μIR -An intermediate representation for transforming and optimizing the microarchitecture of application accelerators

Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu*
Apala Guha, Tony Nowatzki*, Arrvindh Shriraman
Simon Fraser University, * University of California, Los Angeles
https://github.com/sfu-arch/muir

Abstract

Creating high quality application-specific accelerators requires us to make iterative changes to both algorithm behavior and microarchitecture, and this is a tedious and error-prone process. High-Level Synthesis (HLS) tools [5, 10] generate RTL for application accelerators from annotated software. Unfortunately, the generated RTL is challenging to change and optimize. The primary limitation of HLS is that the functionality and microarchitecture are conflated together in a single language (such as C++). Making changes to the accelerator design may require code restructuring, and microarchitecture optimizations are tied with program correctness.

We propose a generalized intermediate representation for describing accelerator microarchitecture, μIR, and an associated pass framework, μopt. μIR represents the accelerator as a concurrent structural graph in which the components roughly correspond to microarchitecture level hardware blocks (e.g., function units, network, memory banks). There are two important benefits i) it decouples microarchitecture optimizations from algorithm/program optimizations. ii) it decouples microarchitecture optimizations from the RTL generation. Computer architects express their ideas as a set of iterative transformations of the μIR graph that successively refine the accelerator architecture. The μIR graph is then translated to Chisel, while maintaining the execution model and cycle-level performance characteristics. In this paper, we study three broad classes of optimizations: Timing (e.g., Pipeline re-timing), Spatial (e.g., Compute tiling), and Higher-order Ops (e.g., Tensor function units) that deliver between 1.5 — 8× improvement in performance; overall 5—20× speedup compared to an ARM A9 1Ghz. We evaluate the quality of the auto-generated accelerators on an Arria 10 FPGA and under ASIC UMC 28nm technology.

1 Introduction

Current High-Level Synthesis (HLS) tools [5, 8, 10, 15, 43] translate a program, typically specified in C-like language, to synthesizable RTL. The RTL is verbose, is hard to modify and optimize, and supports a limited execution model (timing-linked operation schedule). Hence, many HLS tools [37] lift hardware description to the input C program and rely on ad-hoc compiler optimizations as proxies for microarchitecture optimization e.g., loop unrolling to create parallel function units [14, 18]. HLS tools expect designers to sprinkle ad-hoc annotations in the C program to side-step limitations in the compiler and ensure the quality of the final RTL output e.g., Xilinx's `stream<T>` in a C++ program translate to FIFO queues in RTL. Like HLS, Hardware construction languages (HCLs) are also motivated to raise the design abstraction [7, 35]. HCLs require the designer to specify a structural hardware description (as opposed to C-level behavior specification). While this enables the designer to precisely explore the hardware tradeoffs, it leaves unanswered the question of how to derive a good quality hardware description from software. Domain-specific HLS tools have sought to provide a software-feel to hardware construction by introducing higher order hardware controllers (e.g., for loops) [8, 22, 28, 31]. Unfortunately, they restrict the control and data patterns, and only target fixed hardware templates.

Our insight is that perhaps the limiting factor in prior work is their use of a single representation to capture both the behavior of the accelerator (i.e., the operational specification) and its microarchitecture (i.e., the structural specification). HLS toolchains require both to be specified in C variants, while HCLs require both to be specified in a hardware-oriented language. This conflating of microarchitecture and behavior limits the scope of accelerator to loop-based program behaviors [26, 55]. The hardware description generated by prior toolchains also tend to correspond to low-level RTL that only permit a limited set of transformations. **We propose an alternative —- decouple the representation used for accelerator microarchitecture and hardware optimizations from the functional behavior specification.** We develop μIR, a new intermediate representation for the back-end representation of the accelerator microarchitecture. We are motivated by software compilers that have long recognized the importance of an intermediate-layer for enabling optimizations prior to binary creation [51].

μIR is a structural graph that explicitly specifies the accelerator's microarchitecture components and orchestrates data movement between the different components. The higher-level representation (compared to RTL) makes it easier for both localized and global transformations to optimize the accelerator microarchitecture. Figure 1 provides the end-to-end view of our multi-stage framework that generates the RTL for a high-performance accelerator from a program. The multi-stage approach encourages a clear demarcation of behavior optimizations (e.g., loop unrolling), microarchitecture optimizations (e.g., memory banking), and RTL optimizations (e.g., FPGA-specific SRAMs).

These are the primary novelties of μIR, i) *Optimizability:* μIR represents hardware at a higher microarchitecture level of abstraction. In comparison to hardware languages (such as FIRRTL [24] and Verilog), μIR enables computer architects to concisely express

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358292

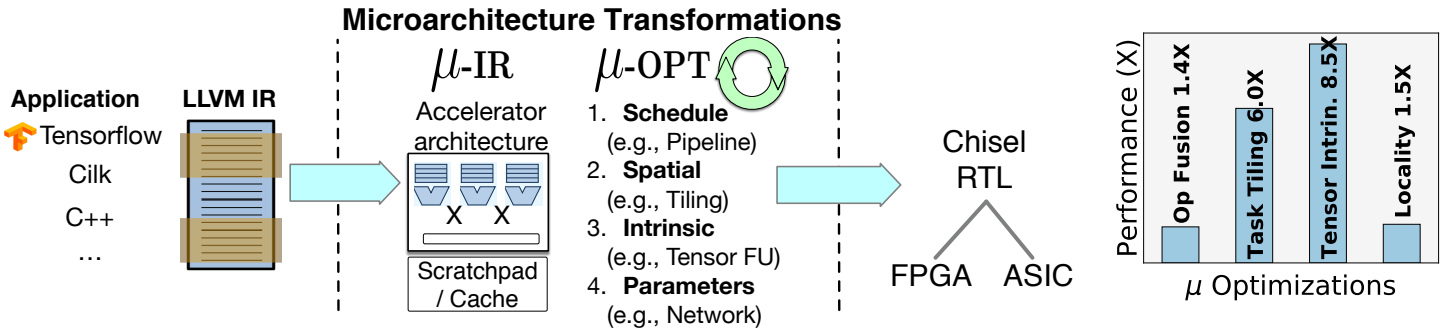


Figure 1: Overview of *μIR* toolchain: Stage 1: We translate C++, Cilk [34, 49], Tensorflow [38] programs to *μIR* graph of the accelerator. Stage 2: *μopt* applies transformations and optimizations on the *μIR* graph. Stage 3: We lower the *μIR* graph to Chisel RTL relying on an internal library of microarchitecture components, for subsequent mapping either to an FPGA or ASIC. Plot: Improvement in performance of application accelerators through μoptimization passes (see § 6 for details).

their optimizations. ii) *Tranformability: μIR* decouples the microarchitectural description from its behavior. Unlike HLS, this enables the microarchitecture to be changed iteratively without affecting behavioral correctness. *μIR* also more precisely describes the hardware and continues to preserve the expected cycle-level performance tradeoffs when translated to RTL (unlike HLS). iii) *Synthesizability*: *μIR*'s abstractions have been purposefully designed targeting heterogeneous parallel dataflow architectures. This enables the baseline *μIR* graph to be derived from software, unlike hardware construction languages such as Chisel [4, Section 1.1]. iv) *Composability:* All the edges in an *μIR* graph are governed by latency-agnostic interfaces. *μIR* exploits this property to ensure correctness when composing optimization passes. Prior work has exploited this property to modularize CPUs [57].

μopt is a toolchain that realizes architecture ideas as iterative transformations of the accelerator microarchitecture graph. We demonstrate that *μopt* is capable of applying three broad classes of optimizations: i)*Timing*, statically assigns operations to hardware units and changes pipelining, ii) *Spatial*, which replicates nodes representing hardware structures in the graph to improve throughput and reduce contention, iii) *Higher-Order Ops* enable a designer to introduce operators on composite data types such as Tensors (and vectors) to increase computational intensity. § 6 discusses the optimizations. During these transformations *μopt* tunes the parameters of *μIR* components to optimize the generated RTL (e.g., operator bit-width, channel width).

The plot in Figure 1 summarizes the benefit of four optimization passes. These microarchitectural changes can result in 1.5 — 8× improvement in performance. (see § 6.1—§ 6.4 for details). We synthesized all our accelerators on Arria 10 SoC board and also push them through a ASIC 28nm umc. Our contributions:

- We have created *μIR*, an intermediate representation for optimizing and generating accelerator microarchitecture. *μIR* is sufficiently generalized to automatically derive a baseline accelerator from unmodified software (currently tested using C++, Cilk and Tensorflow).
- We created an optimization framework, *μopt*, that decouples microarchitecture optimizations from the lower RTL. *μopt* helps designers realize optimizations as an iterative pass of the *μIR*, without having to modify RTL. *μopt* optimization passes can be automatically applied to different accelerators, and multiple passes can be stacked for a specific accelerator.
- We implemented three important classes of optimizations, Timing, Spatial, and Higher-Order Ops on five different components (compute units, concurrency control, memory network, scratchpads, caches). The optimizations result in between 1.4 — 8× improvement in performance over the baseline accelerator.

2 A Case for a Microarchitecture IR

2.1 Limitations of current HLS compilers

Current HLS compilers [10, 14] plug into software compilers and translate C to RTL (similar to generating a binary). Hence, they rely on software compiler's intermediate representation (IR) for representing a model of the hardware and rely on compiler transformations to optimize the microarchitecture.[1] Consider canonical loop unrolling, which HLS tools simply interpret as parallel hardware instances of instructions. Unfortunately, HLS compiler's IRs suffer from two broad limitations that make them ill suited for studying microarchitecture tradeoffs: 1) Transformations are performed using the control-driven Von-Neumann execution model and this limits the types of hardware designs that they can target and C behaviors they can support. For instance, HLS tools primarily focus on lowering loops to statically scheduled circuits [10]. 2) HLS IR represents execution behavior and not the structural components of a microarchitecture. This makes it challenging to understand how a transformation of the IR graph changes the RTL output, and quantify the performance and power tradeoffs (particularly in the presence of multiple transformations).

HLS compilers are aware of the challenges with creating microarchitectural transformations with a software compiler's IR. Hence, they encourage users to lift the microarchitectural descriptions to the C behavior description. However, this closely ties in behavioral correctness with microarchitecture structural description, both of which require different mental models. Further, note in many cases changes to microarchitecture are inter-dependent, and current HLS tools require the C source program to be modified iteratively for each accelerator (a labor-intensive task).

[1] HLS compilers do use an RTL-based hardware representation in the backend, but this primarily targets circuit transformations (e.g., vendor-specific BRAM), not microarchitectural transformations.

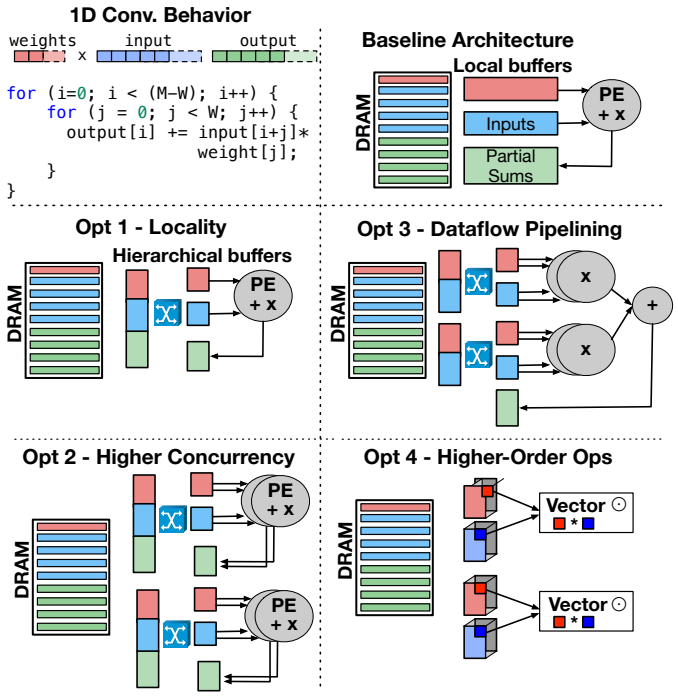


Figure 2: Overview of different microarchitectures that implement a 1D Convolution [2]

We highlight the limitations in HLS by considering microarchitecture-level transformations that a designer may seek to implement. Figure 2 expresses the behavior of a simple 1D convolution in C. The baseline microarchitecture that HLS creates is also shown. HLS lowers `int inputs[],outputs[],weight[]` arrays into local buffers and streams data from the DRAM;the loop body is offloaded to a single processing element on which iterations are time multiplexed.

- **Opt. 1 - Localities:** A common optimization in accelerators is to introduce multiple layers of buffering, with sharing (or not) between the compute units. In HLS, data movement between buffers is specified using copy operations in the C behavioral description. Unfortunately, this does not permit the designer to study how the data moves between the buffers and when the data actually moves. Further, these logical hardware buffers can choose different hardware implementations e.g., FIFOs vs Line-buffers vs Scratchpad. To specify these in HLS the designer has to make changes (potentially correctness affecting) to the input C program. In contrast, *μIR* is a structural-graph that permits the designer to explicitly introduce buffers and the communication logic that implicitly move data between these buffers.
- **Opt. 2 - Higher-level Concurrency:** Designers may seek to capture concurrency synergistically at multiple levels in the microarchitecture. In this example, we could implement the parallelism of the inner loop in a vector fashion, and then replicate each vector block in a fractal fashion to capture outer loop parallelism. HLS compilers IR are sequential and lack the abstractions to represent hierarchical parallelism in the microarchitecture. *μIR* is a concurrent specification and will allow the designer to sweep different tiling factors for each nested level in conjunction with changes to organization of the local RAM buffers.
- **Opt. 3 - Dataflow Pipelining:** HLS compilers do not expose concepts such as operation pipelining, as they primarily target statically scheduled (i.e., fixed latency) circuits. HLS compilers' IR also assume a machine model with unbounded resources and hence scheduling and mapping has to be achieved at the RTL level. This does not allow computer architects to budget resources (cycles, power, or area) of the microarchitecture's dataflow. *μIR* permits a more detailed expression of the microarchitecture and permits designer to: a) explore higher performance dynamically-scheduled pipelines to hide memory latency, and b) schedule operations to manage contention on the datapath's computational and memory resources.
- **Opt. 4 - Higher-Order Ops** Finally, current HLS compilers are rigid in their definition of operations and data types, typically RISC-style 3-operand (as they derived software compiler's IR). However, hardware accelerators are capable of supporting a richer variety of operators on complex shapes such as tensors. This necessitates an IR that permits the introduction of custom operators with transformations. [2]

As shown in Figure 2 when translating a behavior specification to hardware multiple microarchitecture designs that could be generated. *μIR* is essentially a software data structure that canonicalizes the description of microarchitectures and describes an execution model that is better matched for hardware (than HLS compiler IR). This permits a richer set of microarchitecture optimizations and promotes a quantitative clearer understanding of the performance implications. Finally, *μIR* automates the labor-intensive task of optimizing the microarchitecture description of each accelerator.

2.2 Related works

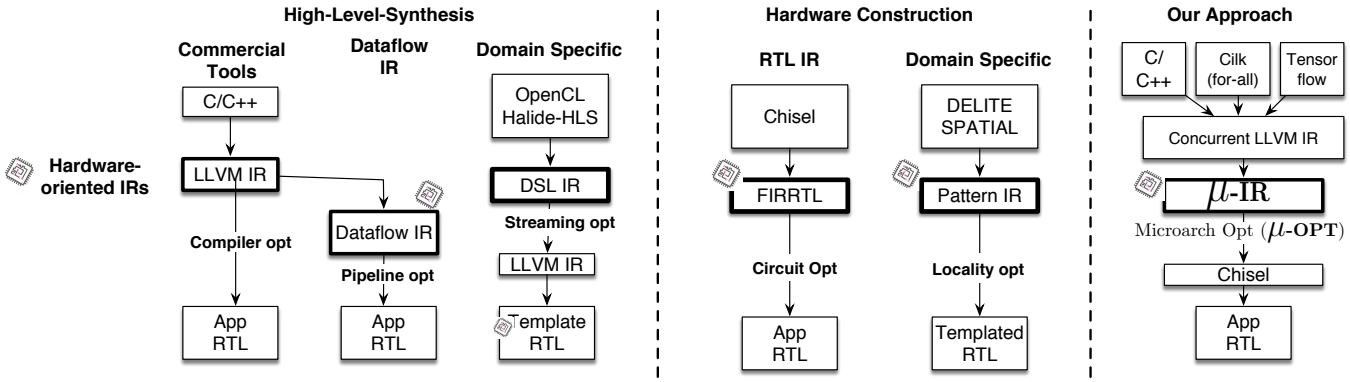
Table 1 summarizes comparison between *μIR* and other IRs that are used to build hardware accelerators.

There has been a number of works that have recognized the mismatch between compiler IRs and hardware execution. Pegasus [9], VSFG [56], and AHIR [48] all have created dataflow-based IRs [54] that unify the control, data, and memory dependency edges. Their primary target is to transform branches in compiler IR to predicates. This enables higher instruction-level-parallelism. The whole accelerator is described as a monolithic dataflow and the nodes in the graph roughly correspond to instructions in the compiler IR. Overall, such IRs are primarily suitable for dataflow pipelining.

There has been a spate of work in domain-specific languages (DSLs) for leveraging concurrency patterns [15, 20, 47, 52]. A common trait in these DSLs is that they embed information on parallel patterns within the compiler IR. The parallel patterns could either be data-parallel (OpenCL's SPIR-V), heterogeneous parallel (e.g., HPVM [30], TAPIR [49]), or domain-specific (e.g., Halide-HLS [42], MLIR [3]). While all IRs are focused on optimized code generation, only some of DSL IRs have an execution model that resembles hardware (e.g., Halide HLS, SPIR-V, Gorilla [32]) and can be fed to an HLS toolchain. However, many of these IRs target a known fixed microarchitecture [12, 31], and closely tie in algorithm and microarchitecture structure (e.g., line buffers [22, 42, 44, 46]). The

[2]HLS tools do introduce vendor-specific IPs, but these IPs are implemented as co-processors.

Table 1: Comparing the Intermediate-Representations



		High-Level Synthesis (HLS)		Hardware construction lang (HCL)		HLS + HCL
		HLS Toolchains Legup [10], Vivado [5, 14], Catapault-C [1], AHIR [48], Pegasus [9]	**Domain-specific IR** OpenCL SPIR [27], Halide-HLS [42], Gorilla++ [32]	**Modular RTL** FIRRTL [24]	**Pattern-based HCL** SPATIAL [28, 52], Pattern [17]	***μIR* and *μopt***
Design	Input	C program	Domain lang.	Chisel	Domain Lang.	C++/Cilk/Tensorflow
	Output	Flat dataflow	Pipeline of kernels	Flat dataflow	Pattern-based dataflow	Hierarchical dataflow
	IR Nodes	Compiler ops (e.g., add)	Kernels, Streams	Logic e.g., Mux,Adder	Concurrency patterns, Memory	Polymorphic ops, Dataflow node, Memory.
Target Opt.	Higher Parall.	—	✓Kernels e.g., stencil	—	✓Patterns e.g., for-all	✓General
	Op Pipelining	✓	—	✓	—	✓
	Locality	—	✓(streams)	—	✓(streams)	✓(partitioned. shared)
	Resource	—	—	—	—	✓(μarch level)
	Higher Ops	—	—	—	—	✓(Tensor ops)

primary benefit of these IRs is the ability to assure the domain-specific program is transformed to a structure well-suited to HLS toolchains.

FIRRTL and other hardware IRs: Recently, there has been work on raising the abstraction of hardware description [7, 35, 53]. The most popular is Chisel, which internally uses an IR, FIRRTL [24]. FIRRTL is closer in spirit to Verilog and many of the known passes only support localized circuit transformations (e.g., common-sub-expression elimination, backend specific RAMs). FIRRTL would present the following challenges if used as a microarchitectural IR — i) it would be challenging and verbose to write transformations that makes changes to overall microarchitecture graph (see Section 7 for details). ii) FIRRTL is not intended to be a backend for HLS and would restrict the types of software behaviors that can be lowered to hardware (e.g. nested loops are unrolled, no nested parallelism). There has been some work on introducing higher-level concurrent patterns in the hardware descriptions [28, 29, 36, 40, 41] and optimizing those patterns prior to lowering to RTL. These, however, tend to be pattern specific and target the organization at a fixed microarchitecture template such as grid-based spatial architectures.

***μIR* Summary:** *μIR*'s abstractions have been designed with a view to two requirements i) *μIR* must support high-level synthesis i.e., the input to our toolchain is a microarchitecture behavior described in software. The main reason being we would like to leverage software transformations such as loop unrolling to expose more opportunity for hardware transformations. ii) *μIR*'s execution model must resemble hardware and include a structural specification that permits hardware transformations, like hardware construction languages. This will ensure the performance characteristics of the transformations we implement on *μIR* will be retained when lowered to the final RTL. *μIR* creates a specific set of abstraction that target the construction of generalized heterogeneous-parallel dataflow architectures. Unlike Chisel [4, page 5], it is precise enough to be the backend target of an HLS system. Unlike FIRRTL[24], *μIR* helps designers express their optimizations in a concise manner. Unlike HLS, it helps designers realize a broad set of microarchitecture-level transformations.

3 Microarchitectural Intermediate-Representation (*μIR*)

Figure 3 provides an overview of the toolflow. The input to the flow is an unmodified software programs specifying behavior. We use the LLVM compiler framework's language bindings for Tapir [49], Cilk/OpenMP programs and Tensorflow [33] for helping translate

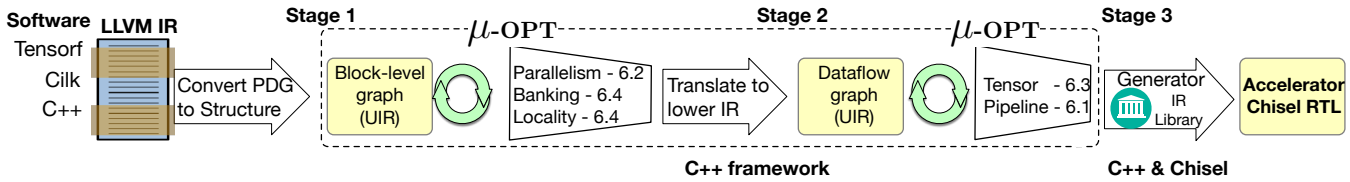


Figure 3: A summary of the code transformation stages in *μIR*. *μIR* is a C++ data structure and *μopt* passes are written in C++.

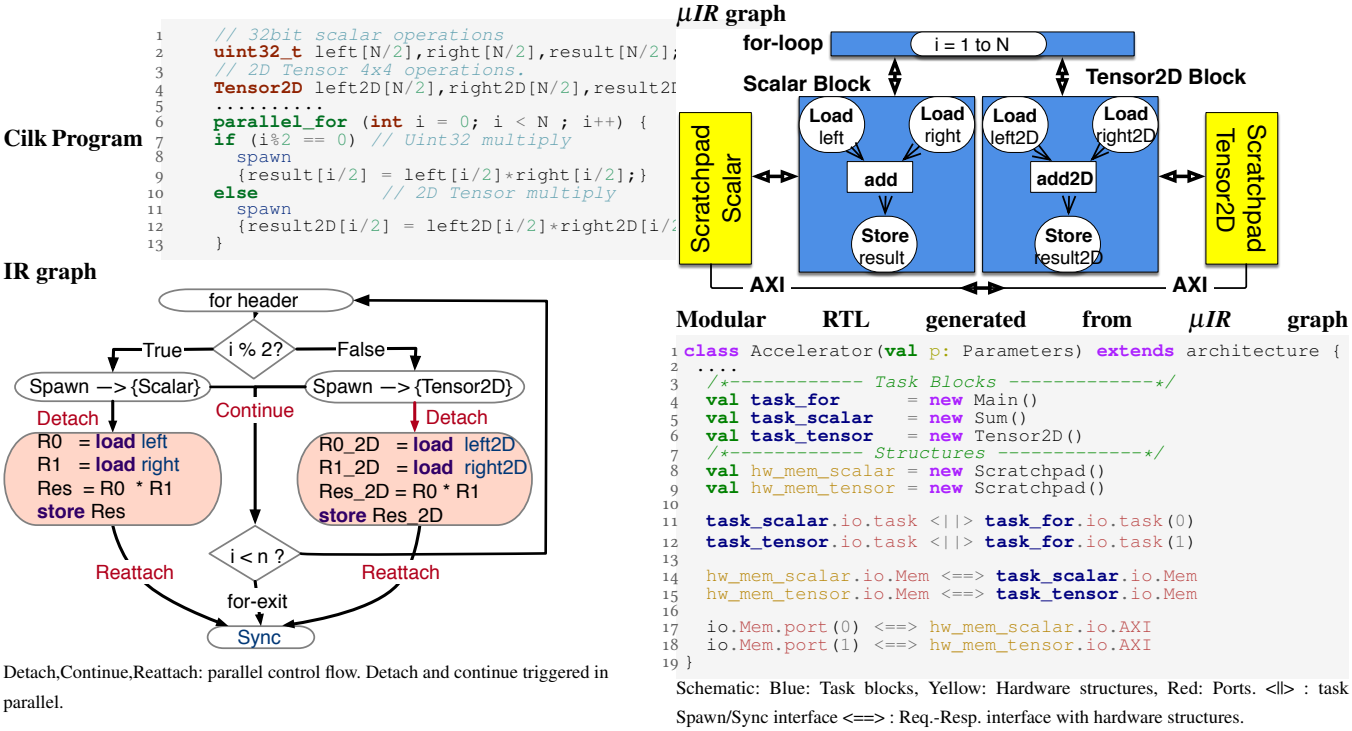


Figure 4: Cilk Example auto-translated to *μIR* specification (please view in color for syntax highlights)

input programs into LLVM compiler IR. Our tool automatically translates the program specification to a *μIR* hierarchical graph using the compiler IR as an intermediary step. We iterate over the *μIR* graph in a hierarchical fashion and lower the *μIR* graph into synthesizable Chisel. Lowering implies as we iterate over the graph, we create the Chisel code instantiating the code and connecting it to other components. This stage relies on a *μIR* library of components. *μIR* itself is simply implemented as a data structure and we provide a *μopt*, a C++ framework, to manipulate and transform the microarchitecture graph prior to the Chisel generation. This will expert designers to create graph hardware optimization passes for which "end-users" could choose some ordering and parameters for existing transformations.

We first describe the overall design of *μIR* (§ 3.1). Then we explain the components used to describe a whole accelerator circuit (§ 3.2) followed by the components used to describe the dataflow within each block (§ 3.3). We provide an overview of memory and control in § 3.4 and § 3.5. In § 3.6 we provide pseudo-code to illustrate converting behavior-oriented compiler IR graph to a structural *μIR* graph.

3.1 *μIR* Design Overview

The *μIR* graph represents the accelerator architecture as a latency-agnostic structural graph. Components in a *μIR* graph execute in parallel and communicate via sequence of atomic tokens passed over unbounded edges. This representation is particularly suited for specifying microarchitecture, because of its “patience“ i.e., timing and latency of individual components has no impact on the functional correctness of the accelerator architecture. The main benefits of *μIR* is i) designers are free to transform the *μIR* graph, prior to RTL generation, and this permits many microarchitectural design options to be explored. ii) components can be locally refined during performance tuning without requiring global schedule changes e.g., change the number of execution units to improve throughput, iii) it promotes modular, re-usable hardware components for an accelerator.

The components in the *μIR* graph are organized in a hierarchy. Our motivation is similar to compilers, which organize their IR in a hierarchy: modules →functions→basic-blocks→instructions. Compilers rely on this hierarchy to demarcate the scope of optimizations e.g., local constant propagation targets functions, while global constant propagation targets whole program. Similarly, we separate concurrency and locality optimizations operating on the whole-accelerator circuit from local function unit optimizations. *μIR* uses a hierarchy of components to separate the data structures, iterators, and API used to implement these transformations.

3.2 Whole-Accelerator Circuit in *μIR*: Task blocks, Structures, and Connections

In this subsection, we use a Cilk [34] parallel program to illustrate the different whole-accelerator components in *μIR*. Figure 4 provides the code listing. A parallel loop; in the odd iterations, the loop spawns a task for performing 2D tensor (2×2 tile) multiplications and in the even iterations it performs an integer multiplication. The spawns in Cilk create a concurrent task while the parallel loop continues onto subsequent iterations. Figure 4 also includes the structural description of the accelerator microarchitecture as an *μIR* graph, generated from the Cilk program. The Chisel RTL is auto-generated from the *μIR* graph. Computer architects do not deal with the RTL; we have shown it here to illustrate how a microarchitecture graph looks when it is lowered to Chisel.

The whole-accelerator circuit is represented as a structural, concurrent graph of task blocks, connections and structures. Tasks represent

an asynchronous (may run and complete concurrently) execution block. In the example, there are three task blocks, the root for-loop task, and two children, a scalar task and a tensor task. (see Figure 4-Line:`5--8` in the structural specification). A task block is analogous to a closure (not unlike a function call) in software which takes arguments and produces a set of values after running to completion. Representing the accelerator as a pipeline of asynchronous task blocks has two benefits i) it helps avoids centralized control stores and stall signals in the hardware ii) we can implement arbitrary heterogeneous parallel patterns (including nested loops and recursion). Tasks are inspired by seminal work on threaded dataflow machines [16].

In the RTL specification in Figure 4 *Line 13—16 specify the task connections (<||>)*. Inter-task connections (<||>) establish a logical parent-child relationships between tasks. Finally, *Line 9—11 declare the scratchpad hardware structures*. In μIR, hardware structures are used to encapsulate elements that have no representation in the software e.g., local FIFO or RAM. In the example, two structures in the μIR encapsulate corresponding local memory spaces for holding the data to be streamed into and out of the task blocks. Lines 20—23 specify connections (<==>) between tasks and the scratchpad. The task blocks interface with scratchpads through a non-blocking request-response interface.

Execution and Memory Model: The execution flows need to be considered at two levels, whole-accelerator level, and the local dataflow within each task. Figure 5 provides an exploded view. μIR represents the whole accelerator as a graph of concurrently running dynamic task blocks. In this example, only the child tasks (`task_scalar` and `task_tensor`) perform actual work, and the parent `task_for` is used only for creation and coordination of the workers. `task_for` creates N/2 instances of `task_scalar` and N/2 instances of the `task_tensor`. μIR models each task block as having a local task queue that stores ready and pending tasks. The task block is free to process the ready tasks in any order. In the overall execution, parents `spawn` children to run concurrently and children terminate and return values to parents at `sync`. Tasks communicate either through memory or through registers in the connection. The memory model is

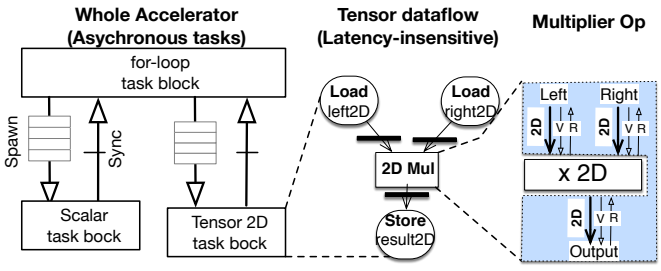


Figure 5: Execution Model of μIR at all levels. We have exploded each component to show the internal execution flow.

a partitioned global address space. A hardware designer could introduce any number of memory spaces in the μIR graph to interface with the task blocks. These address-spaces are incoherent with each other, but coherent with DRAM and the CPU through AXI. μIR includes two abstractions for representing local address spaces, scratchpads and caches. Caches are implicitly managed by a hardware controller, while scratchpads are managed with DMA.

The execution within each task block is modeled as a pipelined latency-agnostic dataflow. Individual nodes in the dataflow handshake with each other through a ready/valid flow-control protocol. The flow-control can apply back pressure to handle stalls (like control signals in a microprocessor pipeline) and permits arbitrary insertion and removal of buffering between nodes. Every node in the dataflow operation is considered to occur completely asychronously. The pipelined dataflow enables multiple concurrent invocations to be outstanding at the same time on dataflow and improves throughput. Unlike a tagged dataflow architecture [6], concurrent invocations complete in-order of invocation. This leads to a simpler RTL implementation.

3.3 A Task's Dataflow, Nodes, and Connections

Figure 6 lists the structural specification of the Tensor2D block's dataflow. *Line 4—8 declare the different nodes within the dataflow and line 10–12 specify the dependency connections.* The abstraction of a node in the μIR intuitively represents a function unit allocated to implement the required operation. Currently, our hardware library supports all operations specified by the LLVM IR, including FP and vector.

Nodes are flexible. They can either represent i) a single-cycle combinatorial logic (e.g., fixed-point add), ii) a multi-cycle latency node, where the node itself is potentially internally pipelined (e.g,, an FP add), iii) or a non-deterministic multi-cycle operation, in which the node only serves as a transit point to route values into the dataflow from an external unit shared amongst multiple nodes (e.g., the ld ops in Figure 6). This final representation is useful for composing hardware blocks, and implementing software patterns as function calls and nested loops. Connections represent "polymorphic" 1-1 dataflow between a producer and consumer nodes. Polymorphism implies that the designer only has to specify the data types of individual nodes, and during RTL generation, μIR implicitly infers and sets up the physical wire widths and flit sizes for the ports. This enables computer architect to perform generic dataflow pipeline transformations without having to consider each type.

```
1 class Tensor2D extends TaskModule(Tensor2D) {
2     ...
3  /*------- Dataflow specification  -------*/
4   val load_0  = new Load(Tensor2D)
5   val load_1  = new Load(Tensor2D)
6   val op_0    = new ComputeNode(opCode = "mul")
7                                        (Tensor2D)
8   val store_0 = new Store(Tensor2D)
9    ....
10  op_0.io.LeftIO     <> load_0.io.Out(0)
11  op_0.io.RightIO    <> load_1.io.Out(1)
12  store_0.io.data    <> op_0.io.Out(0)
13
14  /*------------ Junctions -------------*/
15  val mem_junc = new Junction(R=2,W=1)  (Tensor2D)
16  mem_junc.io.Read(0)  <==> load_0.io.Mem
17  mem_junc.io.Read(1)  <==> load_1.io.Mem
18  mem_junc.io.Write(0) <==> store_0.io.Mem
19 }
20
```

Figure 6: Autogenerated RTL description for Tensor2D task block. Corresponds to line 12:Cilk program in Figure 4

3.4 Memory: Load/Store Operations

Figure 7 illustrates the flow of a memory operation. In μIR, the loads and stores only serve as transit points for passing data between the memories and other computation nodes in the dataflow. "Polymorphism" is the central idea behind memory nodes in μIR. Memory operations nodes can support scalar, vector or tensor loads and stores. The design is based on this observation: although the shape and word layout are different, the fundamental hardware resources, such as

on-chip data storage and interconnect, are very similar. Hence, we centralize all of this logic in the databox component and during RTL generation lower to different implementations.

The databox 1) converts the type (e.g., Tensor2D) to word granularity accesses and issues them to the cache/scratchpad 2) coalesces the responses required from the cache to complete a request (e.g., 4 word responses for a Tensor2D), and 3) shifts and masks the data to handle alignments and sub-word operations. The databox fetches words required by single load in parallel, and fetches multiple loads in parallel.

Finally, a key question is how to interface all the distributed set of memory nodes to the appropriate scratchpad/caches. For this, the μIR includes *Junctions*, which represent generic 1:N, N:1, and M:N connections. In the structural specification of the task in Figure 6, Line 10 declares a 1:N junction and line 15–18 specify the connections between the memory operations and the junctions i.e., all the memory operations in the task are time-multiplexed over the junction. One possible implementation of junctions is a static tree network or a local bus. We have parameterized the junctions in μIR so that the designer can control the physical network that junctions lower to.

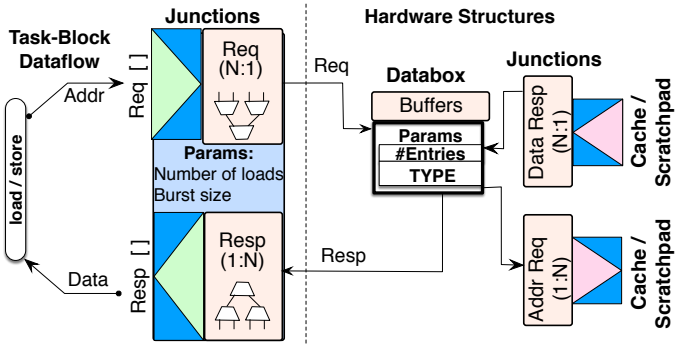


Figure 7: Connections between memory nodes in the task module and transfers to/from a cache or scratchpad.

3.5 Control Flow and Loops

μIR supports arbitrary control-flow from input algorithm. For forward branches, μIR implements dataflow predication i.e., trigger the node in dataflow for flow control, but bypass the actual logic and poison the output. With backward branches and loops there are three challenges i) Live ins: We extract each loop into the a task block, buffer the live ins from other parts of the circuit, and feed it into the dataflow. ii) Loop carried dependencies: We introduce buffering, latency-insensitive edges, and registers to break the combinatorial loop when implementing backward edges. This is similar to Arvind and Nikhil's seminal work on dataflow machines [6]. iii) Loop nests: In μIR each nested loop is disassociated from its outer loop, and is encapsulated within a task block. Intuitively, each nested loop is enclosed in a separate function that can run in pipeline parallel fashion with the parent. To the outer loop, the nested loop appears as a variable latency non-deterministic operation with request-response interface. Finally, recursion is handled similar to loops. We use LLVM to convert recursion to a iterative pattern prior to translating the program into an μIR graph (see recursive mergesort and fib in § 5).

3.6 μIR Front-end: Tranforming programs to μIR graph

The generation of μIR graph from compiler IR proceeds in three stages. In **Stage 1** we transform the compiler IR to a μIR task graph. A task block in μIR represents a set of basic blocks that needs to be asynchronously scheduled in hardware, either because the amount of work is statically unknown or it may be profitable to dynamically schedule.

Algorithm 1: Generating μIR graph from Compiler IR.

```
1  function Stage1_μIR_Taskgraph:
2    μIR_TaskGEdges = Map{}
3    μIR_TaskGNodes = Map{}
4    TaskQueue = {main()}
5    while TaskQueue != ϕ :
6      Current = TaskQueue.pop()
7      μIR_TaskNodes[Current] = List{}
8      for bb in Current.BasicBlocks :
9        if StaticSchedule(bb) :
10         μIR_TaskGNodes[Current].add(bb)
11       else:
12         Child = new Task(bb)
13         μIR_TaskGEdges[Current].add({Current,
           Child]})
14         TaskQueue.push(Child)
15     end
16   end
17 end
18
19 function Stage2_Schedule(Task node):
20   ComputeNodes = {}; DataflowEdges = {}
21   ControlNodes = {}; ControlEdges = {}
22   MemoryNodes = {}
23   for bb in Task.BasicBlocks :
24     for node in bb :
25       if node is Compute :
26         ComputeNode.add(node)
27         DataflowEdges.add({node,
           node.dependents})
28       elif node is Control :
29         ControlNodes.add(node)
30         ControlEdges.add({node,
           node.target})
31       elif node is Memory :
32         MemoryNodes.add(node)
33         GlobalMemory.connect(node)
34     end
35   end
```

Algorithm 1 shows the pseudocode of step 1. $\mu IR\_TaskGEdges$ and $\mu IR\_TaskGNodes$ collectively represent the task-level microarchitecture graph. We iterate over LLVM program-dependence-graph in breadth-first fashion and aggregate basic blocks(line 9: `if block`). Basic blocks that terminate dynamically schedulable regions e.g., loops, function calls, concurrent tasks in Cilk, Tensor-flow intrinsics, start a new task and restart the aggregation process (line 11: else block). Our compiler pass then extracts the task's basic blocks from the surrounding program-graph and creates a closure that captures the scope i.e., live-ins, live-outs and control dependencies. This enables the task region to be invoked through a timing-agnostic asychronous interface. The asynchronous interface lowers to a hardware

issue-queue (during the Chisel elaboration stage). Based on program flow the hardware queue, at run time, determines if an execution tile has to assigned for the task region, and if so which execution tile (see Section 3.2:Execution Model). The design is fully parameterized and a user can vary the number of execution tiles for each task region.

In **Stage 2:** we create the datapath for each task-block. μ`IR_TaskGNodes` is a dictionary that specifies for each node in the task graph, the corresponding region of basic blocks in LLVM. Algorithm 1 lists the pseudocode, `Stage2_Schedule`. In this stage, the body of each task block contains only forward branches[3]. We lower the set of basic blocks to a hyperblock and embed it as subgraph within the node in a task graph. The conversion to μ*IR* graph is a literal translation of the data flow graph. In the baseline, every compiler op lowers to a decoupled node, every node internally implements the functional unit and edges are pipelined connections between the function units. Subsequently, we connect memory operations to a global memory unit at the top-level of the graph. Section 3.5 and 3.4 provides more details on control-flow and memory implementation of stage 2.

4 Microarchitecture optimizations (μ*opt*)

The key benefit of μ*IR* is the ability to generate multiple microarchitectures with different design tradeoffs for the same software functionality. In this section we demonstrate five microarchitecture optimizations from the μ*opt* framework that successively expose opportunity for each other — Figure 8 shows the order in which these passes optimize the design, using the example from Figure 4.

Pass 1: Task Block Queuing (Goal 1: § 2.1). The hardware designer has the ability to modify the queuing and asynchrony between tasks in the whole-accelerator circuit. This permits the individual task blocks to proceed at different rates and enables subsequent optimization passes. This pass is achieved by controlling how inter-task connections (<||>) in μ*IR* map to RTL. One choice would be to introduce FIFO queues on the interface between the for-loop task block and tensor task block only, while leaving the low latency scalar block coupled. The tensor block has higher latency and we require more decoupling to ensure the for-loop block can run at a higher rate.

Pass 2: Execution Tiling (Goal 1,4: § 2.1). The higher latency of the tensor block could potentially lead to longer queuing delays. Hence, we need to increase the throughput of the tensor block by replicating it by N (a tunable parameter). The key challenge that μ*IR* deals with during the RTL generation is creating buses and crossbar to route tasks to different execution units. This change can be achieved locally without affecting the other parts of the accelerator circuit.

Pass 3: Localized Type-specific scratchpads (Goal 3:§ 2.1). The shared scratchpad compromise the execution of both the tensor and scalar blocks, due to contention. The tensor block (multiplying 2×2 tiles) reads 8 words and writes back 4 words per cycle, while the scalar reads 2 words and writes back 1. To solve this, μ*IR* creates local per-task scratchpads. The scratchpads also expose their type to μ*IR* and during RTL generation, we optimize the shape of the data movement over physical wires and change the data organization. Another option would be introducing a separate writeback buffer for writing out the data.

[3]Loops are treated as self-scheduling asychronous tasks.

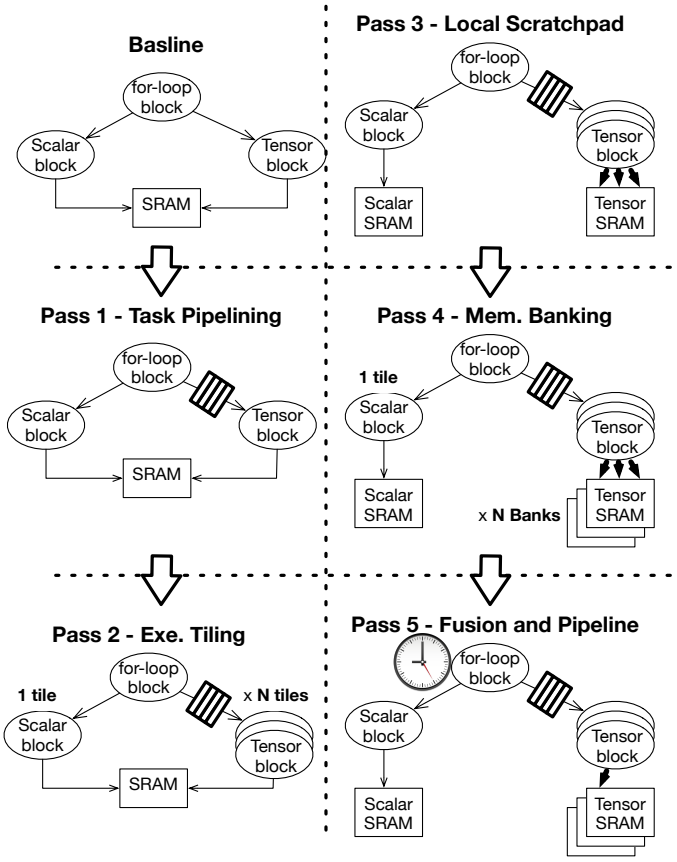


Figure 8: Overview of μ*opt* transformation passes. Each pass applies a specific transformation. Passes run in the order 1→5.

Pass 4: Scratchpad Banking (Goal 4:§ 2.1) To deal with the higher throughput introduced by multiple execution blocks in Pass 2, we also have to increase the throughput of the tensor memory system. For the tensor scratchpad we need to supply 2 tiles (each four words) in a cycle. The options are to either use four dual-port SRAM blocks and stripe the words across them or use a dual-ported SRAM with wide eight-word reads. The scalar scratchpad will use two port SRAM.

Pass 5: Op Fusion and Pipelining (Goal 2:§ 2.1) Finally, the for-loop block is on the critical path of the entire accelerator circuit. The dataflow itself is entirely serial and the pipeline has five stages: Buffer → ϕ → `i++` → `i==0` → `Cond-Branch`. This implies that each iteration takes atleast five cycles, and limits the throughput of the scalar task (only 2 cycles for execution). To re-time the pipeline to two stages, the pass fuses all of the operations into a single node. μ*IR* enables re-time the pipeline with having to modify the RTL.

4.1 μ*opt*: Codifying microarchitecture transformations.

In this section, we illustrate a pass that uses iterators for both the whole-accelerator and local dataflow of each task block. Algorithm 2 shows the pseudocode for this optimization. The optimization codifies pass 3 and 4 in Figure 8 — the goal is to partition the address space and direct un-related loads to different scratchpad banks. The optimization itself requires two sub-passes i) *Analysis*, which identifies the memory space to which each memory operation belongs. ii)

Algorithm 2: Scratchpad Banking

```
   // Temporary map from address space to list of memory
      ops. ID 0: Global. 1--N:Different address spaces
   Global: Mem_groups = Map[ID, List(MemOps)]
   Analysis:
1  def getMemoryAccess(Circuit):
2      foreach task in Circuit do
3          foreach mem in task.getMemops() do
4              space_id = LLVMPointsto(mem)
5              Mem_groups[ space_id ].insert(mem)
6
   Transformation:
7  def scratchpadBanking(Circuit):
8      foreach (ID,items) in Mem_groups do
           // Get memory parameters for each memory space
9          Param = getMemParams(items)
10         Mem = new RAM(Param)
11         foreach op in items do
               // Connecting memory ops to the new RAM
12             op.connect(Mem)
```

Transformation, which creates separate scratchpads in the microarchitecture graph for each memory space. The analysis pass can invoke any helper function, including software compiler (e.g., here we invoke the `LLVMPointsto()` which returns a unique id identifying the memory space). The transformation pass shows the flexibility of *μIR*. We can tune each scratchpad (e.g., number of ports, banks etc). It also demonstrates that *μopt* helps automate repetitive RTL modifications. For instance, the pass also has to repeatedly route each memory operation to its corresponding scratchpad. *μopt* provides helper API (`connect`) to automate the underlying RTL generation.

5 Quantitively Evaluating *μopt* and *μIR*

The primary purpose of *μIR* and *μopt* is to provide a fertile playground for computer architects to see their ideas reflected in RTL of accelerators. Here we try to answer the following: i) What is the quality of the baseline accelerators (no optimizations). Our goal is to establish a performance bar to isolate the benefits of the individual optimizations, ii) How do the baseline accelerator architectures compare to those generated by commercial HLS tools. iii) How do different optimization passes improve the performance. We consider each optimization individually, § 6.1—§ 6.4, and together§ 6.5

5.1 What is the quality of baseline accelerators?

Observation 1: *μIR-generated accelerators can attain high frequency. 200-500 Mhz on FPGA. 1.6—-2.5Ghz on ASIC.* **Observation 2:** *μIR-generated accelerators can achieve low power. 500-1200 mW on FPGA, 20-150 mW on ASIC.*

Here, we try to establish the performance and power characteristics of the baseline accelerator, which we further optimize in § 6. We evaluate on two backends, an FPGA Intel Arria 10 and ASICs synthesized using the Synopsys Design Compiler (UMC 28nm techology). Table 2 summarizes the results. All our workloads were unmodified.

Overall, even pre-optimization, *μIR* produces competitive accelerators. The floating point workloads (benchmarks with ^{F}) attain frequency between 350-400Mhz. For the FP macros, during RTL generation *μIR* plugs in the IP cores and for ASIC we use an in-house version of Berkley hardfloat. Here we use single precision throughout. For FP workloads, ASICs improve the power 20—50x vs. the FPGAs.

Compute intensive workloads use plenty of registers and FPGA ALMs and typically consume 1—1.2W in power on the Arria 10 e.g., COVAR, CONV.,2MM 3MM, SOFTM, FFT. In workloads with higher compute density and simpler operations, SAXPY, CONV, SOFTM, RELU, 2MM[T] the ASICs improved clock frequency 4—6×, compared to the FPGA.

Finally, the Cilk accelerators, achieved a lower target frequency, between 200-300Mhz on the FPGA (compared to non-Cilk workloads). This is primarily a result of the queueing and buffering logic required to manage asynchronous task blocks being on the critical path.

Table 2: Synthesizing Baselinr *μIR* on Arria10 FPGA

	FPGA Backend. Arria 10 SoC.					ASIC. 28nm		
Bench	**MHZ**	**mW**	**ALMs**	**Reg.**	**DSP**	***nm*^{2}**	**mW**	**Ghz**
Polybench or Machsuite [39, 45]								
GEMM^F	373	946	4480	7936	1	42.9	28	1.66
COVAR.^{F}	354	1496	11448	20415	4	73.6	150	1.66
FFT^F	425	1109	6548	12347	4	91.3	47	1.66
SPMV^F	388	868	3386	6048	2	36.9	24	1.66
2MM^F	385	1080	6126	12198	4	84.5	24	1.66
3MM^F	377	1202	8150	15216	1	97.38	52	1.66
Cilk Benchmarks								
FIB	307	751	1818	2614	0	—	—	—[#]
M-SORT	314	959	4943	6767	0	—	—	—[#]
SAXPY	214	609	1667	2252	2	17.4	20	2.5
STENCIL	207	812	4123	6134	2	82.6	94	2.5
IMG. Scale	206	705	2484	3582	2	83.4	96	2.5
Tensorflow Benchmarks [38]								
CONV.	363	1071	6354	10698	1	75.3	60	2.5
DENSE8^F	362	914	4025	7218	1	45.8	37	2
DENSE16^F	381	923	4070	7222	1	46.5	37	2
SOFTM8^F	375	1171	7787	13232	0	60.6	54	2
SOFTM16^F	347	1171	7842	10698	0	61.2	53	2
In-house								
RELU[T]	460	547	840	1169	14	15.9	19	2.5
2MM[T]	496	568	1073	1566	14	91.1	27	2.5
CONV.[T]	397	618	1717	2714	14	14.6	17	2.5

Footnotes: Table 2: ^{F} - Floating point benchmarks $[T]$: Tensor operations. Synopsys DC Compiler. ASIC umc 28nm.

5.2 *μIR* vs High-level synthesis

Observation 1 *Starting from the same program specification, μIR-generated accelerators attain 20% higher Mhz compared to HLS toolchains, due to dataflow execution*

Observation 2 *Many workloads exploit higher clock to achieve overall better performance (10—30%). On some workloads, HLS can generate better streaming buffers and achieves 10% better performance.*

An apples-to-apples comparison with prior HLS toolflows is not feasible since i) HLS primarily targets loop parallelism, and ii) they rely on streaming memory behavior (otherwise memory accesses are serialized). To ensure a fair comparison, we manually modified programs to be HLS compatible. We ported all the Machsuite and Tensorflow to HLS. [4]. Our set-up, i) we switched on all the compiler

[4]We had to rely on two toolchains, Legup [10] and Intel HLS, as neither one supported all the workloads

optimizations, since HLS relies on them), ii) we disable all μopt optimizations and tool-specific optimizations [8, 13, 14, 26], iii) we ensure that for RAMs and FP we use vendor-specific IP.

Figure 9 plots the baseline μIR's (no optimizations) performance normalized to HLS. The main reason for the performance improvement under μIR is the fundamentally different execution models and architecture generated by μIR. HLS relies on a state machine to coordinate execution. μIR however adopts a decentralized dataflow-based execution model. This leads to deeper operation pipelines and hence 20% higher frequency than HLS. In workloads like fft, gemm, 2mm and 3mm, which they have nested loops the μIR's pipeline depth is 30 (2MM) — 40(GEMM) stages; even workloads with few loops such as Dense8 have 15 stages. GEMM, Covar, 2MM, and 3MM: μIR exploits better operation parallelism as HLS serialize the nested loop executions — overall performance improvement is 20–30% (execution cycle improvement and clock frequency improvement contribute equally). In Conv, μIR's dataflow achieves nearly 80% improvement in target clock (overall 60% improvement in execution time). In FFT and Dense, HLS generates streaming buffers and improves the memory system (we were unable to turn it off), μIR relies on a less efficient cache.

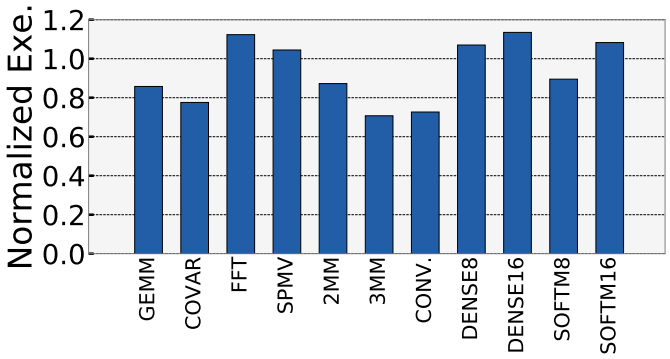


Figure 9: μIR vs HLS. Normalized performance. HLS = 1. < 1: μIR is better. > 1 HLS is better.

6 Evaluating the benefit of passes (μopt)

Table 3 summarizes the different optimization passes we study. It is not our intention to claim that the optimizations themselves are novel. Our goal is to implement and evaluate these optimizations as an μIR graph transformation.

Our microarchitecture transformations can be broadly classified into three categories. i)*Timing*: The microarchitecture graph exposes latency and contention through connection edges. We study three specific optimizations that improve the communication latency between operations (op-fusion, scratchpad/cache banking). ii) *Spatial*: All nodes and structures in the μIR graph can be replicated to improve throughput. We study spatial optimization in four components, function units, scratchpad, cache and task blocks. iii) *Higher-Order Ops*: μIR provides an opportunity for specializing the compute node implementations on a per-type basis (for example, single-cycle tensor operations). For each pass we list the benchmarks that benefited from our optimization.

Table 3: Summary of μopt passes

Opt	Type	Bench.	Sec.	Perf Impro.
Op fusion	*Timing*	FFT, SPMV, COVAR., SAXPY	§ 6.1	1.4×
Task tiling	*Spatial*	STENCIL, SAXPY, IMG. SCALE, FIB, M-SORT	§ 6.2	6×
Tensor Ops	Higher Ops	RELU[T], 2MM[T], CONV.[T]	§ 6.3	8×
Memory localization	*Timing*& *Spatial*	SPMV, CONV., SAXPY, COVAR.	§ 6.4	1.3×
Cache banking	*Timing*& *Spatial*	SAXPY, RELU, RGB2YUV	§ 6.4	1.5×
All Opt			§ 6.5	

6.1 Auto Pipelining and Op-Fusion

Result: *Reduces execution time between 1.17 — 1.7×*
Related research: [11, 21, 23]
μIR scope: Task dataflow, nodes and connections

This pass iterates over and transforms the dataflow graph of a task block. We auto balance the dataflow pipeline and fuse nodes in a greedy fashion. The baseline μIR makes no scheduling decisions and hence requires pipeline handshaking on all dataflow edges. Fusing nodes, to balance the pipeline eliminates the handshaking and pipeline register (Figure 10). The op fusion pass iterates over the dataflow in depth first fashion to search for opportunities to fuse nodes with their successors. During fusion, we seek to try to ensure that the resulting fused pipeline's frequency is not penalized (compared to the baseline). Combining multiple low-latency nodes together reduces the number of pipeline stages (and consequently latency) without introducing frequency-robbing critical stages.

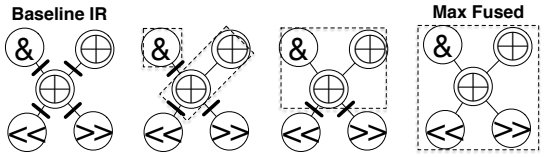


Figure 10: Illustration of Auto-Pipelining and Op-Fusion pass.

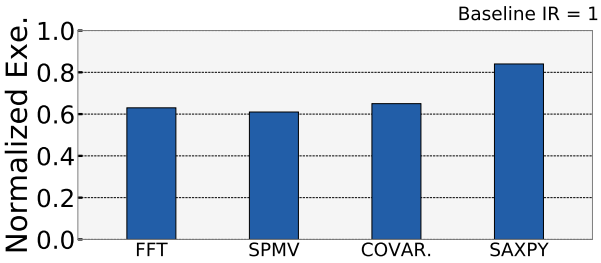


Figure 11: Execution time improvement due to Op-Fusion.

Figure 11 shows the normalized execution comparison with the baseline implementation on four benchmarks, FFT, SPMV, COVAR. and SAXPY. The overall execution time reduces by 1.2× to 1.6×. These were chosen due to their compute intensity. This pass primarily target compute intensive dataflows where there are long chains of fusable nodes and inexpensive operations like shift and bit-wise operations.

6.2 Concurrency Tiling

Result: *Reduces execution time between 1.5—6×*
Related research: [16, 19, 25, 49, 50]
***μIR* scope:** Task blocks

Figure 8:Pass 2 in § 4 provides an overview of this optimization. *μIR* permits each task block to independently increase the number of execution units. This effectively achieves a "multi-core" effect with multiple execution units running and completing in parallel.

Figure 12 plots the performance when varying the number of execution units per task. The baseline accelerator *μIR* specifies 1 execution unit for each task. We only study Cilk benchmarks as they exploit higher-level parallelism. The accelerator exploits all the available parallelism exposed by the applications and scale with increasing FPGA resources (1.5–6×). Saxpy improves with the addition of a second tile, but the benchmarks become quickly memory bound. Stencil and Image-scaling accelerators are more computationally intense (scale up to 8 cores). Both fib and merge-sort have extensive parallelism and scale well upto 4—8-way parallelism before being limited by the memory system.

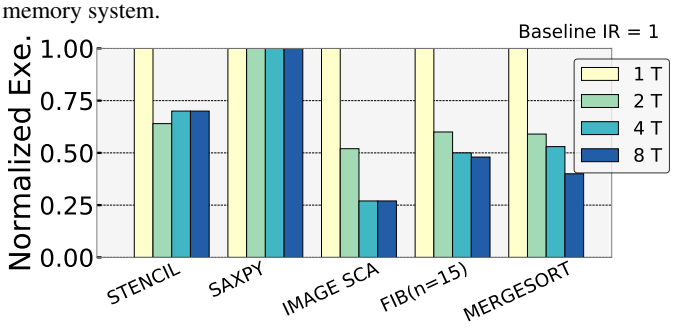


Figure 12: Reduction with in execution time by increasing number of parallel execution units.

6.3 Tensor Higher-Order Ops

Result: *Reduces execution time between 4 — 8×*
Related research: [12]
***μIR* scope:** Nodes and connections

In this section, we introduce tensor operators in the microarchitecture. They are highly optimized hand designed library (in Chisel) of operations that *μIR* can incorporate during the construction of the dataflow. Our *μIR* library includes support for 2D tensors, whose shape the designer can control i.e., for instance 2×2 in this example (Figure 13). In *μIR* all the microarchitecture components are typed and reflect the type specification of the macro operation. The microarchitecture transformations do not have to be involved in the actual implementation of the operations itself, but can flexibly use them within the dataflow. The actual hardware function unit for the tensor operations are incorporated in from the library of components. Figure 13 lists tiled matrix multiplication.

Figure 14 shows an optimized reduction-tree implementation of tensor multiplication for 2×2 shapes. Compared to the baseline which implements the operation through the pipeline, this is more efficient and also embarrassingly parallel. *μIR* also parameterizes the type of the intrinsic itself i.e., one of the parameters for a scratchpad is the shape of the data (2x2 in this case). *μIR* autogenerates RTL for the appropriate RAMs.

We implemented operator for all common tensor math (e.g., +,*,conv) and evaluated their benefit in improving 3 benchmarks RELU[T], 2MM[T], and CONV[T]. We find that the tensor operator increases leads to 4—8× improvement in performance (Figure 15). The cause for this improvement i) ≃4× increase in computational density and DSP blocks compared to baseline ii) the operand networks are all widened to implicitly transfer all the elements of the Tensor2D at one time iii) the fusion of scalar ops into a single higher order operator eliminates the pipeline handshaking.

```
for (int i = 0; i < NTiles; i++)
  for (int j = 0; j < NTiles; j++)
    for (int k = 0; k < NTiles; k++) {
      /**   Tensor Intrinsics  **/
      Tensor2D* a   = loadTile(&A[i][j]);
      Tensor2D* b   = loadTile(&B[i][j]);
      Tensor2D *mul = mulTile(a,b);
      storeTile(addTile(&C[i][j],mul),&C[i][j]);
    }
```

Figure 13: Implementation of 2MM with Tensor ops

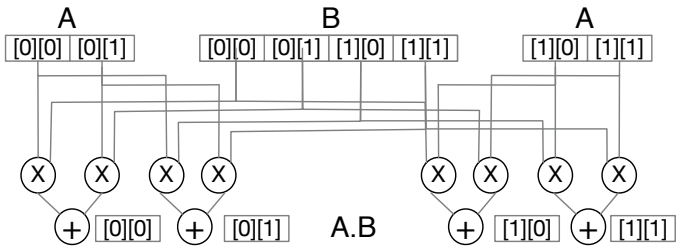


Figure 14: Multiplier unit for Tensor2D. $C_{2\times2} = A_{2\times2} \times B_{2\times2}$

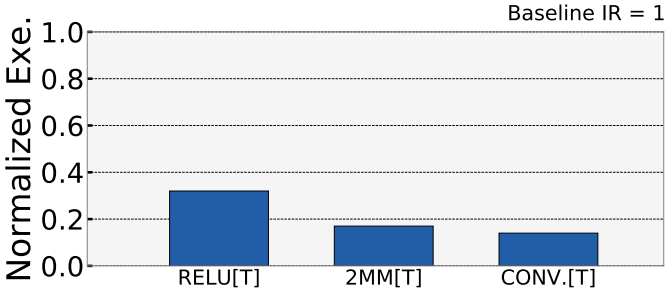


Figure 15: Performance improvement due to tensor ops.

6.4 Localizing and Banking Memory

Result: *Reduces execution time between 1.05—1.8×*
Related research: Universal ***μIR* scope:** Memory

The baseline microarchitecture used a shared scratchpad for local accesses and an L1 cache for all global accesses. Here, we focus on further increasing the number of scratchpads and L1 cache banks. First, we leverage algorithm 2 listed in section 4.1 to create multiple local memory address spaces. Second, we bank the L1 cache to parallelize the global accesses. *μIR* auto-generates the RTL logic for i) for routing loads/stores to the different memory banks, and ii) managing shared ports.

Figure 16 shows the performance improvement of these optimizations. SPMV, SAXPY, and CONV2D benefit from localized scratchpads as they stream data. SAXPY and CONV2D read in two matrices and hence do not benefit from four-way memory partitioning. The amount of improvement for benchmarks depends on memory level parallelism of each workload and whether working set size fit in cache

(64KB here). For instance workloads such as GEMM and FFT benefit from parallel access to local caches. 2MM, and 3MM see no benefit because the data maps to the same cache bank and does not benefit from the increased port and banks. COVAR. is compute intensive.

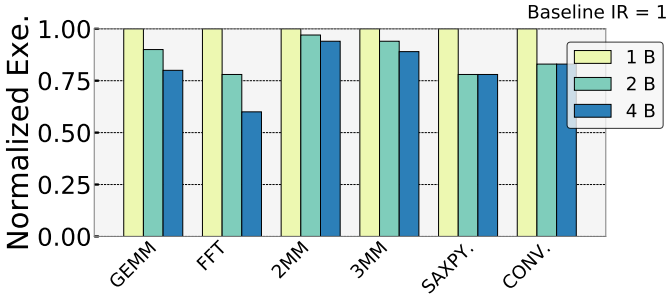


Figure 16: Effect of cache banking (1–4 Banks)

6.5 Stacking Multiple optimizations

Observation : *Applying multiple optimizations, leads to cumulative benefits — overall between 20%—4.2× improvement in performance* Our goal is to study the best performance achievable with the set of optimization we study in this paper. Figure 17 shows the overall performance improvement. We group together all the Cilk accelerators, as the concurrency tiling optimization applies only to them. SAXPY and STENCIL: the tiling pass increases parallelism. To accommodate the higher compute parallelism, memory localization increases the memory parallelism as well. GEMM, FFT and SPMV: Most of the performance improvement is attributable to Op-Fusion since it improves loop initiation interval and improves pipeline parallelism between loops. In 2MM and 3MM, both Op-fusion and localization are helpful to improve the performance. Finally, in COVAR, CONV, DENSE and SOFTM, the primary optimization that results in performance improvement is memory localization and banking.

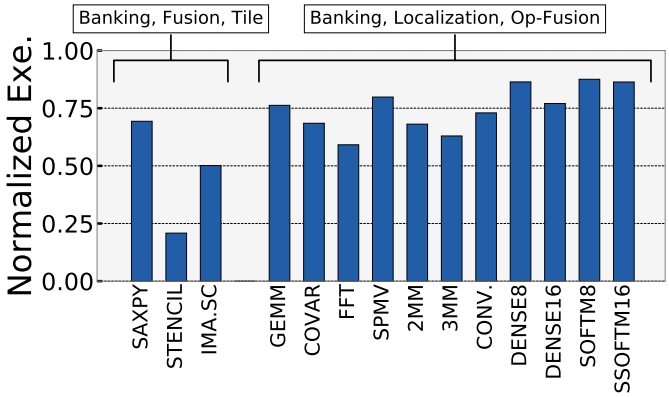


Figure 17: Effect of stacking multiple μopt optimizations

6.6 μIR vs. an ARM A9

μIR accelerators perform 2–17× better than an ARM A9.

Figure 18 compares the perform of μIR-optimized accelerators against an ARM A9 1Ghz dual issue out-of-order processor. In this case, we compare the best version of each accelerator with all the μopt optimizations applied. There are three main reasons for the better performance of μIR accelerators i) More ILP: GEMM, FFT, RELU and 2MM accelerators can issue more operations per cycle than the dual-issue ARM. ii) More compute density: Relu[T], 2MM[T], and Conv[T] leverage tensor function unit to pack more ops/cycle into the execution; CPU pipeline limits compute density. iii) Reduced overhead: the dataflow execution model eliminates the latency penalty of the front-end in CPUs.

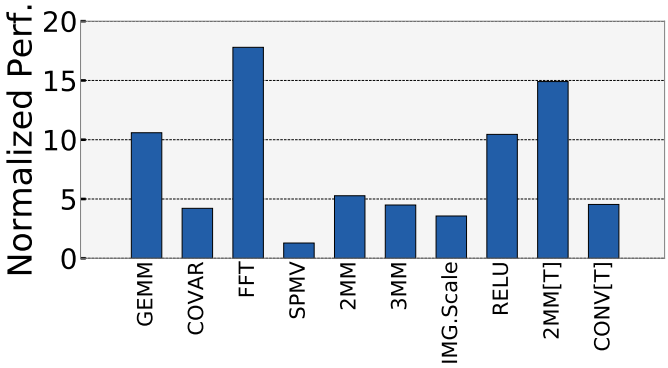


Figure 18: Optimized μIR vs ARM A9 1Ghz. ARM = 1. > 1: μIR is better. < 1 ARM is better. Note: ARM does not support Cilk.

7 Quantifying μIR productivity vs FIRRTL

In this section, we compare μIR against a hypothetical HLS system in which we lower the programs to FIRRTL (a lower level circuit-IR). We compare the % of nodes and edges that would have to be manipulated in a FIRRTL graph compared to a μIR graph to apply μopt optimizations. We believe this would be indicative of the conciseness with which μIR optimizations could be expressed. To conduct this study, We picked the workloads that supported all the optimizations across Section 6 on which we applied all of our optimizations passes, SAXPY, STENCIL and IMAGE scaling. We also quantify the number of graph nodes in FIRRTL and μIR representations of the hardware. Table 4 shows the result of this comparison; μIR is capable of more succinctly expressing and effecting architectural changes.

8 Summary

In this paper, we are proposing a toolchain and methodology to help with complexity of designing hardware accelerators. Our contribution: i) we developed an intermediate-representation, μIR, for designing hardware at microarchitecture (more informative than RTL) level. ii) we also developed μopt, a framework that cast microarchitecture optimization as a transformation of the μIR graph. We will be releasing the framework open-source.

Table 4: Conciseness of μIR vs FIRRTL (All μopt)

	Execution Tile 1 to 2				Add one more SRAM				Fused Operation				# Graph
	μIR		FIRRTL		μIR		FIRRTL		μIR		FIRRTL		$\frac{FIRRTL}{\mu IR}$
	ΔNode	ΔEdge	ΔNode	ΔEdge	ΔNode	ΔEdge	ΔNode	ΔEdge	ΔNode	ΔEdge	ΔNode	ΔEdge	
Saxpy	1	4	39	92	6	18	26	68	4	8	8	18	9.3×
Stencil	1	4	68	144	8	24	38	78	14	9	36	68	12.4×
Image SCA.	1	4	46	128	6	18	26	68	12	8	26	18	8.4×

References

[1] Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[2] Enabling rapid design space exploration and prototyping of dnn accelerators. http://pwp.gatech.edu/ece-synergy/wp-content/uploads/sites/332/2019/02/2_NNDataflowAnalysis.pdf.

[3] Mlir primer: A compiler infrastructure for the end of mooreâĂŹs law. https://github.com/tensorflow/mlir.

[4] Specification for the firrtl language. https://github.com/freechipsproject/firrtl/blob/master/spec/spec.pdf.

[5] Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[6] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, 1990.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a scala embedded language. https://github.com/freechipsproject/chisel3.

[8] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[9] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[10] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. From software to accelerators with LegUp high-level synthesis. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–9. IEEE, 2013.

[11] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *arXiv preprint arXiv:1607.02318*, 2016.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[13] Jongsok Choi, Stephen Dean Brown, and Jason Helge Anderson. From pthreads to multicore hardware systems in legup high-level synthesis for fpgas. *IEEE Trans. VLSI Syst.*, 25(10), 2017.

[14] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4), 2011.

[15] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. 2018.

[16] David E Culler, Anurag Sah, Klaus E Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Proc. of PROC of the 4th ASPLOS*, 1991.

[17] A DeHon, J Adams, M deLorimier, N Kapre, Y Matsuda, H Naeimi, M Vanier, and M Wrighton. Design patterns for reconfigurable computing. In *Proc. of the 12th FCCM*, 2004.

[18] Stephen A Edwards. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design & Test of Computers*, 23(5):375–386, 2006.

[19] Vladimir Gajinov, Srdjan Stipic, Osman S Unsal, Tim Harris 0001, Eduard Ayguadé, and Adrián Cristal. Supporting stateful tasks in a dataflow graph. In *Proc. of PACT*, 2012.

[20] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from Domain-Specific Languages. In *Proc. of FPL*, pages 1–8. IEEE, 2014.

[21] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *PROC of the 44th MICRO*, 2011.

[22] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom - compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):1–11, 2014.

[23] S Hu, I Kim, M H Lipasti, and J E Smith. An approach for implementing efficient superscalar CISC processors. In *PROC of the 12th HPCA*, 2006.

[24] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 209–216. IEEE Press, 2017.

[25] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proc. of the FPGA*, 2018.

[26] N Kapre and H Patel. Applying Models of Computation to OpenCL Pipes for FPGA Computing. *Proceedings of the 5th International Workshop on OpenCL*, 2017.

[27] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.

[28] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the PLDI*, 2018.

[29] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *Proc. of the 43rd ISCA*, pages 115–127, 2016.

[30] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: Heterogeneous parallel virtual machine. In *Proc. of the 23rd PPOPP*, 2018.

[31] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proc. of FPGA*, 2019.

[32] Maysam Lavasani. *Generating irregular data-stream accelerators: methodology and applications*. PhD thesis, 2015.

[33] Chris Leary and Todd Wang. Xla: Tensorflow, compiled! TensorFlow Dev Summit, Feb 2017.

[34] Charles E Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[35] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Proc. of the 47th MICRO*, pages 280–292, 2014.

[36] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *Proc. of the 22nd HPCA*, 2016.

[37] Razvan Nane, Vlad Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu-Ting Chen, Hsuan Hsiao, Stephen Dean Brown, Fabrizio Ferrandi, Jason Helge Anderson, and Koen Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[38] D. H. Noronha, B. Salehpour, and S. J. E. Wilton. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks. *ArXiv e-prints*, July 2018.

[39] Louis-Noel Pouchet and Uday Bondugula. Polybench 3.2. 2013. http://www.cse.ohio-state.edu/~pouchet/software/polybench.

[40] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating Configurable Hardware from Parallel Patterns. In *Proc. of the 21st ASPLOS*, 2016.

[41] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel paterns. In *Proc. of the 44th ISCA*, 2017.

[42] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *TACO*, 14(3):1–25, 2017.

[43] Andrew Putnam. FPGAs in the Datacenter - Combining the Worlds of Hardware and Software Development. *ACM Great Lakes Symposium on VLSI*, 2017.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P Amarasinghe. Halide - a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. of PLDI*, 2013.

[45] Brandon Reagen, Robert Adolf, Sophia Yakun Shao, Gu-Yeon Wei and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[46] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *Proc. of CODES+ISSS*, pages 1–10, New York, New York, USA, 2014. ACM Press.

[47] Hongbo Rong. Programmatic Control of a Compiler for Generating High-performance Spatial Hardware. In *arXiv.org*, November 2017.

[48] Sameer D. Sahasrabuddhe, Sreenivas Subramanian, Kunal P. Ghosh, Kavi Arya, and Madhav P. Desai. A c-to-rtl flow as an energy efficient alternative to embedded processors in digital systems. In *13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2010, 1-3 September 2010, Lille, France*, 2010.

[49] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir - Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *In Proc. of PPOPP*, 2017.

[50] Prakalp Srivastava, Rakesh Komuravelli, Sarita Adve, Maria Kotsifakou, Matthew D Sinclair, and Vikram Adve. HPVM: heterogeneous parallel virtual machine. In *Proc. of ACM SIGPLAN Notices*, pages 68–80. ACM, March 2018.

[51] James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*

[52] Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite - A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.*, 2014.

[53] Richard Townsend, Martha A Kim, and Stephen A Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 76–86. ACM, 2017.

[54] Ken Traub, James Hicks, and Shail Aditya. A dataflow compiler substrate. 1991.

[55] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. FPGA-Oriented Parallel Programming. In *Design of FPGA-Based Computing Systems with OpenCL*. October 2017.

[56] Ali Mustafa Zaidi and David Greaves. A new dataflow compiler ir for accelerating control-intensive code in spatial hardware. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, 2014.

[57] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. Composable building blocks to open up processor design. In *Proc. of the 51st MICRO*, 2018.