Dataflow Incremental Refinement for C for Gently Migrating to High-Performance FPGA Accelerators

ABSTRACT

With the rise of high quality C-to-gates HLS compilation, it is becoming viable to craft high-performance FPGA-accelerated designs in C. However, just as arbitrary Verilog is not synthesizable or efficient, arbitrary C won't necessarily produce efficient and high performance FPGA accelerators. Nonetheless, it should be possible to incrementally refine an application written in C to tune it properly for HLS to FPGA mapping. While incremental refinement with frequent recompilation and testing is a solid strategy for software development, slow (hour long), monolithic FPGA compilation discourages or prevents rapid, incremental development and tuning. We present techniques and methodologies to allow incremental refinement of dataflow-coordinated C computations into efficient FPGA-mapped computations. In particular, we show how supporting a common stream interface on top of separately compiled and linked dataflow operators allows operators to run on either FPGA logic or embedded cores on the Zynq UltraScale+ MPSoC. C can be compiled to the embedded cores quickly (<10 seconds) with no limitations on operator sizing, allowing continuous, rapid validation of functional refinements. As operators are refined and sized for the FPGA logic, they can be incrementally moved to the FPGA with short (<5 minute) compiles, even if parts of the application are still running on embedded cores. We illustrate how this allows developers to refine applications in the Rosetta Benchmark Suite to high performance, dataflow-mapped designs in a short series of incremental refinement steps, each of which compiles and executes quickly.

KEYWORDS

FPGA, Compilation, Streams, Dataflow, Latency Incensitive, Debug

ACM Reference Format:

. 2021. Dataflow Incremental Refinement for C for Gently Migrating to High-Performance FPGA Accelerators. In *Proceedings of ACM ISFPGA (FPGA'2021).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

How do we develop and refine software? We typically start with something minimal that works. We add functionality incrementally, recompiling and testing as we go. Recompilation of software is fast, so the time to recompile between changes is almost unnoticeable. All of our time is spent on development and testing. When we have something functional that works, we benchmark to identify bottlenecks and further refine the code incrementally to improve performance. We build and accelerate through many small changes. We typically have a working piece of software all the time. With the fast edit-compile-debug loop, we can add probes for debugging and timing. After relating the challenges of software development in the original Mythical Man Month, in the 25th Anniversary edition, Brooks reports the one real advance he's noticed is the method for incremental refinement [3]. Modern agile software development promote evolutionary development and continual improvement.

Unfortunately, slow compiles, which can run into hours, make our FPGA accelerator development strategy largely the opposite. Since the edit-compile-debug loop is long, we are encouraged to minimize rebuilds. It is expensive to add targeted probe points for inspection and visibility as we debug. If the design doesn't fit on the hardware, it simply cannot run, so we must first get the design sized to fit before we can test functionality. This lengthens the development time and discourages the addition of features and the aggressive design-space exploration necessary to find the most efficient mappings. This raises our key question: *How can we create a more software-like experience for FPGA accelerator development?*

We can close this gap by strategically using today's SoC FPGAs that include hard-core processors and today's HLS compilation that can compile C to FPGA logic. We can compile operators in C quickly to the embedded processors and migrate them to the FPGA as time permits. With the appropriate discipline (Secs. 5.1.1, 6), operators can be retargeted between the FPGA and the embedded processors by simply changing the target in a header file. To enable this, we develop a common, latency-insensitive stream interface between C-defined operators that functions the same regardless of whether the source and sink are both on processors, both on the FPGA fabric, or one is on a processor and the other is in the FPGA fabric (Sec. 5). This allows us to freely move functionality between the processor cores and the FPGA without changing the operator source code. Combined with an overlay NoC, we can retarget functions to the embedded core without performing any FPGA compilation. Further combined with fast incremental compilation of partial-reconfiguration regions on the FPGA [16], suitably-sized operators can be moved onto the FPGA fabric in minutes.

Together, this enables a path for dataflow incremental refinement of C (DIRC) code that allows the developer to gently[1] migrate applications from pure software to hybrid- or pure-hardware running on the FPGA. Development can proceed like software, retaining the ability to make quick changes and always run the code after every change. If the operator is too big to fit in a target partial reconfiguration region, the developer can refactor the operators to split the operation into multiple operators. If the operator becomes the performance bottleneck, the developer can replicate the operator to increase parallelism. These changes can all be performed as incremental refinements to the operator C graph; graph nodes

FPGA'2021, February 2021, Monterey, CA USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

[1] Dirk Gently is a "Holistic Detective" in Douglas Adams' novels [1, 2] who believes in the "the fundamental interconnectedness of all things."

can always runs on the embedded processors with some nodes running on the FPGA as the operators are sized appropriately. This methodology provides the missing `-O0` compilation strategy, familiar in software that offers a quick, but possibly low-performance, compilation to allow testing and debug to proceed rapidly. It also provides the separate compilation and linking familiar from software development. Sec. 4 illustrates how this methodology can be applied to refine one application from the Rosetta Benchmark Suite. We further show that the ability to remap streams and operators to the embedded processor core enables more software-like instrumentation and debugging (Sec. 7).

We make the following contributions:

- Engineer a high performance stream implementation that provides uniform, latency-insensitive communication between operators whether they reside on an embedded core or the FPGA fabric (Sec. 5)
- Articulate a discipline for dataflow operators that can migrate between software and hardware (Sec. 6)
- Introduce a methodology for incremental refinement of dataflow C programs that builds on this stream implementation (Sec. 4).
- Characterize the compile times and performance achievable with this methodology (Sec. 8).

We provide an Open Source release for our engineered stream implementations. [ideally whole methodology]

2 BACKGROUND

2.1 Dataflow Composition Model

We use a streaming dataflow computational model based on Kahn Processing Networks [8] similar to other reconfigurable architectures [4, 5]. Basic kernel computations are described in C as operators that receive inputs over latency-insensitive streams and produce outputs to latency-insensitive streams. The stream connections link the operators into a computational graph.

2.2 Heterogeneous SoCs

Today's System-on-a-Chip (SoC) FPGAs embed hard core processors along with a traditional FPGA fabric, including Xilinx Zynq, Intel Arria and MicroSemi PolarFire. These include high-speed AXI channels to connect the processor cores and the FPGA fabric. Additionally, vendors support softcore processors on the FPGA fabric (NIOS, uBlaze, and RISC-V).

2.2.1 Zynq UltraScale+ MPSoC. We demonstrate this work on the Zynq UltraScale+ MPSoC [19]. This includes 4×A53 64b ARM core processors and 2×R5 32b "Real Time" ARM core processors. The MPSoC Zynq includes an explicitly managed scratchpad On-Chip Memory (OCM) as well as a cache-hierarchy with L1 and L2 caches for the A53 processors. Nine AXI slave channels and 3 AXI master channels provide interconnect to the FPGA fabric, with 3 channels providing I/O coherence and 1 providing full coherence with the A53 processor hiearchy. Each AXI channel is 128b wide and can operate up to 333 MHz, providing a peak bandwidth ×128b = 5.3 GB/s each direction. Xilinx calls the traditional FPGA fabric, the Programmable Logic (PL) portion of the MPSoC, and the processor cores and peripherals the Programmable System (PS).

2.3 Processor Compilation

Compilation for processor 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 place-and-route 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.

2.4 Out-of-Context and Partial Reconfiguation Compilation

Park [10] and Xiao [16] demonstrated that the separate compilation and linkage idea could be applied to FPGA compilation. They showed that, by dividing the FPGA into separate, small partial reconfiguration regions with standard interfaces, hardware pages, it was possible to compile operators independently for each region, reducing the compile time compared to monolithic compilation of the entire FPGA. To link modules without further compilation, they use a lightweight, packet-switched Hoplite Butterfly Fat Tree (BFT) Network-on-a-Chip (NOC) [9].

While they accelerated compile times, they still saw compiles that took 5–10 minutes. Furthermore, their 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.

2.5 Software-First Methodology

Our approach matches the software-first methodology embraced by GRVI-PHALANX [6, 7], Soft Vector Processors [13], and other refinement methodologies [14]. We provide a single dataflow model that works on pure processors and allows the code to be refined for *automatic* translation to FPGA hardware. While we do require some sizing and shaping of the C code, once sized, no RTL (e.g. Verilog) is required to migrate computation to FPGA-based accelerators.

2.6 Just-in-Time Compilation

Recent work in Just-In-Time (JIT) compilation for Verilog simulates Verilog code on a processor while FPGA compilation occurs, moving the design to the FPGA once an FPGA configuration is generated [12]. We support compilation from C and allow parts of the application to run directly on hardware. Significantly, we introduce a stream interface here that allows high-performance interaction between parts of the design running concurently on the processor and the on the FPGA. A JIT framework is a natural consumer for the techniques we enable.

Seiba also supports processor integration with compiled FPGA logic to accelerate the edit-compile-debug loop [15]. Rather than accelerating the initial development and iterations before the design is suitable for hardware mapping, Seiba supports iterative improvement after the design has been initially mapped. Seiba's model is of

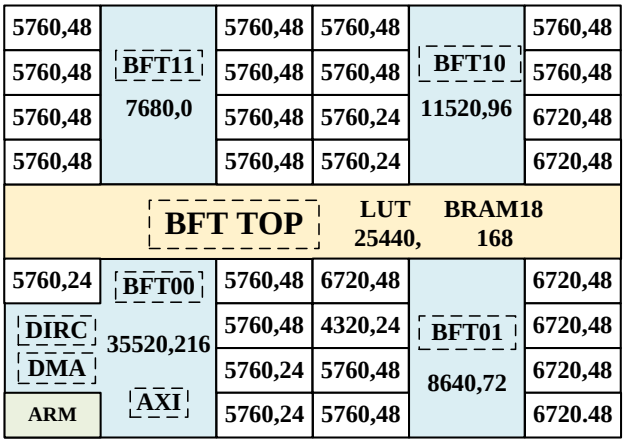

Figure 1: XCZU9EG 30-Page Overlay Design for Separate Compilation and Linkage

serial intraction between the FPGA logic and the processor rather than concurrent operation. WIth our stream support and concurrent composition of operators, the techniques are complementary and once a design is mapped to a page, one could use Seiba's technique within a page.

2.7 Emulation

Xilinx supports emulation where the processor is emulated with QEMU and the logic for the FPGA fabric is simulated at the RTL level [17, 18] . This allows the application to run without waiting for expensive logic synthesis, placement, and routing, but still requires VivadoHLS compilation to compile the C code down to RTL for the RTL simulator. It provides high visibility into the FPGA-accelerated logic, but runs at the slow speed of RTL simulation. Our techniques provide higher throughput computations and shorter time from source-code change to running test.

3 EXPERIMENTAL SETUP

Unless otherwise noted, we use Xilinx SDSoC 2018.2 including the associated Vivado and SDK. We target a ZCU102 board that includes a UltraScale+ Zynq XCZU9EG FPGA (274,080 LUTs, 912 36Kb BRAMs). Our compilations are performed on a 2.7 GHz Intel E5-2680 CPU with 128 GB RAM .

We divide the logic on the XCZU9EG into 30 user logic pages, 5 packet-switched interconnect pages, one page for DMA, and one page for PS↔PL streams as shown in Fig. 1.

4 EXAMPLE DEVELOPMENT FLOW

To illustrate the dataflow refinement, we use the Optical Flow benchmark as given in the Rosetta Benchmark Suite [21]. Their Optical Flow starts as a set of dataflow, stream-connected functions, so it is no large leep to thinking about the dataflow in the design. As given, the design has nine operators (unpack, gradient_xy, gradient_z, weight_y, weight_x, outer_product, tensor_y, tensor_x, compute_flow).

We can initially get this running as operators mapped to the A53 cores communicating with streams. These initial 9 operators take [range of times] to compile and run at XX frames per second. We can also run HLS compilation on each operator taking [range]. From this, we see that operators [list operator and resource requirements] are too large for our overlay that has pages with at most 6,000 LUTs and 30 BRAMs in a page. The other operators can be mapped to physical page in 600 s using partial reconfiguration mapping [16].

For each of the operators that is too big, we can look at splitting it. [describe what need to do on each – callout things like independence of x and y computations in weight_xy allowing that to be split into separate operators]. As we split each operator, we can compile it quickly in [time] s and run the split design on the A53 cores along with the already FPGA-page-mapped operators to validate functionality. [maybe more callout of individual operator optimization and moves]

After decomposing the operators, we have 12 operators that all fit on the pages on our overlay, and the entire design runs on FPGA logic. The user logic runs at 200 MHz. The design achieves YY frames per second. While all the operators are tuned to processing one pixel per cycle (Iteration Interval of 1) for their computations, the [which] operator(s) require XXX bits of input for [what data] on each iteration. Since the input to each page is only 32b at 300 MHz, the input data is the bottleneck. We can further split [operators] into multiple pages by [how] to reduce the magnitude of the bottleneck. We test the logic of the split on the A53 cores with 30 s compiles, then move them to the hardware with 600 s page compiles. As a result, we are finally able to run the design at ZZ frames per second. This is still half the speed of the monolithic design due to the remaining input bandwidth bottleneck into the pages.

As a result, we were able to refine the 9-operator design that did not fit on the fixed-sized pages of the overlay into a 14-operator design that did in 10 incremental steps where we changed one operator at a time and were able to test the functionality with each operator change. This required 10 sequential processor compiles taking a total of ZXY seconds, and 3 sequential FPGA page compiles taking a total of 3×600=1800 s. At the end of this, we could perform a single monolithic compile without the overlay in 2,400 s. If we had to perform monolithic compiles for each of the ten steps in the revision, it would require 24,000 s (almost 7 hours) for the compile time alone. Even using only the separate-compilation page mapping, the ten sequential steps would take over an hour and a half of compile time.

5 ENGINEERING SOURCE-SINK AGNOSTIC STREAMS

5.1 Goals and Requirements

To make migration easy, we develop streams with functionally identical behavior regardless of where the source and sink lies. The code should work if we have an ARM core on the PS sending data to FPGA logic on the PL, PL sending to PS, PL sending to PL, or PS sending to PS. To move an operator from one side to the other, we would like to make minimal to no changes in the source code. We settle on changing only definitions in a header (.h) file that could be replaced by using a different set of includes for PS-targeted operators and PL-targeted operators. Of course, we also want high

performance out of the streams, sacrificing no performance for the PL↔PL connections, and achieving high enough bandwidth from the PS↔PL connections that the stream is not the bottleneck on performance. We will later see (Sec. 8) that a few hundered megabytes per second is the typical limit of what the ARM cores on the UltraScale MPSoC Zynq can use.

5.1.1 Interface. To support our ease of conversion goals, we design a `pr_flow::stream` stream interface for the software side that is compatible with the Vivado `hls::stream` interface [20]. This way, PL↔PL connections can continue to use `hls::stream`, and operators can use the same read and write operations regardless of whether we actually instantiate `hls::stream` or our own `pr_flow::stream`. We provide abstraction macros for stream declaration (STREAM), and stream read (STREAM_READ) and write operations (STREAM_WRITE), so the operator code does not need to change when migrated between hardware and software. STREAM declaration takes and argument indicating the width of data used with the stream. As with `HLS::stream`, simple read and write operations are blocking, providing a latency-insensitive interface.

5.1.2 Why Not DMA?. Standard DMA operations take tens of thousands of cycles to setup. They can be efficient when moving large blocks of data. They are not efficient when operators operate on just a few data items at a time. For purely feed-foward operator graphs, large-scale batching of operations is viable, but introduces the need to identify the size of the data block being transfered, which is a divergence from the simple `hls::stream` interface. For operator graphs with cycles, the latency around the cycle may be a bottleneck for performance; batching will prevent dataflow parallelism and reduce throughput.

5.2 Basic Stream Design

On the software side, the `pr_flow::stream` implements a FIFO in shared memory that can be accessed by the processor cores and the PL. We use memory (main memory, on-chip memory) accessible to the processor cores to take advantage of the low latency and high throughput provide for the cores to access on-chip memory (both the OCM (On-Chip Memory) and caches local to the PS). We use a standard ring-buffer FIFO design with head and tail pointers also maintained in shared memory. Since we use these PS-accessible on-chip memories, the PL must use a slave AXI port to access the shared pointers and data. PS↔PS streams reduce to a shared-memory FIFO and do not consume bandwidth on AXI ports.

Placing the head and tail pointers in shared memory has the disadvantage that, in the worst-case, on every FIFO read and write operation, there is a need to read one pointer (head to make sure there is data to read on a read, tail to make make sure there is space to write on a write) and write another (update tail on a read, update head on a write). In the simplest case this cuts the raw bandwidth in the direction of the stream flow in half. One advantage of this design is that the pointers do not need to be read for every write or read operation for the typical case, only when their previous values might indicate a full or empty FIFO, reducing the throughput impact compared to the worst-case scenario. For example, if the producer reads a tail pointer that is already 16 ahead of the head pointer, it knows it can perform 15 writes before it needs to read the tail pointer again.

We also explored the alternative of using data-presence bits. While the data presence bits worked well on the PL side where we could use the extra parity bits to add data presence onto a 32b or 64b word without requiring extra BRAMs, the fact that the processors only see data in fixed chunks and the AXI channels move 128b-wide data made the data-presence scheme inefficient for PS-side interactions.

5.2.1 Shared Memory Requirements. It is worthwhile to note that this scheme does not need the full capabilities of symmetric shared memory. In particular:

- The head pointer is exclusively written by the consumer.
- The tail pointer is exclusively written by the producer.
- Data in the stream is exclusively written by the producer.
- Data in the stream is exclusively read by the consumer.

This means that:

- There is nothing that must be written by both sides.
- The producer can keep a local copy of the tail pointer, and the consumer can keep a local copy of the head pointer
  - they do not need to prepared for anyone else to change the values
  - they do need to share their updates
- It is functionally acceptable for the updates to the head and tail pointer to be delayed before seen by the reader, as long as they occur **after** the data has been written or read.
  - There is a need to make sure the data operation and associated head or tail pointer update occur in order.
  - This also allows the producer or consumer to write a collection of data and perform a single update to the head or tail pointer, reducing the number of update writes and, thereby, reducing overhead throughput.
- There are no other ordering requirements; in particular there are no ordering requirement among different streams.

Among other things, this means we are not forced to use the ACE AXI channel for communication. The ACP and HPC ports provide adequate functionality in *outer-shareable* mode to share data in the L2 cache.

5.3 Implementation Options

Having selected the ring-buffer FIFO design, we still have a variety of options for how we implement the PS→PL communication. In particular, we have a choice of which memories to use (OCM, L2-caches, DRAM) and which slave AXI ports to use. First, we characterize the peak, raw read and write performance to each of the memories through each class of AXI port as shown in Tab. 1. We perform a tight loop of read (or write) operations the maximum datawidth allowed (32b for R5, 64b for A53, 128b for PL fabric) with the PL running at 300 MHz as shown in App. A. This shows the PS can get high bandwidth from the L2 cache and has an odd asymmetry between the OCM read and write bandwidth. The PL achieves high bandwidth accessing the L2 over the ACP port, but also has decent bandwidth accessing the OCM over an HP port.

Even in a tight loop the PS ARM core must issue 7 instructions per read or write operation (See Listing 1). As a result, many of

Table 1: Peak Raw Read and Write Bandwidth

Unit	Case (PL AXI)	Results in MB/s DRAM r	DRAM w	OCM r	OCM w	L2 r	L2 w
PS: A53	64b	23	18	83	625	720	620
PS: A53	unroll 64b	67	260	91	4900	3750	5000
PS: A53	vector 128b	140	94	180	2100	2900	2100
PS: R5	32b	—	—	77	55	—	—
PS: R5	unroll 32b	—	—	120	80	—	—
PL	HP 128b	460	470	550	440	–	–
PL	HPC 128b	220	380	270	250	200	380
PL	ACP 128b	310	280	–	–	810	710

PL running at 300 MHz. Unroll cases shown for an aggressive unroll factor for 32.

Table 2: Peak Unidirectional Stream Throughput

Units 1	Units 2	AXI	Results in MB/s OCM 1→2	OCM 2→1	L2 1→2	L2 2→1
A53	A53		110		435	
A53	R5		39	29	—	—
R5	R5		32		—	—
A53	PL	HP	95	120	—	—
A53	PL	HPC	95	120	73	130
A53	PL	ACP	—	—	460	360
R5	PL	HP	34	41	—	
R5	PL	HPC	34	41	—	

A53 cores use vector operations to transfer 128b data; buffers are sized to hold 512 64b words. PL running at 300 MHz. Final 2 columns show TCM for R5 and L2 for A53 cores.

Listing 1: ARM -O3 Assembly for Tight Write Loop

```
1  .L3:
2          uxtw    x2, w3
3          add     w3, w3, 1
4          ldr     x4, [x1], 8
5          and     w3, w7, w3
6          str     x4, [x6, x2, lsl 3]
7          cmp     x1, x5
8          bne     .L3
```

Table 3: Loopback Throughput

Loopback Case	Mem.	AXI	Thput (MB/s)
A53→PL→A53	L2	ACP	360
PL→A53→PL	L2	ACP	310
A53→PL→A53	OCM	HP	110
PL→A53→PL	OCM	HP	51
R5→PL→R5	OCM	HP	32
PL→R5→PL	OCM	HP	41

A53 cores use vector operations to transfer 128b data; buffers are sized to hold 512 64b words. PL running at 300 MHz.

the PS side interactions are bound by processor issue cycles not by memory bandwidth. We can reduce the loop overhead by unrolling the communication loops. We can alternately use the vector operations on the A53 NEON units to issue 128b-wide read and write operations from the processor. Tab. 1 also shows the impact of these optimizations on raw memory bandwidth. This shows that the PL bandwidth in the simple tight loop test was limited by instruction issue and that can be overcome by unrolling to achieve bandwidth into the GB/s. Vector operations moving 128b data to and from the L2 cache recover some of the lost bandwidth and exceeds the bandwidth achieved by the PL. The unrolled write operations provide a large throughput gain for the PS cores accessing OCM, but the read bandwidth remains largerly unchanged. Vector operations double the read bandwidth. This suggests the read operations are limited by the architecture and not by inefficient instruction issue.

The throughput of a stream will depend on both the throughput of the producer and the consumer. Furthermore, as previously noted, stream read and write operations will require both the raw reads and writes of data and the reads and write of head and tail pointers which will consume some of the raw communication bandwidth. Tab. 2 shows the peak unidirectional stream throughput for each of the cases using buffers of length 512 64b words. Here, we perform a tight loop of stream operations using 128b NEON vector operations (A53) or 32b operations (R5) as shown in App. A We see the ACP port sharing data in L2 is the fastest at 350 and 450 MB/s; this is a little over half the raw bandwidth available between the PL and the L2 cache.

Finally, we show loopback test performance with the identity operator running on the opposite side of the PS-PL interface in Tab. 3. This directly models the case where input and output stream bandwidth is roughly comparable for an operator. This achieves a little below the minimum of the directional bandwidths on interacting pair over a port.

5.4 Composing Streams

We also have several choices when supporting multiple PS↔PL streams. The simplest solution might be to use an AXI channel for each such stream. However, the FPGA has a limited number of AXI channels and only one ACP AXI channel, which we found provided the highest throughput for our streams. Furthermore, with our peak streams running at 460 MB/s, no stream will saturate the 5.3 GB/s capacity of each AXI channel (Sec. 2.2.1). That means we should be able to share AXI channels without sacrificing perfomance. Furthermore, since there are only four A53 cores, the largest total bandwidth the processing cores can sustain is 1.8 GB/s.

Sharing a single AXI channel with a separate PL-side interface (IP Block) for each stream has a high LUT and BRAM cost per stream as shown in Tab. 4. Note the the table reports the per stream bandwidth; in the 4 stream cases all four cores are sending data at the identified peak bandwidth. Each interface costs around 5,000 LUTs [looks more like 10,000] for logic plus around 1,000 LUTs to add a port to an AXI crossbar. Alternately, we can develop a single PL-side interface that serially processes data from each stream in

Table 4: Multiple Stream Implementation Options

Streams	AXI Chans.	PL IPs	Thrupt (MB/s) PS→PL	PL→PS	Resources LUTs	BRAMs
1	1 ACP	1	460	360	7,300	10
1	1 HP	1	95	120	7,400	10
BFT 1	1 ACP	1	120	170	7,300	10
4	4 HP	4	95	120	43,000	40
4	1 HP	4	95	120	40,000	40
4	1 HP	1	95	96	13,000	12
4	1 ACP	4	342	200	40,000	40
4	1 ACP	1	220	200	12,000	12
BFT 4	1 ACP	1	130	110	12,000	30
BFT 6	1 ACP	1				

All data here for A53 cores using vector operations to transfer 128b data. For the cases marked BFT, the PL-side operator is a leaf page accessed over the BFT. Other cases, the PL-side operator is directly connected to the AXI channel with no BFT in between. Buffers are sized to hold 512 64b words. PL running at 300 MHz.

round-robin fashion (PL IPs of 1 in Tab. 4). Again, since the bottleneck is PS bandwidth, this serialization does not have a large impact on the throughtput of individual streams.

Note that the A53 cores are only achieving the peak bandwidth of 360 and 460 MB/s in tight data send and receive loops. If the operator on the core does anything else, the throughput will drop as the core is the bottleneck. Simply using the core to copy between streams cuts the transfer rate in half.

5.5 Integration with BFT

The PL side of the interface is connected to one or more ports on the overlay NoC that connects the FPGA leaf pages. The 32b, 300MHz BFT provides a raw bandwidth of 1.2 GB/s per leaf port, which, will typically exceed the bandwidth driven by all cores on the PS. The PL interface IP connects to the BFT leaf interface with AXI-S interfaces per stream, similar to any other PRflow user logic. An entire leaf page supporting 6 PS→PL and 6 PL→PS streams requires 5000 LUTs and 20 BRAMs, comparable to a typical PRflow leaf page. As Tab. 4 shows, the BFT interface provides no additional bottleneck, supporting the full throughput of the streams.

5.6 Result

As a result, we are able to achieve over 300 MB/s per stream per direction between the A53 cores and the PL with all four A53 cores operating simultaneously, using a shared L2 cache and sharing the single ACP AXI channel. We can achieve over 30 MB/s per stream each direction for the R5 cores using the OCM and a single HP channel.

6 OPERATOR DISCIPLINE

There is some discipline required to designing operators that are mutually compatible with the processor cores and the PL fabric, particularly when there are more operators assigned to the PS than cores available. Our discipline includes:

- All communication is through streams, using the STREAM, STREAM_READ, STREAM_WRITE operations from Sec. 5
- Operators are limited to 7 input streams and 7 output streams; this is not fundamental, but a particular system will need to pick some MAX_STREAMS to support in the hardware design. This limit impacts both the packet headers for the BFT and the buffers and control allocated for the PL side IP.
- Operators run finite loops inside the operator that are restarted by an outer loop or continuously enabled in hardware; particularly when there are more operators than cores, this guarantees no operator monopolizes a processor.
- Operators should obey standard HLS prohibitions such as no allocation or recursion; if you want to exploit processor-only operations when they are on the process, like `print`, it should be guarded by suitable `ifdef` software macro guards

When used with a full, preemptive schedule, such as when running Linux on the PS, there are no further restrictions. When running on bare metal with more PS operators than cores, care will be required in assigning operators to shared cores and ordering the execution of operators on a single core to guarantee producers run before consumers with adequate stream buffer space and appropriate token production and consumption in each invocation. In bare metal when sharing cores, an atomic invocation of an operator will not yield when stalled on streams, which could deadlock the system.

Listing 2 shows our common C++ source for `outer_product1` in the refined Optical Flow Rosetta Benchmark.

7 DEBUGGING

The ability to dynamically link in processor-mapped operators enables rapid inspection and debugging.

Initially, it allows us to move a single operator over to hardware for testing. The stream links allow the software to feed the operator data and consume its results without additional harness setup to run the operator.

If we have a mapped design where we encounter new bugs, we can move an operator back to software for inspection and add `print` statements just as we might use in software debugging. Note this is done without recompiling any hardware, since the connectivity over the PS NoC is configured with configuration messages over the NoC and does not require the recompilation of the neighbors of the operator moved to software. Similarly, it is possible to interpose an operator in an existing stream link by configuring the producer to send its data to the interposed operator and the interposed operator sending its outpt to the original consumer. This interposed operator can then be programmed to perform inspection and instrumentation, such as counting data tokens or printing them out. The interposed operator can perform the role of an integrated logic-analyzer (e.g., [**?**][Chapter 10–11]), again without the need to recompile the entire design. The interposed page can also act as a larger data buffer. These token counts and stream inspection are often useful in debugging deadlock scenarios that arise in dynamic dataflow streams (e.g. [11]).

On processors, for serious debugging, we will single-step through the programs one instruction at the time, potentially inspecting the program state at each step. For parallel, dataflow processing, a

Listing 2: Sample Operator

```
void outer_product1(STREAM & Input_1,
                    STREAM & Input_2,
                    STREAM & Input_3,
                    STREAM & Output_1)  {
#pragma HLS interface ap_hs port=Input_1
#pragma HLS interface ap_hs port=Input_2
#pragma HLS interface ap_hs port=Input_3
#pragma HLS interface ap_hs port=Output_1
OUTER_OUTER: for(int r=0; r<MAX_HEIGHT; r++)
  {
    OUTER_INNER: for(int c=0; c<MAX_WIDTH; c++)
    {
      #pragma HLS pipeline II=1
      gradient_t grad;
      databus_t temp_x,temp_y,temp_z;
      temp_x =STREAM_READ( Input_1);
      temp_y =STREAM_READ( Input_2);
      temp_z =STREAM_READ( Input_3);
      grad.x.range(31,0) = temp_x.range(31,0);
      grad.y.range(31,0) = temp_y.range(31,0);
      grad.z.range(31,0) = temp_z.range(31,0);
      outer_pixel_t x = (outer_pixel_t) grad.x;
      outer_pixel_t y = (outer_pixel_t) grad.y;
      outer_pixel_t z = (outer_pixel_t) grad.z;
      outer_t out;
      out.val[0] = (x*x);
      out.val[1] = (y*y);
      out.val[2] = (x*y);
      widebus_t wtmp;
      wtmp.range(47,0)=out.val[0].range(47,0);
      wtmp.range(95,48)=out.val[1].range(47,0);
      wtmp.range(143,96)=out.val[2].range(47,0);
      wtmp.range(287,144) = 0;
      bit160 out_tmp;
      out_tmp(159,0) = wtmp.range(159,0);
      STREAM_WRITE(Output_1, out_tmp(31,0));
      STREAM_WRITE(Output_1, out_tmp(63,32));
      STREAM_WRITE(Output_1, out_tmp(95,64));
      STREAM_WRITE(Output_1, out_tmp(127,96));
      STREAM_WRITE(Output_1, out_tmp(159,128));
    }
  }
}
```

logical equivalent is single token-debugging. We can send a single token input into the dataflow-connected operators, let the dataflow computation settle, and inspect the state associated with the operators. A processor core operator can drive the single-token stepping. The state of a processor-implemented operator can similarly be inspected. In the worst-case, partial-reconfiguration readback can be used to extract the state of each register or BRAM [] between single-step input tokens.

8 BENCHMARK EVALUATION

We map the full set of designs from the Rosetta Benchmark Suite [21] and report the results in Tab. 5. We include incremental compilation time and throughput for several cases.

Tab. 5 shows clearly how much faster operators can be mapped to processor cores than to the full FPGA, separated FPGA pages, or even emulation. Monolithic compile time and throughput is the standard flow mapping the complete design to the FPGA with the Vivado tools. This takes from 20 minutes (1150 s) to over almost 3 hours (9790 s). Using the incremental mapping and linking form Xiao [16] individual operators can be mapped to pages can be mapped in 10–14 minutes (600–853 s), and the operators can be mapped in parallel. Compiling to software operators for the A53 core takes seconds (1–3 s), and the operators can be compiled in parallel. Compilation for emulation takes 5–6 minutes (292–388 s), three orders of magnitude slower than compiling the operator for an A53 core.

To illustrate the use in incremental debugging, we quantify the performance for the cases where a single operator is mapped to an A53 core with the rest running on separately compiled and linked FPGA pages. While the A53 core has lower throughput than the full design mapped to FPGA logic, it still runs orders of magnitude faster than mapping the full design to software or using emulation. We also see the full design running only on A53 cores has higher throughput than emulation.

if opportunity, callout example where ok to leave something in sw

Maybe need to callout examples that include ap_int and compile slower to processors than those that do not? (and/or comments about replacing ap_int)

In the final column we show the aggregate throughput required between the PS and PL when the identified page is is running in software. For simplicity, we show the max of the PS→PL and PL→PS throughputs. This shows that streams we engineered (Sec. 5) supporting over 100 MB/s on each of 6 streams is never the performance bottleneck. Rather, throughput limits are set by the computation that must be implemented serial on each A53 processor core between data consumption and production.

9 FUTURE WORK

An obvious extension is to add softcores to the network as fast processor core compilation targets in addition to the embedded cores [6**?**]. This would extend the methodology to FPGAs without embedded hardcore processors and reduce or eliminate the need to be concerned with operator scheduling due to core sharing (Sec. 6). It would be natural to place a softcore operator in a partial reconfiguration FPGA pages until the operator can be mapped to the FPGA logic then reconfigure the page in place to upgrade to the custom, FPGA-mapped logic.

With this discipline in place, it becomes possible to develop a next level of automation. Optimization can explore automated operator splitting and replication and selection of operator mapping

Table 5: Page Mapping Times and Performance Impact of Single Page in Software

Benchmark	App. Thput case	Thput	Page Name	Compile Time (s) PRflow	Compile Time (s) page in SW	Thput page on A53	Total Thput Streams to SW
Rendering	Monolithic (compile)	1150s	data_redir	773	1.41	42 f/s	2 MB/s
	(Thput)	413 f/s	rasterization	853	2.66	48 f/s	14 MB/s
	PRflow all HW		zculling_{top, bot}	774, 774	1.56, 1.52	138, 95 f/s	17, 16 MB/s
	(Thput)	1173 f/s	coloringFB_{bot,top}	772, 743	1.41, 1.41	206, 242 f/s	28, 29 MB/s
	Emulation (compile)	388s					
	(Thput)	1 f/548s					
	All A53 cores						
	(Thput)	10 f/s					
Spam Filter	Monolithic (compile)	1200 s	dotProduct_[1-8]	678–741	4.37–4.53	0.13 i/s	1.3 MB/s
	(Thput)	8 i/2	data_1_4_[1–4]	668–700	2.79–3.37	3–8 i/s	0.7–19.8 MB/s
	PRflow all HW		add_4_1_[1–2]	688–733	2.9–3.1	8 i/s	0.7 MB/s
	(Thput)	20 i/s	add_2_1	666	2.6	10 i/s	0.46 MB/s
	Emulation (compile)	235s	Sigmoid_axi	665	8.18	9 i/s	0.42 MB/s
	(Thput)	1 /6h	data_4_1_[1–2]	697–704	2.8–2.9	20 i/s	10 KB/s
	All A53 cores		data_2_1	673	3.7	20 i/s	20 KB/s
	(Thput)	0.13 i/s	data_in_redir	679	2.3	0.3 i/s	3.4MB/s
Digit Recognition	Monolithic (compile)	1750 s	upate_knn_[1-20]	770–844	2.16–2.32	0.06 i/s	10–46 KB/s
	(Thput)	150 i/s					
	PRflow all HW						
	(Thput)	253 i/s					
	Emulation (compile)	292 s					
	(Thput)	1 i/1h					
	All A53 cores						
	(Thput)	1 i/160s					
BNN	Monolithic (compile)	9790 s	bnn_bin_conv				
	(Thput)	280 i/s	bnn_bin_dense				
	PRflow all HW		bnn_bin_dense_wt_[0–15]				
	(Thput)	57 i/s	bnn_fp_conv				
	Emulation (compile)		bnn_mem				
	(Thput)		bnn_bin_conv_wt_[0-8]				
	All A53 cores						
	(Thput)						
Optical Flow	Monolithic (comile)	2560 s	gradient_xyz_calc	632			
	(Thput)	260 f/s	gradient_weight_y_[1–3]	635–653			
	PRflow all HW		gradient_weight_x[1–3]	643–652			
	(Thput)	39 f/s	outer_product[1–2]	599–608			
	Emulation (compile)		tensor_weight_y1	668			
	(Thput)		tensor_weight_y2	678			
	All A53 cores		tensor_weight_x1	687			
	(Thput)		tensor_weight_x2	689			
			output_fun	606			
Face Detect	Monolithic (compile)	2480 s	imagScaler_{top,bot}				
	(Thput)	58 f/s	sfilter[0–4]				
	PRflow all HW		wfilter[0–4]				
	(Thput)	2.8 f/s	wfilter[0–4]_process				
	Emulation (compile)		weak_data_req_simple				
	(Thput)		strong_classifier				
	All A53 cores		weak_process_new				
	(Thput)						

(processor vs. FPGA page). These optimization would be promising candidates for autotuning.

10 CONCLUSIONS

With a suitable dataflow coordination discipline, combining C-to-gates compilation, embedded processor cores, and separate compilation and linking, we can provide a better FPGA development experience—one that is no longer dominate by long wait times for FPGA synthesis-placement-and-routing. This supports a familiar incremental refinement design style, where a functional design is always running and can be improved through a series of small changes and fast edit-compile-debug turns. The key capability we add to enable this discipline is a producer/consumer agnostic communication stream that support both separate compilation and heterogeneous architectures (e.g., FPGA, Processor) on either side of the communication stream. This allows us to compile C for the operator quickly to processors to give an immediate (tens of seconds) integration of a refined operator and to compile C to a small partial reconfiguration region quickly (10 minutes) to migrate to the FPGA. Operators mapped to processors can continue to run with the separately compiled regions already mapped to the FPGA, providing higher performance integration testing than emulation. In an incremental compilation and refinement development style exploiting separate compilation and linking, once an operator has been migrated to the FPGA, it need not be recompiled as other operators are developed and modified. As a result, the methodology provides the missing -O0, -O1, -O2 compiler optimization options to complement the existing best-effort -O3 option provided by conventional monolithic FPGA compilation.

A STREAM READ AND WRITE LOOPS

For raw memory read and write testing without dataflow handshaking (Sec. 5.3, we use inlined read and write operations with simple loops. Here's the inline write defintion:

```
inline void inline_simple_write(uint64_t data)
{
   this->bench_buff[this->ptr] = data;
   this->ptr++;
   this->ptr = this->ptr & this->buff_size;
}
```

We test with:

```
for(int i = 0; i < loops; i++)
    Input_Stream.inline_simple_write(i);
```

The corresponding inline read definition:

```
inline uint64_t inline_simple_read()
{
   uint64_t temp;
   temp = this->bench_buff[this->ptr];
   this->ptr++;
   this->ptr = this->ptr & this->buff_size;
   return temp;
}
```

We test with:

```
for(int i = 0; i < loops; i++)
    tempa = Input_Stream.inline_simple_read();
```

For NEON Vector operations, we use:

```
inline void inline_vector_write( uint64_t data )
{
   uint64x2_t new_data;
   uint64x2_t new_data2;
   new_data = vdupq_n_u64(data);
   new_data2 = vdupq_n_u64(data+2);
   vst1q_u64((uint64_t*)&this->bench_buff[this->ptr],
             new_data);
   vst1q_u64((uint64_t*)&this->bench_buff[this->ptr+2],
          new_data2);
   this->ptr+=4;
   this->ptr = this->ptr & this->buff_size;
}

inline void inline_vector_read( uint64_t* out_data )
{
   uint64x2_t v_data;
   uint64x2_t v_data2;
   v_data = vld1q_u64((const uint64_t*)
                      &this->bench_buff[this->ptr] );
   v_data2 = vld1q_u64((const uint64_t*)
                       &this->bench_buff[this->ptr+2] )
   vst1q_u64(&out_data[0],v_data);
   vst1q_u64(&out_data[2],v_data2);
   this->ptr+=4;
   this->ptr = this->ptr & this->buff_size;
}
```

We test with:

```
for(int i = 0; i < loops; i++)
   Input_Stream.inline_vector_write(i);
for(int i = 0; i < loops; i++)
   Input_Stream.inline_vector_read( &out_data[0] );
}
```

REFERENCES

[1] Douglas Adams. 1987. *Dirk Gently's Holistic Detective Agency.* Gallery Books.

[2] Douglas Adams. 1988. *The Long Dark Tea-Time of the Soul.* Gallery Books.

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

[4] Michael Butts, Anthony Mark Jones, and Paul Wasson. 2007. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In *FCCM*. 55–64.

[5] 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. *J. Microproc. and Microsys.* 30, 6 (September 2006), 334–354.

[6] Jan Gray. 2016. GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator. In *FCCM*. 17–20. https://doi.org/10.1109/FCCM.2016.12

[7] Jan Gray. 2017. GRVI Phalanx Update: Plowing the Cloud with Thousands of RISC-V Chickens. In *Proceedings of the Seventh RISC-V Workshop*. December. http://fpga.org/wp-content/uploads/2017/12/GRVI-Phalanx-Update-7th-RISC-V-Workshop.pdf

[8] 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.

[9] Nachiket Kapre. 2017. Deflection-routed butterfly fat trees on FPGAs. In *FPL*. 1–8. https://doi.org/10.23919/FPL.2017.8056804

[10] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. 2018. Case for Fast FPGA Compilation using Partial Reconfiguration. *FPL* (2018).

[11] Thomas M. Parks. 1995. *Bounded Scheduling of Process Networks.* UCB/ERL95-105. University of California at Berkeley.

[12] 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 *Proc. ASPLOS*. ACM, 271–286.

[13] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft Vector Processors with Streaming Pipelines. In *FPGA*. 117–126.

[14] Greg Stitt, Frank Vahid, and Walid Najjar. 2006. A Code Refinement Methodology for Performance-Improved Synthesis from C. In *ICCAD*. 716–723. https://doi.org/10.1145/1233501.1233649

[15] David Wilson and Greg Stitt. 2019. Seiba: An FPGA Overlay-Based Approach to Rapid Application Development. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.

[16] 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. *ICFPT* (2019).

[17] Xilinx, Inc. 2018. *UG1169: Xilinx Quick Emulator User Guide.* Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1169-xilinx-qemu.pdf

[18] Xilinx, Inc. 2019. *UG1027: SDSoC Environment User Guide.* Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1027-sdsoc-user-guide.pdf

[19] Xilinx, Inc. 2019. *UG1085: Zynq UltraScale+ Device: Technical Reference Manual.* Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

[20] Xilinx, Inc. 2020. *UG902: Vivado Design Suite User Guide: High-Level Synthesis.* Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives

[21] 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 *FPGA*. 269–278. https://doi.org/10.1145/3174243.3174255