Auto-Partitioning Heterogeneous Task-Parallel Programs with StreamBlocks

Mahyar Emami*
mahyar.emami@epfl.ch
EPFL
Lausanne, Switzerland

Endri Bezati*
endri.bezati@huawei.com
Computing Systems Laboratory, Huawei Technologies
Zurich, Switzerland

Jörn W. Janneck
jwj@cs.lth.se
Lund University
Lund, Sweden

James R. Larus
james.larus@epfl.ch
EPFL
Lausanne, Switzerland

ABSTRACT

FPGAs play an increasing role in the reconfigurable accelerator landscape. A key challenge in designing FPGA-based systems is partitioning computation between processor cores and FPGAs. An appropriate division of labor is difficult to predict in advance and requires experiments and measurements. When an investigation requires rewriting part of the system in a new language or with a new programming model, its high cost can delay design-space exploration. A single-language system with an appropriate programming model and compiler that targets both platforms transforms this tedious exploration to a simple recompile with new compiler directives.

This work introduces StreamBlocks, a unified open-source software/FPGA compiler and runtime that takes dataflow programs written in CAL, and automatically partitions them across heterogeneous CPU/FPGA platforms. The explicit task-parallel semantics of dataflow allows our compiler to simultaneously take advantage of thread parallelism on software and spatial parallelism on hardware.

StreamBlocks is augmented with a profile-guided auto-partitioning tool that helps identify the best hardware-software partitions. We demonstrate the capability of our compiler in finding the right balance between hardware and software execution on both a high-end datacenter accelerator card and an embedded board. Our experiments exhibit a $4-7\times$ speedup over trivial partitions. This speedup is achieved automatically with zero code modifications.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**; *Multicore architectures*; • **Hardware** → *Hardware-software codesign*; • **Computing methodologies** → *Concurrent programming languages.*

KEYWORDS

reconfigurable computing, Actors, partitioning

*Both authors contributed equally to this research.

PACT '22, October 10–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569659

ACM Reference Format:
Mahyar Emami, Endri Bezati, Jörn W. Janneck, and James R. Larus. 2022. Auto-Partitioning Heterogeneous Task-Parallel Programs with StreamBlocks. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3559009.3569659

1 INTRODUCTION

The slowdown in performance gain of general-purpose processors has increased interest in using FPGAs as reconfigurable accelerators, particularly for cloud computing [8, 13, 42, 48].

A widely-accepted approach towards accelerator design is the use of High-Level Synthesis (HLS) tools [2, 26, 35, 41] that allow designers to develop accelerators in a software language such as C/C++. In spite of being marketed as C/C++ compilers, the code written for HLS follows different principles from software and usually depends on specific HLS libraries. That is, to execute a program on an FPGA using HLS, it should be written for HLS from scratch. Furthermore, every time a new functionality is added to or, subtracted from hardware, the hardware-software interface needs to be modified to account for the change in communication patterns.

A key challenge in designing accelerators is partitioning computation between processors and an FPGA. An appropriate division of labor may be difficult to predict and only be revealed by experiments and measurements. When such an experiment requires rewriting part of a system in a new language, or with a new programming model, to run on the other platform, the high cost of exploration may retard studies of different configurations and limit the evaluation.

A possible solution is a single-language system with an appropriately portable programming model and a compiler to generate code for both platforms. In this case, exploring a new system configuration reduces to recompilation with different compiler directives.

A fundamental aspect is the programming model, which must allow parts of a program to run on either a CPU or an FPGA and interact transparently, regardless of where they execute. A unified programming model should be able to take advantage of thread-level parallelism on software and abundant spatial parallelism on hardware. Its communication mechanisms should map well to the available resources on both CPUs and FPGAs. Additionally, the programming model should define encapsulation boundaries such

that a piece of functionality can be freely placed on either hardware or software independent of other concurrent functionality.

The *Actor* model offers a natural fit to our requirements. An Actor-based programming model describes a network of concurrent tasks—actors—where each task has an independent state and only communicates with others through messages. The message-passing style of communication, called dataflow parallelism, is ubiquitous in modern software and, interestingly is the de-facto model of task-parallel programming in HLS [15, 16, 43, 49].

This work introduces StreamBlocks[1], a prototype compiler suite for heterogeneous task-parallel programs that enables an exploratory approach to programming heterogeneous FPGA-accelerated systems. StreamBlocks makes no distinction between software and hardware code. It only restricts tasks with system call invocations (e.g., reading a file) or legacy code to software execution, while allowing any other task to be placed on hardware or software with zero code modification.

StreamBlocks compiles programs written in the CAL Actor language. Programs written in CAL are directed dataflow graphs (cyclic or acyclic) whose nodes are actors and edges are first-in-first-out (FIFO) communication channels. A dataflow program specifies a partial order of computation in which sequencing constraints arise only from data dependencies. As a result, actors can execute concurrently.

A dataflow program can be compiled to run on a processor, FPGA, or a combination of the two. In addition, dataflow semantics do not constrain an FPGA to operate only as a simple call-respond accelerator. Instead, it allows the FPGA to operate as a streaming coprocessor that executes concurrently with a processor and continually communicates with software actors. This generality allows the direct migration of part of a computation from software to hardware without rewriting the (dataflow) program, an essential step in developing, evaluating, or evolving a heterogeneous system.

However, the design space of possible partitions of functionality between the two platforms can be large and complex. An integral part of StreamBlocks is a profile-guided auto-partitioning tool to help identify the best-performing ones by design-space exploration. It formulates the problem of CPU-FPGA partitioning as a mixed-integer linear programming (MILP) model and optimizes mapping of actors to one or more CPU threads, or dedicated hardware on an FPGA for end-to-end performance.

Our work naturally focuses on new applications, without an established code-base. In this situation, a team must choose between a single language (e.g., CAL) or two languages (e.g., C and C-HLS) solution. The former brings abstractions that (1) free developers from concurrency management at the language level and (2) enables automatic design-space exploration. The other, more conventional, approach requires a team to define their abstractions using low-level communication and concurrency mechanisms and does not provide a methodology for exploring the design space of partitioning work between the CPU and FPGA.

Existing applications *must* be rewritten for FPGAs (even when using HLS), so writing code in a new language (e.g., CAL) that targets both platforms helps avoid supporting separate hardware and software implementations. Moreover, StreamBlocks supports code reuse since it can invoke C/C++ code through a thin wrapper actor written in CAL. Our design-space exploration methodology can constrain *legacy actors* to software execution. This is especially useful when developers have prior knowledge that parts of the application will not benefit from hardware execution.

Previous work has demonstrated that CAL can be efficiently compiled for both hardware [29] and software [31]. CAL does not inherently limit the evaluation of our techniques and findings. However, our goal is not to find the optimal implementation of an application in the large space of possible implementations (originating from all mixes of source languages), nor is it to replace a fine-tuned manual hardware-software partitions backed by engineering insight and platform expertise. Rather, Streamblocks is a new compiler that treat an FPGA like other compute resources that may or may not help achieve *automated* acceleration.

An optimizing compiler only needs to satisfy a performance goal, not find the absolute optimal solution. In case auto-partitioning fails to meet the goal, a developer could fallback to lower-level optimization, or even fall back to standard methods of hardware development like fine-tuned HLS or RTL code. This is a standard practice in software development. For instance, many developers choose Python for rapid experimentation and only fallback to C++ development if strict performance constraints are not met.

Note that our work simply uses CAL as medium to facilitate portability and design-space exploration. Any task-parallel programming model that respects the portability requirements described earlier is a viable frontend to which the techniques described in this paper can be applied. The contributions of this work are:

- An open-source dataflow compiler for the CAL programming language that targets heterogeneous platforms.
- An automatic task-partitioning methodology for placing computations on heterogeneous platforms from a single source language.
- A tool that uses this flexibility to find high-performing hardware-software partitions of a system.
- Experimental results demonstrating that our compiler can efficiently use the resources of a heterogeneous platform.

The paper is organized as follows. Section 2 provides an overview of dataflow programming and the CAL programming language. Section 3 describes the partitioning algorithm. Section 4 presents the StreamBlocks compiler and its heterogeneous code generators. Section 5 evaluates the compiler and the partitioning methodology. Section 6 compares StreamBlocks with other systems and the paper concludes with Section 7.

2 ACTORS AND DATAFLOW

Dataflow is a computation model used to express concurrent algorithms operating on streams of data. It has a rich history with many variants (e.g., [6, 21, 33, 39]) often targeted at a specific application domain or designed to facilitate efficient analysis or implementation. In our work, we use a general form of dataflow, in which a program is a *network* of computational kernels, called *actors*, that are connected via input and output *ports* by directed, point-to-point connections or *channels*. Actors may have an internal state that is invisible to other actors. They interact exclusively by sending

[1] https://github.com/streamblocks/streamblocks-platforms

packets of data called *tokens* along the channels. Channels are lossless and preserve the order in which tokens are sent. They are also buffered so that the producer and consumer of a token need not synchronize during a transfer. A token may be consumed any time after it is produced.

2.1 CAL Actor Language

The actors considered in this paper execute in a sequence of atomic steps or *firings* [21]. During each firing an actor may (i) consume input token(s), (ii) produce output token(s), and (iii) modify its internal state. Instead of being written as a sequential process (Kahn processes [33], i.e., a never-ending loop), languages such as CAL are structured around the steps an actor may perform. In CAL, an actor is a collection of *actions*, each describing a step that the actor can perform, along with the *conditions* under which the step should execute.

The CAL actor language [22] permits the description of actors for a wide range of dynamic and static dataflow models, ranging from Kahn process networks [33], dataflow process networks [39], and synchronous and cyclo-static dataflow [6, 7, 38], among others. Parts of the language have been standardized by the ISO/IEC committees *RVC-CAL* [1] as one of the languages used to write the reference implementations of MPEG video decoders.

```
namespace example:
  external function rand(int x) --> int end
  actor Source() int IN ==> int OUT:
    int x := 0;
    action ==> OUT:[rand(x)] guard x < 4096 do
      x := x + 1;
    end
  end
  actor Filter(int param) int IN ==> int OUT:
    function pred(int param, int value) --> bool :
      if (param > value) then true else false end
    end
    t0: action IN:[t] ==> OUT:[t] guard pred(param, t) end
    t1: action IN:[t] ==> end
    priority
      t0 > t1;
    end
  end
  actor Sink() int IN ==> :
    action IN:[x] ==> do println(x) end
  end
 network TopFilter(int param) ==>:
    entities
      source = Source();
      filter = Filter(param=param);
      sink   = Sink();
    structure
      source.OUT --> filter.IN; filter.OUT --> sink.IN;
  end
end
```

Listing 1: A simple network of two actors in CAL.

Listing 1 contains a CAL dataflow program which we use to explain StreamBlocks operational principles. This actor is composed of four entities: three actors and a network. Actor `Source` has a single output port `OUT`. Its single action describes the only step it takes, incrementing a state variable `x` up to 4096 and producing a token to be sent to its output port. `rand` is an external function that links with legacy code.

The `Filter` actor's parameter, `param`, is a number used by the local function `pred`. `Filter` uses an input port `IN` and an output port `OUT`. It comprises two actions. The first, labeled `t0`, includes a *guard* condition, a logical expression that must be true (in addition to an input token being available) for the actor to be able to take the action—copying the input token to the output. The second action, labeled `t1`, also consumes an input token but has no guard and produces no output. A priority rule specifies that action `t0` has a higher priority than action `t1`. This means that whenever conditions for both are satisfied, action `t0` executes. In this way, action `t0` copies to the output all input tokens that satisfy its guard, while action `t1` "swallows" the other tokens. Finally, Actor `Sink` consumes a token from its input port `IN` and prints it to the console.

The `TopFilter` network connects the three actor instances. The dataflow network in `TopFilter` has no input and output ports. It has three entities `source`, `filter`, and `sink`, which are instantiations of actors `Source`, `Filter`, and `Sink`, respectively. The `filter` instance is instantiated with a parameter received from the network.

The `TopFilter` network connects the `IN` input port of the network to the input `IN` of the `rand` instance. The output port `OUT` of instance `source` is connected to the input port `IN` of `filter` instance. The `filter` output port `OUT` is connected to the input port `IN` of the instance `sink`.

3 DESIGN-SPACE EXPLORATION

Finding an appropriate balance between hardware and software execution is a design-space exploration problem. StreamBlocks facilitates this exploration process since an actor written in CAL can run in hardware or software with no code changes.

Hardware-software partitioning can be modeled as a graph partitioning problem with a cost function. In this paper, we focus on *end-to-end execution time as the single metric to optimize*. The cost function used in design-space exploration should estimate the execution time for a given partitioning.

Performance prediction is difficult for a variety of reasons: actors in general can exhibit dynamic behavior, a multi-threaded software runtime can behave unpredictably, software-hardware crossings are difficult to model accurately, and an actors' behavior is data-dependent because of dynamic data dependencies in and among actors.

We introduce an approximate performance model for a CPU-FPGA platform that is directly applicable to any task-parallel system built on top off OpenCL's host (software) and device (hardware) abstraction. We make a few simplifying assumptions: First, we assume a many-to-many mapping of actors to CPU threads and a *single* FPGA. Second, we assume the existence of a *virtual* PartitionLink (PLink) actor which asynchronously transfers data between the hardware and software in a transparent way. We focus our attention to the performance model here; Section 4 gives a technical overview of how StreamBlocks assembles the partitioned system.

Fig.1 outlines a hypothetical dataflow graph on the left and possible hardware-software partition (i.e., an implementation) on the right. Each circle represents an actor with point-to-point channels delineated as arrows. Note that in the partitioned graph, all hardware-software communication should go through the PLink. In addition, the thickness of arrows are proportional to the communication cost which varies based on the source and destination partitions. Communication inside the FPGA is very cheap since FIFOs can be implemented using on-chip memories with single

cycle access latency. In contrast, FPGA-PLink communication is expensive since it usually involves a DMA transfer. FPGA-PLink communication is therefore critical to system performance. Inter- and intra-thread communication latency also differs because actors on the same thread are more likely to communicate through lower level caches, while inter-thread communication has coherency overhead through last level caches or even the off-chip memory.

3.1 Performance Modeling

Our performance model is based on two key metrics: (i) modeling *pure* execution time within and across partitions, and (ii) modeling communication cost penalties within and across partitions.

Pure execution time: We assume software actors mapped to the same thread execute serially, hence their execution time adds up. In contrast, software actors mapped to different threads execute concurrently and their execution time is dominated by the slowest software actor.

Likewise, hardware actor execution time is dwarfed by the slowest actor since all actors on hardware execute in parallel.

Communication penalty: Intra-thread communication (actor-to-actor inside a thread) of different threads overlap since threads run concurrently, and inter-thread communication (actor-to-actor across threads) adds up since they go through a shared communication medium (e.g., L3 cache). These two components serve as a performance penalty for the total execution time.

We assume computation fully overlaps with communication and communication therefore has no cost (i.e., FIFOs act as pipelines). And finally, hardware-software communication time adds up to hardware execution time, but it overlaps with software execution time (since hardware executes concurrently with software).

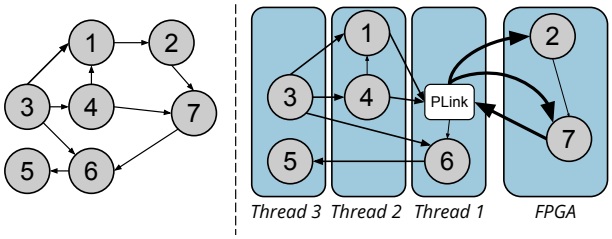


Figure 1: A dataflow graph (left) and a possible partitioning (right). Each arrow's thickness is proportional to the communication cost across that link.

3.2 MILP Formulation

Our performance model has simple ideas that can easily be formulated as a MILP optimization problem. The model is parameterized by profiling information. StreamBlocks provides the mechanisms to profile different system metrics (c.f. Section 4.5).

Given a set of $n+1$ partitions corresponding to n threads from the set $P_{thread} = \{p_1, p_2, ..., p_n\}$, an FPGA partition *accel*, and a set of actors A, we define the decision variables d_p^a with the following constraints:

$$\forall a \in A, \forall p \in P_{thread} \cup \{accel\} : d_p^a \in \{0,1\}$$
$$\forall a \in A : \sum_{p \in P_{thread} \cup \{accel\}} d_p^a = 1$$

The first constraint states that the decision variables are Boolean, i.e., an actor is either on a partition or not, and the second one ensures that each actor a maps to exactly one partition from the set of partitions.

The execution time of an actor a on partition p is given by $exec(a, p)$, which is obtained through software or hardware profiling. The time spent on a thread partition, T_p, is the sum of all actor execution times since actors run sequentially on a thread:

$$\forall p \in P_{thread} : T_p = \sum_{a \in A} d_p^a \times exec(a, p) \quad (1)$$

We delegate the performance modeling of hardware actors to the PLink. We assume that all PLink operations are asynchronous (i.e., data transfer can overlap with computation) and the PLink is scheduled on one of the threads (e.g., on p_1).

$$T_{plink} = max(\{d_{accel}^a \times exec(a, accel) : a \in A\}) + T_{plink}^{read} + T_{plink}^{write} \quad (2)$$

The first term models hardware execution time as the maximum of actor execution times since actors can run concurrently (i.e., in a pipelined fashion) on the FPGA. The other two terms model OpenCL/PLink read and write transfers, which depend on the number of tokens and the OpenCL cost information and buffer size. Although PLink's operations are asynchronous, because there is a logical dependence between write, execution and read, their total time is their sum.

Similar to hardware, we model the multi-threaded execution time as the maximum of individual thread execution times:

$$T_{exec} = max(\{T_p : p \in P_{thread}\} \cup \{T_{plink}\}) + T_{intra} + T_{inter} \quad (3)$$

Observe that T_{plink} is not added to any of thread execution times even though it is scheduled on one of them because we assume the PLink is asynchronous. However, if the hardware is slower than all software threads, then execution time is dominated by hardware, hence we take the maximum. The *max* term models pure actor execution time with no regard for communication cost between actors. Actors that communicate heavily can exploit fast caches if they are assigned to the same thread. Actors on different threads will communicate through the shared and coherent cache or main memory. T_{intra} and T_{inter} model the latency of intra- and inter-core communication. The latter two terms help ensure that heavily communicating actors reside on the same thread.

We omit the full derivation of the communication cost for brevity [2]. Basically, T_{intra} is the maximum of communication cost of individual intra-thread communication since threads execute concurrently. Contrarily, T_{inter} penalizes the actor communication across threads and is the sum of all inter-thread communication costs because inter-thread messages have to traverse a medium shared by all threads.

[2] See the Appendix for details.

Using a MILP formulation enables us to incorporate other constraints into the search. For instance, we can constrain the solutions to have fewer than m FIFOs crossing the hardware-software boundary. This is important when an FPGA platform has a limited number of AXI master ports to off-chip memory. Additionally, we can *pin* certain actors to software if they contain legacy code or perform operations requiring operating system support.

The MILP formulation we have presented can be optimally solved with the current state-of-the-art solvers, and we do indeed find the optimal solutions in Section 5. However, since our model does not capture all possible dynamic behaviors, the solutions are only optimal with respect to the model.

Our MILP model optimizes only for performance. Other metrics, such as power, could be incorporated to obtain pareto-optimal design points. We omitted power considerations from our formulations as our focus is datacenter platforms where power is less of a concern.

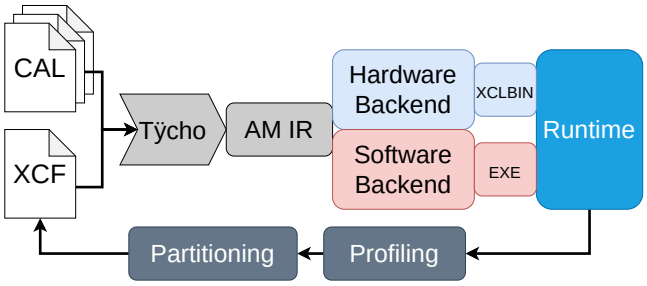


Figure 2: StreamBlocks heterogeneous compilation flow. The compilation and synthesis results in a multi-threaded executable that asynchronously launches an OpenCL RTL kernel (XCLBIN). The XCF configuration file can either be specified by a user or generated automatically by the partitioning tool after profiling.

4 STREAMBLOCKS

We now detail how StreamBlocks stitches a partitioned system together. We start with an overview, then cover hardware and software code generation.

StreamBlocks is a suite of tools for designing, compiling, and optimizing Actor-based applications. The framework consists of three parts: CAL compiler frontend, backend code generators, and a partitioning tool.

4.1 Overview

The CAL front-end is based on Tÿcho [14] with an intermediate representation called the *Actor Machine* (AM) [27].

StreamBlocks extends Tÿcho with a type system, network parameterization, and actor instantiation with list comprehension useful for generating regular dataflow graphs.

Tÿcho produces the AM intermediate representation, which is fed to the homogeneous/heterogeneous StreamBlocks backends to apply platform-specific transformations (Section 4.2 and 4.3). Each platform supports a runtime responsible for actor scheduling. Hardware actors are scheduled in parallel whereas software actors are scheduled on cores following an actor-to-thread mapping. In other words, hardware actors enjoy abundant spatial parallelism and all execute in parallel, while actors on software are co-located on multiple CPU threads. An XML configuration file (XCF) specifies these actor-to-thread or actor-to-FPGA mappings[3].

The final aspect of StreamBlocks is partitioning (Section 3). Each runtime supplies methods for profiling a dataflow application on its platform (Section 4.5). Profiling uses either hardware cycle timers for CPUs or cycle counts from a cycle-accurate RTL simulation. Also, StreamBlocks provides tools for profiling inter-actor communication on a heterogeneous platform. The StreamBlocks partitioning uses profiling information to map actor instances to the CPU threads in a homogeneous platform or across CPU threads and FPGA on a heterogeneous platform.

Currently, StreamBlocks supports code generation for heterogeneous platforms that support the Xilinx OpenCL runtime, i.e., any Xilinx PCIe accelerator board or MPSoC. An overview of the compilation flow is given in Fig. 2.

We perform design-space exploration by going through the compiler twice. First, an application is compiled and profiled for both hardware and software. Then the profiling information is fed to the partitioning tool which emits a set of XCF partitioning configurations. The XCF files along with the application can then be used to get to the implementation.

4.2 Hardware Code Generation

StreamBlocks' hardware backend is responsible for creating an FPGA implementation (i.e., Verilog) of the actors placed on hardware. One choice is to directly translate CAL to Verilog, i.e., perform high-level synthesis. However, doing so requires access to a library of FPGA device-specific timing characteristics to perform operator scheduling, but these are not publicly available. Another choice is to emit a special HLS C++ class for each actor and use an already existing HLS tool to generate Verilog. StreamBlocks takes the latter approach and uses Vivado HLS for RTL generation.

Bear in mind that the current state-of-the-art HLS tools *do not* support the general model of dataflow that StreamBlocks supports and it is not possible to solely rely on HLS for implementation. For instance, Vivado HLS does not allow feedback in a dataflow graph [49]. StreamBlocks works around this problem by interconnecting actors directly in Verilog with custom FIFOs.

For each hardware actor, StreamBlocks emits an HLS C++ class. This class implements a *controller state machine* that governs the action selection process. Upon invocation, the controller state machine checks a set of conditions required to fire an action, performs the selected action, and transitions to a new state with a new set of conditions to check.

Listing 2 outlines the structure of such an HLS actor implementation for actor `Filter` in Listing 1. The top function for HLS is `operator()` that implements the state machine. The `transition` and `condition` functions implement the actions and the conditions' required for action firings.

Our general model of dataflow requires FIFO channels with a side-effect-free *peek* (reading the head of a FIFO without consuming

[3]Provided either by the partitioning tool or manually by the user.

it) and *count* (number of tokens in the FIFO) interface. Unfortunately, Vivado HLS's streaming interface, `hls::stream`, implements neither. To circumvent this limitation, we created a custom Verilog first-word fall-through (FWFT) FIFO whose outputs are its size, count, and head queue element and that is compatible with the `hls::stream` stateful read and write interface. We generate a unique `IO` structure for each actor that contains the extended interface (see Listing 2).

Listing 2: C++ class and top HLS function for the `Filter` actor (Vivado/Vitis HLS).

```
struct IO {
  int32_t IN_peek, IN_count;
  int32_t OUT_size, OUT_count;
};
using Stream32 = hls::stream<int32_t>;
class class_filter {
private:
  // -- State Variables
  bool pred_1(int32_t l_param, int32_t l_value);
  int32_t a_param, a_t__1, program_counter;
  // -- Conditions
  bool condition_0(Stream32 &IN, IO io);  // input
  bool condition_1(Stream32 &OUT, IO io); // output
  bool condition_2();                     // guard
  // -- Transitions
  void transition_0(Stream32 &IN, Stream32 &OUT);
  void transition_1(Stream32 &IN);
public:
  class_filter(int32_t param) { a_param = param; }
  // -- Controller
  int32_t operator()(Stream32 &IN, Stream32 &OUT, IO io);
};
// -- HLS Top Function, instantiated with param=10
int32_t filter(Stream32 &IN, Stream32 &OUT, IO io) {
  static class_filter i_filter(10);
  return i_filter(IN, OUT, io);
} // ------------------------------------------------
```

At each invocation of `operator()`, the actor performs multiple checks to select an action to execute, then transitions to a new state with possibly different enabled actions. This state is recorded in the `program_counter` variable to allow execution to continue between invocations. The controller state machine is designed not to execute the *same* action multiple times in one invocation in order to allow Vivado HLS to schedule all condition evaluations in the first cycle of invocation in parallel. However, the controller can execute *different* actions in a single invocation.

StreamBlocks automatically inserts some HLS directives (i.e., `pragma`) in appropriate places to optimize the hardware. For instance, all scope, condition, and transition methods are inlined in the controller. In addition, loops with fewer than 64 iterations are unrolled to parallelize loops without incurring an unreasonable resource cost. A developer can also directly annotate the Cal code with other directives. For example, *@loop_merge* can be used to merge input-read and output-write loops, and *@external* can place an actor variable in an off-chip memory (e.g., DDR or HBM). In the latter case, the compiler produces appropriate HLS pragmas to add an AXI master interface to the actor.

FIFOs that cross the hardware-software boundary are connected to *Input* and *Output Stage* actors on the FPGA. These special actors transparently pass tokens between hardware and software (see Section 4.4).

Each RTL actor (output of Vivado HLS) is instantiated in a Verilog network along with its *trigger* module. The trigger is a hardware scheduler that enables or disables its actor. Ideally all hardware actors should execute asynchronously to maximize performance. However, hardware should also dynamically detect when forward progress is no longer possible (idleness) and inform software that computation is complete and output data is available.

We use to triggers to perform a synchronized coordination between actors and detect idleness. The triggers continually monitor network state and eventually perform a synchronized idleness check. In this synchronized check, all actors (with variable lantency) on hardware are stopped and then invoked a single time. Synchronization ensures that there are no *unobserved* messages and therefore any enabled action will fire. In other words, if none of the actors end up performing an action then the network is idle and no more forward-progress is possible.

4.3 Software Code Generation

Similar to hardware code generation, StreamBlocks compiler translates a software actor to a standard C++ class (not HLS) that runs on a processor. Conceptually, it may seem appropriate to allocate a thread to each actor since they execute independently. However, the relatively high cost of context switches leads to sub-optimal performance since an actor's controller may not run for long before it starves for data. A better use of resources is to map several actors onto a thread and run them sequentially to amortize the context switch cost over larger computation. The threads are assigned to dedicated physical cores. Each pinned thread has an independent, round-robin scheduling loop for its actors.

A key difference between the generated hardware and software state machines is that the software controller attempts to reduce scheduler overheads by performing as many action firings as possible until it starves for data. This contrasts with generated hardware code that only allows a single action to be triggered at a time to optimize the HLS schedule.

FIFOs are implemented as lockless circular buffers. A structure attached to every actor output port holds global and local FIFO pointers. The global pointers are visible to all partition threads, but a local one is visible only to its thread. Global and local pointers do not change while the actors are running. Updates are cached internally and synchronously applied after full iteration of the round-robin scheduling.

The software runtime provides the threading mechanism and FIFO implementation. In addition, it supports operating system operations—such as file operations, image/video visualization, and other functionality—and links with library and legacy code.

4.4 Hardware-Software Interface

StreamBlocks generates appropriate hardware-software interface to enable heterogeneous execution. Fig. 3 depicts the interface architecture.

The PLink (PartitionLink) is a special actor that (i) transfers software FIFOs to FPGA DDR memory, (ii) starts the execution of hardware, and (iii) receives data from the FPGA memory when the FPGA network becomes idle. PLink treats the hardware network like a coprocessor and is its software controller. It invokes the coprocessor as long as the hardware-software execution can make forward-progress. Underneath, PLink uses OpenCL's API [46], basically treating the hardware coprocessor as an *OpenCL kernel*.

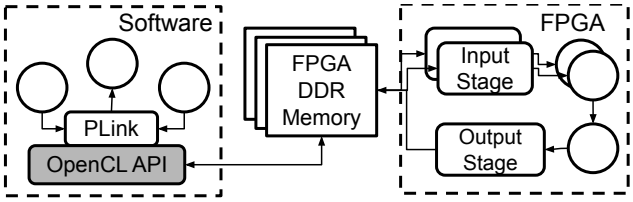


Figure 3: The hardware-software interface. Circles represent application actors. Arrows show communication paths. Input stage, output stage, and PLink are special actors that are instantiated depending on hardware-software actor placement.

FIFOs that cross the hardware-software boundary are allocated in the FPGA DDR memory. We instantiate an input or an output stage actor for any software-to-hardware or hardware-to-software channel respectively. An input stage actor continuously reads the data FPGA DDR memory in bursts and streams it to an on-chip FIFO connected to consumer actors. An output stage actor carries out a similar task and reads an on-chip FIFO and streams it to a DDR bank.

The PLink takes advantage of OpenCL events and an out-of-order command queue for asynchronous operations. Consequently, the PLink never blocks the software thread it is scheduled on such that other actors can perform useful work.

4.5 Profiling

The compiler provides a co-simulation solution that runs software actors on the software runtime and instantiates hardware actors as cycle-accurate SystemC models which is used for hardware profiling. We rely on Verilator to translate Verilog to SystemC and the PLink transparently establishes the interface between the software and SystemC simulation domain. Co-simulation provides per-actor and per-action cycle-accurate metrics. We use OpenCL profile counters to measure the OpenCL read and write bandwidth for a variety of buffer sizes.

Likewise, the software runtime can profile similar information. The runtime uses the `rdtscp` time counter for x86 code, whereas ARM code uses the virtual `cntvct` counter scaled by clock frequency. The runtime also measures inter- and intra-thread communication overhead and logs the number of tokens traversed on each link.

The profiling information obtained from StreamBlocks are then directly injected to the MILP formulas. All that remains is solving the MILP optimization problem which then results in (multiple) heterogeneous partitions.

5 EVALUATION

We performed our experiments on two systems. The first was a single node in the an HACC[4] cluster that contains an Intel Xeon Gold 6234 8-core (16-thread) 3.3GHz processor and an Alveo U250 accelerator card connected to the system through a Gen3 PCIe x16 slot. This system is representative of data center accelerator platforms, with a high-end processor and a large FPGA. To evaluate our approach for embedded systems, we used the ZCU106 development board consisting of an MPSoC with 4 ARM64 cores running at 1.2GHz and a medium-size FPGA.

[4]Heterogeneous Accelerated Compute Cluster

5.1 Benchmarks

We used a suite of benchmarks of varying complexity from different application domains. Table 1 summarizes these benchmarks, with performance on the U250 platform. The benchmarks are:

- **JPEG Blur** performs 8 coarse tasks: parsing, Huffman decoding, dequantization, cosine transform, macro block to raster conversion, Gaussian blur filter [19], and raster to macro block conversion.
- **RVC-MPEG4SP** video decoder is a *reference* implementation of the RVC-MPEG4SP ISO/IEC 14496-2 MPEG-4 standard. This design has been used in many CAL-related publications.
- **Smith-Waterman** is an implementation of the Smith-Waterman string matching algorithm for DNA alignment [10].
- **SHA1** is a straightforward implementation of the SHA1 algorithm with eight SHA1 compute engines. Each engine has a padding actor and a compute actor.
- **Bitonic sort** is an eight-element bitonic sort implementation.
- **FIR** a 64-tap pipelined FIR filter.
- **IDCT** Inverse cosine discrete transformation used in video and image decoding.

The end-to-end throughput numbers in Table 1 reflect three scenarios:

- **hardware.** All actors are placed on the FPGA with the exception of 2 or 3 actors that perform IO operations.
- **single.** All actors are placed on a single software thread.
- **many.** Each actor runs on its own thread, e.g., with 104 actors, 104 threads are used.

The speedup numbers in Table 1 use the **single**-thread performance of each benchmark as its base-line. The speedup is reported as the maximum achieved throughput of either the **hardware** or the **many**-thread implementation. This shows what can be trivially achieved by only taking advantage of task-parallelism in absence of heterogeneous execution.

As evident in Table 1, using a thread per actor frequently degrades performance because of the cost of thread scheduling and inter-thread communication. Furthermore, placing all actors in hardware does not necessarily result in the best performance. The three throughput measurements in Table 1 represent the three corners of the design space, but not necessarily the best performing points. In fact, we demonstrate that when multiple software threads work in tandem with the FPGA better performance is achieved.

Below, we focus only on JPEG Blur and RVC-MPEG4SP, as they are fairly large and contain a set of computational kernels with dynamic behavior.

5.2 Design Space Exploration

5.2.1 Methodology. We use the MILP formulation presented in Section 3 to automatically explore the design space of the JPEG

Benchmark	Actors	FIFOs	hardware	single	many	unit	speedup	Domain
JPEG Blur	104	210	881.96	161.33	127.06	frames/second	5.47	Video decoding/stencil
RVC-MPEG4SP	60	123	1858.62	868.61	472.44	frames/second	2.14	Video decoding
Smith-Waterman	8	30	12911.56	3967.07	204.39	alignments/second	3.25	Sequence alignment
SHA1	20	26	130.64	53.75	177.66	MiB/second	3.31	Encryption
Bitonic Sort	28	57	6443K	5215K	5477K	sort/second	1.23	Hardware sorting
FIR	34	45	56	7.2	0.16	MiB/second	7.8	1D convolution
IDCT	7	9	1612K	979K	2039K	macroblock/second	2.08	Inverse cosine transform

Table 1: An overview of the benchmarks on Intel Xeon 6234 + Alveo U250.

Blur and RVC-MPEG4SP benchmarks on both platforms. To do so, we solve the MILP formulation for a fixed number of threads, with and without an FPGA. We vary the number of threads from 1-8 on the datacenter platform and 1-4 on the embedded platform since the Xeon processor has 8 cores and the ARM processor has 4. We keep the top 8 to 10 optimization solutions given a core count.

The partitioning tool then checks all the solutions in range of available number of cores and extracts the partitions that have the same *set* of hardware actors. This ensures that we compile each FPGA implementation (bitstream) only once. For instance, a heterogeneous multi-threaded partition with 4 cores may use the same set of hardware actors as a multi-threaded partition with 2 cores.

To demonstrate the advantages of heterogeneous auto-partitioning, we also run a configuration in which the MILP optimization is constrained *not to use hardware* to obtain multi-threaded *software-only* partitions.

We allocate 1-4 MiB OpenCL buffers between the CPU and the FPGA to decrease the host-device transfer overheads. Inside the FPGA, FIFO sizes are set by the values in the CAL code or set to 4096 on Alveo U250 and 512 on ZCU106 (default value) if unspecified[5]. Likewise, for multi-threaded software, we do not override the buffer configuration since buffers that do not span software and hardware are not as sensitive to payload size. The selection of buffer size is reflected in the MILP formulation as parameters. Without this design choice, hardware performance is severely bottlenecked by poor hardware-software data transfers and, in fact, the MILP optimizer fails to find meaningful hardware partitions. The outcome of our experiments are summarized in Table 2.

5.2.2 JPEG Blur. Fig. 4 illustrates the various evaluated design points of the JPEGBlur benchmark on the Alveo U250 and ZCU106 platforms.

- **Alveo U250** The MILP optimization finds 34 hardware actor partitions (each containing multiple actors), 67 multi-threaded heterogeneous partitions, and 63 multi-threaded software-only partitions. Note that, the number of multi-threaded heterogeneous partitions is more than the unique hardware actor partitions because for a given hardware actor partition, there are many mappings of software actors to threads (i.e., 67 > 34).
- **ZCU106** Design space exploration yielded 12 hardware partitions, 17 heterogeneous multi-threaded partitions, and 30 multi-threaded software-only partitions.

We now give some insight behind the unexpected scaling behavior shown in Fig. 4.

As expected, the throughput of a multi-threaded software implementation of JPEG Blur scales linearly with the number of cores on both platforms.

Contrarily, scaling is not well-behaved in a heterogeneous system: Alveo U250 exhibits a sub-linear speedup compared to software-only execution, whereas ZCU106 sees a slowdown.

Starting, from an FPGA-only partition, we can remove some actors from hardware and place them on a software thread. If the moved actors have better isolated performance on software, this could result in a net-increase in performance even though by moving actors from "infinitely" parallel hardware to software we are, in effect, *removing parallelism.* Effectively, actors with long sequential actions[6] are likely to perform better on processors due an order of magnitude higher clock frequency.

[5]Buffer dimensioning for optimal performance or deadlock avoidance can be done with external tools, cf. TURNUS [11].

[6]E.g., loops with inter-iteration dependencies or nested loops with dynamic bounds.

	Benchmark		JPEG Blur	RVC-MPEG4SP
U250	**baseline** (fps)		161.33	868.61
	SW	**partitions**	63	70
		speedup	6.90	4.78
	SW + FPGA	**partitions**	67	66
		bitstreams	34	38
		speedup	7.68	4.40
ZCU106	**baseline** (fps)		22.13	109.87
	SW	**partitions**	30	30
		speedup	3.83	3.92
	SW + FPGA	**partitions**	17	14
		bitstreams	12	8
		speedup	27.14	14.44

Table 2: Design space exploration summary of the JPEG Blur and RVC-MPEG4SP benchmarks on the U250 data-center and the ZCU106 embedded platforms. Speedup numbers correspond to the maximum performance achieved with and without hardware. Bitstreams counts the number of unique hardware partitions (which may have multiple software-thread partitions).

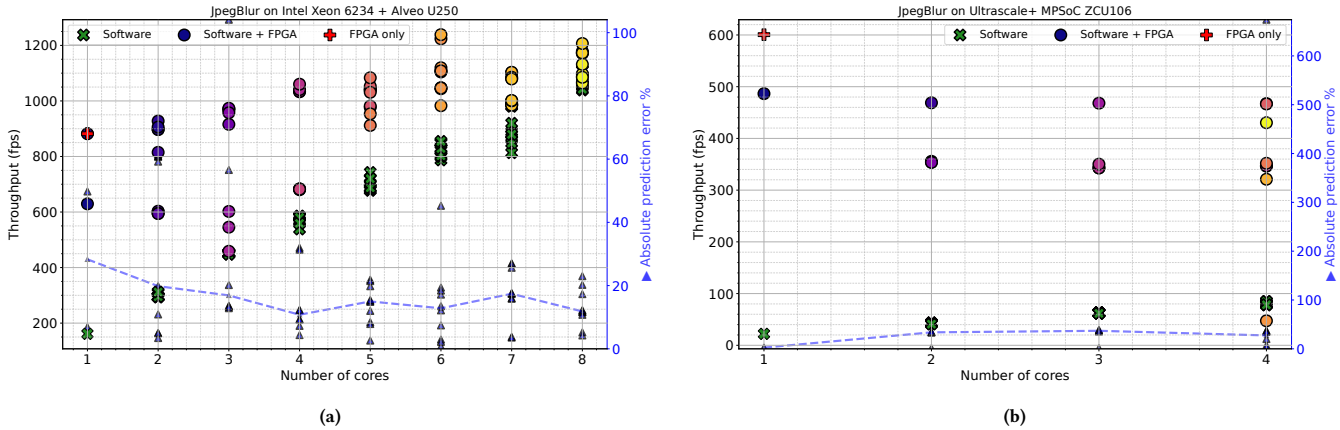


Figure 4: Evaluated JPEG Blur design points on (a) Alveo U250 and (b) ZCU106 platforms. Different colors labeled as Software + FPGA denote different FPGA bitstreams. The right axis shows the relative prediction errors for Software+FPGA points with dashed lines showing median.

Another interesting observation is that the same hardware partition yields different performance depending on how the remaining software actors are partitioned across threads. The heterogeneous points in Fig. 4 are color-coded by their hardware actor partitions, i.e., points that have the same color contain the same set of actors on hardware.

Fig. 4a demonstrates that the optimum design point is not necessarily a hardware-only or software-only configuration, but rather a mixture. Without a common programming model to facilitate exploration of many partitions, exploring these design points would be a tedious process involving code rewriting. This claim is reinforced by Fig. 5, which depicts the best performing partition on Alveo U250. This partition places a substantial number of actors on 6 software threads and boosts the speedup from about 5.5× (all hardware) to 7.7× (heterogeneous). This partition places most of the hardware-friendly actors on the FPGA and finds a balanced thread-partitioning on software.

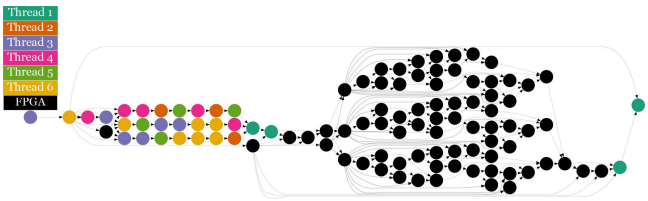


Figure 5: JPEG Blur configuration with the highest throughput.

5.2.3 RVC-MPEG4SP. Fig. 6 depicts our design exploration results for the RVC-MPEG4SP benchmark:

- **Alveo U250** Our search resulted in 38 unique hardware actor partitions, 66 heterogeneous multi-threaded partitions, and 70 multi-threaded software-only configurations.
- **ZCU106** We found 8 hardware actor partitions, 14 heterogeneous multi-threaded partitions, and 30 multi-threaded software-only ones.

Again, we see a an almost linear scaling in the multi-threaded software-only points and sub-linear scaling of multi-threaded heterogeneous points on Alveo U250.

On the ZCU106 platform, software scaling is almost perfectly linear for the 4 available cores. The heterogeneous performance sees a marginal increase with additional cores, unlike Fig. 4b, where heterogeneous performance experiences a slight dip with more cores.

The best heterogeneous Alveo U250 partition is outlined in Fig. 7. Despite the fact that 87% of the actors are faster in isolation on software, our MILP optimization identifies a heterogeneous partitioning that achieves a 4.4× speed up. This is because the MILP optimization models actor communication costs and places heavily-communicating actors on hardware to benefit from fast on-chip FIFOs in tandem with a high degree of spatial parallelism.

Our experiments show the importance of design-space exploration on modern heterogeneous platforms. FPGAs are usually the go-to choice in an embedded platform with "wimpy" processors, whereas powerful datacenter-grade platforms can benefit from heterogeneous partitionings. Again we stress that our approach can achieve this performance automatically.

5.3 Limitations

We close our evaluation by discussing key limitations.

5.3.1 Frontend Language. StreamBlocks simultaneously eases high-level programming of heterogeneous platforms and design-space exploration by starting from a unified and portable programming model. Of course the choice of the high-level language imposes many limitations on the end-to-end quality of results. That said, StreamBlocks is certainly not the be-all-end-all compiler for CPU-FPGA platforms, but a correct step in exposing FPGAs as

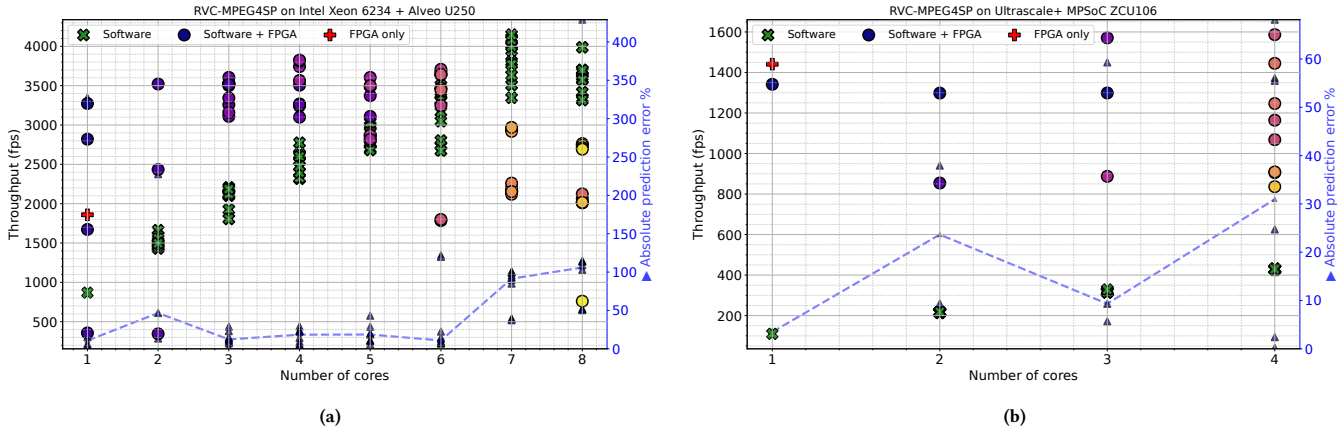


Figure 6: Evaluated RVC-MPEG4SP design points on (a) Alveo U250 and (b) ZCU106 platforms. Different colors labeled as Software + FPGA denote different FPGA bitstreams. The right axis shows the relative prediction errors for Software+FPGA points with dashed lines showing median.

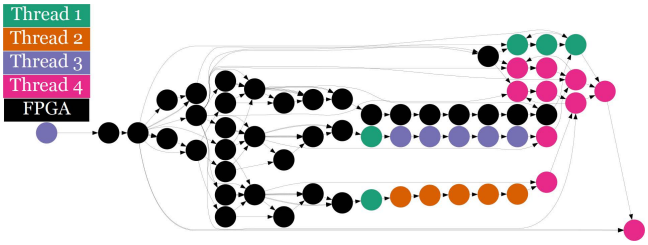


Figure 7: Best performing heterogeneous partition for the RVC-MPEG4SP benchmark.

programmable resources to a wider audience that do not wish to deal with minute hardware optimizations. Furthermore, the Actor model offers an opportunity to readily scale to multi-node CPU-FPGA clusters, a future direction we are investigating.

5.3.2 Optionality. Our modeling makes simplifying assumptions about the execution dynamics of a system in favor of a simple analytical formula. Our results therefore do not reflect *true* global optimal design points. Although we find optimal design points by solving the MILP formulation, the optimally is with respect to our approximate system model.

The right vertical axis in Fig.4 and 6 show the absolute prediction error of our MILP formulation for Software + FPGA partitions. In general, the errors grow with additional cores and can eventually become unreasonably high. Our MILP formulation can not model dynamic behavior and fine interactions between hardware and software, both of which are more likely to happen with increased inter-partition communication.

6 RELATED WORK

Our work explores the possibility of using a unified programming model for heterogeneous computing that allows automatic design-space exploration for task partitioning. We focus on the related research that aims to deliver portability and/or to expedite design-space exploration.

6.1 Languages

Using a single source language to design both hardware and software is a well-trodden research domain. However, most of the work do not handle task-partitioning automatically, or do so in a constrained environment.

TornadoVM [25] is a virtual machine for managing Java-based streaming tasks on heterogeneous platforms incorporating CPUs, GPUs, and FPGAs. It can dynamically reconfigure tasks written in Java on available hardware resources. TornadoVM can dynamically offload a single task to an FPGA based on runtime information. Lime [3] is a dataflow Java-based unified programming language for heterogeneous platform that supports offloading dataflow tasks to an FPGA. SCORE [12, 20] proposes a stream computation model with a system architecture capable of time-domain multiplexing. It enables placing applications that require more spatial resources that are available on an FPGA using loadtime and runtime scheduling and placement techniques. None of the work above envisions a heterogeneous multi-threaded auto-partitioning methodology like StreamBlocks.

LINQits [17] takes a program written in LINQ and accelerates query operation with a template library of hardware accelerators for an embedded heterogeneous MPSoC. Unlike our work, LINQits focuses on accelerating the whole program (or kernel) while StreamBlocks provides a flexible boundary for acceleration determined through design-space exploration.

Although CAL is a high-level language, its guarded actions resemble the guarded rules that the BlueSpec HDL uses to facilitate description of state machines. This has resulted in CAL being used for hardware-software co-design. The early work focused on either hardware code generation [5, 28, 30, 44] or pure software code such as the Orcc compiler [52]. Exelixi [4] is a CAL compiler that targets

both hardware and software but it is tied to an embedded platform without any means for design-space exploration. Unlike prior work, StreamBlocks translate guarded actions to the Actor Machine intermediate representation which codifies action selection and memoizes conditional evaluation between invocations.

Vitis/Vivado HLS/OpenCL, Intel OpenCL SDK, Intel OneAPI, Maxeler MaxJ, and the SOFF [32] OpenCL compiler are HLS compilers with a clear distinction between host and device code. Naturally, programs written in these compilers are not amenable to *automatic* task-partitioning. It is worth noting that most HLS compilers offer streaming primitives that are equivalent to those that are an inherent part of Cal to support task parallelism.

Lastly, HeteroRefactor [37] and HeteroGen [53] try to address the CPU-FPGA interoperability issue by transpiling C to C-HLS. They show an interesting opportunity to automatically generate code that is semantically equivalent and performs better than human translations. StreamBlocks could potentially use the same techniques for transpiling C/C++ code to the AM intermediate representation and reuse existing code for task partitioning.

6.2 Design-Space Exploration

SystemCoDesigner [34] is an automated hardware-software design-space exploration framework and SoC generator for SysteMoC [23] actor programs. SystemCoDesigner generates a prototype FPGA implementation with *soft* processors as the result of exploration. LegUp [9] is an HLS compiler that semi-automatically partitions C programs on a *soft* MIPS core and FPGA accelerated hardware. StreamBlocks differs from the two latter work by targeting a broader range of modern platforms, from heterogeneous, *hard* multi-core SoCs to data center FPGA-accelerated platforms with powerful processors. Our results for embedded platforms corroborates their findings that on embedded systems with a less powerful processor, it is usually best to offload all work to FPGAs.

StreamBlocks focuses on exploiting *task-parallelism* for hardware-software co-design. Accelerating loops by hardware-software co-design has been studied extensively [40]. RIP [54] is a MILP-based SoC hardware-software partitioning and scheduling tool for affine loops using an atomic task model. RIP's model is less expressive than the *general* dataflow model used in StreamBlocks.

TURNUS [11] is a design-space exploration and optimization tool that uses post-mortem execution traces of a dataflow application and profiling information to find partitions only for multi-core platforms through repeated trace simulations and heuristics. StreamBlocks supports both homogeneous and heterogeneous partitioning through our novel MILP formulation, that does not require collecting and simulating huge program traces.

An orthogonal approach to task partitioning is task optimization. Recent work focuses on transforming software-like code [18, 24, 36, 45, 47, 50, 51] to fine-tuned HLS code using design-space exploration to ease accelerator development. StreamBlocks can be vertically integrated with a task-optimizing design-space exploration engine and use more optimized actor implementations as the input to its partitioning tool.

7 CONCLUSION

This work presents StreamBlocks, a new dataflow compiler for FPGA-based heterogeneous platforms. StreamBlocks unifies hardware and software development under a single programming model. StreamBlocks makes expressing complex task-parallel programs that can cross hardware-software boundaries possible without any hardware expertise.

StreamBlocks compiler generates and interconnects computation on software and hardware in heterogeneous platforms. StreamBlocks' main contribution is a profile-guided auto-partitioning algorithm that finds non-trivial heterogeneous partitions with speedups ranging between 4× to 7× single-thread partitions and requires no hardware expertise or code modification from the developer to do so.

We believe StreamBlocks is a step towards democratizing FPGAs as a readily-available compute resource in the cloud, especially for non-hardware developers.

Our future work will include expanding our exploration methodology to multi-node heterogeneous systems and design-space exploration for optimizing individual hardware actors prior to partitioning.

8 ACKNOWLEDGMENTS

We thank Sahand Kashani of Very Large Scale Computing Laboratory at EPFL for his insightful discussion and internal reviews. We thank Gurobi for providing us an academic license to use the Gurobi Optimizer. This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program (formerly known as the XACC program - Xilinx Adaptive Compute Cluster program).

REFERENCES

[1] ISO/IEC 23001-4:2011. 2011. Information technology - MPEG systems technologies - Part 4: Codec configuration representation.

[2] Sassan Ahmadi. 2016. *Toward 5G Xilinx Solutions and Enablers for Next-Generation Wireless Systems.* White paper. Xilinx Inc.

[3] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses: The Programmability of FPGAs Must Improve If They Are to Be Part of Mainstream Computing. *Queue* 11, 2 (feb 2013), 40–52. https://doi.org/10.1145/2436696.2443836

[4] E. Bezati, S. Casale-Brunet, R. Mosqueron, and M. Mattavelli. 2019. An Heterogeneous Compiler of Dataflow Programs for Zynq Platforms. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1537–1541. https://doi.org/10.1109/ICASSP.2019.8682525

[5] E. Bezati, M. Mattavelli, and J.W. Janneck. 2013. High-level synthesis of dataflow programs for signal processing systems. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*. 750–754. https://doi.org/10.1109/ISPA.2013.6703837

[6] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. 1995. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, Vol. 5. 3255 –3258 vol.5. https://doi.org/10.1109/ICASSP.1995.479579

[7] J. T. Buck and E. A. Lee. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 1. 429–432 vol.1. https://doi.org/10.1109/ICASSP.1993.319147

[8] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 109–116. https://doi.org/10.1109/FCCM.2014.42

[9] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. ACM, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[10] S. Casale-Brunet, E. Bezati, and M. Mattavelli. 2017. Design space exploration of dataflow-based Smith-Waterman FPGA implementations. In *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*. 1–6. https://doi.org/10.1109/SiPS.2017.8109982

[11] Simone Casale-Brunet, Abdallah Elguindy, Endri Bezati, Richard Thavot, Ghislain Roquier, Marco Mattavelli, and Jorn W Janneck. 2013. Methods to explore design space for MPEG RMC codec specifications. *Signal Processing: Image Communication* 28, 10 (2013), 1278–1294.

[12] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. 2000. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*. Springer-Verlag, Berlin, Heidelberg, 605–614.

[13] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[14] Gustav Cedersjö and Jörn W. Janneck. 2019. Tÿcho: A Framework for Compiling Stream Programs. *ACM Trans. Embed. Comput. Syst.* 18, 6, Article 120 (Dec. 2019), 25 pages. https://doi.org/10.1145/3362692

[15] Yuze Chi, Licheng Guo, Jason Lau, Young kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. arXiv:2009.11389 [cs.AR]

[16] J. Choi, Ruo Long Lian, S. Brown, and J. Anderson. 2016. A unified software approach to specify pipeline and spatial parallelism in FPGA hardware. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 75–82. https://doi.org/10.1109/ASAP.2016.7760775

[17] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/2485922.2485945

[18] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. *Source-to-Source Optimization for HLS.* Springer International Publishing, Cham, 137–163. https://doi.org/10.1007/978-3-319-26408-0_8

[19] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. 2014. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers. In *Proceedings of the 51st Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2593069.2593090

[20] 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. *Microprocessors and Microsystems* 30, 6 (2006), 334–354. https://doi.org/10.1016/j.micpro.2006.02.009 Special Issue on FPGA's.

[21] Jack B. Dennis. 1974. First version of a data flow procedure language. In *Symposium on Programming*. 362–376.

[22] J. Eker and J. Janneck. 2003. *CAL Language Report.* Technical Report ERL Technical Memo UCB/ERL M03/48. University of California at Berkeley.

[23] Joachim Falk, Christian Haubelt, Jürgen Teich, and Christian Zebelein. 2017. SysteMoC: A Data-Flow Programming Language for Codesign. In *Handbook of Hardware/Software Codesign*, Teich J Ha S (Ed.). Vol. 1. Springer, Dordrecht, The Netherlands, 59 – 97.

[24] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. 2018. Lattice-Traversing Design Space Exploration for High Level Synthesis. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 210–217. https://doi.org/10.1109/ICCD.2018.00040

[25] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/3313808.3313819

[26] Mentor Graphics Inc. 2015. *GOOGLE DEVELOPS WEBM VIDEO DECOMPRESSION HARDWARE IP USING TECHNOLOGY INDEPENDENT SOURCES AND HIGH-LEVEL SYNTHESIS.* White paper.

[27] J.W. Janneck. 2011. A machine model for dataflow actors and its applications. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*. 756–760. https://doi.org/10.1109/ACSSC.2011.6190107

[28] Jörn Janneck, Ian Miller, David Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. 2009. Synthesizing Hardware from Dataflow Programs: An MPEG-4 Simple Profile Decoder Case Study. *Journal of Signal Processing Systems* 63, 2 (2009), 241–249. http://dx.doi.org/10.1007/s11265-009-0397-5 10.1007/s11265-009-0397-5.

[29] Jorn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickael Raulet. 2008. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *2008 IEEE Workshop on Signal Processing Systems*. 287–292. https://doi.org/10.1109/SIPS.2008.4671777

[30] K. Jerbi, M. Raulet, O. Deforges, and M. Abid. 2012. Automatic generation of synthesizable hardware implementation from high level RVC-CAL description. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. 1597–1600. https://doi.org/10.1109/ICASSP.2012.6288199

[31] Khaled Jerbi, Daniele Renzi, Damien De Saint Jorre, Hervé Yviquel, Mickaël Raulet, Claudio Alberti, and Marco Mattavelli. 2014. Development and optimization of high level dataflow programs: The HEVC decoder design case. In *2014 48th Asilomar Conference on Signals, Systems and Computers*. 2155–2159. https://doi.org/10.1109/ACSSC.2014.7094857

[32] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. 2020. SOFF: An OpenCL High-Level Synthesis Framework for FPGAs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 295–308. https://doi.org/10.1109/ISCA45697.2020.00034

[33] Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*. 471–475.

[34] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. 2009. SystemCoDesigner—an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Trans. Des. Autom. Electron. Syst.* 14, 1, Article 1 (Jan. 2009), 23 pages. https://doi.org/10.1145/1455229.1455230

[35] Yongfeng Gu Kiran Kintali and Eric Cigan. 2014. *Model-Based Design Using Simulink, HDL Coder, and DSP Builder for Intel FPGAs.* White paper. Matlab Inc.

[36] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. *SIGPLAN Not.* 53, 4 (jun 2018), 296–311. https://doi.org/10.1145/3296979.3192379

[37] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 493–505. https://doi.org/10.1145/3377811.3380340

[38] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[39] E.A. Lee and T.M. Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (May 1995), 773 –801. https://doi.org/10.1109/5.381846

[40] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. 2000. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 37th Annual Design Automation Conference* (Los Angeles, California, USA) *(DAC '00)*. Association for Computing Machinery, New York, NY, USA, 507–512. https://doi.org/10.1145/337292.337559

[41] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. arXiv:1807.04188 [cs.LG]

[42] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, and et al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 13–24. https://doi.org/10.1145/2678373.2665678

[43] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. 2018. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 9–16. https://doi.org/10.1109/FCCM.2018.00011

[44] N. Siret, M. Wipliez, J.-F. Nezan, and A. Rhatay. 2010. Hardware code generation from dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. 113–120. https://doi.org/10.1109/DASIP.2010.5706254

[45] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2021. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. arXiv:2009.14381 [cs.AR]

[46] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (2010), 66–73.

[47] Qi Sun, Tinghuan Chen, Siting Liu, Jin Miao, Jianli Chen, Hao Yu, and Bei Yu. 2021. Correlated Multi-objective Multi-fidelity Optimization for HLS Directives Design. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 46–51. https://doi.org/10.23919/DATE51398.2021.9474241

[48] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner. 2016. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*. 36–43. https://doi.org/10.1109/FPT.2016.7929186

[49] Xilinx. [n. d.]. *Vivado Design Suite User Guide - High-Level Synthesis*. Xilinx Inc.

[50] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2021. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. arXiv:2107.11673 [cs.PL]

[51] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465827

[52] Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet. 2013. Orcc: Multimedia Development Made Easy. In *Proceedings of the 21st ACM International Conference on Multimedia (MM '13)*. ACM, 863–866. https://doi.org/10.1145/2502081.2502231

[53] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. 2022. HeteroGen: Transpiling C to Heterogeneous HLS Code with Automated Test Generation and Program Repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 1017–1029. https://doi.org/10.1145/3503222.3507748

[54] Wei Zuo, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. 2017. Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062195

A APPENDIX

A.1 MILP Formulation Details

T^{write}_{plink} and T^{read}_{plink} are upper bound estimates of the runtime cost of transferring data between host and FPGA. We represent the set of all connection as the set C and such that $(s,t) \in C$ denotes a connection from source port s to target port t. Furthermore, $s.a$ and $t.a$ denote the actor that the port belongs to.

We obtain the number of tokens traversed a connection (s,t) through profiling and denote it as $n_{(s,t)}$. Every connection also has an associated buffer size $b_{(s,t)}$. OpenCL read and write operations are most efficient if a full buffer (i.e., $b_{(s,t)}$ tokens) are transferred.

We measured the OpenCL transfer operation times from queueing to completion using OpenCL event counters to obtain a function $\xi_r(b)$ and $\xi_w(b)$ that models OpenCL read and write time given a buffer configuration b. Using this function we can estimate the best case time required to write n tokens over buffers with capacity b as:

$$\tau_w(n,b) = \begin{cases} \xi_w(n) & n \leq b \\ \xi_w(b) \times \lfloor \frac{n}{b} \rfloor + \xi_w(n \bmod b) & n > b \end{cases} \tag{4}$$

Here, $\lfloor . \rfloor$ and *mod* are the floor and modulo operators. To estimate read times, a function $\tau_r(n,b)$ is similarly defined by replacing $\xi_w(b)$ with $\xi_r(b)$. Now we can estimate PLink read and write times as follows:

$$\begin{aligned} T^{write}_{plink} &= \sum_{(s,t)\in C} (\neg d^{s.a}_{accel} \wedge d^{t.a}_{accel}) \tau_w(n_{(s,t)}, b_{(s,t)}) \\ T^{read}_{plink} &= \sum_{(s,t)\in C} (d^{s.a}_{accel} \wedge \neg d^{t.a}_{accel}) \tau_r(n_{(s,t)}, b_{(s,t)}) \end{aligned} \tag{5}$$

Where $\wedge$ and $\neg$ are the logical *conjunction* and *negation* operators respectively. The conjunction expressions ensure that read and write times are only accounted for connections with one port on the FPGA and the other on the CPU.

Similarly, for every partition $p \in P_{thread}$, we define

$$t^p_{intra} = \sum_{(s,t)\in C} (d^{s.a}_p \wedge d^{t.a}_p) \tau_{intra}(n_{(s,t)}, b_{(s,t)}) \tag{6}$$

Where τ_{intra} is models the intra-core communication time and is obtained by profiling software FIFO read and write bandwidth. To do this, we measure the roundtrip times of sending a token from one port and receiving in from another port going through a pass-through actor. Therefore, we use the same bandwidth value for both read and write (i.e., time for read *or* write is round-trip time divided by 2).

However, notice that if a connection (s,t) has its source actor on the first partition (i.e., p_1 which also contains the PLink) and its target actor is on the FPGA, the data should first be copied to the PLink and then to the device. This can be modeled by t^{plink}_{intra} as follows:

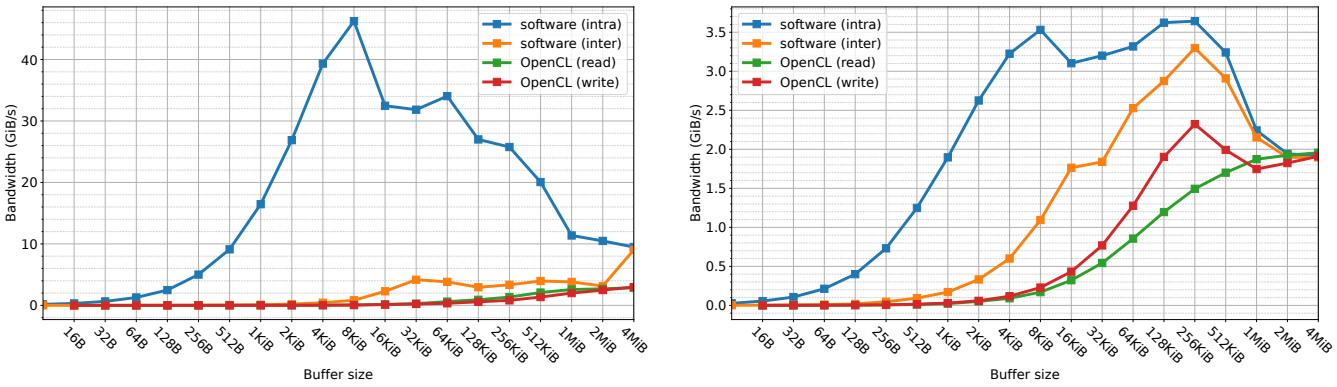
$$t^{plink}_{intra} = \sum_{(s,t)\in C} \left((d^{s.a}_{p_1} \wedge d^{t.a}_{accel}) \vee (d^{s.a}_{accel} \wedge d^{t.a}_{p_1}) \right) \tau_{intra}(n_{(s,t)}, b_{(s,t)}) \tag{7}$$

Where $\vee$ is the logical disjunction operator. We can now define the local core communication cost for each partition $p \in P_{thread}$ as:

$$T^p_{intra} = \begin{cases} t^p_{intra} & p \in P_{thread} \setminus \{p_1\} \\ t^p_{intra} + t^{plink}_{intra} & p = p_1 \end{cases} \tag{8}$$

Since all the threads operate in parallel, then the total intra-core communication time is obtained as the maximum of individual ones:

$$T_{intra} = max(\{T^p_{intra} : p \in P_{thread}\}) \tag{9}$$



(a) (b)

Figure 8: Measured FIFO communication bandwidth on (a) Alveo U250 and (b) ZCU106.

Finally, we estimate the core to core communication cost as follows:

$$
\begin{aligned}
T_{inter} = \sum_{(s,t)\in C} \Bigg(& \\
& \sum_{q \in P_{thread} \setminus \{p_1\}} \Big(\Big((d^{s.a}_{p_1} \vee d^{s.a}_{accel}) \wedge d^{t.a}_{q} \Big) \vee \\
& \Big(d^{s.a}_{q} \wedge (d^{t.a}_{p_1} \vee d^{t.a}_{accel}) \Big) \Big) \\
+ \sum_{p \in P_{thread} \setminus \{p_1\}} & \sum_{q \in P_{thread} \setminus \{p, p_1\}} \Big(\Big(d^{s.a}_{p} \wedge d^{t.a}_{q} \Big) \vee \Big(d^{s.a}_{q} \wedge d^{t.a}_{p} \Big) \Big) \\
& \Bigg) \tau_{inter}(n_{(s,t)}, b_{(s,t)}) \qquad (10)
\end{aligned}
$$

This formula includes a cost for all of the connection that cross a thread. Notice how the first partition p_1 which contains the PLink is treated slightly differently to handle connections from a thread $q \neq p_1$ to *accel* or p_1 and vice versa.

With the derivation of T_{inter}, T_{intra} and T^{read}_{plink} and T^{write}_{plink} our formulation is complete.

A.2 Communication Cost Measurements

In the MILP formulation, the communication cost is parameterized as a function $\xi(b)$ that represents the measured read or write times. If we plot the communication bandwidth as $b/\xi(b)$, then we obtain the Fig. 8a and Fig. 8b for the Alveo U250 and ZCU106 platforms respectively.

Notice the large difference between inter- and intra-core communication bandwidth in Fig 8a. This is because when the two ends of a FIFO are on the same core (i.e., pinned thread), the communication goes through the private caches (e.g., L1 and L2) without coherence traffic. When the two ends of a FIFO are on two different cores, the tokens travel at least to the shared last-level cache (i.e., L3), which incurs a coherence cost. As expected, the intra-core bandwidth increases with larger buffer transfers But there is when payloads become too large for private caches, then bandwidth drops.

Furthermore, we can observe that OpenCL read and write operations are considerably slower than software FIFO operations and they only partially catch up around 1MiB.

The communication cost terms in the MILP formulation penalize using too many threads or placing everything in hardware without accounting for the considerable overhead of OpenCL read and write operations or inter-core communication.