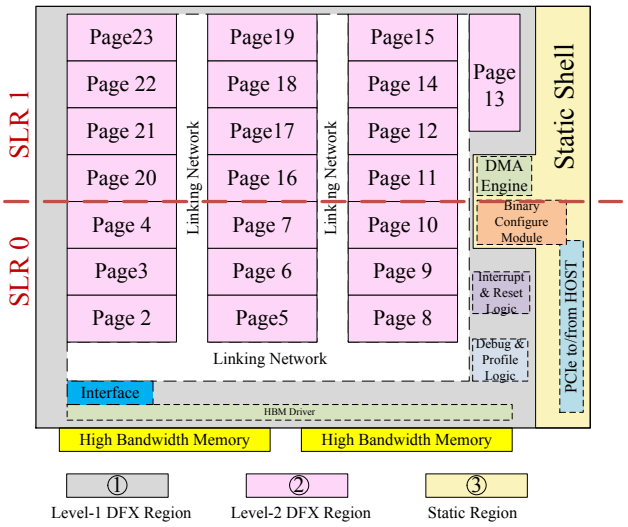


Figure 3: PLD FPGA Decomposition: Pages, Linking Network, and Support Infrastructure

PLD uses a Hoplite, lightweight, deflection-routed [18, 46], single-flit packet, packet-switched network [34] using a Butterfly Fat Tree (BFT) topology [32] as shown in Fig. 3. We set control registers in the leaf interface to add appropriate packet destination headers to data so they will be routed through the network. These control registers can be changed with control packets on the network, so that operators can be re-linked without recompiling the source or destination pages. As such, it is only necessary to send a few packets per page to link it into the network.

A modest packet-switched network is deployed in PLD for the fastest linking. The standard leaf interface connection into the network can become a performance bottleneck when the operator needs higher bandwidth than a single network port provides. The streaming dataflow abstraction admits to a wide range of linking network architectures that could provide different compile-time versus performance and overhead points. Wider network interfaces and networks with less link sharing will support higher performance at the expense of larger overhead. By cutting the network region up into switch pages, it would be possible to dedicate unshared wire connections between operators and separately compile the application-customized linking switch pages, as well [64].

5 SOFTCORE INTEGRATION

Mapping to small FPGA pages is still slower than mapping to a processor. However, we can always configure portions of the FPGA, including an FPGA page, as a processor. The processor serves as a simple overlay architecture that admits to fast compilation. Since users start with C code, PLD also allows them to easily compile

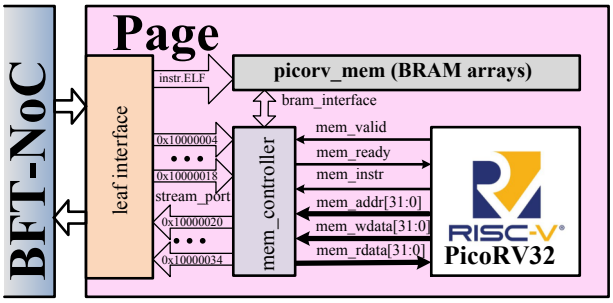


Figure 4: RISC-V Integration for One PR Page - mem_controller is responsible for connecting native memory interface, stream interface and BRAM access interface; processor instructions (ELF) are loaded from the BFT-NoC via the leaf interface.

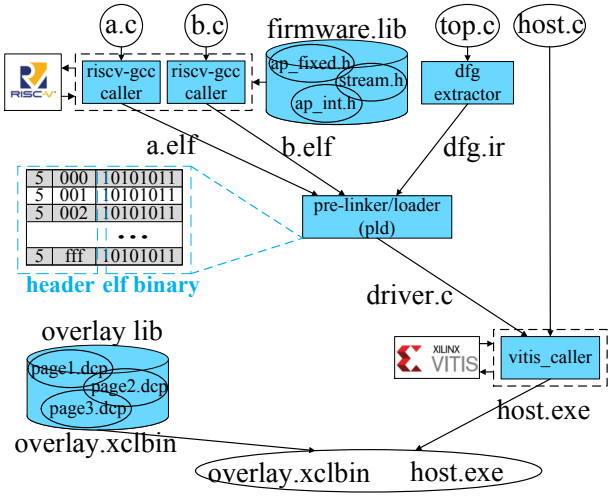


Figure 5: -O0 Compile Flow

the same C source to an overlay processor as an `-O0` mapping to quickly get code changes integrated and running on the accelerator.

5.1 Processor Prototype Implementation

As a prototype implementation, PLD pre-loads each page with a PicoRV32 soft processor [62], as it is area-efficient and easy to set up. PicoRV32 with 32-bit multiplier includes a simple native memory interface running at 200 MHz to match the overlay linking network. In this configuration, the PicoRV32 needs 2K LUTs, which easily fits in the DFX pages along with leaf interface logic. As the PicoRV32 uses a unified instruction and data memory, different instruction and data memory size can be allocated according to the operator's number of input and output stream ports and the applications need. PLD pages support at most 192 KB (96 BRAM18s) of unified memory.

5.2 Streams Support and Compatibility Libraries

Key to integrating the processors in place of FPGA pages is supporting streaming communications and the standard interface to the linking network. PLD supports `hls::stream` by defining general peripheral ports with global memory address and support logic to interface with the leaf interface FIFOs. As shown in Fig. 4, each stream datawidth is 4-bytes, matching the datawidth of the 32b processor. To take advantage of customization features, FPGA HLS C++ code often use datatypes from `ap_int` and `ap_fixed` libraries. These arbitrary-precision variables can be mapped to FPGA logic using minimum LUTs, rather than 32b or 64b datapaths on processors. However, Xilinx `ap_int` and `ap_fixed` libraries use more than the minimum number of bits to represent these types, which can be a challenge when our partial reconfigurable pages only have 48 or 96 BRAM18s. Therefore, we develop our own, more memory efficient, `ap_int` and `ap_fixed` libraries that are compatible with the existing Xilinx HLS C++ code. Mapping all the operators from the Rosetta Benchmarks (Sec. 7.2), the code and data footprint for each operator is typically 30–60 KB, consuming 16–32 BRAM18s.

6 PLD TOOL FLOW

To realize our separate compilation and linking, PLD provides an automated tool flow to map operators to pages and configure the linking network. PLD uses Vitis as a backend to compile from C to placed-and-routed logic for individual pages, hiding all the details of DFX compilation and linking. PLD also automatically switches between compiling FPGA page targets (-O1, Sec. 4.1) and softcore processor targets (-O0, Sec. 5) based on compilation flags and directives. We develop a standard `Makefile` configuration so only the pages with changing logic must be recompiled, and build parallelism can launch separate page compiles concurrently. An operator with mapping control directives is shown in Fig. 2(a) Line 3. Each operator has a line with a target specification. Changing the target will change whether the page is loaded as a native FPGA partial bitstream or a standard processor overlay loaded with a compiled processor instruction stream. Changing the target also sets up the compiler dependencies to build the appropriate bitstream or instruction stream. PLD can run on both local machines and the Google Cloud Platform. PLD can be easily extended to other cloud platforms.

6.1 -O0: Fast Mapping to RISC-V

Fig. 5 shows PLD toolflow for the `-O0` optimization. The input includes separate C files (a.c and b.c), top function (top.c) that describes the operators' connections, and host.c that uses the Vitis OpenCL interface to instantiate kernels and launch DMA to interface with the kernels.

When the target in the header file for an operator is set to `RISCV` (Fig. 2(a) Line 4), PLD launches the `-O0` compile flow, and calls the RISC-V toolchain to compile each operator's C file to a standalone binary in standard ELF (executable and linkable format) format. A pre-linker/loader (pld) packs the binary with headers that indicate the final page number and the memory address for each binary byte. A dfg extractor produces a data flow graph intermediate file (dfg.ir) from top.c that pld uses with the the binary components to construct the driver.c file to orchestrate partial bitstream loading and

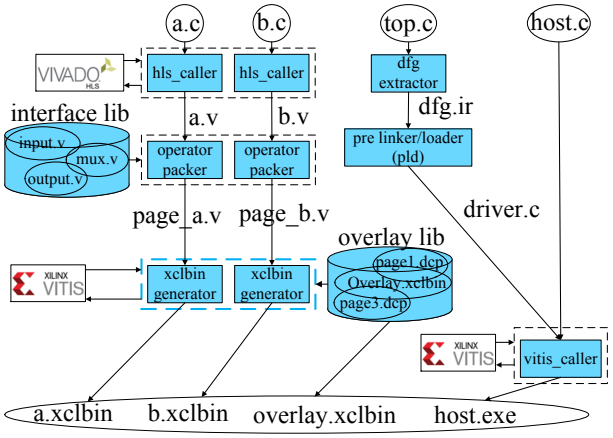


Figure 6: -O1 Compile Flow

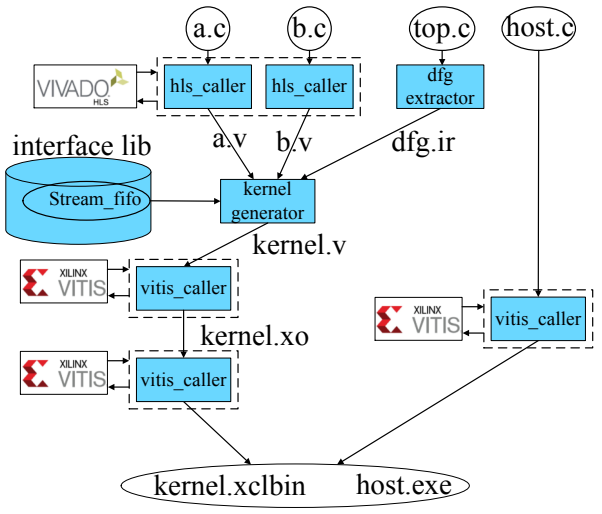


Figure 7: -O3 Compile Flow

linking network configurations. For `-O0` compiled operators, the driver loads the packed ELF binaries into the appropriate softcore memories. This driver.c and host.c are compiled by the `Vitis` software compiler to generate the top-level host executable, host.exe. Executing host.exe along with the overlay bitstream (overlay.xclbin) on the Linux host will run the application.

6.2 -O1: Incremental Mapping to FPGA Fabrics

With the same source code, PLD can launch the `-O1` compile flow (Fig. 6) if the target pragma is `HW` for that operator (Fig. 2(a) Line 3). The hls_caller of PLD generates customized scripts to direct `Vitis_HLS` to compile the C source to Verilog files. The operator packer wraps each operator's Verilog files with our pre-defined leaf interface (Sec. 4.1). This interface is used for the communication between the linking network and the page logic. Just as in the `-O0` flow, the dfg extractor generates a dataflow graph intermediate (dfg.ir), and the pld module uses it to generate driver.c that is responsible to configure the linking network by sending configuration packets through the network. Host.c and driver.c are all compiled by the `Vitis` software compiler to generate the host.exe. To run the application, the Linux host needs to load the overlay.xclbin (L1 binary) first to set up our overlay. Then it loads all the page xclbins (L2 binary) to map the operators to real FPGA fabrics.

The partial bitstream (xclbin file) for each operator can be compiled independently. The xclbin generator along with the pre-compiled overlay library are the key to accelerating compilation time for the operators. According to the page target of the operator, which represents the physical page to which the operator will be mapped, the xclbin generator selects the appropriate abstract shell from our pre-compiled overlay library and generates a customized script to guide `Vitis` to place the operator in the appropriate DFX region. The customized script directs `Vitis` to only place and route that single operator on the small target physical page region. Since the `Vitis` abstract shell only loads the minimal context related to the page, `Vitis` can perform much less work during placement and routing than the monolithic flow. All the operators' compilations can be performed in parallel, since they are implemented on different physical locations with no overlapping area. When all the xclbin compilations are performed in parallel, the compilation time is determined by the longest individual one instead of the total compilation time.

Fig. 5 and 6 show compiling all operators as `-O0` or `-O1` for simplicity. The tool flow allows any combination of operators, each independently mapped `-O0` or `-O1`. The common linker (pld) will pack together the binaries from the `-O0` flow and the common linking dataflow graph and load the appropriate combination of `-O1` mapped page L2 DFX xclbins and RISC-V page L2 DFX xclbins for the `-O0` mapped operators.

6.3 Monolithic (O3) Linking

The `-O1` compilations are fast and suitable for incremental development especially during debugging and design-space exploration stage. However, the application's performance may be constrained by the limited bandwidth of the linking network. Moreover, the pre-defined overlay infrastructure can also increase the area overhead. To overcome the above limitations, PLD provides the users with the `-O3` compile option that can compile the same application source monolithically as the normal Vitis flow.

With the same source code, PLD can launch the `-O3` compile flow (Fig. 7) when the `-O3` target is selected. PLD generates the Verilog files for all the operator by calling `Vitis_HLS` just as in the `-O1` flow. The same dfg extractor still generates a dataflow graph intermediate file (dfg.ir). The `-O3` flow adds a kernel generator module that uses the dataflow intermediate graph specification to connect all the operators back together with hardware FIFO streams at the Verilog level according to the datawidth for each link. PLD calls `Vitis` on the kernel Verilog files to package up a Xilinx object file (`kernel.xo`) and perform synthesis, placement, and routing to generate the kernel.xclbin bitstream. The host.c can be compiled by the normal `Vitis` software flow. Finally, the users can run the application by loading the kernel.xclbin and executing host.exe, just as with the normal Xilinx Vitis flow. Since the design is mapped to

Table 1: Resource Distribution

Page Type	Type-1	Type-2	Type-3	Type-4
LUTs	21,240	17,464	18,880	18,560
FFs	43,200	35,520	38,400	37,440
BRAM18s	120	72	72	48
DSPs	168	120	144	144
Number	7	7	7	1

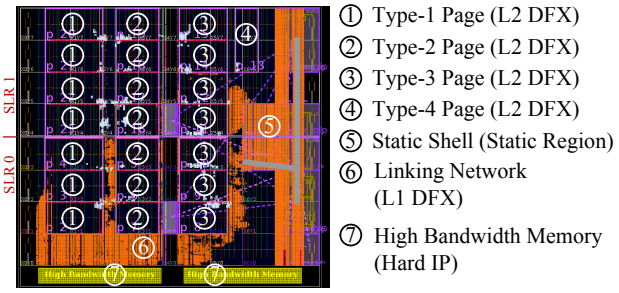


Figure 8: Physical Layout Floorplan

the raw FPGA fabric, instead of our pre-defined overlay, the area overhead and bandwidth limitations of the overlay are removed.

7 BENCHMARK EVALUATION

7.1 Methodology

We evaluated PLD by targeting a Xilinx Alveo U50 data center card with a Virtex UltraScale+ XCU50 FPGA. As Xilinx has already implemented its firmware shell as the static region for bitstream configuration and host-fabric communications, the available resources for the developer includes 751,793 LUTs, 2.300 18Kb BRAMs and 5,936 DSPs in two SLRs (Sec. 2.5). PLD uses Xilinx Vitis 2021.1 including the associated `Vivado` and `Vitis_HLS` as the backend compiler. All experiments, both PLD decomposed compiles and Vitis monolithic compiles, use the same, standard Vitis shell logic. All the compilation experiments are conducted on the Google Cloud Platform. Slurm [21] is deployed to construct the scheduler cluster, where each compute node is equipped with 4 dual-thread, 2.8 GHz Intel Xeon Intel(R) Cascade Lake processors and 32 GB RAM. For monolithic compile, we use a compute node with 15 dual-thread, 3.1 GHz Intel Xeon Intel(R) Cascade Lake processors and 128 GB RAM.

Fig. 8 shows the layout floorplan. We divide the chip into 22 user logic pages and an interface module that combines a portion of the packet-switch network with DMA (Fig. 3). The page resource and types are listed in Tab. 1. PLD uses a Hoplite BFT [32] for the packet-switched network, running at 200 MHz with 32b data payload. The interface leaf logic in each partial-reconfigurable page allows configuration of the consumer address by packets on the BFT network.

7.2 Benchmark Set

For evaluation, we use the Rosetta Benchmark suite [74] containing a diverse range of graphics, image processing, and classification tasks. The original Rosetta Benchmarks were written for monolithic HLS synthesis. We decomposed the benchmarks into operators and added dataflow stream links for communication among the operators. We report results for both the original HLS synthesis version and our automatically generated `-O3` monolithic version that compiles from our decomposed version in the following three subsections.

rendering – a simple triangle rendering pipeline that includes projection to a 2D viewpoint, rasterization, and Z-buffering. We decomposed by the pipeline stages, then decomposed large pipeline stages by image region.

digit recognition – a classification task for hand-written digits 0–9 that uses matching to a training set to identify each candidate digit. We refactored the computation as a systolic pipeline with each pipe stage operating on a subset of the training set.

SPAM filtering – a classification task that identifies the likelihood of SPAM based on a set of feature vectors. We decomposed the data-parallel feature vectors into separate dot product operators and provided operators for decomposition and data reduce.

optical flow – an image processing task that identifies the movement of objects among a set of frames. The original computation already had the shape of a dataflow task graph. We started with each task as an operator. We decomposed large operators that did not fit onto a single page by separable components (e.g., x, y, and z computations).

face detection – an image classification task that identifies faces in images. We decomposed the two main stages of the computation (strong and weak filtering), then decomposed the strong filtering by image region and the weak filtering by filter sets.

bnn – a binarized neural network performing image classification of 10 images based on the CIFAR-10 dataset [39] that uses 6 convolutional levels and 3 fully connected levels. The first convolutional level operates on fixed-point inputs and produces binary outputs, while the remaining levels operator on binary signals. We moved the weight coefficients to on-chip memory and made each stage and operation its own operator.

For the original benchmark, most of the kernels require more than 1000 lines of code to describe.

7.3 Compile Time

We initially compile the original benchmarks with the standard Vitis flow as our baseline. Then we use PLD to compile modified source code with different optimization level by only changing the pragmas and compiler options and summarize all compile times in Tab. 2.

The original Vitis flow has compilation times of 1–2 hours. We further see that placement and routing (p&r) requires roughly half of compilation time. When we compile our decomposed benchmarks with `-O3` option, our `-O3` compiled decomposed designs take about the same 1–2 hours for compilation.

When all of the operators are mapped with our separate compilation and linkage `-O1` option, compile times reduce to 10–20 minutes, a factor of 4.2–7.3 speedup. Pages have varying mapping times, so the actual `-O1` mapping downtime will vary based on the page being recompiled. Fig. 9 shows the distribution of page mapping times. The design with the worst-case 20 minute compile page also

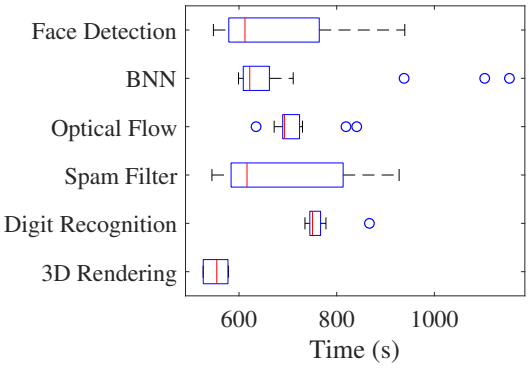


Figure 9: Operators Mapping Time for PLD with -O1

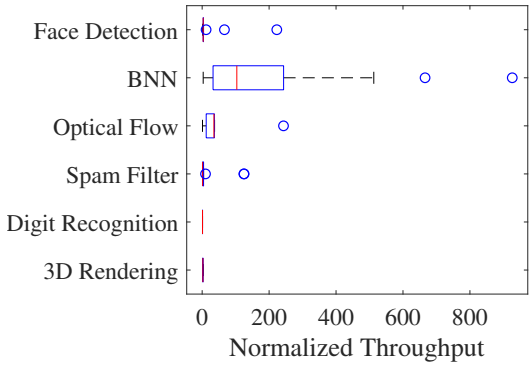


Figure 10: Speedup Distribution with One Softcore (`-O0`) and Rest on FPGA Pages (`-O1`) Compared to All Softcore (`-O0`)

has pages that compile in 10 minutes. Designers can deliberately refactor into smaller operators to decrease compile time during development.

Tab. 2 also includes the case where all of the operators are mapped with our `-O0` option. Compile times reduce below 4 seconds, demonstrating that this option allows PLD to support nearly instant compilation for rapid functional debugging.

7.4 Performance

Tab. 3 lists the performance comparison between Vitis and PLD. Compiled with the PLD `-O3` option, our decomposed designs achieve comparable performance to the original, undecomposed designs, showing that decomposition for fast mapping does not degrade the performance of the application. Some of our `-O3` designs (Optical flow and BNN) can run at higher clock rates because of the FIFOs and pipelined interconnect between operators that isolate the operator delay and long interconnect between operators, whereas the original monolithic designs may suffer from long wires and slow SLR crossings (Sec. 2.5).

For PLD's page-decomposed `-O1` option, applications run 1.5–10× slower than the monolithic cases. Most slowdowns are due to data bandwidth bottlenecks between the leaf pages and the linking network. This is partly due to our general overlay that can map a wide range of benchmarks and that has been tuned for mapping speed over performance. This limitation can be easily overcome by using `-O3` optimization in PLD as shown above. Tab. 3 also shows Xilinx Vitis Emulation time (Vitis Emu) and the native application runtime on the X86 host, showing that the `-O1` FPGA page compilation is providing substantial speedups over host execution or emulation.

As expected the processor-mapped `-O0` designs run three to five orders of magnitude slower than the monolithic, FPGA-mapped cases (Tab. 3). While much slower, this allows developers to quickly find interface and logic bugs introduced in revisions. While some full frame times are long, many bugs can be identified after seeing the results of a few pixels or elements. During steady-state debugging, a common practice will be to recompile only the single operator being debugged with `-O0`. Fig. 10 shows the distribution of speedups compared to the all `-O0` case for cases where a single operator is mapped with `-O0`. Here, we see a range of performances.

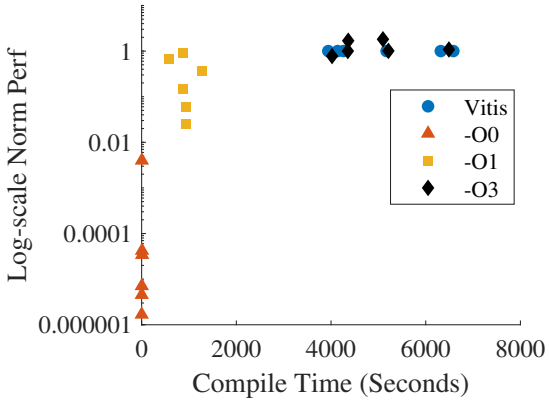


Figure 11: Performance vs. Compile Time

When the bottleneck operator is mapped with `-O0`, performance can approach the all `-O0` case, but many cases are faster, falling between the all `-O0` and all `-O1` cases, highlighting the benefit of being able to co-operate with portions of the design mapped natively to FPGA pages. The PicoRV is a slow, unpipelined core, and performance can easily be improved by replacing it with a higher frequency, pipelined softcore processor.

Fig. 11 shows how the performance and compile-time compare among these options. This clearly shows that PLD provides interesting new points in the compile-time versus performance trade space, providing developers with new control options to support rapid edit-compile-debug loops, without giving up their single C source or their access to high quality designs when they are ready to invest in longer compile time.

7.5 Area Evaluation

Tab. 4 shows the resource breakdowns for Vitis and PLD implementation. For PLD `-O1`, we add up all the operators' resource including leaf interfaces. We can see `-O1` and `-O3` both have higher resource utilization than the Vitis flow, since both use FIFOs to link operators together. This can consume a large number of BRAMs along

Table 2: Rosetta Benchmark Compile Time (in seconds)

	Vitis Flow Compile with 30 Threads					PLD -O3 Compile with 30 Threads					PLD -O1 Compile with 8 Threads for each Operator					PLD -O0 Compile
	hls	syn	p&r	bit	total	hls	syn	p&r	bit	total	hls	syn	p&r	bitgen	total	riscv g++
3D-rendering	104	1190	2364	606	**4264**	36	1306	2359	662	**4363**	35	119	294	130	**578**	**1.0**
Digit Recognition	144	1627	2673	729	**5173**	30	1701	2827	654	**5212**	29	114	560	164	**867**	**1.5**
Spam Filter	69	1308	1867	698	**3942**	20	1397	2243	695	**4355**	12	139	597	177	**925**	**3.1**
Optical Flow	84	1293	2094	668	**4139**	19	1531	2854	693	**5097**	18	102	583	177	**880**	**2.4**
Face Detection	542	1738	3280	728	**6288**	20	1219	2115	668	**4022**	24	212	545	158	**939**	**2.1**
Binary NN	485	2946	2430	723	**6584**	225	2153	3292	820	**6490**	225	415	372	140	**1152**	**3.4**

Table 3: Rosetta Benchmark Performance

	Vitis Flow		PLD -O3		PLD -O1		PLD -O0		X86 g++	Vitis Emu
	Fmax	per input	Fmax	per input	Fmax	per input	Fmax	per input	per input	per input
Rendering	300 MHz	1.6 ms	300 MHz	0.9 ms	200 MHz	1.4 ms	200 MHz	3 s	0.1 s	3 s
Digit Reg	300 MHz	10.5 ms	300 MHz	3.9 ms	200 MHz	6.2 ms	200 MHz	137 s	824.0 s	7400 s
Spam	300 MHz	18.6 ms	300 MHz	20.0 ms	200 MHz	68.7 ms	200 MHz	752 s	4.5 s	100 s
Optical	200 MHz	13.6 ms	300 MHz	4.8 ms	200 MHz	48.4 ms	200 MHz	10 935 s	7.3 s	151 s
Face	300 MHz	24.1 ms	300 MHz	31.0 ms	200 MHz	125.0 ms	200 MHz	527 s	17.5 s	352 s
BNN	150 MHz	5.1 ms	300 MHz	4.7 ms	200 MHz	7.1 ms	200 MHz	983 s	135.0 s	7100 s

Table 4: Rosetta Benchmark Area Consumption

	Vitis Flow			PLD -O3			PLD -O1				PLD -O0			
	LUT	B18	DSP	LUT	B18	DSP	LUT	B18	DSP	PAGE#	LUT	B18	DSP	PAGE#
3D-rendering	4225	64	13	17696	128	26	22823	106	18	6	119208	576	864	6
Digit Recognition	36070	382	1	50595	406	0	63923	441	0	20	393224	1680	2832	20
Spam Filter	9616	34	224	21011	126	256	50965	204	256	16	291480	1176	2088	16
Optical Flow	26974	136	158	27278	192	312	43231	211	312	16	313752	1296	2256	16
Face Detection	51549	156	97	127890	322	192	164385	296	145	20	393224	1680	2832	20
Binary NN	26724	46	5	44077	1130	5	64093	1197	4	22	437768	1920	3168	22

(B18: BRAM18)

with LUTs to support those BRAMs and their synchronization. One promising solution is to use Relay Station [64] to connect operators together, instead of stream FIFOs. However, this requires care to set the buffer sizes appropriately to avoid introducing deadlock that was not part of the original design. We leave this alternative and optimization to future work. The `-O0` cases use large total resources because they use a single, one-size-fits-all processor and memory organization with large memory capacity to handle the worst-case memory requirements of any operator.

7.6 Discussion

The raw results in the previous section should give a flavor of how faster compilation will impact development. If the compile time did not impact the compiles the developer made, we could look at ratios between monolithic compile times and PLD `-O1` and `-O0` compile times and estimate reductions in compile time between one and three orders of magnitude. However, developer behavior is not likely to be the same. Most developers will simply avoid FPGAs given the daunting long compile times. Those that do, are likely to perform less optimization and refinement since they do not have the orders of magnitude more time to invest in development. As such, we expect PLD provides a qualitatively different experience, enabling more software engineers to access FPGA acceleration, and allowing those that do to develop more highly optimized implementations.

8 RELATED WORK

Many designs use the idea of dividing the FPGA into separately managed physical regions to allow independent logic to be mapped to the FPGA [4, 9, 11, 35, 43, 44, 50, 72, 73]. However, these designs do not address compile time reduction or support for high-level compilation from C.

Cascade [52] and SYNERGY [41] also aim at improving the FPGA programming experience, allowing applications to run immediately in simulation and supporting unsynthesizable Verilog primitives (`$printf` or `$finish`). Subprograms can be replaced over time by hardware engines when FPGA-target compilations finish, hiding the compilation time along with software runtime. However, these

approaches demand code be rewritten in a lower-level language, Verilog, rather than incrementally refined within a single source language. Moreover, the compilation-hiding scheme still requires monolithic compiles, each of which can take hours before the FPGA-acceleration can be incorporated into the computation. Since the Verilog simulation is slow compared to native C compiled to processors, developers must maintain and synchronize two separate version of the code if they wish to maintain flexibility on whether a subprogram executes on the FPGA or a processor core. Our backend works from RTL, so can likely provide separate compilation benefits for the Cascade/SYNERGY flow.

One path to faster compilation has been to pre-define overlay architectures for the FPGA [3, 14, 19, 27, 36–38, 42, 61, 73]. These overlays provide a higher-level, typically more coarse-grained, architecture than the FPGA that is an easier compile target. With fewer, coarse-grained operators to map and many of the low-level details already addressed in the low-level definition of the overlay components, compilation can be faster. However, overlays typically come at a large cost (3–10×) in capacity since the computation isn't directly exploiting the FPGA resources. Many overlays are specialized to particular domains [19, 73], allowing them to achieve higher capacity in the domain, but then they do not support applications outside of the domain. The more specialized overlays must often be programmed in their own domain-specific language or instruction set, further meaning they cannot be programmed with the same source code that can be compiled to processors and native FPGA logic.

Implementing processors, including scalar [40, 54, 70], vector [53, 71], and specialized VLIW [26, 33, 56, 60] processors, on top of FPGAs can also be seen as a form of overlay that allows faster compilation using standard compilation tools and techniques. We build on this idea for our fastest compile option (Sec. 5), and PLD could use a wider range of overlays with our general approach in the future.

Our prior work [49, 65] hints at the promise of separate compilation for FPGAs, but only runs on application-dedicated embedded platforms (no operating system, not support OpenCL interface or data-center cards). While that work accelerated compile times using these smaller, decomposed mapping tasks, it still saw compiles that took 5–10 minutes. Significantly, the designs only run when the operators have already been sized to fit into the pre-sized hardware regions. This means that development may go through long periods of time when the design cannot be tested directly on the hardware. This also does not support incremental refinement and leaves the developer without running code throughout refinement process.

9 FUTURE WORK

We have demonstrated the potential for fast, incremental compilation using a specific page size and linking network architecture. The streaming dataflow model gives us considerable freedom to change the microarchitecture of the system without changing the functional behavior of the application. Multiple infrastructure overlays with different resources can be pre-computed and stored as alternate compile-time and quality targets available to the software developer.

A natural extension would be to expand the potential overlays available for fast mapping, including a range of pre-compiled processors with different specializations (e.g., floating-point, multipliers, vectors), customized VLIW processors or coarse-grained arrays with diverse resource mixes, and even more specialized, domain-specific overlays that can be used when they match operator needs.

Decomposing and sizing operators to fit into pages is an added developer burden. It would be useful to develop more automation and high-level pragmas to allow the programmer to guide page partitioning without restructuring code, the same way programmers can use pipelining and unrolling pragmas in HLS to guide implementations without rewriting loops.

While our demonstration is with vendor tools with developer refined C, our tools could serve as accelerated backends for emerging compilation flows that create efficient, pipelined dataflow operators but also suffer from long monolithic place-and-route times using standard vendor tool flows [23, 28, 59].

10 CONCLUSIONS

PLD offers developers a new set of compile-time versus performance points. To complement the hours-long, high quality compilation provided by vendor tools, PLD offers fast, native-FPGA compile options that compile in 10 minutes and near immediate compilation of the same source to softcore processors that complete in seconds. These speedups are enabled by extending an idea from processor compilation of separate compilation and linkage of program operators, allowing the compiler to work on smaller tasks that can be completed quickly and abstracting the details of each compiled operator so they can be composed. PLD uses a streaming dataflow abstraction and a fast linking network to connect the separately compiled operators back into a functional design. When combined with program descriptions in C compiled by modern high-level synthesis tools, this provides a more familiar development experience for software engineers. They can refine C programs to run well on FPGAs through a series of incremental modifications to the original source. Fast compilation turns allow a fast edit-compile-debug loop, providing rapid feedback on program functionality. The compiler can produce successively more performant implementations over time, but the developer always has an executable version of the evolving program available for testing. PLD tools work with modern data-center FPGAs and interfaces to hide the low-level of details of FPGA CAD flows, microarchitecture, and interfacing, providing an interface closer to familiar compilation and linking for processors and GPUs.

ACKNOWLEDGEMENTS

This work is funded in part by a Google Faculty Research Award and the Office of Naval Research under grant N000141812557. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Google or the Office of Naval Research. Xilinx donated Vivado and Vitis tools for use in this work.

REFERENCES

[1] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming*

systems languages and applications. 89–108. https://doi.org/10.1145/1932682.1869469

[2] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained*. Addison-Wesley.

[3] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*. IEEE, 93–96. https://doi.org/10.1109/FCCM.2012.25

[4] Gordon Brebner. 1997. The Swappable Logic Unit: A Paradigm for Virtual Hardware. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 77–86. https://doi.org/10.1109/FPGA.1997.624607

[5] Pietro Bressana, Noa Zilberman, and Robert Soulé. 2020. Finding hard-to-find data plane bugs with a PTA. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 218–231. https://doi.org/10.1145/3386367.3431313

[6] Frederick P. Brooks, Jr. 1995. *The Mythical Man-Month: Essays on Software Engineering* (25th anniversary ed.). Addison Wesley Logman, Inc., Chapter 19.

[7] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems* 20, 9 (2001), 1059–1076. https://doi.org/10.1109/43.945302

[8] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. 2019. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1073–1086. https://doi.org/10.1145/3297858.3304066

[9] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. 2000. Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (LNCS)*. Springer-Verlag, 605–614. https://doi.org/10.1007/3-540-44614-1_65

[10] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[11] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10. https://doi.org/10.1145/2597917.2597929

[12] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 327–338. https://doi.org/10.1145/2967938.2967969

[13] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[14] James Coole and Greg Stitt. 2010. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 13–22. https://doi.org/10.1145/1878961.1878966

[15] Philippe Coussy and Adam Morawiec. 2010. *High-level synthesis*. Vol. 1. Springer.

[16] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream Computations Organized for Reconfigurable Execution. *Journal of Microprocessors and Microsystems* 30, 6 (September 2006), 334–354. https://doi.org/10.1016/j.micpro.2006.02.009

[17] Amazon EC2. 2017 (Accessed: 2020-11-16). *Amazon EC2 F1 Instances*. https://aws.amazon.com/ec2/instance-types/f1/

[18] Chris Fallin, Chris Craik, and Onur Mutlu. 2011. CHIPPER: A low-complexity bufferless deflection router. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 144–155. https://doi.org/10.1109/HPCA.2011.5749724

[19] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Press, 1–14. https://doi.org/10.1109/ISCA.2018.00012

[20] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweely, and Daniel Lopresti. 1991. Building and Using a Highly Programmable Logic Array. *IEEE Computer* 24, 1 (January 1991), 81–89. https://doi.org/10.1109/2.67197

[21] Google. 2021 (Accessed: 2021-08-10). *Deploying a Slurm cluster on Compute Engine*. https://cloud.google.com/architecture/deploying-slurm-cluster-compute-engine

[22] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. 2002. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 291–303. https://doi.org/10.1145/635508.605428

[23] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/3431920.3439289

[24] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/3020078.3021745

[25] Intel 2018. *AN 797: Partially Reconfiguring a Design on Intel Arria 10 GX FPGA Development Board*. Intel. https://www.altera.com/documentation/ihj1482170009390.html

[26] Christian Iseli and Eduardo Sanchez. 1993. Spyder: A Reconfigurable VLIW Processor using FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*. 17–24. https://doi.org/10.1109/FPGA.1993.279483

[27] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. 2016. DeCO: A DSP Block Based FPGA Accelerator Overlay with Low Overhead Interconnect. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 1–8. https://doi.org/10.1109/FCCM.2016.10

[28] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. 2020. SOFF: An OpenCL High-Level Synthesis Framework for FPGAs. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Press, 295–308. https://doi.org/10.1109/ISCA45697.2020.00034

[29] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IFIP CONGRESS 74*. North-Holland Publishing Company, 471–475.

[30] Gilles Kahn and David B. MacQueen. 1977. Coroutines and Networks of Parallel Processes. In *Proceedings of the IFIP CONGRESS 77*. North-Holland Publishing Company, 993–998.

[31] Nachiket Kapre. 2015. Custom FPGA-based soft-processors for sparse graph acceleration. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 9–16. https://doi.org/10.1109/ASAP.2015.7245698

[32] Nachiket Kapre. 2017. Deflection-routed butterfly fat trees on FPGAs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*. 1–8. https://doi.org/10.23919/FPL.2017.8056804

[33] Nachiket Kapre and André DeHon. 2011. VLIW-SCORE: Beyond C for Sequential Control of SPICE FPGA Acceleration. In *Proceedings of the International Conference on Field-Programmable Technology*. IEEE, 1–9. https://doi.org/10.1109/FPT.2011.6132678

[34] Nachiket Kapre and Tushar Krishna. 2018. FastTrack: Leveraging Heterogeneous FPGA Wires to Design Low-Cost High-Performance Soft NoCs. In *Proceedings of the International Symposium on Computer Architecture*. 739–751. https://doi.org/10.1109/ISCA.2018.00067

[35] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 107–127.

[36] Robert Kirchgessner, Alan D George, and Greg Stitt. 2015. Low-overhead FPGA middleware for application portability and productivity. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8, 4 (2015), 1–22. https://doi.org/10.1145/2746404

[37] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. 2012. VirtualRC: a virtual FPGA platform for applications and tools portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. 205–208. https://doi.org/10.1145/2145694.2145728

[38] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. 2013. An efficient FPGA overlay for portable custom instruction set extensions. In *2013 23rd international conference on field programmable logic and applications*. IEEE, 1–8. https://doi.org/10.1109/FPL.2013.6645517

[39] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report. University of Toronto.

[40] Charles Eric LaForest and Gregory Steffan. 2012. Octavo: an FPGA-Centric Processor Family. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 97–106. https://doi.org/10.1145/2145694.2145731

[41] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. 2021. Compiler-Driven FPGA Virtualization with SYNERGY. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 818–831. https://doi.org/10.1145/3445814.3446755

[42] C. Liu, H. C. Ng, and H. K. H. So. 2015. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *Proceedings of the International Conference on Field-Programmable Technology*. 56–63. https://doi.org/10.1109/FPT.2015.7393130

[43] Mateusz Majer, Jurgen Teich, Ali Ahmadinia, and Christophe Bobda. 2007. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 47, 1 (2007), 15–31. https://doi.org/10.1007/s11265-006-0017-6

[44] Théodore Marescaux, Vincent Nollet, Jean-Yves Mignolet, Andrei Bartic and W. Moffat, Prabhat Avasare, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. 2004. Run-Time Support for Heterogeneous Multitasking on Reconfigurable SoCs. *INTEGRATION, The VLSI Journal* 38, 1 (2004), 107–130. https://doi.org/10.1016/j.vlsi.2004.03.002

[45] Microsoft. 2021 (Accessed: 2021-8-1). *Microsoft Azure Goes Back To Rack Servers With Project Olympus*. https://azure.microsoft.com/en-us/

[46] Thomas Moscibroda and Onur Mutlu. 2009. A Case for Bufferless Routing in On-Chip Networks. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. 196–207. https://doi.org/10.1145/1555754.1555781

[47] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, USA, 416–429. https://doi.org/10.1145/3140659.3080255

[48] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: a Model of Computation for Intelligent Memory. In *Proceedings of the International Symposium on Computer Architecture*. https://doi.org/10.1109/ISCA.1998.694774

[49] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. 2018. Case for Fast FPGA Compilation using Partial Reconfiguration. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*. https://doi.org/10.1109/FPL.2018.00047

[50] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 389–402. https://doi.org/10.1145/3079856.3080256

[51] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2678373.2665678

[52] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 271–286. https://doi.org/10.1145/3297858.3304010

[53] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft Vector Processors with Streaming Pipelines. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 117–126. https://doi.org/10.1145/2554688.2554774

[54] D. Sheldon, F. Vahid, and S. Lonardi. 2007. Soft-core Processor Customization using the Design of Experiments Paradigm. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2007.364392

[55] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 639–651. https://doi.org/10.1145/3373376.3378495

[56] Ilian Tili, Kalin Ovtcharov, and J. Gregory Steffan. 2017. Reducing the Performance Gap Between Soft Scalar CPUs and Custom Hardware with TILT. *ACM Transactions on Reconfigurable Technology and Systems* 10, 3, Article 22 (June 2017), 23 pages. https://doi.org/10.1145/3079757

[57] Ramshankar Venkatakrishnan, Ashish Misra, and Volodymyr Kindratenko. 2020. High-Level Synthesis-Based Approach for Accelerating Scientific Codes on FPGAs. *Computing in Science & Engineering* 22, 4 (2020), 104–109. https://doi.org/10.1109/MCSE.2020.2996072

[58] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé Touati, and Philippe Boucard. 1996. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems* 4, 1 (March 1996), 56–69. https://doi.org/10.1109/92.486081

[59] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/3431920.3439292

[60] Qiang Wang and David Lewis. 1997. Automated Field-Programmable Compute Accelerator Design using Partial Evaluation. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 145–154. https://doi.org/10.1109/FPGA.1997.624614

[61] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. 2014. Embedding FPGA overlays into configurable systems-on-chip: ReconOS meets ZUMA. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, 1–6. https://doi.org/10.1109/ReConFig.2014.7032514

[62] Claire Wolf. 2021 (Accessed: 2021-08-10). *PicoRV32 - A Size-Optimized RISC-V CPU*. https://github.com/cliffordwolf/picorv32

[63] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, David A. Patterson, and Anthony D. Joseph. 2019. FPGA Accelerated INDEL Realignment in the Cloud. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 277–290. https://doi.org/10.1109/HPCA.2019.00044

[64] Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon. 2020. Fast Linking of Separately-Compiled FPGA Blocks without a NoC. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 196–205. https://doi.org/10.1109/ICFPT51103.2020.00035

[65] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, and André DeHon. 2019. Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks. In *Proceedings of the International Conference on Field-Programmable Technology*. 153–161. https://doi.org/10.1109/ICFPT47387.2019.00026

[66] Xilinx, Inc. 2020. *UG1145: Xilinx Vitis Unified Software Platform User Guide*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1145-sdk-system-performance.pdf

[67] Xilinx, Inc. 2021. *UG1120: Alveo Data Center Accelerator Card Platforms*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1120-alveo-platforms.pdf

[68] Xilinx, Inc. 2021. *UG909: Vivado Design Suite User Guide: Dynamic Function eXchange*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf

[69] Xilinx, Inc. 2021. *UG947: Vivado Design Suite Tutorial: Dynamic Function eXchange*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug947-vivado-partial-reconfiguration-tutorial.pdf

[70] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. 2007. Exploration and Customization of FPGA-Based Soft Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007 2007), 266–277. https://doi.org/10.1109/TCAD.2006.887921

[71] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. 2012. Portable, Flexible, and Scalable Soft Vector Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 8 (2012), 1429–1442. https://doi.org/10.1109/TVLSI.2011.2160463

[72] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–858. https://doi.org/10.1145/3373376.3378491

[73] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 123–134. https://doi.org/10.1145/3445814.3446699

[74] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 269–278. https://doi.org/10.1145/3174243.3174255