HLS-Compatible, Embedded-Processor Stream Links

Eric Micallef, Yuanlong Xiao, and André DeHon
Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
Email: micallef@seas.upenn.edu, ylxiao@seas.upenn.edu, andre@ieee.org

***Abstract*—Fine-grained dataflow streaming between parallel compute operators provides both a simple form of concurrency and high performance operation. These streams are regularly used to support concurrency within HLS computations on the FPGA. We provide a compatible stream API and implementation that allows FPGA operators to interoperate with operators implemented on the embedded, hardcore processors on SoC FPGAs. With our stream interface, individual operators can be written in C and compiled to either the embedded core or the FPGA from a single source file, and neither FPGA-mapped nor processor-mapped operators need to know whether the other side of the stream is implemented on an embedded core or on the FPGA. This capability also eases processor integration for debugging and development. Our streams support over 100 MB/s per core between the ARM A53 cores on the Zynq UltraScale+ and the FPGA fabric even when all four A53 cores concurrently share a single AXI channel.**

I. Introduction

Latency-insensitive dataflow streaming interfaces allow a producer and consumer to operate concurrently. They integrate synchronization and tolerate variable timing in the producer and consumer [1]. These are heavily used within High-Level Synthesis (HLS), both explicitly using `hls::stream` in Vivado HLS and implicitly using `DATAFLOW` pragmas to stream between functions and loops. We also use them when communicating with IP cores and some softcore processors (e.g., microBlaze) [2]. They provide a way to connect and coordinate concurrently operating components on our FPGAs.

Unfortunately, we do not have an out-of-the-box solution to provide stream connections between our embedded, hardcore processors on our System-on-a-Chip (SoC) FPGAs and our streaming FPGA blocks. If we did, we could use the streams to link an embedded processor into the computational dataflow in the place of an FPGA block. We could also compile C-based operators to either the embedded ARM cores or to the FPGA fabric from the same source code. This is useful for development, where we may work out functionality of the operator with fast compiles to the processor before migrating the operator to the FPGA. It is also useful for debugging, where we might move a faulty operator to the processor for testing and to provide greater visibility into application data.

To address this need, we develop stream support for the embedded, hardcore processors. This includes a compatible stream definition that we can swap in for `hls::stream` when the operator C-code is compiled to the ARM core and an efficient implementation that bridges streams across AXI channels between the ARM core and FPGA when producer and consumer are implemented on different targets.

We make the following contributions:

- Characterize raw communication paths between embedded processors cores and the FPGA fabric on an UltraScale+ Zynq SoC (Sec. V-A)
- Introduce `ps::stream` API for embedded-processor-mapped operators
- Provide associated FPGA interface logic to implement a source-sink agnostic, latency-insensitive communication link between operators when one operator resides on an embedded core and the other resides on the FPGA fabric (Sec. III)
- Characterize the performance of the API implementation on an UltraScale+ Zynq SoC (Sec. V)

We provide an open-source release for our `ps::stream` implementations: https://github.com/icgrp/estream4fccm2021

II. Background

A. Heterogeneous SoCs

Today's System-on-a-Chip (SoC) FPGAs embed hardcore 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.

We demonstrate this work on the Zynq UltraScale+ MPSoC [3]. This includes 4×A53 64b ARM core and 2×R5 32b "Real Time" ARM cores. 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 master channels and 3 AXI slave channels provide interconnect to the FPGA fabric, with 3 master channels providing I/O coherence and one providing full coherence with the A53 processor hierarchy. Each AXI channel is 128b wide and can operate up to 333 MHz, providing a peak bandwidth 333×128b = 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).

B. Processor Integration and Task Migration

A smoother continuum of processor integration with FPGA computations and the migration of computational elements between the processor and the FPGA has long been the goal for FCCMs. Keller makes the case for integrating processors

© 2021 IEEE

into the FPGA design to replace low throughput tasks or when the processors have favorable features [4], [5]. To ease design, many developers recommend a software-first methodology including GRVI-PHALANX [6], [7] and Soft Vector Processors [8]. *Seiba* supports sequential migration of functionality back to a processor for debugging [9]. We help further enable this vision by (1) providing the ability to compile code from a single source to either an embedded processor or the FPGA fabric, (2) supporting concurrent dataflow operation of processors and FPGA logic, and (3) providing high throughput communication between concurrent operations through streaming.

III. 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 (e.g., different `-I` include paths) 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.

To support our ease of conversion goals, we design a `ps::stream` stream interface for the software side that is compatible with the Vivado `hls::stream` interface [10]. 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 `ps::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 an 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.

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 work on just a few data items at a time. For purely feed-forward operator graphs, large-scale batching of operations is viable, but introduces the need to identify the size of the data block being transferred, which is not compatible with the existing, 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. DMA requires the inclusion of `mm2s` and/or `s2mm` blocks on the FPGA that require a couple of thousands LUTs each and AXI interface logic the requires about four thousands LUTs, for a total around 8K LUTs.

IV. Basic Stream Design

On the software side, the `ps::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 master 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 operation, there is a need to read one pointer (tail to make sure there is data to read on a read, head to make make sure there is space to write on a write) and write another (update head on a read, update tail 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.

A. 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, nor do they need to share their updates.
- It is functionally acceptable for updates to the head (tail) pointer to be delayed before seen by the producer (consumer), as long as it occurs **after** the data has been read (written).

  - The data operation most complete before the associated head or tail pointer update, which we assure with the data synchronization barrier (`dsb`) instruction.
  - 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; there are no ordering requirement among different streams.

Among other things, this means we are not forced to use the ACE (AXI Coherence Extension) AXI channel for communication. The UltraScale+ Zynq ACP (Accelerator Coherence Port) and HPC (High Performance Coherence) ports provide adequate functionality in *outer-shareable* mode to share data in the L2 cache.

V. Implementation

Our experiments use the Ultra96-v2 board with an UltraScale+ Zynq XCZU3EG FPGA. The Stream IP block that bridges across the PS→PL interface is written in C and compiled with VivadoHLS. We use Xilinx Vivado 2018.3 including the associated VivadoHLS and SDK.

A. Raw Memory Performance

We 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 master 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. I. Working on bare-metal designs, we perform a tight loop of read (or write) operations at the maximum datawidth allowed (32b for R5, 64b for A53, 128b for PL fabric) with the PL running at 300 MHz. When used, L2 is set to outer-shareable mode. We use ARM cycle counters to measure the read, write, or transfer time for batches of 1,000,000 64b words unless noted otherwise. Tab. I shows the PS can get high bandwidth from the L2 cache (620 MB/s write) and has an odd asymmetry between the OCM read and write bandwidth (83 MB/s read vs. 625 MB/s write). The PL achieves high bandwidth accessing the L2 over the ACP port (710 MB/s write), but also has decent bandwidth accessing the OCM over an HP port (440 MB/s write).

Even in a tight loop the PS ARM core must issue 7 instructions per read or write operation. As a result, many of 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. I also shows the impact of these optimizations on raw memory bandwidth. This shows that the PS bandwidth in the simple tight loop test was limited by instruction issue (620 MB/s write) and that bottleneck can be overcome by unrolling to achieve bandwidth into the GB/s (5 GB/s write). Vector operations moving 128b data to and from the L2 cache recover some of the lost bandwidth (2.1 GB/s write) and exceeds the bandwidth achieved by the PL.

TABLE I
Peak Raw Read and Write Bandwidth

Unit	Case (PL AXI)	DRAM r	DRAM w	OCM r	OCM w	L2 r	L2 w
		Results in MB/s					
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 unroll factor of 32. Measured for transfer of 1,000,000 words.

The R5 cannot access the APU's L2, so we must use the OCM when connecting streams to the R5 processors when they communicate with A53 cores or the PL. The unrolled write operations provide a large throughput gain for the A53 cores accessing the OCM, but the read bandwidth remains largely unchanged. Unrolling improves the bandwidth for the R5 access as well, but the impact is not as large, and the read bandwidth gain is more significant than write. A53 vector operations double the read bandwidth. The A53 results suggests the OCM read operations are limited by the architecture and not by inefficient instruction issue.

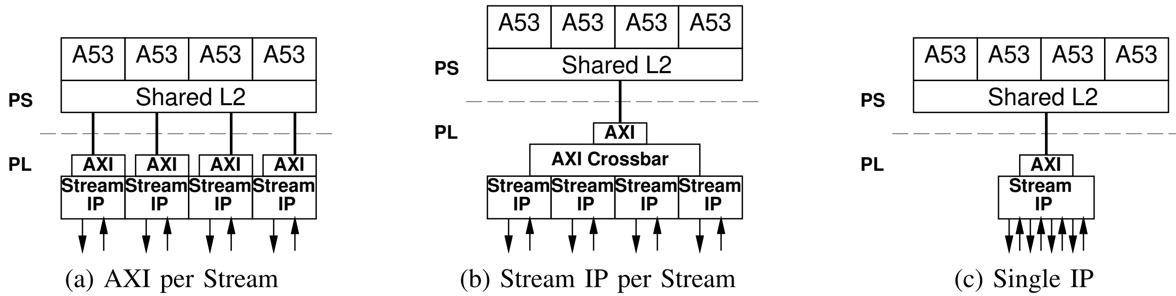
B. Full Stream Links

The throughput of a stream will depend on both the throughput of the producer and the consumer and include additional head and tail maintenance overhead as previously noted. Tab. II 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). We see the ACP port sharing data in L2 is the fastest at 360 and 460 MB/s; this is a little over half the raw bandwidth (810 MB/s read, 710 MB/s write, Tab. I) available between the PL and the L2 cache.

C. Multiple 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 (Fig. 1(a)). 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 (Tab. II). 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. II-A). We should be able to share AXI channels without sacrificing performance. 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 saves AXI channels, but using a separate PL-side interface (IP Block) for each stream



(a) AXI per Stream (b) Stream IP per Stream (c) Single IP

Fig. 1. Multiple PS↔PL Stream Options

TABLE II
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
PS:A53	PS:A53		110		435	
PS:A53	PS:R5		39	29	—	—
PS:R5	PS:R5		32		—	—
PS:A53	PL	HP	95	120	—	—
PS:A53	PL	HPC	95	120	73	130
PS: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 Tightly-Coupled Memory for R5 and L2 for A53 cores. Measured for transfer of 1,000,000 128b words.

TABLE III
MULTIPLE STREAM IMPLEMENTATION OPTIONS

Strms	AXI Chans.	PL IPs	Fig.	**Thrupt (MB/s)** PS→PL	PL→PS	**PL IP Resources** LUTs	RAMs	**PL AXI** LUTs
1	1 ACP	1		460	360	2,639	24	7,100
1	1 HP	1		95	120	3,353	24	7,000
4	4 HP	4	1(a)	95	120	13,432	96	19,000
4	1 HP	4	1(b)	95	120	13,432	96	16,000
4	1 HP	1	1(c)	95	96	3,604	28	7,100
4	1 ACP	4	1(b)	342	200	13,412	96	16,000
4	1 ACP	1	1(c)	170	120	3,604	28	7,100

In multiple stream cases, stream throughput in columns 5 and 6 is for *each* stream. All data here for A53 cores using vector operations to transfer 128b data using L2 cache for shared buffers. Buffers are sized to hold 512 64b words. PL running at 300 MHz. RAM count is for 18Kb BRAM blocks. Measured for transfer of 1,000,000 128b words.

(Fig. 1(b)) has a high LUT and BRAM cost per stream as shown in Tab. III (PL IPs of 4). We see that each interface costs around 3,500 LUTs for logic plus 3,000–4,000 LUTs to add a port to an AXI crossbar.

Alternately, we develop a single PL-side interface that serially processes data from each stream in round-robin fashion (Fig. 1(c), PL IPs of 1 in Tab. III). Using the ACP AXI channel, the single PL-side interface supports 4 PS→PL and 4 PL→PS streams at over 100 MB/s each with around 10K LUTs, only about 1K LUTs more than the interface that supported a single PS→PL and PL→PS stream, putting both in about the same footprint as the minimal logic for a DMA interface. Fig. 2 shows throughput as a function of transfer length for this Fig. 1(c) ACP case supporting 4 streams each direction. At these rates, the stream throughput is never the bottleneck once the processor does any computation between stream operations.

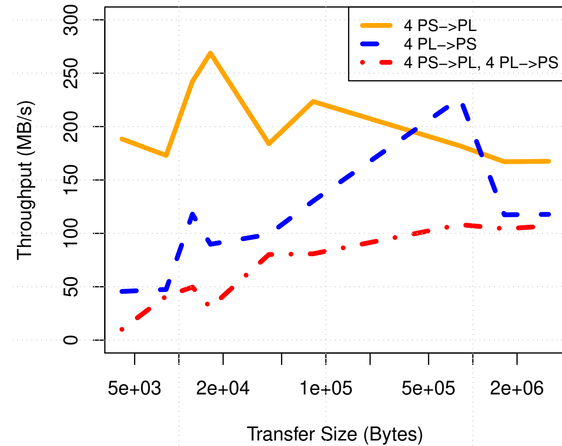


Fig. 2. Throughput as a Function of Transfer Length for Fig. 1(c) ACP AXI Channels supporting 4 Streams each Direction

VI. CONCLUSIONS

By providing a `ps::stream` API and implementation that is compatible and interchangeable with the `hls::stream`, we can compile operators from a single source definition to either an embedded processor or the logic fabric on an SoC FPGA. Operators using the compatible stream interface need not know whether the operators on the other side of the interface is implemented on an embedded processor or on the FPGA fabric. This allows the construction of dataflow pipelines and graphs that mix embedded processors and FPGA resources. It also eases the movement of operators between the FPGA fabric and the embedded processors during development and debug. The associated `ps::stream` IP block bridges communication across an AXI channel, supporting over 100 MB/s communication bandwidth simultaneously on 4 streams in each direction.

ACKNOWLEDGMENTS

This work was supported in part by a Google Faculty Fellowship. Xilinx donated Vivado tools for use in this work.

REFERENCES

[1] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[2] *UG984: MicroBlaze Processor Reference Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf

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

[4] E. Keller, G. Brebner, and P. James-Roxby, "Software decelerators," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2003.

[5] P. James-Roxby, G. Brebner, and D. Bemmann, "Time-critical software deceleration in a FCCM," 2004, pp. 3–12.

[6] J. Gray, "GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2016, pp. 17–20.

[7] ——, "GRVI phalanx update: Plowing the cloud with thousands of RISC-V chickens," in *Proceedings of the Seventh RISC-V Workshop*. December, 2017. [Online]. Available: http://fpga.org/wp-content/uploads/2017/12/GRVI-Phalanx-Update-7th-RISC-V-Workshop.pdf

[8] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2014, pp. 117–126.

[9] D. Wilson and G. Stitt, "Seiba: An FPGA overlay-based approach to rapid application development," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.

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