\( \mu \text{IR} \) - 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, \( \mu \text{IR} \), and an associated pass framework, \( \mu \text{opt} \). \( \mu \text{IR} \) 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 \( \mu \text{IR} \) graph that successively refine the accelerator architecture. The \( \mu \text{IR} \) 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 — 8x improvement in performance; overall 5—20x 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 confining 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 \( \mu \text{IR} \), 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].

\( \mu \text{IR} \) 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 \( \mu \text{IR} \), i) Optimizability: \( \mu \text{IR} \) represents hardware at a higher microarchitecture level of abstraction. In comparison to hardware languages (such as FIRRTL [24] and Verilog), \( \mu \text{IR} \) enables computer architects to concisely express...
their optimizations. ii) Transformability: 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 architectural changes can result in 1.5 — 8× improvement in performance. (see § 6.1—§ 6.4 for details). We synthesized all our accelerators on Arriva 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).

¹HLS 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.
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 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. 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. \(^2\)

\(^2\)HLS tools do introduce vendor-specific IPs, but these IPs are implemented as co-processors.

---

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

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]).
**Table 1: Comparing the Intermediate-Representations**

<table>
<thead>
<tr>
<th>Hardware-oriented IRs</th>
<th>High-Level-Synthesis</th>
<th>Domain Specific</th>
<th>Hardware Construction</th>
<th>Our Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial Tools</td>
<td>HLS toolchains</td>
<td>Modular RTL</td>
<td>Domain Specific</td>
<td>Chisel</td>
</tr>
<tr>
<td>C/C++</td>
<td>Legup [10], Vivado [5, 14], Catapult-C [1], AHIR [48], Pegasus [9]</td>
<td>OpenCL SPIR [27], Halide-HLS [42], Gorilla++ [32]</td>
<td>Chisel</td>
<td></td>
</tr>
<tr>
<td>Compiler opt</td>
<td>OpenCL SPIR</td>
<td>FIRRTL [24]</td>
<td>DELITE SPATIAL</td>
<td>C++/Cilk/Tensorflow</td>
</tr>
<tr>
<td>Dataflow IR</td>
<td>Pipeline IR</td>
<td>Pattern IR</td>
<td>Pattern IR</td>
<td>Concurrent LLVM IR</td>
</tr>
<tr>
<td>Pipeline opt</td>
<td>Streaming opt</td>
<td>Logic e.g., Mux, Adder</td>
<td>Flat dataflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Heterarchical dataflow</td>
<td>Pattern-based dataflow</td>
<td></td>
</tr>
</tbody>
</table>

**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
input programs into LLVM compiler IR. Our tool automatically translates the program specification to a \( \mu IR \) hierarchical graph using the compiler IR as an intermediary step. We iterate over the \( \mu IR \) graph in a hierarchical fashion and lower the \( \mu IR \) graph into synthesizable Chisel. Lowering implies as we iterate over the graph, we create the Chisel code instantiating the code and connecting to other components. This stage relies on a \( \mu IR \) library of components. \( \mu IR \) itself is simply implemented as a data structure and we provide a \texttt{hopt}, 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 \( \mu IR\) (§ 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 \( \mu IR \) graph.

3.1 \( \mu IR \) Design Overview

The \( \mu IR \) graph represents the accelerator architecture as a latency-agnostic structural graph. Components in a \( \mu IR \) 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 \( \mu IR \) is i) designers are free to transform the \( \mu IR \) 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 \( \mu IR \) graph are organized in a hierarchy. Our motivation is similar to compilers, which organize their IR in a hierarchy: modules \( \rightarrow \) functions \( \rightarrow \) basic-blocks \( \rightarrow \) 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. \( \mu IR \) uses a hierarchy of components to separate the data structures, iterators, and API used to implement these transformations.

3.2 Whole-Accelerator Circuit in \( \mu IR \): Task blocks, Structures, and Connections

In this subsection, we use a Cilk [34] parallel program to illustrate the different whole-accelerator components in \( \mu IR \). Figure 4 provides the code listing. A parallel loop: in the odd iterations, the loop spawns a task for performing 2D tensor (2 x 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 \( \mu IR \) graph, generated from the Cilk program. The Chisel RTL is auto-generated from the \( \mu IR \) 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 avoid 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_for) 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.](image)

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 asynchronously. 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 fitt sizes for the ports. This enables computer architect to perform generic dataflow pipeline transformations without having to consider each type.

```class Tensor2D extends TaskModule(Tensor2D) {
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  task( Tensor2D )
8  val store_0 = new Store(Tensor2D)
9  val mem_junc = new Junction((R=2,W=1)) (Tensor2D)
10  mem_junc.io.Read() <<< load_1.io.Mem
11  mem_junc.io.Write() <<< store_0.io.Mem
```

![Figure 6: Autogenerated RTL description for Tensor2D task block. Corresponds to line 12:Cilk program in Figure 4](image)

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 \( \mu IR \) includes \textit{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 \( \mu IR \) so that the designer can control the physical network that junctions lower to.

3.5 Control Flow and Loops

\( \mu IR \) supports arbitrary control-flow from input algorithm. For forward branches, \( \mu IR \) 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 dependences: 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 \( \mu IR \) 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 \( \mu IR \) graph (see recursive mergesort and fib in § 5).

3.6 \( \mu IR \) Front-end: Transforming programs to \( \mu IR \) graph

The generation of \( \mu IR \) graph from compiler IR proceeds in three stages. In Stage 1 we transform the compiler IR to a \( \mu IR \) task graph. A task block in \( \mu IR \) 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.

![Figure 7: Connections between memory nodes in the task module and transfers to/from a cache or scratchpad.](image)

**Algorithm 1:** Generating \( \mu IR \) graph from Compiler IR.

```
function Stage1_\mu IR_Taskgraph:
    \mu IR_TaskGEdges = Map()
    \mu IR_TaskGNodes = Map()
    TaskQueue = {main()}
    while TaskQueue != φ:
        Current = TaskQueue.pop()
        \mu IR_TaskGNodes[Current] = List()
        for bb in Current.BasicBlocks:
            if StaticSchedule(bb):
                \mu IR_TaskGNodes[Current].add(bb)
            else:
                Child = new Task(bb)
                \mu IR_TaskGEdges[Current].add({Current, Child})
                TaskQueue.push(Child)
    return \mu IR_TaskGNodes, \mu IR_TaskGEdges

function Stage2_Schedule(Task node):
    ComputeNodes = {}; DataflowEdges = {}
    ControlNodes = {}; ControlEdges = {}
    MemoryNodes = {}
    for bb in Task.BasicBlocks:
        node = bb:
        if node is Compute:
            ComputeNode.add(node)
            DataflowEdges.add({node, node.dependents})
        elif node is Control:
            ControlNodes.add(node)
            ControlEdges.add({node, node.target})
        elif node is Memory:
            MemoryNodes.add(node)
            GlobalMemory.connect(node)
    return ComputeNodes, DataflowEdges, ControlNodes, ControlEdges, MemoryNodes
```

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: \textbf{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 asynchronous 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. μIR_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. 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 μIR 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 (μopt)

The key benefit of μIR 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 μopt 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 μIR 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 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 μIR 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, μIR creates local per-task scratchpads. The scratchpads also expose their type to μIR 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.

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. μIR enables re-time the pipeline with having to modify the RTL.

4.1 μopt: 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)

Transformation:
7 def scratchpadBanking(Circuit):
8    foreach (ID,items) in Mem_groups do
9        // Get memory parameters for each memory space
10       Param = getMemParams(items)
11       Mem = new RAM(Param)
12       foreach op in items do
13          // Connecting memory ops to the new RAM
14          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 Quantitatively 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 technology). 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 Berkeley hardfloat. Here we use single precision throughout.

For FP workloads, ASICs improve the power 20—50x vs. the FPGAs.

Table 2: Synthesizing Baseline μIR on Arria10 FPGA

<table>
<thead>
<tr>
<th>Bench</th>
<th>FPGA Backend. Arria 10 SoC.</th>
<th>ASIC. 28nm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MHz</td>
<td>mW</td>
</tr>
<tr>
<td>Polybench or Machsuite</td>
<td>[39, 45]</td>
<td></td>
</tr>
<tr>
<td>GEMM²</td>
<td>373</td>
<td>946</td>
</tr>
<tr>
<td>COVAR</td>
<td>354</td>
<td>1496</td>
</tr>
<tr>
<td>FFT²</td>
<td>425</td>
<td>1109</td>
</tr>
<tr>
<td>SPMV²</td>
<td>388</td>
<td>868</td>
</tr>
<tr>
<td>2MM²</td>
<td>385</td>
<td>1080</td>
</tr>
<tr>
<td>3MM²</td>
<td>377</td>
<td>1202</td>
</tr>
</tbody>
</table>

Cil benchmarks

<table>
<thead>
<tr>
<th>Bench</th>
<th>FPGA Backend. Arria 10 SoC.</th>
<th>ASIC. 28nm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MHz</td>
<td>mW</td>
</tr>
<tr>
<td>Polybench or Machsuite</td>
<td>[39, 45]</td>
<td></td>
</tr>
<tr>
<td>FIB</td>
<td>307</td>
<td>751</td>
</tr>
<tr>
<td>M-SORT</td>
<td>314</td>
<td>959</td>
</tr>
<tr>
<td>SAPPY</td>
<td>214</td>
<td>609</td>
</tr>
<tr>
<td>STENCIL</td>
<td>207</td>
<td>812</td>
</tr>
<tr>
<td>IMG. Scale</td>
<td>206</td>
<td>705</td>
</tr>
</tbody>
</table>

Tensorflow Benchmarks [38]

<table>
<thead>
<tr>
<th>Bench</th>
<th>FPGA Backend. Arria 10 SoC.</th>
<th>ASIC. 28nm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MHz</td>
<td>mW</td>
</tr>
<tr>
<td>Polybench or Machsuite</td>
<td>[39, 45]</td>
<td></td>
</tr>
<tr>
<td>CONV</td>
<td>363</td>
<td>1071</td>
</tr>
<tr>
<td>DENSE²</td>
<td>362</td>
<td>914</td>
</tr>
<tr>
<td>DENSE16²</td>
<td>381</td>
<td>923</td>
</tr>
<tr>
<td>SOFTM²</td>
<td>375</td>
<td>1171</td>
</tr>
<tr>
<td>SOFTM16²</td>
<td>347</td>
<td>1171</td>
</tr>
</tbody>
</table>

In-house

<table>
<thead>
<tr>
<th>Bench</th>
<th>FPGA Backend. Arria 10 SoC.</th>
<th>ASIC. 28nm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MHz</td>
<td>mW</td>
</tr>
<tr>
<td>Polybench or Machsuite</td>
<td>[39, 45]</td>
<td></td>
</tr>
<tr>
<td>RELU[T]</td>
<td>460</td>
<td>547</td>
</tr>
<tr>
<td>2MM[T]</td>
<td>496</td>
<td>568</td>
</tr>
<tr>
<td>CONV[T]</td>
<td>397</td>
<td>618</td>
</tr>
</tbody>
</table>

Footnotes: Table 2: F - Floating point benchmarks [7]; Tensorflow operations. Synopsys DC Compiler. ASIC umc 28nm.

5.2 μIR vs. level-high synthesis

Observation 1 Starting from the same program specification, μIR-generated accelerators attain 20% higher MHz compared to HLS toolchains, due to dataloop 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 \( \mu_{opt} \) 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 \( \mu IR \)'s (no optimizations) performance normalized to HLS. The main reason for the performance improvement under \( \mu IR \) is the fundamentally different execution models and architecture generated by \( \mu IR \). HLS relies on a state machine to coordinate execution, \( \mu IR \) however adopts a decentralized dataflow-based execution model. This leads to deeper operation pipelines and hence 20% higher frequency than HLS. In workloads like ift, gemm, 2mm and 3mm, which they have nested loops the \( \mu IR \)'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: \( \mu IR \) 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, \( \mu IR \)'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), \( \mu IR \) relies on a less efficient cache.

### Table 3: Summary of \( \mu_{opt} \) passes

<table>
<thead>
<tr>
<th>Opt</th>
<th>Type</th>
<th>Bench.</th>
<th>Sec.</th>
<th>Perf Impro.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op fusion</td>
<td>Timing</td>
<td>FFT, SPMV, COVAR.</td>
<td>§ 6.1</td>
<td>1×</td>
</tr>
<tr>
<td>Task tiling</td>
<td>Spatial</td>
<td>STENCIL, SAXPY, IMG.</td>
<td>§ 6.2</td>
<td>6×</td>
</tr>
<tr>
<td>Tensor Ops</td>
<td>Higher Ops</td>
<td>RELU[T], CONV[T]</td>
<td>§ 6.3</td>
<td>8×</td>
</tr>
<tr>
<td>Memory Ops</td>
<td>local Timing</td>
<td>SPMV, CONV, SAXPY, CO-</td>
<td>§ 6.4</td>
<td>1.5×</td>
</tr>
<tr>
<td>Cache banking</td>
<td>Timing</td>
<td>SPMV, RELU, RGB2YUV</td>
<td>§ 6.4</td>
<td>1.5×</td>
</tr>
<tr>
<td>All Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.1 Auto Pipelining and Op-Fusion

**Result:** Reduces execution time between 1.17 — 1.7×

**Related research:** [11, 21, 23]

\( \mu IR \) 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 \( \mu IR \) 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 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 fusible 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 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 up to 4—8-way parallelism before being limited by the memory system.

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 (2×2 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++)
    Tensor2D a = loadTile(sA[i][j]);
    Tensor2D b = loadTile(sB[i][j]);
    Tensor2D *mul = mulTile(a, b);
    storeTile(addTile(sC[i][j], mul), sC[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}](image)

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

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
6.6 \( \mu IR \) vs. an ARM A9

\( \mu IR \) accelerators perform 2–17\( \times \) better than an ARM A9.

Figure 18 compares the perform of \( \mu IR \)-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 \( \mu opt \) optimizations applied. There are three main reasons for the better performance of \( \mu IR \) 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.

### Table 4: Conciseness of \( \mu IR \) vs FIRRTL (All \( \mu opt \))

<table>
<thead>
<tr>
<th>Function</th>
<th>( \mu IR )</th>
<th>FIRRTL</th>
<th>Add one more SRAM</th>
<th>Fused Operation</th>
<th># Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saxpy</td>
<td>ANode 1, Edge 4</td>
<td>ANode 39, Edge 92</td>
<td>ANode 6, Edge 26, Edge 68</td>
<td>ANode 4, ANode 6</td>
<td>1B: 3.1×</td>
</tr>
<tr>
<td>Stencil</td>
<td>ANode 1, Edge 4</td>
<td>ANode 68, Edge 144</td>
<td>ANode 6, Edge 26, Edge 68</td>
<td>ANode 14, ANode 9, Edge 36, Edge 68</td>
<td>1B: 12.4×</td>
</tr>
<tr>
<td>Image SCA.</td>
<td>ANode 1, Edge 4</td>
<td>ANode 46, Edge 128</td>
<td>ANode 6, Edge 26, Edge 68</td>
<td>ANode 12, ANode 8, Edge 26, Edge 18</td>
<td>1B: 8.4×</td>
</tr>
</tbody>
</table>
References


