HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description

Kingshuk Majumder
Computer Science and Automation
Indian Institute of Science
Bangalore, India
kingshukm@iisc.ac.in

Uday Bondhugula
Computer Science and Automation
Indian Institute of Science
Bangalore, India
udayb@iisc.ac.in

ABSTRACT

The emergence of machine learning, image and audio processing on edge devices has motivated research towards power-efficient custom hardware accelerators. Though FPGAs are an ideal target for custom accelerators, the difficulty of hardware design and the lack of vendor agnostic, standardized hardware compilation infrastructure has hindered their adoption.

This paper introduces HIR, an MLIR-based intermediate representation (IR) and a compiler to design hardware accelerators for affine workloads. HIR replaces the traditional *datapath + FSM* representation of hardware with *datapath + schedules*. We implement a compiler that automatically synthesizes the finite-state-machine (FSM) from the schedule description. The IR also provides high-level language features, such as loops and multi-dimensional tensors. The combination of explicit schedules and high-level language abstractions allow HIR to express synchronization-free, fine-grained parallelism, as well as high-level optimizations such as loop pipelining and overlapped execution of multiple kernels.

Built as a dialect in MLIR, it draws from best IR practices learnt from communities like those of LLVM. While offering rich optimization opportunities and a high-level abstraction, the IR enables sharing of optimizations, utilities and passes with software compiler infrastructure. Our evaluation shows that the generated hardware design is comparable in performance and resource usage with Vitis HLS. We believe that such a common hardware compilation pipeline can help accelerate the research in language design for hardware description.

KEYWORDS

HDL, HLS, MLIR, Verilog, accelerator, FPGA

ACM Reference Format:
Kingshuk Majumder and Uday Bondhugula. 2023. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623278.3624767

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03…$15.00
https://doi.org/10.1145/3623278.3624767

1 INTRODUCTION

The growing compute demands of machine learning and other high-performance computing (HPC) domains coupled with the need for power efficiency on edge computing devices has motivated the increased use of specialized hardware accelerators. Custom accelerators on reconfigurable computing platforms such as FPGAs are able to achieve high power efficiency [41] and performance but the difficulty associated with programming them is seen as a major roadblock towards their mass adoption.

High-level synthesis (HLS) [6] offers a promising solution towards making custom accelerator design more approachable. Domain specific languges (DSLs) for hardware design offer a higher level of abstraction to the algorithm developers compared to general purpose HLS languages, and benefit from being able to employ HLS compiler optimizations tailored for their domain. Numerous domains-specific as well as general-purpose languages and tools supporting HLS have been built over the past two decades by the academia and industry [1, 8, 9, 13, 15–17, 26, 27, 30, 36]. Nearly all electronic design automation vendors now provide suites supporting HLS, examples of which include Xilinx Vitis HLS, Cadence C-to-silicon, Synopsys Synphony, and Mentor Graphics Catapult HLS. A comprehensive survey of various HLS approaches was conducted by Bacon et al. [3].

All these HLS efforts have to create their own intermediate representations, thus duplicating and re-implementing a lot of the representation and transformation infrastructure that could otherwise be shared. Similarly, DSLs [14, 15, 20, 31] either rely on vendor provided HLS tools or require to reimplement the entire HLS compiler pipeline, including many standard optimizations, to translate the design into a hardware description language (HDL) like VHDL, (System)Verilog or Chisel [2]. The problem is even more pronounced while developing new optimizations. Due to lack of a standard compiler infrastructure, any new optimization work [39] has to implement its own language frontend and code generator.

The availability of an open IR standard for hardware design would help in decoupling the problem of designing suitable language abstractions for high level synthesis from the problem of optimization and code generation. LLVM [21] is a great example of applying this approach to software compilation flow. In this work, we attempt to solve a part of this puzzle by designing an intermediate representation for high level synthesis compilers and implementing a compiler around it to generate SystemVerilog. We limit the scope of our compiler to affine programs only. This allows the compiler to perform precise dependence analysis of the affine memory accesses.

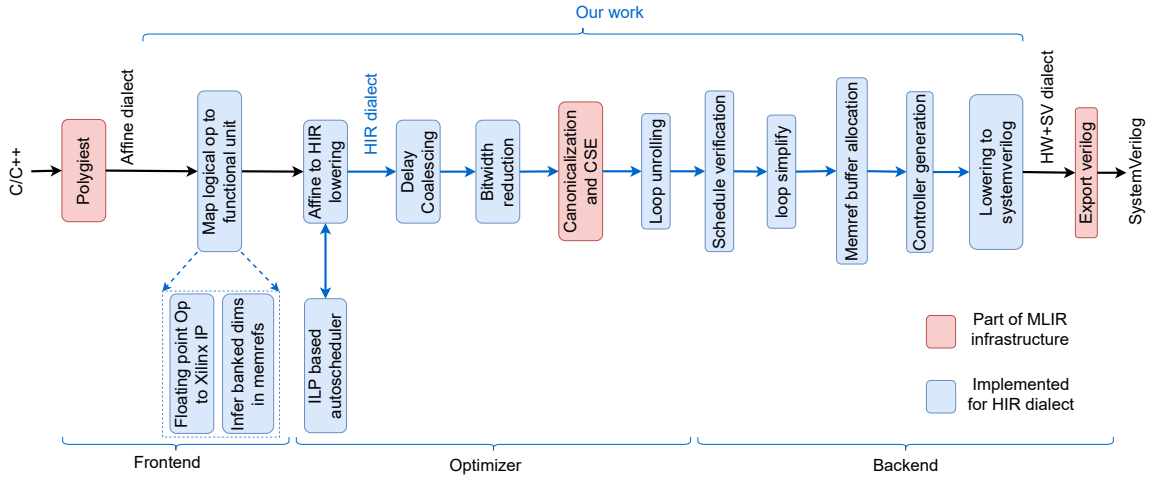


Figure 1: HLS compilation flow.

MLIR [22, 23], (multi-level intermediate representation) is a new compiler infrastructure designed to serve the needs of high-level domain-specific as well as general purpose programming languages and models at one end, and specialized hardware including custom accelerators at the other. Early promising results have been reported on software code generation as well as on building or migrating compiler stacks to MLIR [4, 7, 19, 28]. Interestingly, CIRCT [10] is an LLVM sub-project and an ongoing umbrella initiative to use and adapt MLIR and its methodology for HLS: it brings a compiler style approach to a field where programming and debuggability experience is vastly different. We built our work utilizing the CIRCT infrastructure.

The intermediate representations (IR) used in high level synthesis compilers can be categorized based on their level of abstraction. The language frontends usually target a high-level, latency insensitive IR. Many previous HLS compilers [6, 42] have reused software IRs for this purpose. These IRs are lowered by inserting handshake signals which result in a dynamically scheduled circuit. Alternatively, IRs such as Calyx [32] can also generate statically scheduled circuit if the delay of all operations are known at compile time. On the other extreme, low-level IRs are one step above the final HDL code generation. Multiple recent works [10, 37] have proposed new IRs for this purpose. These IRs capture the structural description of the hardware and are suitable for optimizations such as logic minimization.

In this paper we propose a new intermediate representation HIR, as a mid-level IR for high level synthesis of affine kernels. HIR supports high level control flow and multi-dimensional arrays to simplify lowering from a high-level dialect, while capturing enough information (explicit schedule, memory type and number of ports) for the backend to generate the final synthesizable SystemVerilog design. It isolates the frontend and scheduler from low level details of hardware design. HIR is also useful for DSLs that have their own custom schedulers [14, 15]. These DSLs can directly target HIR and benefit from the optimization and SystemVerilog code generation passes.

Figure 1 shows the compiler we built around the HIR IR. Our primary focus in this paper is on the design of the IR and how it can express a rich variety of parallelism. We also discuss about the rest of the compiler and the optimizations. A detailed discussion about our integer linear programming (ILP) based automatic scheduler/parallelizer can be found elsewhere [25]. Our contributions can be summarized as follows:

- We design an intermediate representation, HIR, on top of the CIRCT/MLIR infrastructure, to capture high level specification of a hardware design.
- We introduce the concept of time variables to express precise schedule of operations. This allows HIR to express different types of parallelism.
- Our IR captures the scheduing contract between the caller and callee in the function type signature. This allows us to implement a schedule verification pass to detect incorrect schedules at compile time.
- We build a compiler around this IR to generate SystemVerilog.

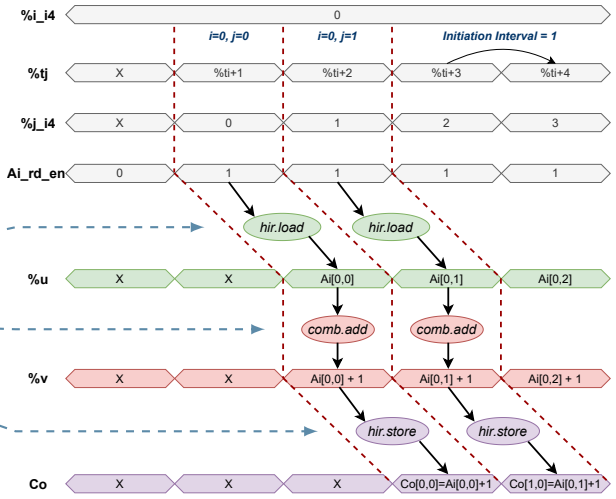
The rest of this paper is organized as follows. Section 2 describes the HIR intermediate language followed by its advantages in Section 3. In Section 4, we show how a rich variety of parallelism can be captured in HIR. Section 5 goes into the details of each individual compiler pass. An evaluation is presented in Section 6 and related work is discussed in Section 7. We summarize our work in Section 8

2 HIR INTERMEDIATE LANGUAGE

The HIR intermediate representation is implemented as a dialect in the MLIR [22] compiler infrastructure. As such, it inherits all the usual benefits provided by the core MLIR infrastructure, such as a round-trippable and human-readable textual representation that could be parsed, printed, and verified [23]. All our HIR operations have a custom pretty-printed form for readability and the convenience of compiler developers. HIR borrows its syntax from software programming languages. Like LLVM, all variables in HIR are SSA variables. The HIR IR looks very similar to a high-level

```
1 #bram_r = {"rd_latency"=1}
2 #bram_w = {"wr_latency"=1}
3 hir.func @transpose_hir at %t(
4   %Ai :!hir.memref<16x16xi32> ports [#bram_r],
5   %Co :!hir.memref<16x16xi32> ports [#bram_w]){
6     %c0 = hw.constant 0:i5
7     %c1 = hw.constant 1:i5
8     %c16 = hw.constant 16:i5
9     %c1_i32 =hw.constant 1:i32
10    hir.for %i : i5 = %c0 to %c16 step %c1
11        iter_time(%ti = %t + 1 ){
12      %tf =hir.for %j : i5 = %c0 to %c16 step %c1
13          iter_time(%tj = %ti + 1){
14        %i_i4 = comb.extract %i from 0:(i5)->(i4)
15        %j_i4 = comb.extract %j from 0:(i5)->(i4)
16        %u =  hir.load %Ai[port 0][%i_i4, %j_i4]
17          at %tj : !hir.memref<16x16xi32> delay 1
18        %j1 = hir.delay %j_i4 by 1 at %tj: i4
19        %v = comb.add %u, %c1_i32 :i32
20        hir.store %v to %Co[port 0][%j1, %i_i4]
21          at %tj + 1 : !hir.memref<16x16xi32> delay 1
22        hir.next_iter at %tj + 1
23      } //j-loop is temporal and pipelined.
24      hir.next_iter at %tf + 1
25    }//i-loop is temporal and sequential.
26    hir.return
27 }
```

(a) Increment and transpose function in HIR.



(b) Timing diagram

Figure 2: Matrix increment and transpose.

software IR, with the addition of explicit scheduling. The explicit schedule allows the IR to represent different types of parallelism such as instruction-level parallelism, loop pipelining and task-level parallelism. As a part of the MLIR infrastructure, HIR can interoperate with other MLIR dialects. The HIR IR allows arithmetic, logical, slicing and casting operations from *hw* and *comb* dialect inside the function body. This allows us to reuse the canonicalization and constant folding methods of these operations.

2.1 Time variable and schedules

The HIR compiler generates a hardware design with a single clock domain, i.e., all operations in the hardware design are synchronized with the positive edge of a single hardware clock. This clock provides the notion of time in the hardware design. Each tick (positive edge) of the clock represents a time instant. Each occurance of an operation in the HIR kernel is called a *static instance* of the operation. A static instance is uniquely identified by its syntactic position in the kernel. A specific static intance of an operation may be executed multiple times during the execution of the kernel. A *dynamic instance* of an operation is one execution of the operation. Thus, a static instance of an operation may be associated with multiple dynamic instances. For example, the *load* operation in line number 16 of Figure 2 is a static instance. Since the operation is inside a loop, each iteration of the loop would execute the load operation. Each of these executions of the operation corresponds to a dynamic instance of the operation (as shown in Figure 2b). Similarly, multiple invocations of the enclosing function also lead to multiple dynamic instances for each operation inside the function body. A *schedule* maps dynamic instances of each operation to a time instant of the clock. The corresponding dynamic instance of the operation is executed at that clock edge. A naive way to define a schedule is to specify the absolute time instant of each operation using an integer. But this would create a mapping from static instances of the operations to time instants. It does not allow us to map different dynamic instances of the same operation to different time instants.

To solve this problem, we introduce the concept of **time variables**. Each region (enclosed in curly braces) in an HIR kernel is associated with a time variable. This time variable represents the time at which the region starts its execution. All operations within the region must execute after this time. A region may be executed multiple times. For instance, a function's body is a region and each call to the function would execute the operations within the region. Thus, just like operations, a single region is also associated with multiple dynamic instances, each corresponding to one execution of the region. For each such dynamic instance of the region, the time variable represents the time instant at which the region started execution. For example, in Figure 2, the *transpose* function defines its start time as *%t* (line 3) and the *i*-loop defines the iteration start time with the time variable *%ti* (line 10). For each function call, the time variable *%t* would correspond to a different start time. Similarly, for each iteration of the loop within the function call, the time variable *%ti* would correspond to the start time of the *specific* iteration within the *specific* function call. The schedule for each operation within a region is defined as a constant delay from the region's start time. Since the region's start-time for each dynamic execution of the region is different, the dynamic instances of the operations within the region are also mapped to different time instants. In this way, the time variables allow HIR to associate different time instants to different dynamic instances of an operation.

Each region defines its own start time variable to indice the start of execution of operations within the region. A time variable can be defined in two ways. In Figure 2, the function definition operation (*hir.func*) defines the time variable *%t* using the *at* syntax, and the j-loop defines the time variable *%tj* using the *iter_time syntax*. The

$\%t$ time variable specifies the time at which the function execution starts and $\%tj$ specifies the time at which an individual iteration of the j-loop starts. The value of $\%tj$ is different for each iteration of the j-loop: $\%tj = \%ti+1$ for iteration $(i = 0, j = 0)$ and $\%tj = \%ti+2$ for iteration $(i = 0, j = 1)$. In addition, loops return a time variable as an output value. This time variable represents the time at which *hir.next_iter* was called in the last loop iteration. The time variable $\%tf$ in line number 12 is the time variable returned by the j-loop. A time variable can be accessed only within the region in which it is defined. Time variables of outer regions are not accessible in the inner regions. For example, $\%t$ is not accessible to operations in the i-loop's body. They can only access $\%ti$ and $\%tf$ time variables defined in the i-loop's body. Similarly, operations in the j-loop's body can only access the $\%tj$ time variable.

Time variables are used to schedule the start time of operations. The *'at'* keyword represents a *use* of a time variable except in the function definition. In the function definition, the *at* keyword defines the time variable to represent the time at which the function starts its execution. In any other operation, the *at* keyword represents the use of a time variable. The time specified after the *at* keyword is the time at which the operation is scheduled to start. An optional delay maybe added to a time variable at the *use* site. For instance, the *load* and *store* operations in Figure 2 are scheduled to start at time $\%tj$ and $\%tj+1$, where the time variable $\%tj$ is defined within the body of the j-loop and represents the start time of each loop iteration.

In addition to operations, all SSA variables of integer and float types are also associated with a time instant. The SSA variables contain a valid value only at the specified time instant. SSA variables of *hir.memref* type do not have any specific time instant associated with them. Memory elements can be read from or written to at any time instant. Each operation that produces an output value in HIR also specifies the delay from the start of the operation to the time at which the output is available. The SSA variable $\%u$ in Figure 2 has a valid value at time $\%tj + 1$ because the *load* operation starts at time $\%tj$ and it requires a delay of one cycle to load the value from the memory.

The HIR dialect utilizes operations from other MLIR dialects as well. For example, it uses the *comb* dialect for combinatorial operations (operations that complete in the same clock cycle) such as integer arithmetic and logical operations, multiplexing and extracting bit-vectors from larger bit-vectors. This allows us to reuse existing dialect operations and operation-specific canonicalization functions without having to reimplement them for the HIR dialect. To schedule a hardware design, only the operations in HIR dialect need to be scheduled explicitly. Since combinatorial operations complete their execution within the same clock cycle, their schedule can be calculated from their input values. For example, in Figure 2, the *comb.add* operation takes $\%u$ as input which is valid at time $\%tj + 1$. Thus, the *comb.add* operation is scheduled at the same clock cycle. In addition, the output $\%v$ is also produced in the same clock cycle $\%tj + 1$. In this way, we can extrapolate the schedule of combinatorial operations from their input values if the schedule of the HIR operations are known.

The *hir.delay* operation adds a delay of specified clock cycles to the input. As shown in line number 18 of Figure 2, the value $\%j\_i4$ is valid at time $\%tj$. It is delayed by one clock cycle to generate the value $\%j1$ which is valid at time $\%tj + 1$.

```
hir.func @multiplyAccumulate at %t
  (%a :i32, %b: i32, %c:i32 delay 3) -> (%d: i32 delay 4){
    //Func body. Time variable %t is available inside.
}

%res_d=hir.call "multAcc" @multiplyAccumulate(%op_a, %op_b, %op_c)
       at %t_call:!hir.func<(i32,i32,i32 delay 3) ->(i32 delay 4)>
```

Figure 3: HIR function definition.

2.2 Functions

A function is converted to a SystemVerilog module after lowering. Functions call can be scheduled at a specific time instant similar to other operations in HIR. The function input can be of a primitive type like integer or float, or it can be a *memref*. The function outputs are limited to integers and floats.

In addition to the type of operands and results, a function signature captures the schedule of its operands and results. The function signature associates a *delay* with each operand of primitive type. The delay specifies the time at which an operand is read by the function, relative to the function's execution start time. Figure 3 shows the function definition of a multiply-accumulate operation. The function signature implies that the function starts executing at time $\%t$. The inputs $\%a$ and $\%b$ are read when the function starts. Input $\%c$ is read after a delay of 3 clock cycles i.e., at time $\%t + 3$. Similarly, function output is also associated with a delay to specify when it is ready. For instance, the *multiplyAccumulate* function in Figure 3 returns the output at time $\%t + 4$, i.e. 4 clock cycles after the function is invoked.

The delay specified for each operand and result in the function signature provides a **scheduling contract** between the caller and the callee. In a statically typed programming language, the type checker uses the function's type signature to verify, at compile time, that the caller is providing operands of correct type to the callee. Similarly, we use the relative schedule of operands and results in the function signature to verify that the caller-provided operands are valid at the time instant when the callee expects them. This is implemented as a separate schedule verification pass (Section 5.3) in our compiler.

Explicit scheduling contract in function signature is also useful when the callee is a Verilog/SystemVerilog module and only the callee's function declaration is available. For example, Xilinx provides Verilog implementation of floating point multipliers and adders. While using them as external functions, the declaration captures the delay of these operations. The schedule verification pass can check that the caller is using the output only at the correct time instant.

2.3 Control flow operations

HIR provides multiple primitives to capture the control flow in a design. The HIR compiler automatically synthesizes state machines to implement the control flow.

Figure 4 shows the syntax of control flow operations in HIR. The *for* loop takes a lower bound, an upper bound and a step size.

```
//%t_end: time at which last iteration calls next_iter.
%tf =hir.for %i : index = %0 to %16 step %1
  iter_time(%ti = %t + 1) {
    //...
    %x = comb.mul %a, %b :i32
    %x1 = hir.delay %x by 5 at %ti : i32
    %y = comb.add %x1, %y_prev: i32
    //...
    hir.next_iter at %ti + 5
}

%res = hir.if %cond at time(%t_inner = %t) -> (i1) {
      %true_0 = hw.constant true
      hir.yield(%true_0) : (i1)
    }else {
      %false = hw.constant false
      hir.yield(%false) : (i1)
    }
```

Figure 4: Control flow ops in HIR.

```
#bram_r = {"rd_latency"=1}
#bram_w = {"wr_latency"=1}
#bram_rw = {"rd_latency"=1, "wr_latency"=1}
#reg_r = {"rd_latency"=0}
#reg_w = {"wr_latency"=1}

//single port block ram.
%bram_1p = hir.alloca bram
  : !hir.memref<8x8xi32> ports[#bram_rw]

// simple dual port block ram.
%bram_s2p = hir.alloca bram
  : !hir.memref<(bank 4)x8xi32> ports[#bram_r, #bram_w]

%reg = hir.alloca reg
  : !hir.memref<(bank 2)xi8> ports[#reg_r,#reg_w]

//%0 is of Index type.
%v =  hir.load %bram_s2p[port 0][%0, %i] at %t
  : !hir.memref<(bank 4)x8xi32> delay 1
hir.store %v to %bram_1p[port 0][%i, %j] at %t + 1
      : !hir.memref<8x8xi32> delay 1
```

Figure 5: HIR memref type.

The *iter_time* syntax specifies that the loop starts at time $\%t + 1$. The *next_iter* operation determines the initiation interval of the loop by specifying the start time of next iteration with respect to the current iteration. The *for* loops in HIR can be either temporal loops (multiple iterations on same hardware at different times) or spatial loops (loop body unrolled). Since, *index* type represents compile time constant integers in HIR (Section 2.4), spatial loops are represented by specifying a loop induction variable of *index* type.

The *if* statement takes two input variables - the boolean condition variable (*%cond*), and a time variable (*%t*) to represent the time at which the condition is checked. It also defines a new time variable (*%t_inner*), which is available inside the *then* and *else* bodies to schedule operations inside them.

2.4 Types

HIR supports primitive data types such as arbitrary bitwidth **integers** and **floats**. It uses the **time type** (*hir.time*) to represent the type of time variables.

The **index type** represents constant integer value known to the compiler. Index variables can be defined using the *arith.constant* operation or, as loop induction variables of unrollable for-loops, as shown in Figure 4.

All the available memory resources in hardware are represented via the **memref type** (*hir.memref*). A *memref* can be viewed as a pointer or reference to a multi-dimensional tensor. The tensor may be placed in an array of buffers (such as distributed or block RAM) or registers. The *memref* type abstracts its implementation details, such as the exact type of hardware memory, and provides a uniform interface for memory accesses.

Figure 5 shows two *alloca* operations that allocate new memory elements (block RAMs and registers). The *memref* datatype defines the dimensions of the tensor, the data type of its elements and memory banking. The ports of the memory are specified using MLIR's dictionary attributes. These attributes (*bram_r*, *bram_w*) specify the type of the port and the latencies of the read/write operations. Each port has its own address bus. If a port attribute specifies both read and write latencies then it is treated as a read/write port with a common address bus. This approach of representing memory ports gives HIR the flexibility to instantiate memories with arbitrary number of ports of different types. For example, block RAMs in Xilinx FPGAs are dual ported. A memref can specify separate read and write ports to instantiate a simple-dual-port RAM or have read+write permission in both ports for a true-dual-port RAM.

If the *memref* is an input operand to the function, then the compiler will synthesize the correct data and address buses based on the number and type of ports. For example, in Figure 2, the input argument *%Ai* has only one port (line 4) which is read-only (line 1). The compiler would generate an 8-bit *output* address bus port, a 32-bit *input* data bus port, and a one bit *output* enable bus (for read-enable) port in the generated Verilog module. For *%Co* the address bus would be the same, but the data-bus would be an *output* port and the enable bus would carry the write-enable signal.

The *hir.memref* can optionally choose to distribute its elements in multiple banks. The dimension(s) to be banked are marked using the *bank* keyword. The banked dimensions of a *memref* can only be indexed using *index* type which is guaranteed to have a constant value known to the compiler (after loop unrolling). This ensures that the compiler knows which specific bank is accessed. We discuss the advantages for memory banking in Section 4.4. Another restriction on *memrefs* is that only one *load/store* operation can be scheduled every cycle for a given port of a bank.

2.5 Undefined behavior

HIR also borrows the concept of undefined behavior from software programming languages. The HIR compiler makes the following assumptions:

- Lower bound of a for-loop is never greater than the upper bound.
- A new instance of a for-loop is not scheduled unless the previous instance has completed all iterations. For this reason, the inner loop in Figure 2 is pipelined but the outer loop is sequential.
- There will never be multiple accesses to the same port of a memref bank in the same clock cycle.
- All `load` operations happen on initialized memory, i.e., the memory must be written-to before reading a value from it.

Each call to a function resets all memory elements (such as registers and RAMs) instantiated in the function to uninitialized state. The HIR language does not support persistent state (equivalent to static scope in C) across function calls.

- The *comb.extract* operation is used by our compiler to cast integers to smaller bitwidths. If *nowrap* attribute is specified in the operation then the casting should not change the value.

Violation of any of the above assumptions is treated as undefined behavior. Optimizers can exploit the undefined behaviour to implement more aggressive optimizations that do not violate the semantics. In Section 5.2, we exploit the undefined behaviour associate with the *nowrap* attribute to reduce bitwidth of loop induction variables.

3 STRENGTHS OF HIR INTERMEDIATE LANGUAGE

In this section, we discuss the advantages of using HIR as an intermediate language for HLS compilers.

3.1 High level design

HIR borrows control flow constructs such as loops, function calls and conditional statements directly from imperative programming languages. These features make it easy to convert software algorithms into hardware designs. They also help in representing high level optimizations such as loop pipelining and overlapped kernel execution.

3.2 Explicit scheduling

Previous work [14] has shown that statically scheduled designs are more efficient in their hardware utilization since they do not have to implement extra control logic to dynamically communicate between hardware modules. Although such designs are more efficient, the IRs that precisely capture the schedule are usually designed at the abstraction level of hardware description languages. Thus, like HDLs, they require state machines that generate the control signals to determine the order of execution of operations in the data-path. Without these control signals, every operation in the hardware will execute every cycle.

Instead of describing a hardware design as datapath + FSM, HIR describes it as datapath + schedule. The hardware design specifies the relative time of execution of each operation and the compiler automatically generates the required FSM. Explicit schedules simplify the code generation after automatic scheduling (Figure 1). The scheduling pass does not have to create FSM logic to implement the parallel schedule.

3.3 Deterministic parallelism

Many HLS languages [33] and latency insensitive IRs [32] borrow non-deterministic parallelism from software programming languages. This kind of parallelism often requires some kind of a synchronization mechanism. For example, in Vitis HLS a producer and a consumer task can execute in parallel if the producer is transferring its outputs to the consumer via streams (implemented as FIFOs in hardware). This requires handshaking (a form of synchronization) between the producer and consumer. If the two tasks are working in lock-step, i.e., every fixed number of cycles with the producer task generating one output and the consumer task consuming it, then there is no need for synchronization between the two tasks. Both HIR and low level IRs [37] can express this kind of deterministic, synchronization-free, task level parallelism. Section 4.3 shows an example of synchronization free task level parallelism.

```
hir.func.extern @stencil_A at %t(
    %Ai:!hir.memref<64xi32> ports[#bram_r],
    %Bw:!hir.memref<64xi32> ports[#bram_w])
hir.func.extern @stencil_B at %t(
    %Br:!hir.memref<64xi32> ports[#bram_r],
    %Co:!hir.memref<64xi32> ports[#bram_w])

hir.func @task_parallel at %t(
%Ai :!hir.memref<64xi32> ports[#bram_r],
%Co : !hir.memref<64xi32> ports [#bram_w]){
  %B = hir.alloca "bram"
    : !hir.memref<64xi32> ports[ #bram_r, #bram_w]
  %Br = hir.memref.extract %B[port 0]
    : !hir.memref<64xi32> port[ #bram_r]
  %Bw = hir.memref.extract %B[port 1]
    : !hir.memref<64xi32> port[ #bram_w]
  // Execution of stencilB is overlapped with stencilA.
  hir.call @stencil_A(%Ai, %Bw) at %t
    : !hir.func<(!hir.memref<64xi32> ports [#bram_r],
                 !hir.memref<64xi32> ports[#bram_w])>
  hir.call @stencil_B(%Br, %Co) at %t + 8
    : !hir.func<(!hir.memref<64xi32> ports [#bram_r],
                 !hir.memref<64xi32> ports [#bram_w])>
  hir.return
}
```

Figure 6: Overlapped execution of tasks.

3.4 External hardware modules

The ability to use externally defined hardware circuits is essential in order to specialize a design for the given hardware platform. FPGA vendors provide their own custom libraries for many common circuits such as floating-point arithmetic and multi-ported RAMs. Additionally, there are several third party libraries that may need to be reused or inter-operated with.

HIR's ability to capture precise scheduling information in the function signature makes it easier to integrate external Verilog modules with HIR's design. In languages where the schedule is not a part of the language semantics, external modules usually require additional handshake signals. External modules that have fixed latency can interface with HIR without the overhead of handshaking.

4 EXPRESSING DIFFERENT TYPES OF PARALLELISM IN HIR

HLS compilers are expected to exploit domain knowledge to find potential optimization opportunities [15]. A good intermediate language should provide mechanisms to express these optimizations so that the compiler backend can generate the desired circuit. In this section, we discuss varius standard hardware optimization techniques and how they can be expressed in HIR.

4.1 Instruction level parallelism

The HIR IR can capture fine grained instruction level paralellism. For instance, multiple independent operations can be scheduled in the same time to improve parallelism. Similarly, operator chaining can be used to schedule multiple dependent operations in the same clock cycle, which would otherwise span multiple cycles. In case multiple dependent operations can not be scheduled in same cycle in order to meet frequency targets, pipeline registers can be added between instructions. An HIR design uses `hir.delay` operation to add pipeline registers between dependent operations.

4.2 Loop pipelining and unrolling

Loop pipelining is a key optimization in high level synthesis. In loop pipelining, the next iteration of the for loop starts before the previous iteration completes. This allows multiple loop iterations to execute in parallel. Loop pipelining does not add extra hardware overhead. A *for* loop with a constant initiation interval is shown in Figure 4.

Unrolling replicates the loop body in hardware. This allows an HIR design to scale with available hardware resources if there is enough loop parallelism. A loop where the induction variable is of *IndexType* is fully unrolled. HIR does not support partial unrolling of loops. Partial unrolling can be represented by strip-mining the *for* loop and completely unrolling the resultant inner loop.

4.3 Task level parallelism

In addition to exploiting loop pipelining, multiple tasks can be executed in parallel to further improve performance. The "task_parallel" function in Figure 6 shows an example of task level parallelism expressed in the HIR dialect. Since the stencils read the input array and write to the output sequentially, the second stencil does not have to wait for the first stencil to complete. It can start its operation as soon as there is enough data to calculate its first output. After that both the stencil run in locksteps i.e. in each cycle stencilA produces one value and stencilB consumes one value. Overlapping the execution of the tasks reduces the overall latency of the top level function. Like loop pipelining, overlapped execution of multiple tasks does not need any runtime synchronization. The explicit schedule ensures that both the loops run in lockstep.

4.4 Memory banking and multi-port RAMs

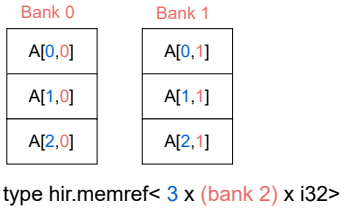


Figure 7: Memory banking in a memref type.

Hardware accelerators use on-chip buffers to reduce DRAM accesses. In order to execute operations in parallel, these designs may perform multiple reads and writes every cycle. FPGAs offer multi-ported on-chip RAMs to support parallel access. An HIR design can instantiate a multi-ported RAM in order to parallelize memory accesses. For workloads where the parallel memory accesses are usually guaranteed to be separated by a fixed stride, an HIR design may employ memory banking instead. In this approach, the data is distributed among multiple buffers in such a way that parallel accesses occur on distinct buffers. Figure 7 shows how elements of a banked memref are spread across multiple buffers.

5 COMPILER FLOW

Figure 1 gives an overview of our compilation flow. The compiler passes are divided into a frontend to lower C programs into affine dialect, an optimizer for automatic parallelization and other optimizations, and a backend to lower the design to SystemVerilog.

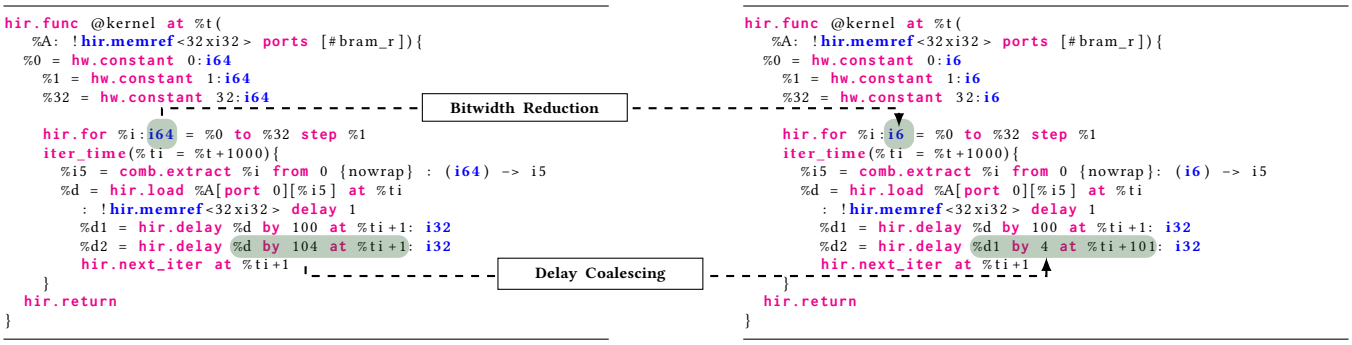
5.1 Frontend

We use Polygeist [29] to lower C programs into the affine dialect of MLIR. The generated IR does not contain any information to map logical operators and memories into FPGA resources. In the next step, our compiler adds a mapping from floating point operations to specific library implementations. This library of operators is written in Verilog for Xilinx FPGAs but can be replaced with a different set of operators for other FPGA vendors. The mapping also contains the information about the delay of each operation. In addition, this pass also determines which of the *memref* dimensions are banked. It analyses the loop induction variables used to index into the corresponding dimensions and, based on whether the loop is marked for unrolling or not, determines whether to bank the dimension. Since Polygeist currently does not have support for specifying loop unrolling factor or initiation intervals, we manually add those to the Polygeist generated IR (as loop attributes) before processing it further. We expect these limitations to go away when Polygeist adds support for pragmas (similar to Vitis HLS). The frontend ensures that there is all the necessary information (like operator delay and memory banking), before we the run automatic scheduling pass.

5.2 Optimizer

The optimizer lowers the affine IR into HIR and optimizes the generated HIR IR. All programs in HIR dialect have an explicitly specified schedule which may capture different types of parallelism in the design, but affine dialect assumes sequential execution. In order to lower Affine dialect to HIR, we use an ILP based automatic scheduler that generates a parallel schedule.

Auto-Scheduling Our compiler uses an ILP based automatic scheduler [25] to calculate a parallel schedule for the input sequential affine program. The objective of the scheduler is to find the additional delays for each operation, relative to the start time of its parent region. The scheduler performs scheduling in two steps. In the first step it formulates an ILP to find actual dependencies and the required delay between dependent operations in clock cycles. For each pair of load/store operations that may have a dependence, the scheduler formulates an ILP which calculates the required delay between the source and sink operations of the dependence. This dependence may be intra-loop, loop-carried or between two loop nests. Since ILP uses the affine memory access subscripts in the constraints to test for dependence, if a pair of load/store operations



(a) Unoptimized IR. **(b) After bitwidth reduction and delay coalescing.**

Figure 8: Optimizations after the automatic scheduling.

never access the same memory location, the ILP yields no solution. The scheduler ignores these false dependencies.

In the second step, the scheduler formulates a combined ILP to calculate the schedule. This ILP uses the previously calculated delays between source and sink operations of actual dependencies as constraints. In addition to memory dependencies, the scheduler also captures dependencies between the *def* and *use* operations of SSA variables of integer and float types. For example, if the output of a function call is used by a *load* operation, and the function call takes 3 cycles to calculate the output, then there is an additional constraint that the load operation must execute atleast 3 cycles after *call* operation. It also captures the user provided loop initiation intervals as constraints in the ILP formulation to satisfy the performance constraints. The ILP solution provides a valid schedule (i.e. a schedule that does not violate any memory dependencies) of all HIR operations such that all the loops are pipelined with the user provided inititation intervals. The affine-to-HIR lowering pass uses the schedule to generate corresponding HIR IR. The lowering pass also inserts additional delay operations if the delay between the *def* and *use* operations is more than the required delay. In the previous example, if the *load* operation is scheduled 5 cycles after the *call* operation then an *hir.delay* operation would be inserted to ensure that the output of the *call* operation is available when the *load* operation executes.

Delay Coalescing: The affine-to-HIR lowering pass may insert a lot of delay operations that can be optimized away. These delays are implemented using shift registers in hardware which consumes extra register or lookup table. We implement a delay coalescing pass in order to reduce the depth of the shift registers. Figure 8 shows an example program before and after optimization. In this code, the two *delay* operations generating *%d1* and *%d2* use the same input *%d*. The *delay coalescing* pass reuses the first delay operation's output in the second delay. As a result of this optimization, the delay op corresponding to *%d2* only requires a shift register of depth 4 instead of 104.

Bitwidth Reduction: Another optimization performed on the generated IR is reducing the bitwidth of loop induction variables (IVs). The input C program to Polygeist uses *int* type for loop IVs. As a result the generated HIR code use 64 bit integers for the loop IVs. This pass analyzes all loop bounds and all the uses of the loop IVs and based on that determines if the bitwidth of the IV can be reduced. The *i-loop* in Figure 8 initially uses a 64 bit integer for the loop induction variable, but based on the loop bounds, the pass decides to reduce the bitwidth to 6 bits. Note that even if the loop bounds were not constants, the pass can still reduce the bitwidth of the *%i* variable because its only use is to downcast it to a 5 bit integer and the *nowrap* attribute (specified for the *comb.extract* operations in Figure 8) guarantees that cast does not change the value (else its undefined behavior). Thus the *%i* variable can only take values less than 32 during runtime. We use 6 bits because the non-inclusive upper bound is 32 which requires 6 bits to represent. This is an example of exploiting undefined behavior to enable new optimizations.

The bitwidth reduction pass does not require scheduling information. Thus, it could be applied on the unscheduled IR before lowering into HIR. But this would imply that if a DSL with its own custom scheduler (such as Darkroom[15]) directly lowers to HIR dialect, it has to reimplement the bitwidth reduction pass in its frontend. Thus, it is better to perform an optimization at the lowest level IR possible. We can not go below HIR because SV and HW dialect do not have a notion of for-loops. The combination of high-level control flow and explicit schedule allows HIR to implement the bitwidth reduction optimization at an abstraction level where both unscheduled HLS languages and DSLs with custom scheduler can benefit from the optimization.

In addition, the compiler reuses the constant folding, constant subexpression elimination and canonicalization passes from the MLIR infrastructure.

5.3 Schedule verification

We propose HIR as an intermediate representation to develop an LLVM-like open compiler infrastructure for HLS. Such a compiler infrastructure will be targeted by many HLS/DSL frontends of different levels of maturity. Furthermore, these frontends may implement their own domain specific schedulers [15]. We exploit HIR's design to implement a schedule verification pass that can catch scheduling

errors during compilation and improve the debugging experience while integrating new frontends and schedulers.

A schedule is invalid if the design may violate data dependence (read-after-write, write-after-read and write-after-write) between operations. Reading output of an operation before the operation completes or reading from a memory which is not written to yet are examples of incorrect scheduling.

As a part of our compiler, we implemented a schedule verification pass. Our verifier checks valid use of SSA values (integer and float variables). But it does not check memory operations. Thus, it can catch pipeline imbalance but it can not statically detect reading uninitialized memory address.

As described in Section 2.1, each SSA variable of integer or float type is associated with a time instant. We calculate this time as follows:

- For each SSA variable, if the defining operation is in an outer scope or a constant then assume that the variable is always valid.
- For each variable, find the start time of the defining operation.
- Each HIR operation with an output specifies the delay in generating the output from the start of the operation. For example, Figure 3 shows how the function call operation specifies the relative delay of the result. Use this delay along with the defining operations start time to calculate the time instant at which the SSA variable is valid.
- If the defining operation is a combinatorial operation then copy the delay information from one of its operands. Output of a combinatorial operation is valid in the same clock cycle in which all its inputs are valid.

Once we have the mapping from SSA variables to time, we can verify the schedule as follows:

- For *hir.store* and *hir.delay* operations, check that the input SSA variable should be valid in the time as the operation start time.
- For combinatorial operations make sure all the inputs are valid at the same time.
- For *hir.call* operation, use the operand delays specified in the function signature to calculate the time at which each operand SSA variable is expected. Match the SSA variable's time with the expected time.
- For SSA variables captured by *hir.for* or *hir.if* operation, assume that the variable is used by the operation at the operation's start time. Check if the variable is valid at that time.

In this pass, we only consider the SSA variables of integer and float type. We do not associate time with SSA variables of *hir.memref* type. We perform the schedule verification pass before the IR is submitted to the backend for code generation. This allows us to detect bugs introduced by the scheduler or any other optimization pass.

5.4 Backend

The backend of our compiler converts the verified HIR program to synthesizable SystemVerilog. This lowering process happens in multiple passes.

Loop simplification pass converts *for* loops into *while* loops. It instantiates a loop counter for the induction variable and extra registers to hold the loop bounds and the step size. It adds extra logic to check if the induction variable is within the loop bounds. The output of this check is used to break out of the while loop when the induction variable is no longer within bounds.

Buffer allocation pass involves instantiating multiple hardware buffers (such as block RAMs or registers) to implement a banked memory. The compiler only declares the functions corresponding to the memories and assumes that the actual implementation is provided by external vendor-specific Verilog libraries. We wrote a wrapper Verilog library to instantiate Xilinx FPGA block RAMs. HIR's ability to interface with arbitrary external Verilog modules allows us to use both memory and floating-point IPs provided by the FPGA vendor (Xilinx). One key difference between HIR and traditional HDLs (such as Verilog) is that HIR allows multiple writers for the same memory irrespective of the number of ports. It achieves this by assuming that those writes are not happening in the same cycle to the same memory port (else it is undefined behavior). The buffer allocation pass also inserts the multiplexing logic to select the correct drivers for the address and data buses.

Controller synthesis pass instantiates a finite state machine to implement the schedule specified in the HIR program using time variables. Since hardware is inherently parallel, each hardware unit in the datapath will operate every cycle in the absence of a controller - a multipliers will perform multiplication and a storage element (register or on-chip RAM) will write data every cycle irrespective of whether the input is valid or not. HDL designs are usually implemented as a combination of a datapath and a finite state machine to control/schedule the datapath operations. Since HIR contains the explicit schedule of each operation, the compiler backend can generate this FSM automatically.

The **hir-to-hw** lowering pass converts HIR into a combination of SV, Comb and HW dialects. The *hir.delay* operations are converted into shift registers. The number of shift registers depends on the *delay* and the initiation interval of the outer scope as discussed in Section 5.2. The *hir.while* operation is replaced with a state-machine that generates the control signal for the time variable of the loop body. Arithmetic operations on integers are replaced with the Verilog equivalents. Floating point arithmetic operations are already converted into calls to external Verilog functions during the frontend preprocessing (Section 5.1). All function calls are converted into Verilog module instantiations. Once all the operations in the function body are lowered to a combination of SV, Comb and HW dialects, the enclosing HIR function is converted into *hw.module*(Verilog module). The *export-verilog* pass generates SystemVerilog from the SV+Comb+HW dialect. The SV, Comb and HW dialects and the export-verilog pass are a part of the larger CIRCT infrastructure and not our contribution.

Transpose	Transpose a 16x16 matrix
Stencil	Convolution of a 64 element array with a kernel of size 2
Matmul-16x16	A 16x16 2D systolic array for matrix multiplication.
Convolution	2D Convolution of an 8x8 image with 2x2 kernel
gesummv	From Polybench benchmarks suit
floyd-warshall	From Polybench benchmarks suit
Unsharp Mask	Sharpening an image
Harris	Harris corner detection
DUS	Down sampling then upsampling an image
Optical	Lucas Kanade optical flow algorithm

Table 1: Benchmarks.

6 EVALUATION

We implemented HIR as an MLIR dialect using mainline MLIR infrastructure. We also implemented a code generator that transforms the HIR IR to *HW* and *SV* dialect of CIRCT. We utilize the existing *export-verilog* pass in the CIRCT infrastructure to generate SystemVerilog from the *hw* and *sv* dialect. The dialect implementation, optimization, lowering and verification passes together were implemented in approximately 8K lines of C++. Our code is open-sourced and is available [24] on GitHub.

HIR is designed to be a mid-level IR in an HLS pipeline. Thus, its ability to represent different kinds of parallelism such as instruction level parallelism, loop pipelining and running all iterations of spatial loops in parallel (*do-all* parallelism) will determine how well a scheduling pass can optimize the design. Additionally, the generated hardware should also be area efficient.

We decided to compare HIR against the Vitis HLS [17] compiler. We implemented our benchmarks in C. We used the same C code (with pragmas) for Vitis HLS with same levels of loop pipelining and unrolling and similar types of memories with same number of ports. We further simulated the generated SystemVerilog to verify that the designs achieve same level of performance. We synthesized and implemented the HIR and HLS generated (System)Verilog designs for the Xilinx VC709 FPGA evaluation platform using the Vivado synthesis tool. All results reported are obtained using Vivado and Vitis HLS version 2021.1. Both the Vitis HLS and HIR designs were synthesized for 200MHz.

We compare the quality of generated hardware on ten benchmarks. Table 1 describes the benchmarks. The first six benchmarks are are simple workloads to evaluate the resource usage at iso-performance. Among these, matrix multiplication is a particularly important benchmark for hardware acceleration due to its use in machine learning. Thus, we report the results for matrix multiplication for different sizes of the systolic array. The remaining four benchmarks are image processing kernels. These benchmarks contain multiple producer-consumer loop nests. We use them in our evaluation to show the benefit of overlapped execution of producer-consumer loop nests.

Table 2 compares the resource utilization of these benchmarks against Vitis HLS and Verilog implementations at iso-performance. Our resource usage of DSP blocks is always the same as Vitis HLS. The results differ only in the utilization of lookup tables and the number of registers. In the matmul benchmark, we used LUTRAM instead of BRAMs for both HLS and HIR since the buffers were small and heavily banked. For matrix transpose, our register usage is much lower than that of the HLS compiler. We believe this is because the HLS compiler pipelines the design more than what is necessary to achieve the desired frequency target. In the Matmul benchmark, HIR uses substantially more LUTs compared to the HLS design at smaller kernel sizes and more registers at higher sizes. This shows that there is still room for improving the LUT and register usage of HIR's lowering passes. Overall, the results in Table 2 show that the HIR compiler can generate Verilog designs that are comparable to Vitis HLS in resource usage while matching the performance.

We evaluate the effectiveness of overlapped execution on the remaining four benchmarks. Vitis HLS can overlap execution of producer consumer loop nests iff:

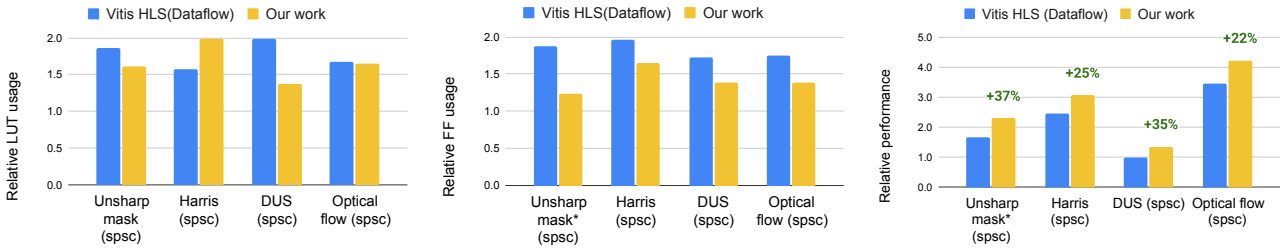
- There is only one producer and one consumer of each array.
- The consumer reads the data in the same order in which the producer writes it to the array.

Vitis HLS could not perform dataflow optimizations on the original program due to the single-producer-single-consumer rule violation. We had to insert additional array duplication loops for whenever there were multiple consumers. The resultant programs follow the single-producer-single-consumer rule. These transformations were required only to enable dataflow optimizations in Vitis HLS. Our scheduler does not have either of the above limitations. We also added the *dataflow* pragma which directs the Vitis HLS compiler to overlap execution of producer-consumer loops.

Figure 9c shows the performance of our compiler and Vitis HLS with dataflow optimization. Figure 9a and 9b shows the resource usage. All the numbers are relative to Vitis HLS without dataflow optimization. This shows the importance of dataflow optimizations. For example, in Unsharp mask, Vitis HLS with dataflow optimization gets a 1.7x performance improvement and our compiler gets 2.3x performance improvement, compared to Vitis HLS without dataflow optimization. The higher performance of HIR compiler is due to its ability to overlap producer-consumer loop nest's execution in the presence of arbitrary affine read/write order. Since, Vitis HLS requires the consumer to read the data in the same order in which the producer writes, it could not fuse some of the loop nest pairs. Vitis HLS replaces the arrays between producer and consumer loop nests with FIFO. At runtime, FIFO synchronizes between the two loop nests to ensure that the consumer always reads valid data. In contrast, our ILP based static scheduler inserts a fixed delay in the consumer's schedule to ensure that the consumer always reads correct data. Due to the extra synchronization required in the Vitis HLS synthesized design, it consumes more lookup tables (Figure 9a) and registers (Figure 9b) with the exception of LUT usage of Harris corner detector. To conclude, Figure 9c shows the performance and resource usage advantage of overlapped execution and static scheduling for affine workloads.

Benchmark	Vitis HLS/Verilog				HIR			
	LUT	FF	DSP	BRAM	LUT	FF	DSP	BRAM
Transpose	7	51	0	0	16	22	0	0
Stencil	152	237	6	0	113	125	6	0
Convolution	80	132	3	0	86	112	3	0
gesummv	276	353	12	0	128	283	12	0
floyd-warshall	168	162	0	0	119	169	0	0
Matmul-4x4	861	2019	48	0	1173	1802	48	0
Matmul-8x8	3553	6914	192	0	3684	7294	192	0
Matmul-16x16	14495	24538	768	0	12645	29062	768	0

Table 2: FPGA resource usage and comparison with Vitis HLS/Verilog.



(a) LUT usage. **(b) FF usage.** **(c) Perf relative to Vitis HLS.**

Figure 9: Performance improvement and resource usage of our compiler.

7 RELATED WORK

Static Single Assignment (SSA) [12] based intermediate representations are a standard in software compilation pipelines. Both LLVM [21] and GCC [34] use SSA based IRs for their compilation pipeline. Some HLS compilers [6, 35] also reuse these software IRs for high-level synthesis.

7.1 Hardware intermediate representations

Like HIR, LLHD [37] is another intermediate representation for hardware description that was later migrated to MLIR and is hosted as part of CIRCT [10]. LLHD attempts to cover all stages of hardware synthesis. It provides IR constructs to describe behavioral and structural circuits as well as netlists. LLHD is a low-level IR with similar abstraction level as HDLs (Verilog/VHDL). It does not have loops and it expects that loop unrolling is done by the language frontend. While LLHD is suitable for targeting HDLs and as a low-level IR in the HLS pipeline, we believe that HIR, due to its high-level abstractions and decoupled schedule, is a better target for scheduling optimization passes as a mid-level IR. Currently, HIR lowers to SystemVerilog, but LLHD could serve as an alternative code generation target. We intend to contribute HIR back to the CIRCT project repo, and develop it further within its community.

The FIRRTL [18] IR is designed along with the Chisel [2] hardware construction language. FIRRTL designs are represented as an abstract syntax tree. FIRRTL offers features like type and width inference for easier Chisel interoperability. Similar to LLHD and in contrast to HIR, FIRRTL is also a low level IR which can be a suitable target for HDLs.

μIR [38] decouples the micro-architectural representation of the accelerator from its behavioral specification. Optimizations like pipelining and retiming are implemented as transformations of structural graph. Calyx [32] represents an accelerator using a structural sub-language and a control sub-language. The control sub-language offers *while* loops and *if* statements. In addition to these, the control sub-language also has *seq* and *par* blocks which together provide fork-join parallelism to the IR. Unlike Calyx, HIR has a unified representation. It does not separate control and structural sub-languages. Calyx can capture latency insensitive designs. It uses handshake signals to enforce the correct schedule at runtime. Calyx can optionally generate statically scheduled circuits if the latency of all operations are provided. HIR designs are always statically scheduled. Though the original Calyx paper did not support pipelining, it has since then added new operators to support it. The primary difference between Calyx and HIR are the time variables and scheduling contract embedded in the HIR function signature. As Calyx is an unscheduled IR, it can not interface with external verilog modules that have specific scheduling requirements on the inputs. For instance, Xilinx's DSP blocks [40] support multiply accumulate ($a * b + c$) operation where the adder input (c) must arrive

after a fixed number of cycles from the other inputs (similar to Figure 3).

HIR can complement these IRs in an HLS pipeline where the latency insensitive IRs can represent the design before the scheduling pass and HIR can represent the hardware design after the instruction scheduling is done. In fact, MLIR was designed to be a common infrastructure where multiple intermediate languages can interoperate and frontends can use a mix of IRs to generate the most optimal design.

7.2 HIR IR for HLS compilation

In this section, we briefly discuss high-level synthesis languages and how HIR could potentially fit in as an IR in their compilation flow.

Darkroom [15] generates structural Verilog circuits from programmer specified high-level description of image processing kernels. The programmer can represent the image processing kernel in a high-level DSL and the Darkroom compiler will automatically find a parallel schedule and lower it down to a hardware design. Aetherling [14] is another interesting work on high-level synthesis. It introduces space-time types that can be used to describe dataflow pipelines with exact throughput and latency requirements. The language attempts to remove the need for synchronization overhead between different stages of the pipeline. Aetherling generates the hardware in Chisel [2] which can then be lowered to Verilog using the Chisel compiler.

Both the Darkroom and Aetherling compilers implement their own custom schedulers, tailored to the DSL's programming model. Thus, they can not target Vitis HLS, or an unscheduled IR, such as Calyx and μIR. Due to our LLVM like, pass-based compiler and the HIR IR, a DSL can choose to target our compiler at any stage depending on how much information the DSL frontend wants to convey to the compiler. For instance, as Darkroom and Aetherling use custom schedulers, they can directly target the HIR IR with their custom schedule. This allows the DSLs to take advantage of optimizations such as constant folding, bitwidth-reduction and delay coalesing, as well as reuse the compiler backend to generate SystemVerilog. Additionally, the schedule verification can help in identifying bugs in the implementation of the the DSL-specific schedulers early in their development.

AutoSA [39] and PolySA [11] are two DSLs designed to synthesize affine kernels into systolic arrays. Unlike Darkroom and Aetherling, these compilers do not perform scheduling. Instead, they rely on a high-level synthesis compiler such as Vitis HLS to generate the final Verilog design. They use polyhedral analysis to perform high-level loop transformations, such as strip-mining and loop permutation, to generate a C/C++ program that can be better optimized by the HLS compiler. For instance, if the innermost loop has a loop-carried dependence then AutoSA finds an outer parallel loop, strip-mines it and permutes it to the innermost loop. The resulting design does not have a loop-carried dependence in the innermost loop and thus, can be fully pipelined. As these DSLs do not have a custom scheduler, they can target the Affine dialect in our compiler stack and reuse the ILP based auto-scheduler of the HIR compiler.

In addition, many general-purpose and domain-specific languages have been proposed for high-level synthesis. Bluespec-Verilog [33] represents a circuit as a set of atomic rules. The order in which these rules can fire is non-deterministic. A recent work [5] attempts to enhance the language for user-level control over the schedule and predictable performance. The Dahlia[31] language is inspired from the observation that HLS languages generate unpredictable designs due to their excessive flexibility. Dahlia and its affine type system enforces the high-level design to respect the limitations of hardware. For example, a valid design is guaranteed to never have multiple reads and writes on the same memory in the same cycle. Rigel [16] extends Darkroom with dynamic scheduling to handle multi-rate image pipelines. Our compiler stack can not be used for these languages since they support non-affine workloads and/or require dynamic scheduling.

8 CONCLUSION

We introduced HIR, an intermediate representation to describe FPGA based hardware accelerator designs. We also build a compiler using HIR for affine kernels. HIR has support for multi-dimensional arrays and high level control flow such as loops. The IR captures computation schedules explicitly, which makes it an ideal target IR for automatic scheduler passes. HIR is built in a rigorous manner drawing from best practices learnt from compiler IR design in the open-source communities of LLVM and MLIR. Preliminary results demonstrate the effectiveness of our IR as a mid level intermediate representation for post-scheduling optimizations and the backend code generator.

ACKNOWLEDGEMENTS

We would like to thank the Science and Engineering Research Board (SERB), India for a grant under the EMR program (EMR/2016/008015) which has supported this research work in part.

REFERENCES

[1] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. 2012. A Compiler and Runtime for Heterogeneous Computing. In *Design Automation Conference*. 271–276.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[3] David F. Bacon, Rodric M. Rabbah, and Sunil Shukla. 2013. FPGA programming for the masses. *Commun. ACM* 56, 4 (2013), 56–63.

[4] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. arXiv:2003.00532 [cs.PF]

[5] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

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

[7] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, and Nicolas Vasilache. 2021. Progressive Raising in Multi-Level IR. In *International Symposium on Code Generation and Optimization (CGO)*. ACM.

[8] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-Level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/3373087.3375297

[9] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs. In *International Conference on Parallel Architectures and Compilation (PACT)* (Haifa, Israel). 327–338.

[10] The CIRCT community. 2020. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt.

[11] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Diego, CA, USA). IEEE Press, 1–8. https://doi.org/10.1145/3240765.3240838

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.

[13] C. Dase, J.S. Falcon, and B. MacCleery. 2006. Motorcycle control prototyping using an FPGA-based embedded control system. *Control Systems, IEEE* 26, 5 (2006), 17–21.

[14] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983

[15] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[16] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible Multi-Rate Image Processing Hardware. *ACM Trans. Graph.* 35, 4, Article 85 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925892

[17] Xilinx Inc. [n. d.]. Vivado High-Level Syntehsis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[18] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[19] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. 2020. Compiling ONNX Neural Network Models Using MLIR. arXiv:2008.08272 [cs.PL]

[20] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, , and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain-Specific Computation. In *International symposium on Code Generation and Optimization (CGO)*.

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL]

[24] Kingshuk Majumder and Uday Bondhugula. 2021. HIR source code. https://github.com/mcl-csa/hir-dev

[25] Kingshuk Majumder and Uday Bondhugula. 2023. Automatic multi-dimensional pipelining for high-level synthesis of dataflow accelerators. arXiv:2309.03203 [cs.AR]

[26] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 245–257. https://doi.org/10.1109/MICRO.2018.00028

[27] matlab-hdl-coder [n. d.]. MATLAB HDL Coder. The MathWorks Inc. http://in.mathworks.com/products/hdl-coder//.

[28] MLIR. 2020. MLIR: Talks and related publications. https://mlir.llvm.org/talks/.

[29] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[30] Walid A. Najjar, Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. 2003. High-Level Language Abstraction for Reconfigurable Computing. *Computer* 36, 8 (Aug. 2003), 63–69.

[31] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[32] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[33] R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[34] Diego Novillo. 2003. Tree SSA—a new high-level optimization framework for the gnu compiler collection. (01 2003).

[35] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*. IEEE, 1–4. https://doi.org/10.1109/FPL.2013.6645550

[36] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. 2014. Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators. In *2014 International Conference on Hardware/Software Codesign and System Synthesis*. Article 17, 17:1–17:10 pages.

[37] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-level Intermediate Representation for Hardware Description Languages. arXiv:2004.03494 [cs.PL]

[38] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μIR -An Intermediate Representation for Transforming and Optimizing the Microarchitecture of Application Accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 940–953. https://doi.org/10.1145/3352460.3358292

[39] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3431920.3439292

[40] Xilinx. 2018. User guide: 7 Series DSP48E1 Slice. https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[41] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[42] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. *AutoPilot: A platform-based ESL synthesis system*. 99–112. https://doi.org/10.1007/978-1-4020-8588-8