An MLIR-Based High-Level Synthesis Compiler for Hardware Accelerator Design

A Thesis

Submitted For the Degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Kingshuk Majumder

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

JANUARY 2023

TO

My parents

Acknowledgements

I am thankful to my advisor, Uday Bondhugula, for his constant support and guidance. He was always keen to discuss about new ideas and gave me complete freedom in my research direction. He always believed in me through all the tough times during my research. I would also take this opportunity to thank Prof. R Govindarajan, Y.N Srikant and Uday Bondhugula for teaching computer science to an electrical engineer. Without their execellent teaching and support, I would not be able to acquire the necessary skills to complete my Ph.D.

I would like to thank Prateek Jain for giving me the opportunity to work in a world class machine learning team at Google as a student researcher. I would also extend my thanks to my team at Google for all the technically stimulating discussions and for their patience in teaching me the basics of modern machine learning.

I would take this opportunity to thank Prof. Arkaprava Basu for inviting me to his lab's technical discussions and the stimulating conversations. I would like to thank my friends Niladri Das, Daniel Sanju, Narmada Naik, Prateek, Ajay Naik and Shweta Pandey for their support and kindness.

I would conclude by acknowleding the unwaivering support, and countless sacrifices of my parents, Swapan and Madhumita Majumder, without which this work would not have been possible.

Vita

Bachelor of Engineering :	Electrical Engineering Department Jadavpur University, West Bengal, India. 2008 - 2012
Master of Engineering :	Signal Processing, Electronics and Communication Engineering Department, Indian Institute of Science, Bangalore, India. 2012 - 2014
Hardware ASIC Engineer :	Nvidia, Bangalore, India. 2014 - 2016
Project Associate :	Multicore Computing Lab, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India. 2016 - 2017
Student Researcher :	Google Research, Bangalore, India. March-Oct 2022

Publications based on this Thesis

1. Kingshuk Majumder, Uday Bondhugula. Feb-2021. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. [arXiv:2103.00194][code].

2. Kingshuk Majumder, Uday Bondhugula. Sep-2022. An ILP-based Automatic Scheduler for High-Level Synthesis of Domain-specific Accelerators [code][Under review].

3. Kingshuk Majumder, Uday Bondhugula. Oct-2022. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. [arXiv:2103.00194][code] [**Accepted in ASPLOS'24**].

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. High-level synthesis (HLS) offers a more compiler-centric alternative to the traditional Verilog-based hardware design improving developer productivity.

Though HLS offers many advantages over traditional HDL-based hardware design flow, it is still not a mature ecosystem. There is a need for research in both programming abstraction for hardware design and compiler optimizations to meet the efficiency of hand-optimized HDL designs. In the software world, LLVM has enabled rapid prototyping of programming languages. A new programming language can target the LLVM compiler and benefit from the existing optimizations and backend code generation. LLVM also enables the development of new compiler optimizations. A new optimization pass can be plugged into the existing compiler pipeline to evaluate its benefits on existing programming languages and benchmarks. This decoupling of different stages of the compiler pipeline can be largely attributed to the LLVM intermediate representation.

The high-level synthesis ecosystem still lacks such extensible modular compiler infrastructure which could be used for the development of new HLS programming languages and optimizations. In this work, we propose an MLIR based end-to-end HLS compiler and an intermediate representation that is suitable for the design and implementation of domain-specific accelerators for affine workloads. Our compiler brings similar levels of modularity and extensibility to the HLS compilation domain, which LLVM brought

in the area of software compilation. A modular compiler infrastructure offers the advantage of incrementally introducing new language frontends and optimization passes without the need to reinvent the whole HLS compiler stack.

Our compiler converts a high-level description of the accelerator specified in the C programming language into a register-transfer-level(RTL) design (SystemVerilog). We use memory dependence analysis and integer-linear-program(ILP) based automatic scheduling to improve loop pipelining, and introduce parallelization between producer-consumer kernels. Our ILP-based optimizer beats the state-of-the-art Vitis HLS compiler by 1.3x in performance over a representative set of benchmarks, while requiring fewer FPGA resources.

Contents

List of Tables

List of Figures

Keywords

MLIR, HIR, HLS, Accelerator, HDL, Verilog, Compiler, Pipelining, Parallelization, CIRCT, ILP.

Notation and Abbreviations

BRAM - Block RAM

CGRA - Coarse-Grained Reconfigurable Array

FIFO - First In First Out (Queue)

FMA - Fused Multiply Add FPGA - Field Programmable Gate Array

FSM - Finite State Machine

GLPK - GNU Linear Programming Kit

GPGPU - General Purpose Graphic Processing Unit

HDL - Hardware Description Language

HIR - Hardware Intermediate Representation

HLS - High-Level Synthesis

ILP - Integer Linear Program

IP - Intellectual Property

IR - Intermediate Representation

LUT - Lookup Table

MUX - Multiplexer

MLIR - Multi-Level Intermediate Representation

RAM - Random Access Memory

SPSC - Single Producer Single Consumer

Chapter 1

Introduction

The interest in specialized hardware accelerators has seen a steady rise in the past decade. The growing compute demand for machine learning and image processing in data centers and edge devices, coupled with the need for high energy efficiency, has motivated both industry and academia to explore domain-specific accelerators that can outperform traditional general-purpose compute hardware. To that end, many new accelerators have been proposed by both industry [Jouppi et al. (2017); Choquette et al. (2021); Fricker (2022); Gaide et al. (2019)] and academia [Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne (2016); Wang et al. (2021); Chen et al. (2014); Kwon et al. (2018); Liu et al. (2015)]. Even General purpose GPUs (GPGPUs), which are designed for efficient execution of highly parallel workloads, introduced specialized accelerators [Markidis et al. (2018)] to address the growing compute demand of machine learning workloads.

Accelerators can optimize the compute and the memory hierarchy to better match the requirements of the workload compared to a general-purpose processor. For instance, both CPUs and GPUs employ hardware-managed caches to capture data reuse. In addition, prefetchers are often used in the hardware to predict complex data access patterns and fetch them to the caches. Prefetchers help in overlapping the compute with the data transfer to/from the memory. This reduces the compute stalls and improves performance. The extra logic required to manage these hardware structures consumes

hardware resources and extra power. On the other hand, accelerators are designed for specific workloads. This simplifies the caching and prefetching mechanisms a lot. Accelerators often use scratchpad memories as software-managed caches and the software/-firmware takes care of the prefetching based on the known data access pattern of the target workload. This results in comparable or better data reuse in accelerators while using fewer hardware resources. Similarly, depending on the workload, accelerators may not require special function units (in GPUs) or support for out-of-order execution. All these application-specific optimizations yield more power and area efficient architectures.

Prior work [Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne (2016)] has shown that different types of on-chip memory accesses can have very different energy costs. Accessing smaller memories such as registers may require an order of magnitude less energy compared to scratchpad memories even though both are on-chip SRAMs. This is attributed to the greater wire length and wider multiplexers for data selection. Systolic arrays are a great example of utilizing this observation to improve power efficiency. The streaming data in a systolic array move from register to register along the width (or height) of the systolic array, reducing the number of scratchpad memory accesses. This improves the power efficiency of the architecture. Such optimizations are not always possible in general-purpose architectures.

With the advantages of these accelerators, comes certain disadvantages as well. Accelerators are not as adaptable as CPUs and GPGPUs. The exact algorithms used for a given workload change/improve over time. For instance, many researchers are trying to improve the performance of machine learning models by exploiting sparsity in the weight and activation matrices. Models are trained to increase sparsity without losing accuracy. These sparsity-related algorithmic improvements do not benefit the actual performance of the neural network if an architecture is not designed to exploit the sparsity and improve memory bandwidth utilization or compute efficiency (by skipping zeroes). CPUs and GPUs require only changes in the software, but accelerators need to be redesigned from scratch to take advantage of these algorithmic advances. In case of ASIC implementations [Jouppi et al. (2017)], designing a new accelerator for modest algorithmic

improvements may not even be an economically feasible approach.

The other disadvantage of accelerators is becoming prominent with the increase in heterogeneity of modern system-on-chips (SOCs). Current generation processors employ dozens of accelerators for different workloads such as image processing, video encoding/decoding, encryption and video compression/decompression. Even though not all the accelerators are used at all times, they consume valuable die area. This partially negates the advantage of area efficiency of custom accelerators over general-purpose architectures.

FPGAs offer an interesting tradeoff between the generality of CPUs and GPUs and the efficiency of custom-designed accelerators. They are more power efficient and have more parallelism than CPUs while being completely reconfigurable. They achieve this at the cost of greater die area and lesser operating frequency compared to an ASIC or CPU. Though FPGAs can never compete in performance or power efficiency with an ASIC, when we factor in the previously discussed issues of adapting to iterative algorithmic improvements and taking advantage of the complete die area in the presence of accelerator heterogeneity, the FPGAs offer a promising alternative. As FPGAs can be reconfigured any number of times, architectures can better adapt to algorithmic improvements. This also incentivizes further research into algorithmic improvements such as increased sparsity, block sparse and low-precision compute. Additionally, different accelerators can time share the whole FPGA instead of statically partitioning the die area among all accelerators.

Custom accelerators on reconfigurable computing platforms such as FPGAs are able to achieve high power efficiency [Zhang et al. (2016)] and performance but the difficulty associated with programming them is seen as a major roadblock towards their mass adoption. High-level synthesis [Canis et al. (2011a)] offers a promising solution towards making custom accelerator design more approachable. DSLs for hardware design can offer an even higher level of abstraction to the algorithm developers and benefit from being able to employ the right high-level synthesis (HLS) pipelines.

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 [Auerbach et al. (2012); Hegarty et al. (2014, 2016); Reiche et al. (2014); Chugh et al. (2016a); Inc. ([n. d.]); matlab-hdl-coder ([n. d.]); Dase et al. (2006); Najjar et al. (2003); Cong et al. (2011)]. Nearly all electronic design automation vendors now provide suites supporting HLS, examples of which include Xilinx Vivado, Cadence C-to-silicon, Synopsys Synphony, and Mentor Graphics Catapult HLS. A comprehensive survey of various HLS approaches were conducted by Bacon et al. (2013) and Cong et al. (2022).

All these HLS compilers have to create their intermediate representations, thus duplicating and re-implementing a lot of the representation and transformation infrastructure that could otherwise have been shared. Similarly, DSLs [Nigam et al. (2020); Hegarty et al. (2014); Koeplinger et al. (2018); Durst et al. (2020)] either rely on vendor-provided HLS tools or require reimplementing the compiler pipeline, including many standard optimizations, to translate the design into a hardware description language (HDL) like Verilog or Chisel [Bachrach et al. (2012a)].

The problem is even more pronounced while developing new optimizations. Due to lack of a standard compiler infrastructure, any new optimization work [Wang et al. (2021)] has to implement its 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 [Lattner and Adve (2004)] is a great example of applying this approach to software compilation flow.

In this thesis, we propose the HIR compiler infrastructure and the HIR intermediate representation for high-level synthesis of affine workloads. Affine kernels are present in both image and audio processing as well as machine learning workloads. Often these affine kernels are the performance bottleneck in these workloads. For instance, the majority of the computation in both convolutional neural networks and neural networks for language processing such as transformers are from matrix multiplications. Affine memory accesses also enable the compiler to perform a more precise memory dependence analysis.

The HIR intermediate representation is designed to represent hardware accelerators. It decouples the functional design from the exact schedule giving the compiler greater freedom to choose the implementation for the state-machine-based controller. The IR has a custom data type to represent multi-ported, banked tensors and high-level control flow operations such as loops and if-else statements.

We also implement an ILP-based scheduler for automatic parallelization of the input sequential hardware description in C language. Our scheduler can perform multi-level loop pipelining. It uses the memory dependence analysis to pipeline across producer-consumer loop nests. We also implemented a backend code generator to generate the accelerator in SystemVerilog from the HIR description. Overall, our compiler takes a sequential C implementation of the accelerator with directives for the level of pipelining and parallelism from the programmer and generates a parallel hardware design in SystemVerilog. We have open-sourced the implementation of the HIR compiler [Majumder and Bondhugula (2022)].

The rest of the thesis is organized as follows. In Chapter 2, we provide the necessary background on MLIR, HLS compilers and FPGAs. Chapter 3 introduces the HIR intermediate representation, the optimization passes and the backend. Chapter 4 provides a detailed discussion on our ILP-based automatic scheduler. We evaluate our HLS compiler against Vitis HLS in Chapter 5. We discuss the related work in Chapter 6. Finally, we conclude our thesis in Chapter 7.

Chapter 2

Background

In this chapter, we will cover the necessary background for this thesis. Section 2.1 describes the internal design of FPGAs and the challenges in programming them. In Section 2.4, we discuss the Vitis HLS compiler and its dataflow optimization.

2.1 FPGA

Field programmable gate arrays (FPGAs) are used to implement hardware design. An FPGA contains multiple types of hardware resources such as arithmetic and logic units, block RAMs, registers, multiplexers and lookup tables in a grid. These components are connected via an on-chip reconfigurable network as shown in Figure 2.1. The FPGA can

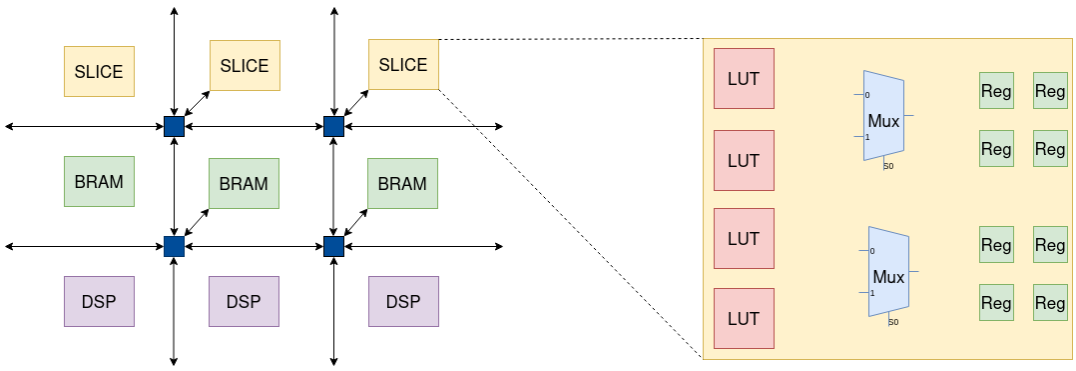


Figure 2.1: Internal design of an FPGA.

be configured to connect any component to any other component. A hardware design can be implemented on the FPGA by connecting various hardware components using the on-chip network.

The FPGAs differ from CPUs and GPUs in a few important ways. FPGAs do not have any hardware-managed caches. All the on-chip memory is exposed as scratchpad buffers and registers. CPUs and GPUs have a unified address space. Each memory location is associated with a unique address and any part of the program can access any memory location using its address. FPGAs do not have a unified address space. As a result, memory locations do not have a unique global address. In a hardware design, each operation is usually connected to only a subset of the scratchpad memories and can only access these memories. The optimal memory hierarchy is dependent on the dataflow patterns of an application. For instance, certain image-processing applications may require point-to-point communication between functional units [Hegarty et al. (2014, 2016)] and matrix multiplication may require scratchpad memories and registers to capture temporal reuse of the input matrices [Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne (2016); Jouppi et al. (2017)].

In CPUs and GPUs, the hardware components such as on-chip memory (caches, registers and scratchpad memories), arithmetic and logic units and the on-chip interconnect are abstracted away from the programmer. The instruction set architecture (ISA) exposes a well-defined, and restricted view of the processor internals. As a result, the software can target the CPU/GPU without caring about the internal components. For instance, in modern CPUs, the ISA exposes a set of logical registers. But the number of physical registers available in the processor is far greater than that. The processor internally maps the architectural registers to the physical registers during the program execution. Similarly, many details such as the number of execution cores, the synchronization latency between cores and the size of caches are not exposed as part of the programming model of GPUs. The programmer may use this information to do additional optimizations on the GPU, but it is not required to write correct parallel programs.

FPGAs do not have an instruction set. All the hardware components such as registers,

on-chip memories, arithmetic and logic units, on-chip interconnect and their latencies (number of cycles) are exposed as a part of the FPGA's programming model. A hardware designer needs to specify the exact connection between different hardware components to realize a specific accelerator design.

Since, hardware is inherently parallel, ordering between the execution of different operations must be enforced using state machines. In contrast, each thread in a CPU/GPU program is guaranteed to execute its instructions sequentially. An FPGA design has to schedule different operations in parallel to exploit the instruction level parallelism in the design. CPUs and GPUs also exploit instruction-level parallelism but do not expose it to the programmer. While FPGAs offer fine-grained parallelism, CPU and GPU programs have coarse-grained parallelism in the form of threads.

An FPGA design usually has two components. The datapath specifies how data moves and what operations are performed and how the logic elements are connected to the on-chip buffers. The controller is responsible for specifying the time at which any operation in the datapath is triggered. It is implemented as a finite state machine (FSM).

Another important goal of a hardware design is to achieve a high frequency. Unlike, CPUs and GPUs, the connection between different hardware components is not fixed in an FPGA. As a result, the maximum frequency achieved by an FPGA design depends on which components are connected and on the length of the connecting wires.

Figure 2.2 shows an example of a Verilog module implementing the multiply-accumulate operation. The registers are assigned in an *always* block. For example, the register *a1* is assigned from the wire *a*. The value in *a* will propagate to *a1* at the next positive edge of the clock. In contrast, the assignment to *out* from *acc3* happens immediately (in the same clock cycle). The whole operation of multiply-accumulate takes 3 cycles. The variables a and b are expected to arrive at the first clock cycle. Input variable c is expected to arrive after one clock cycle delay. Verilog modules do not capture this information about the expected arrival time of each input and the time at which the output is ready. As a result, in a larger design, one multiply-accumulate unit can not be replaced with another unit of different latency even though they are functionally equivalent.

```
module multiply_accumulate(
    input wire[31:0] a,
    input wire[31:0] b,
    input wire[31:0] c,
    output wire[31:0] out,
    input wire clk
);
reg [31:0] a1;
reg [31:0] b1;
reg [31:0] ab2;
reg [31:0] c2;
reg [31:0] acc3;
always @(posedge clk) begin
    a1 <= a;
    b1 <= b;
end

wire [31:0] ab = a1 * b1;
always @(posedge clk) begin
    ab2 <= ab;
    c2 <= c;
end

wire acc = ab2 + c2;
always @(posedge clk) begin
    acc3 <= acc;
end
assign out = acc3;
endmodule
```

Figure 2.2: Verilog module for the multiply-accumulate operation.

2.2 MLIR

In this section, we will provide a background on the MLIR compiler infrastructure [Lattner et al. (2021)]. There are a lot of common building blocks that can be reused across different compilers. Every compiler requires intermediate representations (IR), pass infrastructure for analysis and transformation and printing and parsing support for the IR. Providing a common infrastructure for developing IRs and analysis/transformation passes on these IRs reduces the effort in developing new compilers. MLIR is a compiler infrastructure that can be used to design custom intermediate representations for compilers. It provides a pass infrastructure to develop custom passes of the IR and support for custom parsing and pretty-printing.

An intermediate representation in MLIR is called a *dialect*. MLIR allows combining multiple dialects in a single program. Figure 2.3 shows an example code snippet in the*Affine* dialect of MLIR along with some operations from the *Func* and *Arith* dialects. An MLIR program consists of *operations* such as *arith.addf* in Figure 2.3a. Each operation can have input arguments (*%arg2* and *%2* for arith.addf operation). Operations can produce one or more results. The input operands and the results are *static-single-assignment* (SSA) variables, i.e. the variable can not be reassigned. In addition to the input operands, an MLIR operation may have *attributes*. These attributes provide extra information for the operation. For instance, the constant value of 0 in the *arith.constant* operation in Figure 2.3a is an attribute. The operation name is always qualified by the dialect name, i.e. the operation *arith.constant* belongs to the *Arith* dialect.

Operations can have their own *regions*. This allows MLIR dialects to have high-level control flow operations. For example, the *affine.for* operation in Figure 2.3a has one region that contains all the operations in its body. Other operations such as *if-else* may require multiple regions. Regions are allowed to nest - the *func.func* operation defines its own region for its body and the *affine.for* op defines a region nested inside this region.

In addition to custom operations, dialects may define their custom data types. The *memref* type in Figure 2.3a shows an example of a custom data type. The *memref* type is not qualified by a dialect name because it is part of the *builtin* dialect. Types

```
module {
  func.func @reduce(%arg0: memref<1024xf32>) -> f32 {
    %cst = arith.constant 0.000000e+00 : f32
    %0 = affine.for %arg1 = 0 to 10 step 2 iter_args(%arg2 = %cst) -> (f32) {
      %1 = affine.load %arg0[%arg1] : memref<1024xf32>
      %2 = arith.addf %arg2, %1 : f32
      affine.yield %2 : f32
    }
    return %0 : f32
  }
}
```

(a) Custom pretty printer.

```
#map0 = affine_map<(d0) -> (d0)>
#map1 = affine_map<() -> (0)>
#map2 = affine_map<() -> (10)>
"builtin.module"() ({
  "func.func"() ({
  ^bb0(%arg0: memref<1024xf32>):
    %0 = "arith.constant"() {value = 0.000000e+00 : f32} : () -> f32
    %1 = "affine.for"(%0) ({
    ^bb0(%arg1: index, %arg2: f32):
      %2 = "affine.load"(%arg0, %arg1) {map = #map0} : (memref<1024xf32>, index) -> f32
      %3 = "arith.addf"(%arg2, %2) : (f32, f32) -> f32
      "affine.yield"(%3) : (f32) -> ()
    }) {lower_bound = #map1, step = 2 : index, upper_bound = #map2} : (f32) -> f32
    "func.return"(%1) : (f32) -> ()
  }) {function_type = (memref<1024xf32>) -> f32, sym_name = "reduce"} : () -> ()
}) : () -> ()
```

(b) Generic printer.

Figure 2.3: Example of MLIR Affine dialect.

defined in any other dialect are qualified by the dialect name similar to the dialect operations. Types may contain attributes. For instance, the *memref* type stores the tensor dimensions and the element's datatype as attributes.

MLIR provides a generic parser and printer for each dialect. In addition to this, it also provides necessary infrastructure such as lexers and printing/parsing-related helper functions to write custom parsers and printers for each dialect. Dialects may want to define their pretty printing format for improved readability. Figure 2.3b shows the generic printer output for the program in Figure 2.3a. It is easier to understand the input operands and attributes of a function from the generic format. For instance, *arith.constant* operation has no operands and one attribute named *value*, and *affine.load* has two operands (*%arg0* and *%2*) and one attribute named map (an *affine-map* attribute).

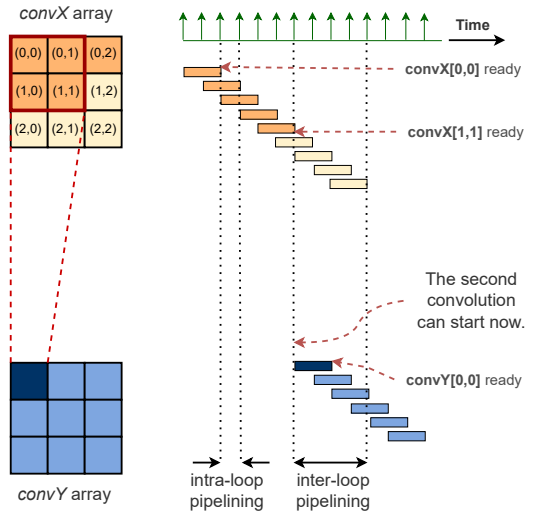
MLIR uses LLVM's *tablegen* infrastructure to allow defining dialects, operations, types, printers and parsers for the operations and types in a succinct domain-specific language. The tablegen infrastructure generates the necessary class definitions from the tablegen description. Tablegen is also used to define custom passes. A pass recursively visits the operations in the program for analysis and transformation. MLIR also provides pattern-based graph rewriting for simple transformations such as peephole optimization. Many of the MLIR passes can be reused by custom dialects. For instance, dialects can provide custom implementation of certain interfaces for its operations to allow transformation passes such as canonicalization and constant subexpression elimination to run on the dialect operations. This reduces the duplicate effort required to implement these basic passes for each dialect.

Overall MLIR provides a robust infrastructure to develop intermediate representations and compilers. It reduces the programmer's effort by providing many of the common building blocks for compiler development. We built our HLS compiler on top of the MLIR compiler infrastructure.

```
//Producer loop-nest.
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++)
#pragma pipeline
    for (int u = 0; u < 2; u++)
#pragma unroll
      for (int v = 0; v < 2; v++)
#pragma unroll
        convX[i][j] += image[i+u][j+v] * wx[u][v];
}

//Consumer loop-nest.
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++)
#pragma pipeline
     for (int u = 0; u < 2; u++)
#pragma unroll
       for (int v = 0; v < 2; v++)
#pragma unroll
         convY[i][j] += convX[i+u][j+v] * wy[u][v];
}
```

(a) Two convolution operations in a series.



(b) Data dependence & Timing diagram.

Figure 2.4: Example of a producer and consumer loop nest. *The consumer loop can start execution when the producer has written the first two pixels in row R1 of array convX.*

2.3 Dataflow optimization

Image-processing and machine-learning workloads are often composed of multiple smaller kernels such as convolution and matrix multiplication. For example, the *Unsharp mask* algorithm contains a series of convolution operations. Machine learning models also often chain multiple window-based (such as convolution or max-pooling) and point-wise (such as RELU) operations together. These tensor operations, represented as a loop nest, may have a producer-consumer relationship among them. Figure 2.4a shows an example of two convolution operations applied on an image successively. The second convolution is dependent on the output of the first convolution. The output of the first convolution loop-nest is written to the *convX* array by the first loop-nest. The second loop-nest consumes the *convX* array as input for the second convolution. To improve performance, we unroll the inner two loops and pipeline the *j-loop* in both convolutions. Figure 2.4b highlights the elements of the array *convX* that are required to calculate the first element of the *convY* array. It also shows the timing diagram of iterations of the *j-loop*. Each iteration of the j-loop is independent of the previous iterations. Thus,

before an iteration of the loop completes, the next iteration can be started as shown in the timing diagram. Overlapping the execution of multiple *j-loop* iterations improves parallelism. In addition to pipelining between loop iterations (*intra-loop pipelining*), there is also an opportunity to pipeline between the producer and consumer loop nests (*inter-loop pipelining*). The second convolution (consumer loop nest) does not have to wait for the first convolution to complete. It can start as soon as enough data is written to the *convX* matrix to calculate the first *convY* value. Pipelining between the producer and consumer convolutions overlaps their execution as shown in Figure 2.4b, which further improves parallelism.

2.4 Vitis HLS

Vitis HLS (also known as Vivado HLS) is a high-level synthesis compiler toolchain from Xilinx for their FPGAs. We compare our work against Vitis HLS in this thesis due to its widespread use and maturity. Vitis HLS takes C/C++ as input and generates Verilog as its output. Vitis HLS is built on top of the LLVM infrastructure and utilizes the LLVM IR for many of its optimizations and lowering passes.

The hardware design is expressed as a single-threaded C/C++ program. Vitis HLS supports adding pragmas to control the generated hardware. For instance, there are pragmas for loop unrolling and pipelining. These pragmas help the hardware designer choose the required parallelism and resource usage. An unrolled loop will duplicate the resources (such as multipliers and adders) used inside the loop, but may offer greater parallelism and hence performance. Pragmas are also used to specify the kind of hardware buffers used to realize arrays in hardware - such as block RAMs, ultra RAMs and LUT RAMs.

2.4.1 Limitations of dataflow optimization

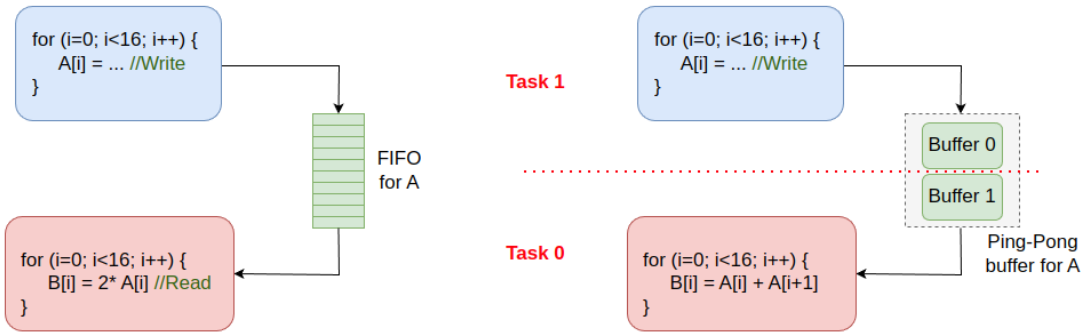
Vitis HLS offers support for both intra-loop pipelining, as well as pipelining across producer-consumer loops. However, the dataflow optimization feature of Vitis HLS for

pipelining the execution of producer-consumer loops within a single function invocation has certain limitations:

- The consumer must read the data in the same order in which the producer generates it.
- **Single-producer-single-consumer (SPSC)** The intermediate arrays used to communicate between the producer and consumer tasks must have only one producer (i.e., only one task can write to it) and one consumer (only one task can read from it).
- The intermediate arrays must be instantiated in the function itself. Array arguments to a function can not be used to write the intermediate values.

In Figure 2.4a, the single-producer-single-consumer (SPSC) constraint is satisfied since the *intermediate array*, *convX*, is produced by the first convolution and consumed by the second. However, the second convolution does not read the *convX* array in the same order in which the first convolution writes to it. In fact, as the consumer task is a stencil operation, the number of reads on *convX* is more than the number of writes. Due to the abundance of stencil operations in both image processing and machine learning applications, this is a very common scenario.

To understand the reason behind the above limitations *dataflow* optimization, we first need to discuss how Vitis HLS implements it. For each producer-consumer task pair, Vitis HLS *dataflow* optimization checks the memory accesses patterns of the intermediate array. If the elements of the intermediate array are read by the consumer in the order in which they are written by the producer, then Vitis HLS can replace the array with a FIFO without altering the program's behavior. The FIFO ensures that the consumer gets the data in the same order in which the producer is generating it. Now, Vitis HLS can start the execution of the consumer loop along with the producer loop without having to worry about data dependence violation. If the producer has not generated and pushed the next value to the FIFO, the consumer automatically stalls while attempting to read that value from the empty FIFO. Thus the read-after-write dependence between



(a) FIFO (b) Ping-Pong buffer

Figure 2.5: Dataflow optimization of Vitis HLS - *Vitis HLS adds (a) FIFO to pipeline/overlap within a single function call (improves latency) if the producer and the consumer read the data in the same order. (b) Otherwise, it adds a ping-pong buffer for pipelining between consecutive function calls (improves throughput).*

the producer and consumer tasks in the original program is enforced at runtime by the FIFO's synchronization logic. If, in the original program, the order of reads and writes to the intermediate buffer do not match, then Vitis HLS can not replace the intermediate array with a FIFO. In such cases, it instantiates a ping-pong buffer to replace the intermediate array. This optimization helps in pipelining between multiple invocations of the function but does not pipeline between the producer and consumer tasks within a single function invocation. Thus, in the absence of the same read and write order, the dataflow optimization does not improve the overall performance/latency of a single function invocation.

When multiple consumer loops read from the same intermediate array, each consumer loop has to wait unless all other consumers have read the current data before reading the next value. As a result, using a single FIFO with multiple consumers may lead to unexpected stalls, and even deadlock in the presence of additional data dependencies between the consumers. Vitis HLS does not duplicate the FIFOs for multiple consumers. This is probably to ensure that the BRAM usage does not explode after the dataflow optimization. The FIFO-based implementation can not handle multiple producers either since at runtime, two producers running in parallel will insert the data in arbitrary order,

and the consumer will not be able to read the data in the expected order. Due to the above-mentioned limitations of the FIFO-based approach, *dataflow* optimization in Vitis HLS is limited to single-producer-single-consumer (SPSC) workloads.

Similarly, if the intermediate array was accessed via function argument, the optimizer would not have access to the array and would not be able to replace it with a FIFO, leading to the third constraint. This leads to the two hard constraints for dataflow optimization.

Based on the above discussion, we can identify two key attributes of the dataflow optimization,

- Vitis HLS performs a very basic static analysis just to check if the producer and consumer access the intermediate array in the same order. Its static analysis does not handle more complex data access patterns.

- The dataflow optimization relies on runtime synchronization to enforce read-after-write dependencies between producer and consumer tasks.

While the lack of better static analysis leads to missed parallelization opportunities for Vitis HLS in the presence of complex memory access patterns, enforcing memory dependence using runtime synchronization can lead to more resource usage. We quantify the effect of these design decisions on the performance and resource usage in Chapter 5.

2.5 Summary

In this chapter, we discussed the necessary background to understand the high-level synthesis compiler landscape. We gave a brief overview of the FPGA internals and discussed the problems associated with designing FPGA accelerators using traditional hardware description language. We also discussed the MLIR compiler infrastructure which we use for developing our own HLS compiler. Finally, we discussed the Vitis HLS compiler and its dataflow optimizations.

Chapter 3

HIR Intermediate Representation

An intermediate representation (IR) helps in decoupling the different stages of a compiler pipeline. We need an IR for our compiler that can capture the important properties of a hardware design. A hardware design has three major components, the algorithm, the schedule of computation and the resource binding. Among these, the schedule and the resource binding are unique to a hardware design. The schedule of a computation specifies the time at which each computation is executed. It captures the parallelism in the design and ensures that data dependencies are not violated. The IR must capture a precise schedule after scheduling to synthesize the state machines that control the data path in the hardware design. An FPGA has many different resources such as multipliers, multi-ported block RAMs and registers. For the backend to generate the hardware design, the IR needs to capture the resource binding information, i.e. which resources are used for which operations? For instance, a tensor may be implemented using a single or dual-ported RAM. Similarly, two multiplications scheduled at different clock cycles can reuse the same multiplier.

A single intermediate representation is not suitable at every stage of a compiler pipeline. Transformations at different stages of the compiler pipeline require different levels of abstraction. For instance, certain transformations (such as operator strength reduction) may occur before the scheduling pass. These do not require the scheduling information. Other optimizations such as resource sharing will require a precise schedule

to identify which resources can be reused between two computations. We introduce a new IR called HIR to capture scheduled design with resource-binding information.

In Section 3.1, we discuss the design of the HIR intermediate language in detail. We also discuss various types of undefined behavior in our compiler. We discuss the strengths of HIR in Section . Section 3.3 explains how different types of parallelizations can be captured in HIR. In Section 3.4.2, we discuss the different optimizations applied to an HIR design. Section 3.4.3 discusses the schedule verification pass. Finally, we explain the details of the compiler backend that lowers HIR to SystemVerilog in Section 3.4.4.

3.1 HIR Design

The HIR intermediate representation is implemented as a dialect in the MLIR [Lattner et al. (2021)] compiler infrastructure. As such, it inherits all the usual benefits provided by the core MLIR infrastructure, such as a human-readable textual representation that could be parsed, printed, and verified [Lattner et al. (2020)]. All the 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 software IR. It has multi-dimensional arrays and high-level control flow operations such as *for/while* loops and *if-else* statements. HIR enhances a software-style IR in two important ways. The first is an abstraction of memory that is suitable for hardware. HIR supports memory banking (mapping a logical memory to multiple physical memories for parallel access) and multi-port RAMs. The second feature that distinguishes it from a software IR is that all HIR designs contain an explicit schedule of all computations. The fine-grained 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

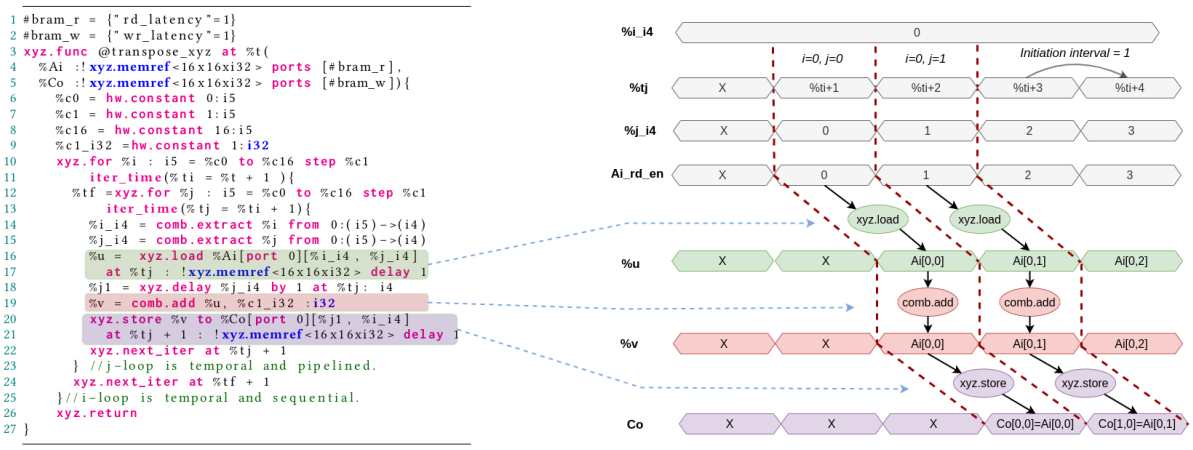


Figure 3.1: Matrix increment and transpose.

the canonicalization and constant folding function hooks of these operations.

3.1.1 Time variable and schedules

The HIR compiler generates a hardware design with a single clock domain. All operations in the hardware design are synchronized with the positive edge of the 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 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 *dynamic instance* of an operation is one execution of the operation. A specific operation may execute multiple times during the execution of the kernel. 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 3.1 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. Similarly, multiple calls to 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 3.1, the *transpose* function defines its start time as $\%t$ and the i-loop defines the iteration start time with the time variable $\%ti$. 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.

A time variable can be defined in two ways. Each region defines its own start time variable. In Figure 3.1, 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 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 3.1 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 3.1 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 the 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 3.1, 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.

The *hir.delay* operation adds a delay of specified clock cycles to the input. As shown in line number 18 of Figure 3.1, 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 $\%tj + 1$.

3.1.2 Functions

```
// Function signature captures input and output delays relative
// to the start time.
hir.func @multiplyAccumulate at %t
  (%a :i32, %b: i32, %c:i32 delay 3) -> (i32 delay 4){
    //Func body. Time variable %t is available inside the body.
}

// RAM port parameters.
#bram_rw = {rd_latency = 1 : i64, wr_latency = 1 : i64}
#bram_rd = {rd_latency = 1 : i64}

// Memref args have port parameters to synthesize the right buses.
hir.func @conv at %t (%img : !hir.memref<16x16xf32> ports [#bram_rw],
                      %kernel : !hir.memref<3x3xf32> ports [#bram_rd]{
    //Func body. Time variable %t is available inside the body.
}
```

Figure 3.2: HIR function definition.

Function signatures capture the time at which the input values (of integer and float type) are expected by this function (callee) and the time at which the output values are available to the caller. Figure 3.2 shows the function definition of a multiply-accumulate

operation. The function signature implies that the function starts executing at time instant $\%t$. The inputs $\%a$ and $\%b$ are read when the function starts. Input $\%c$ is read after a delay of three clock cycles i.e., at time $\%t + 3$. The function returns the output after four clock cycles (at $\%t + 4$). Capturing the schedule of function arguments and results enables us to verify the schedule of the complete design without performing an inter-procedural analysis. Analogous to how a function's signature captures sufficient type information for the caller to perform type-checking without inspecting the callee's body, capturing the schedule allows the caller to verify the schedule of operations in its body without analyzing the callee's body. Memory (*memref* type) arguments specify a list of ports as shown in the *conv* function's signature. The port parameter (captured in the attributes *bram_rw* and *bram_rd*) specifies the delay in clock cycles associated with each operation. Each port can be a read-only port (ex: *bram_rd*), a write-only port or a read-write port (ex: *bram_rw*). Function arguments of *memref* type do not have a delay associated with them because on-chip memory buffers (such as BRAM) are read in multiple clock cycles whereas values can be read in one clock cycle. Thus, we can not associate a single time instant with them.

3.1.3 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 3.3 shows the syntax of control flow operations in HIR. The *for* loop takes a lower bound, an upper bound and a step size. The *iter_time* syntax specifies that the loop starts at the time $\%t + 1$. The *next_iter* operation determines the initiation interval of the loop by specifying the start time of the next iteration relative to the current iteration. The *for* loops in HIR can be either temporal loops (multiple iterations on the same hardware at different times) or spatial loops (loop body unrolled). Since *index* type represents compile time constant integers in HIR, spatial loops are represented by specifying the loop induction variable as *index* type.

The *if* statement takes two input variables. The boolean condition variable ($\%cond$)

```
//%t_end: time at which last iteration calls next_iter.
%t_end = hir.for %i : i32 = %lb  to %ub step %c1_i32
    iter_time(%ti = %t + 1) {

    // The first iteration starts at time %ti=%t + 1,
    // Initiation interval is 5 clock cyles.
    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 3.3: Control flow ops in HIR.

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.

3.1.4 Types

HIR supports primitive data types such as arbitrary bit-width integers and single and double-precision floating-point types. It uses *index* type to represent constant integers and *hir.time* type to represent the type of time variables. In addition to these simple datatypes, HIR defines a container type to represent multidimensional memories in hardware.

All the available memory resources in hardware are represented via the *hir.memref* data type. 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 away and provides a uniform interface for memory access.

Figure 3.4 shows two *alloca* operations that allocate new memory elements (block

```
#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>
hir.store %v to %bram_1p[port 0][%i, %j] at %t + 1
      : !hir.memref<8x8xi32>
```

Figure 3.4: HIR memref type.

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. If a port attribute specifies both read and write latencies then it is treated as a read/write port with a common address bus. Each port has a dedicated address bus. This approach of representing memory ports gives HIR the flexibility to instantiate memories with an 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.

The *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. This ensures that the compiler knows which specific bank is accessed. We discuss the advantages of memory banking in Section 3.3.4. Only one *load/store* operation can be scheduled every cycle for a given port of a bank.

3.1.5 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. We use this to simplify the *for-loop* state machine. Also, our scheduler can only handle constant loop bounds, which makes this condition always true.

- A new instance of a *for-loop* is not scheduled unless the previous instance has completed all iterations. Each instance of a loop requires its loop counter. Since only one counter is instantiated per loop, the hardware can not execute multiple loop instances in parallel. Note that multiple in-flight iterations of a single loop instance, i.e. loop-pipelining, is a perfectly valid optimization.

- There will never be multiple accesses to a *memref* in the same clock cycle unless they occur in different banks or different ports. A memory port has only one address bus. Thus, it can only handle one memory request every cycle. This constraint ensures that there will never be a port conflict in the hardware.

- 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 an uninitialized state. The HIR language does not support persistent state (equivalent to *static* variables in C) across function calls. Uninitialized memory may have any values, including values from the previous run of the function. We do not zero-initialize buffers on every function call since that would lead to multi-cycle delay overhead for each call.

Thus, a program that reads from uninitialized memory may execute in unexpected ways.

- The timing behavior of external Verilog modules captured in a forward function declaration must match with the implementation. There is no way for the compiler to analyze external Verilog modules. Thus, the compiler relies on the timing information in the function signature (such as when the inputs are expected by the callee and when the output should be ready) to generate correct schedules.

Violation of any of the above assumptions is treated as undefined behavior. optimizers can exploit the undefined behavior to implement more aggressive optimizations that do not violate the semantics. Section 3.4.2 shows one such optimization pass exploits undefined behavior to reduce bitwidth of loop induction variables. All of these undefined behaviors, except the undefined behavior related to external Verilog modules, can be checked during simulation. Our compiler optionally (enabled by a compilation flag) adds extra assertions to check undefined behavior during simulation. This is equivalent to undefined-behavior sanitizers available for languages like C++. These checks improve the functional verification of the final design. We can only perform these checks during the simulation as these assertions are not synthesizable.

3.2 Strengths of HIR Intermediate Language

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

3.2.1 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.

3.2.2 Explicit Scheduling

Previous work Durst et al. (2020) 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 code generation after automatic scheduling. The scheduling pass does not have to create FSM logic to implement the parallel schedule.

3.2.3 Deterministic Parallelism

Many HLS languages [Nikhil (2004)] and latency insensitive IRs [Nigam et al. (2021)] borrow non-deterministic parallelism from software programming languages. This kind of parallelism often requires a synchronization mechanism. For example, in Vivado 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 [Schuiki et al. (2020)] can express this kind of deterministic, synchronization-free, task-level parallelism. Section 3.3.3 shows an example of synchronization-free task-level parallelism.

3.2.4 External Hardware Modules

The ability to use externally defined hardware circuits is essential to specialize a design for the given hardware platform. FPGA vendors provide custom libraries for many common circuits such as floating-point arithmetic and multi-ported RAMs. Additionally, several third-party libraries 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.

3.2.5 Predictable QoR

A predictable quality of result (QoR) is essential for a hardware intermediate representation. This allows DSL/HLS compilers to generate hardware with a predictable performance and resource usage. Control over resource usage is also important when the generated accelerator has to share the FPGA with other Verilog IPs such as PCIe controllers, DRAM controllers and soft CPU cores. An HIR design contains a description of the precise schedule of all operations. Additionally, all resources are explicitly instantiated in the design. Together, these ensure that the performance (amount of parallelism) and resource usage are predictable.

3.3 Expressing Different Types Of Parallelism In HIR

HLS compilers are expected to exploit domain knowledge to find potential optimization opportunities [Hegarty et al. (2014)]. 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 various standard hardware optimization techniques and how they can be expressed in HIR.

```
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 3.5: Overlapped execution of tasks.

3.3.1 Instruction Level Parallelism

The HIR IR can capture fine-grained instruction-level parallelism. For instance, multiple independent operations can be scheduled at 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 at the same cycle 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.

3.3.2 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 3.3.

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

3.3.3 Task Level Parallelism

In addition to exploiting parallelism in loops, multiple tasks can be executed in parallel to further improve performance. The "task_parallel" function in Figure 3.5 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 lock steps i.e. in each cycle

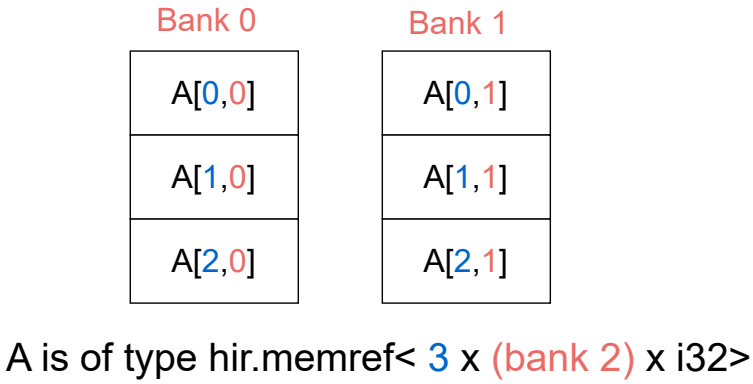


Figure 3.6: Memory banking in a memref type.

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 loops run in lockstep.

3.3.4 Memory Banking And Multi-Port RAMs

Hardware accelerators use on-chip buffers to reduce DRAM accesses. To execute operations in parallel, these designs may perform multiple read and write operations at every cycle. FPGAs offer multi-ported on-chip RAMs to support parallel access. An HIR design can instantiate multi-ported RAMs for parallel 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 3.6 shows how elements of a banked *memref* are spread across multiple buffers.

3.4 The compiler pipeline

Figure 3.7 depicts the comprehensive compiler pipeline based on MLIR. We utilize the Polygeist Moses et al. (2021) C/C++ frontend to lower C programs to the affine dialect.

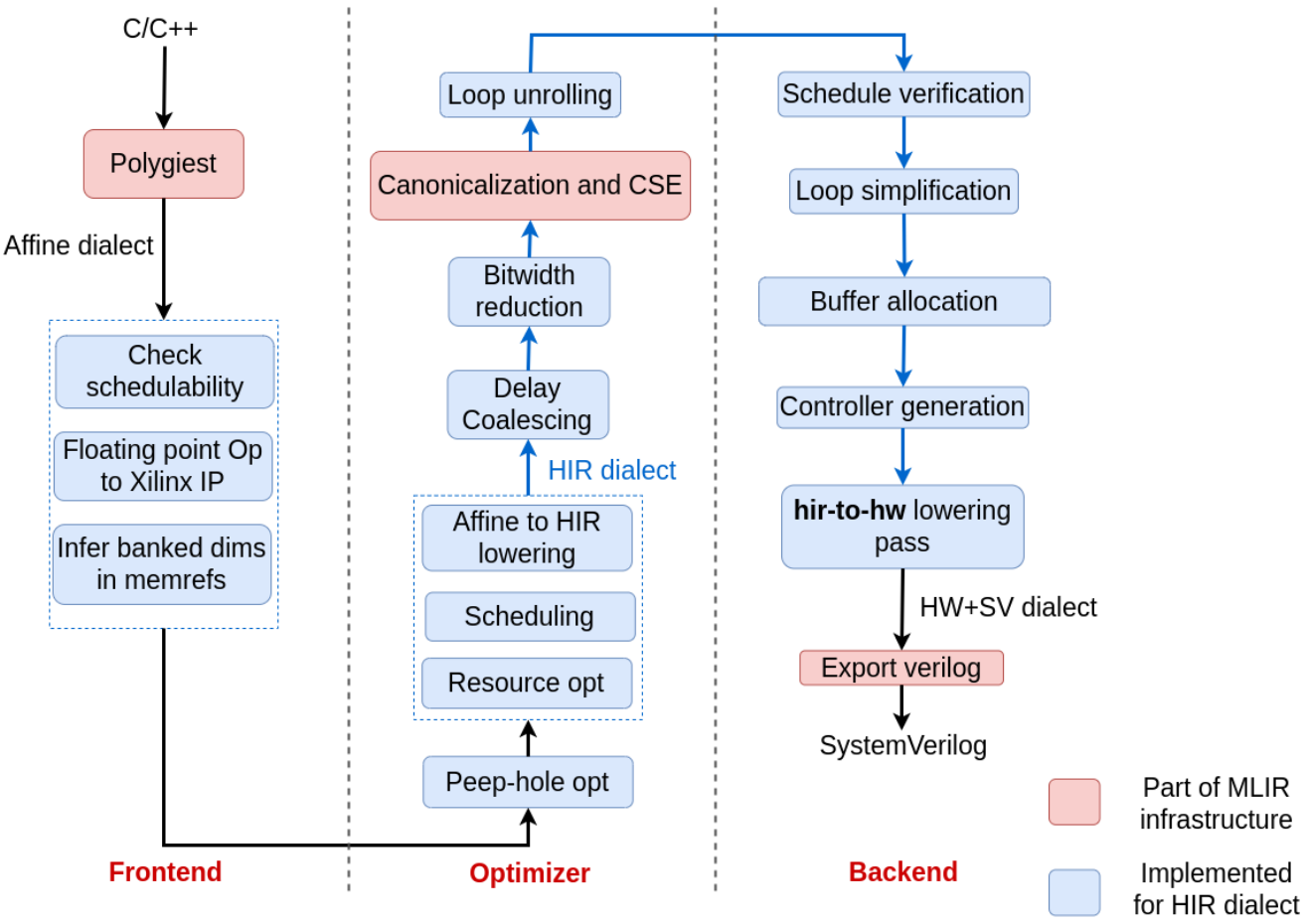


Figure 3.7: HLS compilation flow.

The programmer specifies the target pipelining and hardware resources using pragmas in the C program. In case the programmer does not provide explicit specifications, we have implemented a basic auto-tuner to search for different loop pipelining options. Additionally, we have implemented an affine-to-HIR pass to lower the program to the HIR dialect. The HIR backend is then used to optimize the design, such as bit-width optimization, and generate SystemVerilog.

3.4.1 Polygeist frontend

We utilize the *C* language as the input programming language for our compiler toolchain. The Polygeist Moses et al. (2021) frontend is employed to lower *C* programs into MLIR. We have extended the Polygeist compiler to include support for HLS-related pragmas, akin to those found in Vitis HLS. We support the following pragmas in our compiler:

- **pipeline** pragma is used to specify the loop initiation interval.
- **unroll** pragma is used to specify whether a loop is unrolled or not. We only support complete unrolling at present. Partial unrolling can be achieved by manually strip-mining the loop and unrolling the resulting inner loop completely.
- **bind_storage** pragma is used to specify the type of hardware buffer such as block RAM or LUT used to implement an array and the number of ports.
- **array_partition** pragma is used to specify memory banking along a specific dimension of a multidimensional array. We only support complete partitioning of a dimension. Block and cyclic partitioning are not supported yet.
- **interface** pragma defines the type of interface used to implement a function argument. We use this pragma with array arguments to specify the number of memory ports and latency of the read/write operation.
- **bind_op** pragma is used to specify the latency of an arithmetic operation. This information is required by the scheduler later to calculate a valid schedule. This

allows us to use external FPGA vendor-specific IPs for efficient implementation of floating point arithmetic.

- **extern_func** pragma specifies the latency of external functions. We can use this pragma to call external Verilog modules. Currently, we only support external modules with one output and a constant latency.

We use the one-dimensional convolution kernel shown in Figure 3.8 as our running example to explain the compiler pipeline.

The *bind_op* pragma specifies the latency of the floating-point addition (five clock cycles) and multiplication (four clock cycles) operations. The *interface* pragma specifies the type of memory interface that will be generated for the function arguments. The *pipeline* pragma specifies the pipelining initiation interval (seven cycles) for the j-loop. The *scop* pragma instructs the Polygeist frontend to generate Affine dialect operations for the loops and load/store operations inside the scop.

The Polygeist frontend generates programs in the affine dialect, with pragma information preserved as attributes. Following that, a preprocessing pass is applied to convert all floating-point operations into function calls, utilizing information from the bind_op pragma. It also inserts appropriate function declarations into the affine program. The implementations for these function declarations are provided by external Verilog floating point modules. The frontend also inlines function calls for which definitions are available. The resulting output of the preprocessing for the one-dimensional convolution is depicted in Figure 3.9.

Next, we proceed with scheduling, aiming to identify a schedule that satisfies the programmer-specified initiation intervals (II) without altering the semantics of the input sequential affine program. The autotuner employs a simple binary search technique to determine the optimal II for each loop that lacks a programmer-specified II. For each target II, the scheduler is executed to verify if the design can be feasibly scheduled. The outcome of this step is a parallel program.

The affine-to-HIR pass transforms the affine program to HIR using the scheduling information, and the *binding* pragma attributes which determine the type of hardware

```
#pragma HLS bind_op op = add_f32 latency = 5
#pragma HLS bind_op op = mul_f32 latency = 4

void conv(float outp[16], float inp[16], float wt[2]){
#pragma HLS interface port= outp storage_type=ram_1p \
                      rd_latency=1 wr_latency=1
#pragma HLS interface port= inp storage_type= ram_1p rd_latency = 1
#pragma HLS interface port= wt storage_type= ram_1p rd_latency = 1

#pragma scop
  for (int i = 0; i < 16; i++) {
    for (int j = 0; j < 2; j++) {
#pragma HLS pipeline II = 7
      outp[i] = outp[i] + inp[i + j] * wt[j];
    }
  }
#pragma endscop
}
```

Figure 3.8: One-dimensional convolution kernel with HLS specific pragmas.

buffers to be used for each multi-dimensional array. Figure 3.10 illustrates the scheduled HIR design for the one-dimensional convolution. The initiation interval of this design cannot be reduced below seven clock cycles due to a loop carried dependence between the *store* and the *load* operations on $\%arg0$. Within the same iteration of the j-loop, there is a six-cycle gap between a *load* and a *store* on $\%arg0$, resulting from a one-cycle *load* delay and a five-cycle delay of the floating point *add* operation. To maintain the read-after-write dependence between the *load* and *store* operations, the next iteration's *load* cannot be scheduled before the current iteration's *store* is completed, which requires an additional clock cycle. As a result, the next iteration can only commence after a delay of seven clock cycles. For the sake of brevity, certain operations such as delay and casting

```
func.func private @mul_f32(f32, f32)
  -> (f32 {hir.delay = 4})
func.func private @add_f32(f32, f32)
  -> (f32 {hir.delay = 5})
func.func @conv(%arg0: memref<16xf32>
        {hir.memref.ports = [{rd_latency = 1 : i64, wr_latency = 1 : i64}]},
        %arg1: memref<16xf32>{hir.memref.ports = [{rd_latency = 1 : i64, wr_latency = 1
            : i64}]},
        %arg2: memref<2xf32>{hir.memref.ports = [{rd_latency = 1 : i64, wr_latency = 1 :
             i64}]}){
  affine.for %arg3 = 0 to 16 {
    affine.for %arg4 = 0 to 2 {
      %0 = affine.load %arg0[%arg3]
      %1 = affine.load %arg1[%arg3 + %arg4]
      %2 = affine.load %arg2[%arg4]
      %3 = func.call @mul_f32(%1, %2)
      %4 = func.call @add_f32(%0, %3)
      affine.store %4, %arg0[%arg3]
    } {II = 7 : i64}
  } {II = 14 : i64}
  return
}
```

Figure 3.9: After preprocessing.

```
hir.func.extern @mul_f32 at %arg2 (%arg0 : i32, %arg1 : i32) -> (%out : i32 delay 4)
hir.func @conv at %arg3 (%arg0 : !hir.memref<16xi32>
                          ports [{rd_latency = 1 : i64, wr_latency = 1 : i64}],
                         %arg1: ..., %arg2: ...){
  %t_end = hir.for %i = %c0 to %c16 step %c1 iter_time( %arg5 = %arg3){
    %res, %t_end_0 = hir.for %j = %c0 to %c2 step %c1 iter_time( %arg8 = %arg5){
      %4 = hir.load %arg0[port 0] [%i] at %arg8+4
      %s = comb.add %i, %j : i4
      %8 = hir.load %arg1[port 0] [%s] at %arg8
      %10 = hir.load %arg2[port 0] [%j] at %arg8
      %12 = hir.call @mul_f32(%8, %10) at %arg8+1
      %sum= hir.call @add_f32(%4, %12) at %arg8+5
      hir.store %sum to %arg0[port 0] [%i] at %arg8 + 10
      hir.next_iter at %arg8 + 7 : (i64)
    }
    hir.next_iter at %arg5+14
  }
  hir.return
}
```

Figure 3.10: After scheduling and lowering to HIR.

operations are omitted in Figure 3.10.

3.4.2 Optimizer

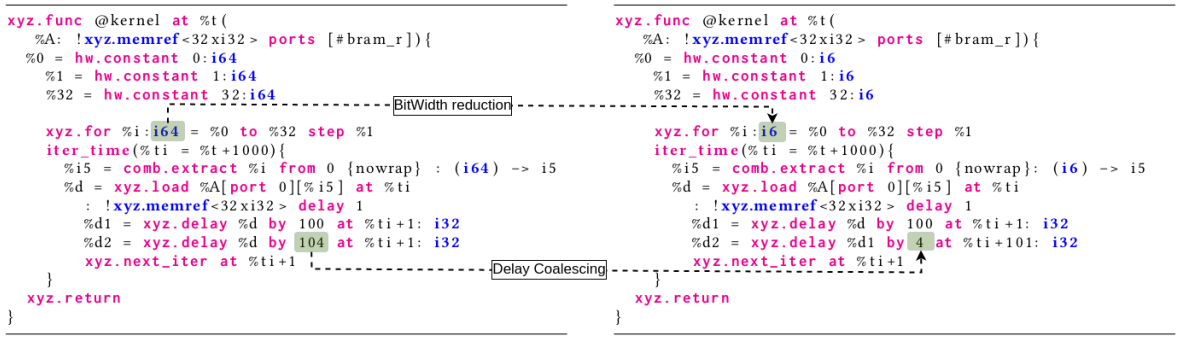
The optimizer lowers the affine IR into HIR and optimizes the generated HIR IR. All programs in the HIR dialect have an explicitly specified schedule which may capture different types of parallelism in the design, but the affine dialect assumes sequential execution. To lower the Affine dialect to HIR,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

Peep-hole optimization Currently our compiler supports one peep-hole optimization. Before scheduling, we look for multiply-accumulate patterns and replace them with a single function call to an external multiply-accumulate Verilog module. Vivado offers IP for fused-multiply-add (FMA) operation which consume fewer resources. The FMA IP from Vivado provides the implementation of the external Verilog module for

the multiply-accumulate operation.

Auto-Scheduling Our compiler uses an ILP-based automatic scheduler 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 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 at least 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 initiation intervals. The affine-to-HIR lowering pass uses the schedule to lower from the *Affine* dialect to the HIR dialect. 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



(a) No optimization.

(b) Delay coalescing and Bitwidth reduction.

Figure 3.11: Optimizations after the automatic scheduling.

an *hir.delay* operation would be inserted to ensure that the output of the *call* operation is available when the *load* operation executes. We discuss our ILP-based automatic scheduler in greater detail in Chapter 4.

Delay Coalescing: The affine-to-HIR lowering pass may insert a lot of delay operations to eliminate pipeline imbalance. These delays are implemented using shift registers in hardware that consumes extra registers and lookup tables. We implement a delay coalescing pass to reduce the depth of the shift registers. Figure 3.11 shows an example program before and after optimization. In this code, the two delay ops 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.

In addition to this, we can also reduce the number of shift registers required to implement a delay operation. If the loop initiation interval of the *i-loop* in Figure 3.11(b) was two instead of one, we could implement the second *hir.delay* operation using only one shift register. Instead of shifting every cycle, the shift register would shift values every two cycles. To the shift register, it is as if the hardware clock is ticking at half the frequency. Thus it can realize a four-cycle delay of the original clock by using just two shift registers. Note that the output of the shift register now holds each value for two clock cycles instead of one cycle. In the general case, if a delay operation with delay *d*

is inside a region with an initiation interval of *II*, the number of shift registers required to realize the delay can be calculated using the formula:

$$num\_shift\_registers = \left\lceil \frac{d}{II} \right\rceil$$

This optimization further reduces the required number of shift registers in the final design.

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 uses 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 3.11 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 i variable because its only use is to downcast it to a 5-bit integer and the *nowrap* attribute guarantees that there will be no integer wrapping during the cast (else its undefined behavior). This is another example of exploiting undefined behavior to enable new optimizations. In addition, the compiler reuses the constant folding, constant subexpression elimination and canonicalization passes from the MLIR infrastructure. 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 custom scheduler (such as DarkroomHegarty et al. (2014)) directly lowers to HIR dialect, it will have 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 dialects 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

schedulers can benefit from the optimization.

Loop unrolling: Hardware resources can be replicated to increase the amount of available parallelism in a design. Loop unrolling replicates the hardware blocks associated with the loop body so that loop iterations can execute in parallel. The loop unrolling pass unrolls a loop completely. We do not support partial unrolling. Partial unrolling may be realized before lowering to HIR by strip-mining a loop and fully unrolling the resulting inner loop of a smaller trip count. We perform loop unrolling after the scheduling pass to reduce the number of ILP constraints and variables. Unrolling before scheduling (for example, unrolling loops in the affine dialect itself) may lead to duplication of ILP constraints and variables related to the loop body.

3.4.3 Schedule verification

Verification is a key aspect of hardware design. For an IR designed to enable LLVM-like open compiler infrastructure, which can be targeted by many HLS/DSL frontends of different levels of maturity, it is important to perform verification on the generated IR.

One major source of error in hardware design is the wrong scheduling of operations. A schedule is invalid if the design may read invalid data. For example, reading an output of an operation before the operation completes or reading from a memory location that is not initialized yet.

Figure 3.12 illustrates this with an example design of a multiply-accumulate operation where a two-stage integer multiplier is replaced (commented out in the code) with a multiplier that has three pipeline stages. This kind of optimization may be required in the final design to meet the frequency requirements. For the adder to work properly, both its inputs must arrive at the same clock cycle. The added pipeline stage in the multiplier delays $\%m$ by one cycle leading to a malfunctioning design. Since function signatures in HIR embed the delays of each input and output value w.r.t a start time, the schedule verification pass can calculate the time instant (relative to the start time of the function) when the SSA vars have valid values, and use this information to detect

```
// Two stage multiplier
//hir.func.extern @mult_2stage at %t
//   (%a:i32, %b:i32)->(%result:i32 delay 2)

// Three stage multiplier
hir.func.extern @mult_3stage at %t
   (%a:i32, %b:i32)->(%result:i32 delay 3)
hir.func @mac at %t
   (%a :i32, %b :i32, %c :i32) -> (%result: i32 delay 3){

  //Previous 2-stage multiplier.
  //%m = hir.call @mult_2stage (%a,%b) at %t
  //       : !hir.func<(i32, i32) -> (i32 delay 2)>

  //Replaced with a 3-stage multiplier.
  %m = hir.call @mult_3stage (%a,%b) at %t
        : !hir.func<(i32, i32) -> (i32 delay 3)>

  %c2= hir.delay %c by 2 at %t : i32

  %res = comb.add %m, %c2 : i32
  %res1 = hir.delay %res by 1 at %t + 2 : i32

  hir.return (%res1) : (i32)
}
```

Figure 3.12: Example of a pipeline imbalance.

pipeline mismatches.

As a part of our compiler, we implemented a schedule verification pass. Our verifier checks the 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 addresses.

As described in Section 3.1.1, HIR has a notion of validity associated with data. The operations that generate an SSA variable of primitive type (*int* and *float*), specify the time instant (relative to a time variable) at which they have a valid value. The schedule verification pass checks the uses of these variables to verify that the use site expects the value at the same time when the data is produced. Next, we discuss the algorithm for

verifying the schedules. Our algorithm can be divided into two parts. First, we calculate the time associated with each variable 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 3.2 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. The 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, the second phase of the algorithm verifies the schedule as follows:

- For *hir.store* and *hir.delay* operations, check that the input SSA variable should be valid at the operation's 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.

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 auto-scheduler or any other optimization pass.

3.4.4 Backend

The backend of our compiler converts the verified HIR program to 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 library to instantiate Xilinx FPGA block RAMs. HIR's ability to interface with arbitrary external Verilog modules allows us to use both floating point and memory IPs provided by the FPGA vendor (Xilinx). One key difference between HIR IR 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 (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 IR 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 3.4.2. 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 were already converted into calls to external Verilog functions during the frontend preprocessing pass as discussed in Section 3.4.1. All function calls are converted into Verilog module instantiations. Once all the operations in the function body are converted into a combination of SV, Comb and HW dialect, the enclosing HIR function is converted into *hw.module*(Verilog module). The SV, Comb and HW dialects are a part of the larger MLIR infrastructure and not a contribution of this work. The export verilog pass generates SystemVerilog from the SV+Comb+HW dialect.

3.5 Summary

In this chapter, we introduced HIR, an intermediate representation to describe FPGA-based hardware accelerator designs. The IR captures computation schedules explicitly,

which makes it an ideal target IR for automatic scheduler passes. We discuss the various high-level language constructs such as for-loops, if-else conditions and multidimensional tensors to capture the design. We also discuss the notion of time variables to specify precise schedules in the IR. We discuss how the *memref* captures the type and port information of the underlying hardware buffer. We show that the IR can capture instruction-level parallelism, loop pipelining and task-level parallelism using the time variables. We also discussed various optimizations that are performed on the IR such as bitwidth reduction and delay coalescing, to improve its resource usage. We implemented a pass to capture pipeline imbalance in the hardware design and a backend to convert HIR to SystemVerlog. In the overall compiler flow, we use HIR for post-scheduling optimizations and the backend code generator.

Chapter 4

ILP-Based Automatic Scheduling

In this chapter, we will discuss the scheduling technique implemented in the HIR compiler. Scheduling is the process of assigning a time instant at which a dynamic instance of an operation would execute. We formulate the scheduling problem as an integer linear program (ILP). The scheduling pass in our compiler analyzes the unscheduled Affine program and generates the ILP constraints. The scheduler then uses an ILP solver to find a valid parallel schedule. We use GLPK as the ILP solver and are currently migrating to Google OR tools for better integration with the rest of the CIRCT infrastructure.

The scheduler's task is to calculate these time instants. In hardware, a memory buffer can have multiple ports for read and write. Some of these ports may be read-only or write-only and others may allow both read and write. Since each port has dedicated address and data buses, multiple load/store operations can be performed using multiple ports on the same memory in the same clock cycle. But a single port can not serve more than one read/write request per cycle. The scheduler also has to ensure that it does not schedule two memory operations on the same port in the same clock cycle. This makes the port assignment problem (which load/store operation uses which port of the memory) entangled with the scheduling problem. Thus the scheduler also calculates the port assignments in addition to the schedule.

We next discuss the correctness criteria for a schedule and the optimization opportunities available to the scheduler. We provide an ILP formulation to calculate both the

schedule and the port assignments.

4.1 Correctness Criteria

For the final design to execute correctly, the schedule must be valid. The input to the scheduler is a kernel in affine dialect. The semantics of this kernel is that each operation executes sequentially. We call this a sequential schedule for the kernel. The scheduler has to find a parallel schedule that would not change the observable behavior of the kernel. The parallel schedule generated by the scheduler can not change the order of dependent operations i.e. if operation Y depends on the results of operation X, then operation Y must be scheduled after operation X has calculated the result. The data communication between two operations can occur in two ways - using SSA variables of simple value types (such as int or float) or using memory elements. Figure 4.1 shows examples of both these types of dependencies. Statement *S1* is dependent on *S0* because of the variable *c_prev*. There is a dependence from statement *S2* of one loop iteration to *S0* of the next iteration because of the array *C*. A valid schedule would ensure that within a loop, an instance of *S1* occur after *S0* and in case of loop pipelining, the load *S0* of the next iteration does not execute until the store *S2* is complete. The scheduler's analysis pass must find all data dependencies between operations. It can be conservative, i.e. falsely report a dependency when there is no actual data dependence between the operations. With a more accurate analysis (less falsely reported dependencies), the scheduler can generate more efficient schedules. For instance, a simple analysis pass could just look at the array variables and ignore the indices to decide if a load and a store operation have a dependence between them. This would capture all the dependencies but may also falsely report a dependence. For instance, if statement *S0* and *S1* were accessing *C[2i][j]* and *C[2i+1][j]*, then there is no actual data dependence between them, but our analysis would have reported a dependence.

Another criterion for correctness is to ensure that there are enough resources to perform the operations scheduled at the same cycle. In HIR, it is undefined behavior for

two memory operations to access the same memory port in the same clock cycle because it can not be realized in hardware. The scheduler also calculates the *port assignment* for each load and store operation to ensure that there are no *port conflicts*.

4.2 Scheduling Objective

Multiple schedules can satisfy the correctness criteria. The original sequential schedule of the affine kernel is trivially correct - the data dependencies are correct by definition and since no two operations occur in the same cycle, there are no port conflicts. The scheduler's task is to find a parallel schedule that can complete the kernel in a lesser number of clock cycles. As long as the parallel schedule does not violate any of the correctness criteria, it would produce the same result as the original sequential schedule.

There are different types of parallelization opportunities in a hardware kernel. The most basic optimization is to pipeline the innermost loops. This means that successive iterations of the loop can start before the previous iteration has completed all its operations. The *initiation interval* is the number of clock cycles between the start of two successive loop iterations. Pipelining the loop improves parallelism by overlapping the execution of successive iterations. We can also pipeline outer loops. In case of time-iterated loops, pipelining outer loops only helps if the inner loop's trip count is small. In such a case, the filling and draining overhead of the inner loop pipeline may become significant if the outer loop is not pipelined. There is another advantage of pipelining the outer loop. If the inner loop is supposed to be fully unrolled, then the outer loop has to be pipelined to take advantage of the parallel hardware generated by the unrolled inner loop. There are two approaches to achieving outer loop pipelining when the inner loop is to be completely unrolled. Either the inner loop should be unrolled first and then the scheduling is performed for pipelining the outer loop (which is the inner loop now) or the scheduler can perform outer loop pipelining. We take the second approach because the number of constraints in our scheduling ILP is proportional to the number of operations in the body of the loop. Unrolling-then-pipelining increases the number of

constraints in the ILP solver making it slower. Thus, our scheduler performs pipelining at all the loop levels before unrolling the loops.

4.3 ILP Formulation

In this section, we describe our ILP formulation of the scheduling problem. Our scheduler lowers programs in *affine* dialect to *HIR* dialect. The original program in the affine dialect has a sequential schedule. To utilize the hardware resources effectively, we attempt to create a parallel schedule while lowering from affine to HIR dialect. Loop attributes in the affine dialect specify the target initiation intervals. The scheduler's job is to find the start time of each operation in the generated HIR dialect such that the target initiation intervals are achieved, execution of loop nests are maximally overlapped for better parallelism and the memory dependencies are not violated, to ensure that the semantics of the original sequential program is preserved after the automatic parallelization.

We represent the problem of finding a valid schedule as a set of ILP formulations. We first solve a smaller *memory dependence ILP* for each conflicting memory access (load/store operations on the same array). The solution of the ILP is then used to formulate the *scheduling ILP*. The scheduling ILP then calculates the start time of each operation such that data dependencies are not violated.

An operation in the program is represented by $S*$ (for example, $S0$). Each operation is executed multiple times during the program execution. Each such execution of an operation S is a *dynamic instance* of the operation and is denoted by $S(i, j, k)$ where i, j, k are the values that the induction variables of the enclosing loops take. This uniquely identifies a dynamic instance of the operation. We denote the initiation interval specified for a loop with induction variable i as II_i. The start time of each operation is specified relative to its parent region. So the start time of operation $S0$ inside a loop represents the delay after which $S0$ will execute relative to the start time of the current loop iteration, and is denoted by t_{S0}. Loops are also treated like any other operation. The start time of the loop with induction variable i is denoted by t_i. These variables denoting the start

time of an operation are called time variables. We now explain our ILP formulations using a few examples.

4.3.1 Intra-Loop Dependence

Figure 4.1 shows a matrix multiplication kernel. We note that there is a write-after-read (WAR) dependence from statement S0 to statement S2 and a read-after-write (RAW) dependence from S2, to S0 of next iteration due to the C[i][j] array. More precisely, we have a dependence from $S2(i', j', k')$ to $S0(i, j, k)$ iff,

- **Address conflict:** Both statements are accessing same element of array C,

$$i = i', j = j'. \tag{4.1}$$

- **Happens before:** If the program ran sequentially $S2(i', j', k')$ would occur before $S0(i, j, k)$,

$$i' * 100 + j' * 10 + k' > i * 100 + j * 10 + k. \tag{4.2}$$

In case of such a dependence, we want to ensure that after scheduling, $S2(i', j', k')$ still occurs after $S0(i, j, k)$. Given the target initiation intervals of each loop, the time instant at which the statement *S0(i,j,k)* will be executed after scheduling is:

$$T_{S0}(i, j, k) = t_i + i * II_i + j * II_j + k * II_k + t_{S0}. \tag{4.3}$$

For a valid schedule, we need to ensure that:

$$T_{S0}(i', j', k') \geq T_{S2}(i, j, k) + 1. \tag{4.4}$$

The extra one cycle is because the store operation on array C due to statement $S2$ would take one cycle to complete. At this point, we may think that in Equations 4.1, 4.2 and 4.4, we have the necessary set of linear constraints to ensure that the resulting ILP solution does not violate the RAW dependence from $S2$ to $S0$. But this is not true.

```
// floating point addition delay=5
// floating point multiplication delay=4
// Load and store to A,B,C take one cycle each.

// Loop-i starts first iteration at time T_i
for (int i = 0; i < 10; i++) {

  // Initiation interval of loop i is II_i.
  for (int j = 0; j < 10; j++) {

    // Initiation interval of loop i is II_j.
    for (int k = 0; k < 10; k++) {
      // Initiation interval of loop i is II_k.

    a = load(A[i][j]);
    b= load(B[i][j]);
    m = a * b;

    // S0 is executed t_s0 cycles after the current
    // k-loop starts.
    S0: c_prev = load(C[i][j]);
    // S1 is executed t_s1 cycles after the current
    // k-loop starts.
    S1: c = c_prev + m;
    // S2 is executed t_s2 cycles after the current
    // k-loop starts.
    S2: store (c, C[i][j]);
    }
  }
}
```

Figure 4.1: Intra-loop memory dependence.

If we put these equations in our ILP formulation, the ILP solver would find one possible set of values for (i, j, k, i', j', k') such that the constraints are satisfied. But for a valid schedule, we need that every (i, j, k, i', j', k') that satisfies Equation 4.1 and 4.2, also satisfy Equation 4.4. In order to formulate the correct ILP, we first expand Equation 4.4 using Equation 4.3 and reorder the terms as follows:

$$\begin{aligned} t_{S2} - t_{S0} \quad \leq \quad & (i' * II_i + j' * II_j + k' * II_k) \\ & -(i * II_i + j * II_j + k * II_k) - 1. \end{aligned} \tag{4.5}$$

We define a new variable called *slack* as follows:

$$\begin{aligned} slack = minimize((i' * II_i + j' * II_j + k' * II_k) \\ - (i * II_i + j * II_j + k * II_k) - 1). \end{aligned} \tag{4.6}$$

such that (i, j, k, i', j', k') satisfy Equation 4.1 and 4.2.

If $t_{S2} - t_{S0} \leq slack$ then the dependence from $S2$ to $S0$ is never violated. We calculate *slack* by solving Equation 4.6 as a minimization ILP problem with Equation 4.5 and 4.2 as the constraints. We also add constraints for the loop bounds on the induction variables (i, j, k, i', j', k'). For each potential memory dependency, we create such an ILP, called the *memory dependence ILP*. If the ILP does not have a solution then there is no actual dependency. Otherwise, we use the calculated slack to add constraints on the time variables in the *scheduling ILP*. For port conflicts, we assume that all operations on the same port of the same memory bank have a data dependence. This added dependence ensures that the resulting schedule does not have port conflicts, i.e., it does not schedule two memory access operations on the same port of the same memory bank in the same clock cycle.

In addition to these constraints related to memory dependence, we also add constraints related to SSA dependence and operator delay in the scheduling ILP. For example, due to the SSA variable *c_prev*, there is a dependence from statement $S0$ to $S1$.

```
// Loop-i1 starts at t_i.
for (int i = 0; i < 10; i++){
  // Initiation interval = II_i.
  for (int j = 0; j < 10; j++){
    // Initiation interval = II_j.

    // S1 is scheduled at t_s1 cycles after the
    // current iteration of j starts.
    S1: store (val, A[i][j]);
  }
}

// Loop-i2 starts at t_u
for (int u = 0; u < 10; u++){
  // Initiation interval = II_u.
  for (int v = 0; v < 10; j++){
    // Initiation interval = II_v.

    // S2 is scheduled at t_s2 cycles after the
    // current iteration of v starts.
    S2: val = load (A[u][v]);
  }
}
```

Figure 4.2: Inter-loop memory dependence.

Since a load operation takes one clock cycle to complete, $S1$ must wait for one cycle before $S0$. We capture this using the constraint:

$$t_{S1} - t_{S0} > 1.$$

Note that t_{S0} and t_{S1} are the start time of the respective operations relative to the start time of the current iteration of the k-loop. But since both the operations are in the same loop-nest, this constraint is enough to ensure that $S1(i, j, k)$ happens after $S0(i, j, k)$ for all values of (i, j, k).

4.3.2 Inter-Loop Dependence

In the previous example, we discussed how we handle memory dependence within a loop nest. Next we will discuss how our technique generalizes for memory dependence across loop nests. Figure 4.2 shows an example of a pair of producer-consumer loop nests. Statement $S2$ has a read-after-write dependence on statement $S1$. Similar to the previous example, we calculate the absolute time T_{S1} and T_{S2} as follows:

$$T_{S1}(i,j) = t_i + i * II_i + j * II_j + t_{S1}, \tag{4.7}$$

$$T_{S2}(u,v) = t_u + u * II_u + v * II_v + t_{S2}. \tag{4.8}$$

For an actual memory dependence between two dynamic instances $S1(i,j)$ and $S2(u,v)$, they must access the same element of the array A, i.e.,

$$u = i, v = j.$$

Unlike the previous example, we do not need the *happens before* constraint here because all instances of $S1$ happen before all the instances of $S2$ in the *sequential schedule*. In general, the *happens before* criterion is required only if there is at least one common loop between the two statements.

The memory dependence is not violated iff:

$$\begin{aligned} & T_{S2}(u,v) \geq T_{S1}(i,j) + 1 \\ \forall \quad & 0 \leq u \leq 10, \quad 0 \leq u \leq 10 \\ & 0 \leq i \leq 10, \quad 0 \leq j \leq 10, \\ & u = i, \quad v = j. \end{aligned} \tag{4.9}$$

Similar to the previous example, we calculate the slack using the following ILP,

$$\begin{aligned} slack = minimize(u * II_u + v * II_v - i * II_i - j * II_j - 1) \\ \text{such that} \quad 0 \leq u \leq 10, \quad 0 \leq u \leq 10 \\ 0 \leq i \leq 10, \quad 0 \leq j \leq 10, \\ u = i, \quad v = j. \end{aligned} \tag{4.10}$$

The constraint added to the scheduling ILP to ensure that the memory dependence is not violated is:

$$t_i + t_{S1} - t_u - t_{S2} \leq slack.$$

4.3.3 Resource Constraints

Operations in a hardware design may need to share available hardware resources. This includes memory ports for load/store operations and expensive hardware resources such as floating point adders and multipliers. The ILP needs to ensure that a shared resource is not required by multiple operations in the same clock cycle. For instance, multiple read/write accesses to the same memory port can not be serviced in the same clock cycle. Similarly, if multiple floating point operations are using the same floating point multiplier in the FPGA, then these operations must be scheduled in different clock cycles. Collectively, these types of constraints on the number and type of each hardware resource available at each clock cycle is called *resource constraint*. Unlike dependence constraints, resource constraints do not impose an ordering. For instance, if two memory operations use the same port then the resource constraint requires that the two operations can not be scheduled in the same clock cycle. In contrast, a dependence constraint between the two memory operations enforces an ordering between the two operations. This key difference makes resource constraints harder to represent in an ILP.

To understand how we encode resource constraints in an ILP, we take the example program in Figure 4.2. If array A has only one read/write port then we need to ensure that any instance of statements $S1$ and $S2$ do not occur at the same clock cycle i.e.

$$T_{S1}(i,j) \neq T_{S2}(u,v) \quad \forall i,j,u,v \tag{4.11}$$

We can not use this constraint directly in the ILP because the ILP will enforce the constraints only for one set of (i,j,u,v), i.e. integer linear programs can not directly handle universal quantifiers ($\forall$). We need a way to convert this to an existential quantifier ($\exists$). We observe that if there exists integers $(d, r1, r2)$ such that,

$$T_{S1}(i,j) = r1 \mod d \quad \forall i,j$$

$$T_{S2}(u,v) = r2 \mod d \quad \forall u,v$$

then,

$$r1 \neq r2 \implies T_{S1}(i,j) \neq T_{S2}(u,v) \quad \forall i,j,u,v$$

We define $d = gcd(II_i, II_j, II_u, II_v)$, where gcd calculates the greatest common divisor. We can rewrite Eq 4.7 as follows,

$$T_{S1}(i,j) = t_i + i * n_i * d + j * n_j * d + t_{S1}, \tag{4.12}$$

where $II_i = n_i * d$ and $II_j = n_j * d$. We know such n_i and n_j exists because d is a common divisor of II_i and II_j by definition. Thus,

$$\begin{aligned} T_{S1}(i,j) \mod d &= (t_i + i * n_i * d + j * n_j * d) \mod d + t_{S1} \mod d \quad \forall i,j \\ &= t_{S1} \mod d \quad \forall i,j \\ \implies \quad r1 &= t_{S1} \mod d \quad \forall i,j \end{aligned} \tag{4.13}$$

Thus, we can define $(d, r1, r2)$ for $S1$ and $S2$ as follows,

$$\begin{aligned} d &= gcd(II_i, II_j, II_u, II_v) \\ r1 &= t_{S1} \mod d \\ r2 &= t_{S2} \mod d \end{aligned} \tag{4.14}$$

such that,

$$r1 \neq r2 \implies T_{S1}(i,j) \neq T_{S2}(u,v) \quad \forall i,j,u,v$$

We can calculate d, given II_i, II_j, II_u, II_v. In order to calculate $r1$ and $r2$, we add the following constraints in the ILP:

$$\begin{aligned} t_{S1} &= n_1 * d + r1 \\ 0 &\leq r1 < d \\ t_{S2} &= n_2 * d + r2 \\ 0 &\leq r2 < d. \end{aligned} \tag{4.15}$$

We can not add the inequality $r1 \neq r2$ directly to the ILP. Thus, we introduce a set of binary variables, c_{10}, c_{11}, c_{20} and c_{21} with the following constraints on them:

$$\begin{aligned} &r1 = 0 * c_{10} + 1 * c_{11}, \\ &r2 = 0 * c_{20} + 1 * c_{21}, \\ &0 \leq c_{10}, c_{11}, c_{20}, c_{21} \leq 1, \\ &c_{10} + c_{11} = 1, \\ &c_{20} + c_{21} = 1. \end{aligned} \tag{4.16}$$

Now, we can enforce the constraint $r1 \neq r2$ using these binary variables as follows,

$$\begin{aligned} c_{10} + c_{20} &\leq 1, \\ c_{11} + c_{21} &\leq 1. \end{aligned} \tag{4.17}$$

In general, we can represent the resource constraint as follows,

$$\begin{aligned} r_i &= \sum_j j * c_{ij}, \\ \sum_j c_{ij} &= 1, \\ \sum_i c_{ij} &\leq N_{res}, \\ 0 \leq c_{ij} &\leq 1 \quad \forall i, j. \end{aligned} \tag{4.18}$$

N_{res} is the number of copies of the hardware resource that are available. For instance, in the case of load/store operations on a memory, N_{res} may be the number of ports that the memory has.

The resource ILP constraints can also be used to share other hardware resources across multiple operations. For instance, the same ILP constraints can be used to specify that a set of floating-point multiplication operations use a fixed number of hardware multipliers. By specifying N_{res} equal to the number of available hardware multipliers, we can force the scheduler to ensure that no more than N_{res} number of multiplication operations are scheduled in the same clock cycle. Time-sharing the hardware resources may lead to designs that are more resource efficient.

4.3.4 Minimization Objective

The scheduling ILP guarantees a valid schedule but it does not ensure resource efficiency. Consider the example in Figure 4.3. In the first loop, the schedule is valid but there is an unnecessary delay of thousand cycles between the load and the dependent store op. To ensure that the value of variable x is not overwritten by the time the store operation happens, a delay of 999 cycles is introduced using the hir.delay operation. This is implemented in hardware using shift registers which is a waste of resources. If the store op is scheduled at $ti + 1$, then a delay is not required as the load op produces the value at $ti + 1$.

```
//Inefficient schedule.
hir.for %i:i6 = %c0 to %c32 step %c1 iter_time(%ti=%t){
  %x = hir.load %A[port 0][%i]
  %xx = hir.delay %x by 999 at %ti
  hir.store %xx to %B[%i] at %ti+1000
  hir.next_iter at %ti+1
}

//Efficient schedule.
hir.for %i:i6 = %c0 to %c32 step %c1 iter_time(%ti=%t){
  %x = hir.load %A[port 0][%i]
  hir.store %x to %B[%i] at %ti+1
  hir.next_iter at %ti+1
}
```

Figure 4.3: Resource-inefficient and -efficient schedules.

To achieve a more resource-efficient schedule, we calculate the delay required by an SSA variable as the difference in the start time of the operation consuming the SSA variable and the operation producing it. We use the sum of the required delays as our minimization objective.

4.4 Summary

In this chapter, we presented our ILP-based scheduler. The scheduler performs analysis on the hardware kernel written in the affine dialect of MLIR. We use the analysis results to create an ILP for each data dependency via memory elements (arrays). The ILP calculates the available slack between the source and destination operations of the dependence. We then use the slacks calculated for each dependence pair to formulate one ILP for the complete kernel. These constraints ensure that the data dependencies are not violated. For operations in the def-use chain of an SSA variable, we add constraints to ensure that the *defining* operation is scheduled before the *use* operations. We also add constraints to ensure that there are no port conflicts, i.e. two memory operations on the same port of the same memory bank are not scheduled in the same clock cycle.

The result of our ILP formulation is the schedule for the kernel, i.e. a mapping from dynamic instances of operations to the time at which these operations would execute. In addition, the ILP also calculates the port assignments to avoid any port conflict between memory access operations.

Chapter 5

Evaluation

In this section, we evaluate the performance and resource usage of our HIR high-level synthesis compiler against the Vitis HLS compiler. We break the evaluation into two parts. We first evaluate the overhead added by our backend code generator. To quantify this, we write some small benchmarks by hand in the HIR dialect. We compare this with the HLS implementation of these benchmarks in C++ language. We then evaluate the complete end-to-end compiler. For this evaluation, we use four image-processing applications. We write the programs in C which is fed to our compiler to generate the final SystemVerilog output. This evaluation helps us understand the benefits of our ILP-based automatic scheduler compared to Vitis HLS.

5.1 Evaluation of the HIR Backend

In this section, we evaluate the backend code generator for the HIR compiler. The backend is responsible for converting HIR into SystemVerilog. It performs various tasks such as generating the state machines to implement the schedule and control flow and lowering multi-ported, multi-banked memrefs into hardware memory buffers. The purpose of this evaluation is to find out the amount of extra hardware overhead added by the compiler backend.

The code generator transforms the HIR IR to *hw* and *sv* dialect of CIRCT which is

Benchmark	**Description**
Transpose	Transpose a 16x16 matrix
Stencil	Convolution of a 64-element array with a kernel of size 2
Histogram	Histogram of a 16x16 image
Matmul-16x16	A 16x16 2D systolic array for matrix multiplication.
Convolution	2D Convolution of an 8x8 image with 2x2 kernel
gesummv	Polybench
floyd-warshall	Polybench

Table 5.1: Description of the handwritten HIR benchmarks for evaluating the backend.

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
Histogram	404	187	0	1	335	55	0	1
Convolution	80	132	3	0	86	112	3	0
gesummv	276	353	12	0	328	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 5.2: FPGA resource usage and comparison with Vitis HLS.

then lowered to SystemVerilog using the *emit-verilog* pass. The *hw* and *sv* dialects are designed to represent hardware at an HDL level of abstraction. The *sv* and *hw* dialects and the *export-verilog* pass are part of the CIRCT project which aims to generate HLS tools on top of the MLIR infrastructure. 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 [Majumder and Bondhugula (2022)] 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 an area-efficient design.

We decided to compare HIR against Vitis HLS [Inc. ([n. d.])] compiler. We implemented our benchmarks in both the HIR dialect and C++ (for Vitis HLS). We added the necessary pragmas in the C++ benchmarks to direct the Vitis HLS compiler for a performance-optimized design (as opposed to an area-optimized design). Both the HLS and HIR implementations used the same level of loop pipelining and unrolling and similar types of memories with the same number of ports. We further simulated the generated SystemVerilog to verify that the designs achieve the same level of performance. This shows that HIR can represent the necessary hardware optimizations that an HLS compiler performs.

To check the resource utilization, 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 seven benchmarks. 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.

Table 5.1 describes the benchmarks for evaluating our backend. We used the modified Polygeist frontend and our ILP-based scheduler to generate HIR for all the benchmarks except the Histogram benchmark. As Histogram is not an affine workload, we wrote the kernel in HIR by hand. We did not include all benchmarks from Polybench as our compiler only supports constant loop bounds (Chapter 4). We believe that this set of benchmarks is sufficient to show the performance of our compiler backend. In Section 5.2 we use more complex image processing benchmarks with multiple loop nests for the end-to-end evaluation of the entire HIR compiler. Table 5.2 compares the resource utilization of our benchmarks against Vitis HLS implementations at iso-performance, i.e. at the same clock frequency (200MHz) and the same level of loop pipelining/unrolling. 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 more aggressively pipelines the design 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 and 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 5.2 show that the HIR compiler can generate Verilog designs that are comparable to Vitis HLS in resource usage while matching the performance.

5.2 Evaluation of the Complete HIR Compiler

We extend the Polygeist C/C++ frontend with pragmas similar to Vitis HLS. The Polygeist frontend passes the pragma information as attributes in the Polygeist generated Affine dialect. We use the HIR as our backend for the generation of SystemVerilog. Our ILP-based scheduler is invoked when lowering from Affine dialect to the HIR dialect. We evaluate the effectiveness of the resulting end-to-end (C-to-SystemVerilog) compiler

on a set of benchmarks by generating SystemVerilog from both our compiler and Vitis HLS. We report the performance by measuring the total number of cycles required for each design to complete execution. We report the resource usage by running synthesis and implementation using Vivado on the generated SystemVerilog designs. All generated designs are synthesized for the VC709 Xilinx FPGA board with a target frequency of 200MHz. The correctness of the generated design is validated on random inputs by comparing the simulation results with Vitis HLS. We also verify that our designs can meet the timing closure after implementation (i.e., after place-and-route). We evaluate our compiler on the following set of benchmarks. The source code for these benchmarks can be found in Appendix A.

Unsharp mask is an image processing pipeline to sharpen the image at the boundaries. It involves calculating the image blur along X and Y direction and then applying a pointwise sharpening and masking filter. We use a 32x32 patch in our benchmark.

Harris is a classic corner detection algorithm [Harris and Stephens (1988)]. It involves multiple stencil operations such as calculating gradients along the x and y axis. We use a 32x32 patch of the image as input for the benchmark.

DUS is an image pipeline where we downsample the image by a factor of two and then upsample it. Downsampling involves blurring and the upsampling uses linear interpolation, both of which are stencil operations. Both the operations are done on each axis separately resulting in four loop nests. Down-sampling and upsampling are very common in image processing pipelines and offer a unique challenge since the producer and consumer loop nests may not have the same number of iterations. We use a 32x32 image patch to evaluate the benchmark.

The **Optical flow** benchmark implements the Lucas-Kanade dense optical flow algorithm [Lucas and Kanade (1981)]. We implemented the single-scale version of the algorithm and used a 32x32 image patch for our evaluation. The benchmark is a mix of pointwise and stencil operations.

2mm is a benchmark from polybench. It involves two matrix multiplications in a series. Both the intermediate and the final matrix are written to the output. This benchmark shows that our compiler can handle non-stencil memory access patterns as well. We use an 8x8 matrix to evaluate the benchmark.

Benchmark	LUT	FF	BRAM	DSP	cycles
Unsharp mask (spsc)	1010	1590	9	8	62472
Harris (spsc)	2010	2607	20	12	270729
DUS (spsc)	1379	1985	7.5	11	2786
Optical flow (spsc)	4084	5175	24	18	382250
2mm	594	1068	0	5	8671

(a) Vitis HLS without dataflow optimization.

Benchmark	LUT	FF	BRAM	DSP	cycles
Unsharp mask (spsc)	1876	2976	12	20	37344
Harris (spsc)	3157	5124	42	43	110083
DUS (spsc)	2747	3589	9	26	2783
Optical flow (spsc)	6821	9088	65	59	110475

(b) Vitis HLS with dataflow optimization.

Benchmark	LUT	FF	BRAM	DSP	cycles
Unsharp mask (spsc)	1634	1969	9	16	27201
Harris (spsc)	4005	4314	20	34	88327
DUS (spsc)	1901	2762	1.5	24	2057
Optical flow (spsc)	6741	7194	24	54	90543
2mm	2667	2063	0	12	3589

(c) HIR with dataflow optimization.

Table 5.3: Resource usage and performance of Vitis HLS and HIR. All designs are synthesized at 200MHz. Clock cycles are measured via simulation. Resource usage numbers are post-implementation in Vivado. Vitis HLS can not apply dataflow optimizations to 2mm.

5.2.1 Results

In our evaluation, we try to quantify the advantages of our scheduler over Xilinx Vitis HLS (a.k.a Vivado HLS), a state-of-the-art commercial high-level synthesis compiler . Specifically, we compare our scheduler's ability to reduce the overall latency by overlapping the execution of multiple producer-consumer loop nests.

To motivate the importance of overlapped execution of producer-consumer loop nests, we first quantify the performance improvements due to overlapping of producer-consumer loop nests. We then compare the performance of the designs generated by our compiler against Vitis HLS with its *dataflow* optimizations. We also compare the resource usage of both approaches to understand the overheads associated with the handshaking logic implemented by Vitis HLS for its dataflow optimization.

In our evaluation, we also try to understand the limitations of Vitis HLS dataflow optimizations and quantify their impact on the actual performance. We demonstrate that our compiler can perform overlapped loop nest execution even in situations where Vitis HLS fails to apply dataflow optimization, and report the performance improvements enabled by our technique in such situations. Using the raw performance and resource usage data from Table 5.3 we compare HIR with Vitis HLS.

Q. Does overlapped execution of loop nests provide any meaningful performance improvement?

A key optimization enabled by our scheduling technique is the overlapped execution of producer-consumer loop nests. But we need to understand how much performance improvement can be attributed to this optimization compared to loop pipelining. To understand this, we generate the scheduled HIR and calculate the total latency (number of clock cycles) of each loop nest takes by multiplying the initiation interval of the outermost loop with its trip count. We then add these latencies to get the total kernel execution latency. This would be the latency of the kernel if each loop was pipelined but there was no overlapping between different loop nests. We use this to quantify the performance improvement of a kernel with all loop nests maximally overlapped (without

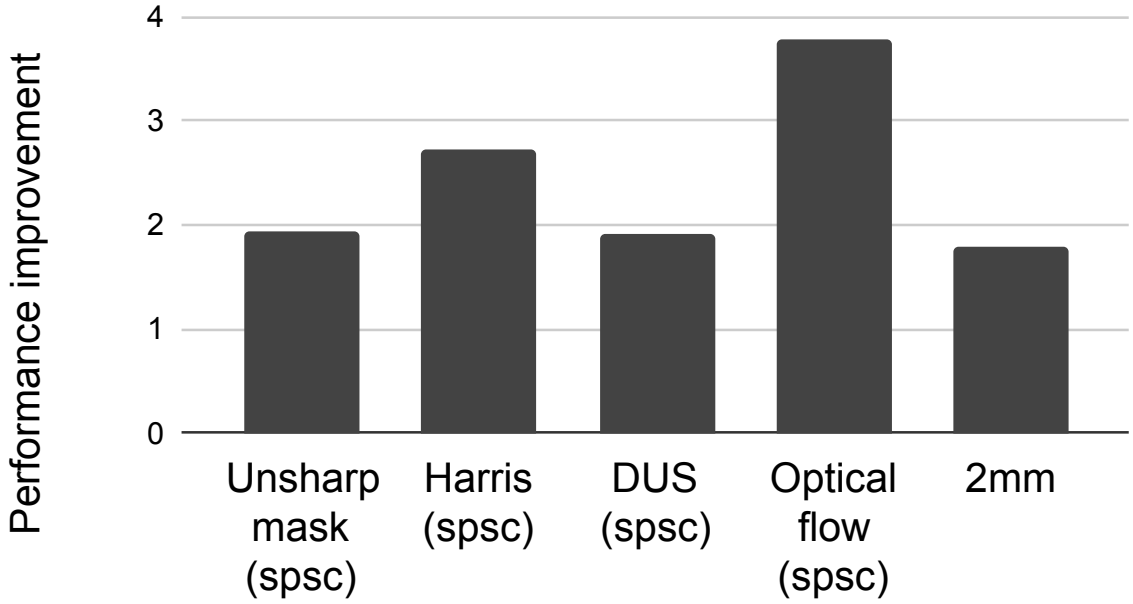


Figure 5.1: Performance improvement due to overlapped execution of loop nests.

violating data-dependencies) compared to a kernel with just intra-loop pipelining and no overlapping of loop nests. Figure 5.1 shows the actual performance of our generated design compared to the design where loop nests are not overlapped. The performance gain due to overlapping varies between 1.7x and 3.7x highlighting the practical importance of this optimization.

Q. How does our scheduler compare against Vitis HLS dataflow optimizations?

Vitis HLS *dataflow* optimization also tries to overlap the execution of producer-consumer loop nests. We compare our performance gains against Vitis HLS in Figure 5.2. Vitis HLS *dataflow* pragma only works if the intermediate arrays between producer-consumer loops follow the single-producer-consumer rule i.e., only one loop nest writes to the array and only one loop nest reads from it. DUS already satisfies the single-producer-consumer constraint. We convert unsharp mask, Harris corner detection and optical flow to single-producer-consumer workloads by inserting copying loops that duplicate the intermediate arrays that are consumed by multiple loop nests. Another limitation of Vitis HLS dataflow optimization is that it can not handle read/write to function

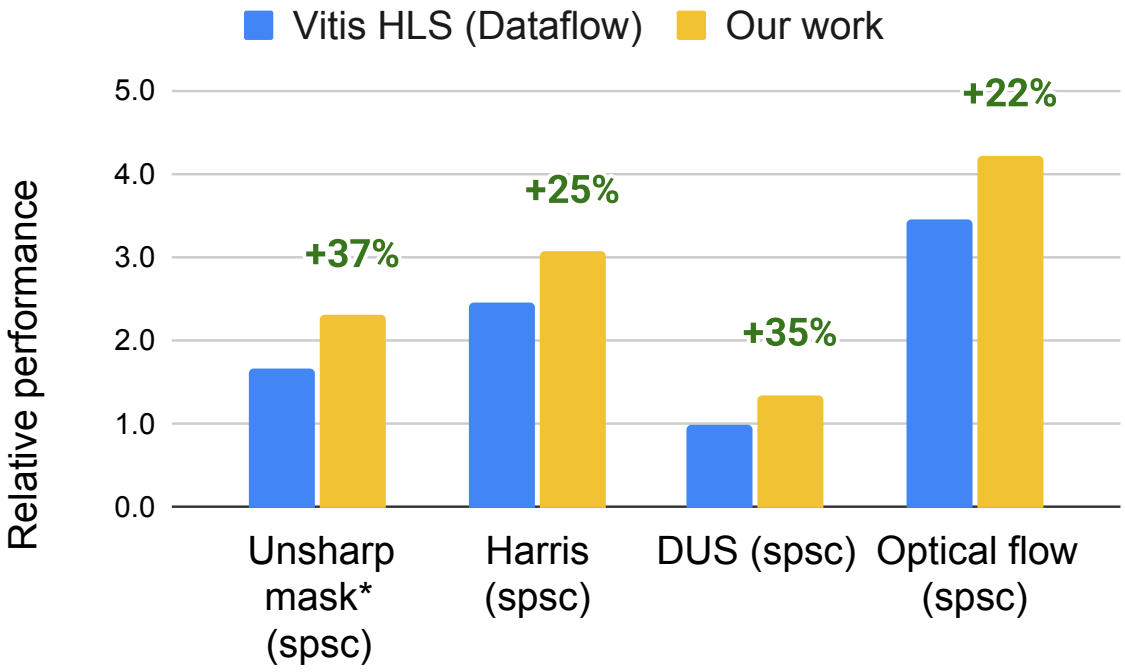
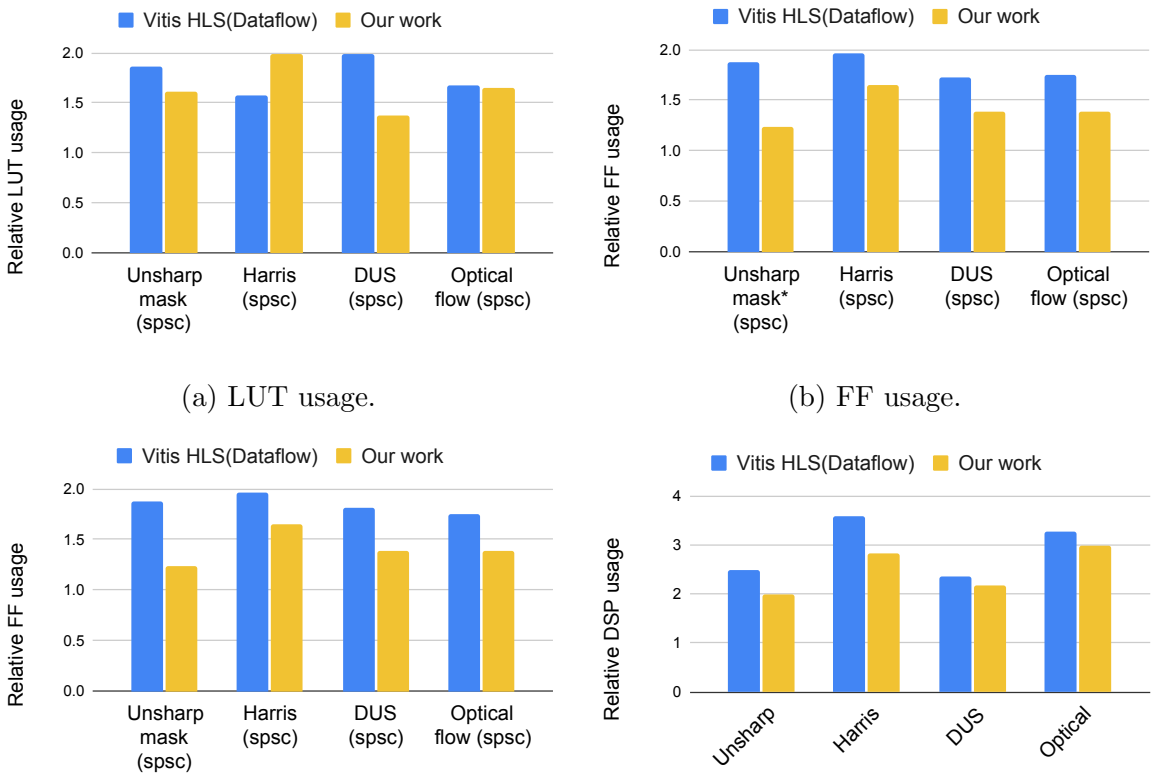


Figure 5.2: Performance comparison between Vitis HLS (with dataflow directives) and our work. The baseline is Vitis HLS without dataflow directives.

arguments in the dataflow region. Due to this limitation of Vitis HLS, we could not use the 2mm benchmark for this specific evaluation, since it writes the intermediate matrix multiplication output to one of the function arguments.

Figure 5.2 reports the performance of both our work and Vitis HLS with dataflow optimization enabled. The performance numbers are relative to Vitis HLS without the dataflow optimization directive. The first thing we observe is that the Vitis HLS *dataflow* directive indeed improves the overall latency of the kernels for single-producer-consumer workloads. However, the designs generated by our compiler provide additional performance gains of upto 37% on top of the dataflow optimized Vitis HLS designs. This showcases the effectiveness of our ILP-based scheduler. But the performance improvement is only one aspect of a good design. We also need to evaluate our resource usage against Vitis HLS.



(a) LUT usage.

(b) FF usage.

(c) BRAM usage.

(d) DSP usage.

Figure 5.3: Resource usage of Vitis HLS with dataflow pragma and our work relative to Vitis HLS without the dataflow directives.

Q. How does our resource usage compare against Vitis HLS?

Figure 5.3 shows the resource usage of Vitis HLS dataflow designs and the designs generated by our compiler relative to Vitis HLS generated designs without the *dataflow* pragma. As expected, both Vitis HLS dataflow and our design consumes more resources compared to the non-dataflow design since they both perform much better than the non-dataflow design.

Interestingly, our design consumes fewer resources in all benchmarks (except for LUT usage in Harris corner detection) while, simultaneously, outperforming the Vitis HLS dataflow optimized design. We see the greatest improvement in BRAM usage. This can be attributed to the synthesis of ping-pong buffers by Vitis HLS for more complex

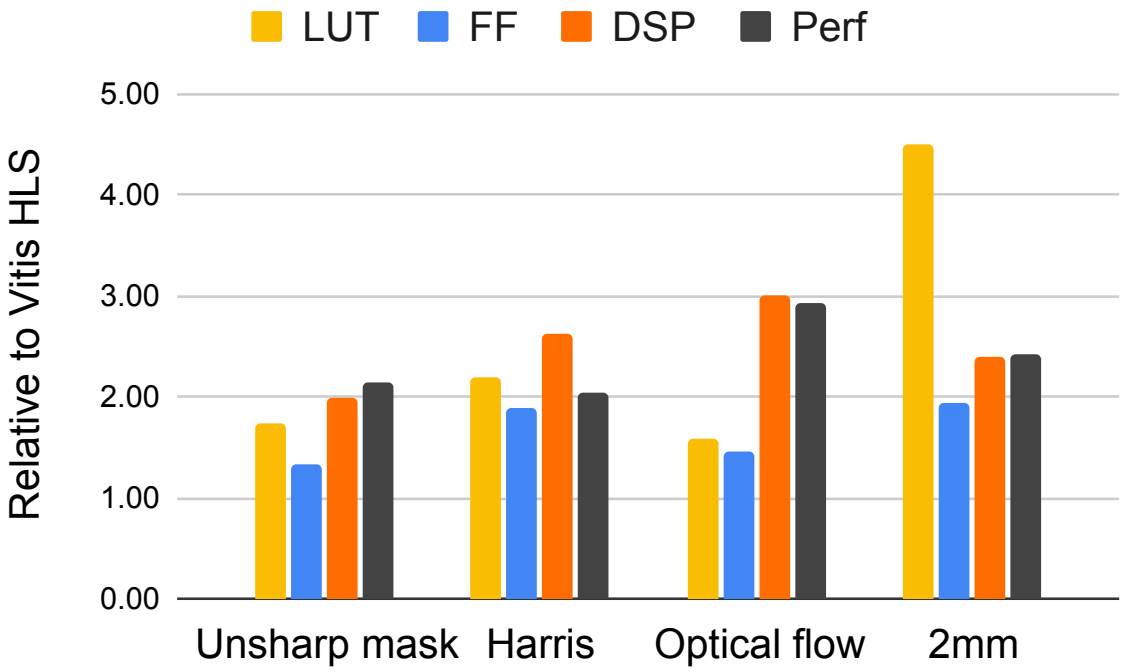


Figure 5.4: Resource usage and performance of our work relative to Vitis HLS for workloads without SPSC dataflow pattern.

dataflow patterns which results in more BRAM usage compared to the baseline non-dataflow Vitis HLS design. Our design, on the other hand, does not instantiate any extra RAMs. Thus our BRAM usage is always the same as the non-dataflow Vitis HLS design (except for DUS where we have even less BRAM usage than the non-dataflow design).

Depending on the data access pattern, Vitis HLS synthesizes either ping-pong buffers or FIFOs to replace the intermediate arrays accessed by producer and consumer loops. Runtime synchronization ensures that the consumer reads data only after the producer has written the data to the intermediate buffer. Though it is impossible to isolate the exact cause of the extra LUT and FF overhead, one possible source of the extra resource usage can be attributed to the extra logic required to implement these synchronizations. On the other hand, our compiler statically schedules the producer and consumer loops in such a way that the data dependencies between them are guaranteed to be never

violated, avoiding the need for any extra synchronization logic at runtime.

Q. What are the limitations of Vitis HLS dataflow optimization and does our technique work in those scenarios.

To understand why we outperform Vitis HLS in Figure 5.1, we need to understand when Vitis HLS can overlap producer-consumer loop nests. We have already mentioned two limitations of the dataflow pragma in Vitis HLS:

- All intermediate arrays must have a single producer loop nest and a single consumer loop nest.
- In a dataflow region, there should not be any read/write to the function arguments.

Although DUS satisfies both the above criteria, we observe that Vitis HLS with dataflow does not offer any improvement in the overall latency of the DUS operation compared to the non-dataflow version (as shown in Figure 5.2). To overlap the execution of producer and consumer loop nests, Vitis HLS tries to replace all intermediate array accesses with FIFOs. If the intermediate arrays are replaced with FIFOs, the consumer can start its execution and on every read of the intermediate array, it can wait until the producer inserts data into the FIFO. This runtime synchronization between the consumer read and producer write (using the FIFO) ensures that the data dependency will not be violated. But the limitation of this approach is that the intermediate arrays can only be replaced with FIFOs if the consumer reads the data from the array in the same order in which the producer is writing. The loop nests in DUS violate this since both upsampling and downsampling loop nests access a neighborhood of pixels (a stencil operation). Because of this, Vitis HLS can not overlap the execution of the loop nests present in downsampling and upsampling. Since our compiler uses an ILP-based scheduler to find the dependence distances between producer and consumer loop nests, we can handle arbitrary affine accesses on the intermediate arrays. As a result, we see a 35% performance improvement in DUS over Vitis HLS. Other benchmarks have a mix of loop nests, some of which can and can not be overlapped by Vitis HLS due to the data access patterns. For instance, in Unsharp mask, there are two convolution operations in a

chain that Vitis HLS can not overlap (since the consumer convolution reads a window of pixels similar to DUS) but it can overlap the sharpening and masking loop nests which are both pointwise access. Our approach, on the other hand, can safely overlap both types of producer-consumer pairs. This accounts for the difference in performance between the two approaches even after converting the programs to single-producer-single-consumer (SPSC) style to satisfy Vitis HLS's dataflow optimization criteria.

We converted our benchmarks to SPSC to enable Vitis HLS dataflow optimizations. But since our scheduler claims to handle arbitrary affine accesses, it is natural to question whether we would see any performance improvement on the unmodified programs i.e., programs with multiple consumers and different data access ordering of the producer and consumer. Figure 5.4 demonstrates the effectiveness of our technique on the unmodified benchmarks. We achieve a speedup between 2x and 2.9x over Vitis HLS (without dataflow directive since dataflow can not handle these unmodified workloads) across all benchmarks, which demonstrates that our compiler can aggressively overlap the loop nests in the unmodified programs. We report the resource usage of our design relative to Vitis HLS for completeness. Vitis HLS can reuse resources such as DSP units across loop nests since the loop nests execute one after the other. As a result, we consume more resources to provide additional performance gains. We include 2mm for this evaluation since our compiler can handle function argument accesses as well.

5.3 Summary

In this chapter, we evaluated the HIR compiler infrastructure against Vitis HLS compiler from Xilinx. We perform the evaluation in two steps. We first show that the compiler IR can represent different types of parallelism and the backend code generator can generate efficient hardware using the IR description. We then compare the end-to-end compiler infrastructure against Vitis HLS. For this, we use a set of image-processing pipelines. We use the Polygeist MLIR frontend to lower the benchmarks from C language to the *Affine* dialect of MLIR. We identify a missing optimization opportunity in the Vitis HLS

compiler namely overlapped execution of producer-consumer loop nests for arbitrary affine data access patterns. We show that our ILP-based the scheduler can leverage this optimization opportunity to generate designs that perform significantly better than Vitis HLS while consuming fewer FPGA resources. Overall the evaluation shows that the HIR compiler infrastructure can produce designs that are comparable in quality with Vitis HLS and for specific types of workloads (multiple kernels with producer-consumer relation) it can generate more performant designs.

Chapter 6

Related work

In this chapter, we discuss the previous work on hardware design and high-level synthesis.

6.1 Hardware description languages

Hardware description languages (HDLs) such as VHDL and Verilog provide developers with the capability to represent complex designs with fine-grained control over the instantiated hardware and scheduling. Examples of HDLs implemented as embedded domain-specific languages (DSLs) on top of existing software programming languages including Chisel [Bachrach et al. (2012b)], MyHDL [Decaluwe (2004)], and VeriScala [Liu et al. (2017)]. These DSLs leverage the meta-programming capabilities of the host language to represent highly parameterized hardware designs. Due to meta-programming these languages provide a higher level of abstraction than the traditional HDLs like Verilog. But they also require the developer to describe the hardware at the register-transfer level, i.e. specify the computations and memory read/write operations for each clock cycle. A hardware design description in an HDL uses state machines to explicitly schedule each computation and read/write operation. Control flow such as for-loops is also implemented by instantiating state machines explicitly. The problem with such designs is that any change in the timing behavior of one component can silently introduce errors in the overall design even if the functional behavior of the component is the same.

Bluespec-Verilog [Nikhil (2004)] represents a circuit as a set of atomic rules. A recent work [Bourgeat et al. (2020)] attempts to enhance the language for user-level control over the schedule and predictable performance. We believe that HIR's ability to express a design at a higher level of abstraction than RTL while providing complete control over the schedule would be a good fit as a target intermediate language for the BSV compiler. The Dahlia [Nigam et al. (2020)] language is inspired by the observation that HLS languages generate unpredictable designs due to their excessive flexibility. Dahlia and its affine type-system enforce the high-level design to respect the limitations of the hardware. For example, a valid design is guaranteed to never have multiple reads and writes on the same memory in the same cycle. This ensures that the design lowered to HIR would not have undefined behavior. Many of Dahlia's language features such as memory banking, *for* loops and loop unrolling have a direct equivalent in HIR.

6.2 HLS languages & compilers

Several HLS compilers, such as Vitis HLS [Inc. ([n. d.])], Intel OpenCL stack [Intel ([n. d.])], and the LegUp compiler [Canis et al. (2011b)], have repurposed existing general-purpose software languages like C/C++ and OpenCL for hardware description, while others, such as Dahlia [Nigam et al. (2020)], Spatial [Koeplinger et al. (2018)], and HeteroCL [Lai et al. (2019)], have opted for a clean-slate DSL design specialized for high-level synthesis.

Dahlia emphasizes performance predictability by using time-sensitive affine types to check and prevent conflicting resource usage during compilation. Its key observation is that in case of port conflicts, traditional high-level synthesis compilers dynamically introduce extra delay at runtime to sequentially perform the conflicting operations, which leads to schedule-dependent unpredictable performance. Dahlia's type system forbids scheduling two memory access operations using the same port on the same clock cycle.

It accomplishes this by using Affine types. *Affine types* are a concept in type theory that restrict objects to at most one *use*. A language can define what constitutes a *use*. Dahlia provides a language construct to specify that there is one clock cycle delay between two operations. Operations such as accessing a memory port constitute a *use* of the resource (port). Within a clock cycle, the compiler statically enforces that there can be only one *use*. At the beginning of every cycle, the use count is reset to zero for all resources.

Spatial is a high-level synthesis language with hardware-centric abstractions to improve programmer productivity and design performance. Some of these abstractions include control flow constructs such as finite-state machines, scheduling directives, support for different types of hardware memories and bus interfaces, and support for design space exploration. It can target both FPGAs and CGRAs.

HeteroCL is a Python-based programming language targeting FPGAs. It decouples algorithmic specification from hardware customizations of compute, data type and memory architecture. HeteroCL's code-generator targets the Merlin compiler [Cong et al. (2016)] as its general-purpose backend and the SODA Chi et al. (2018) framework to implement stencil computation.

SODA-OPT [Agostini et al. (2022)] is an MLIR-based HLS compiler framework. The SODA-OPT compiler automatically searches, outlines, tiles and pre-optimizes relevant code regions to generate hardware accelerators. Backend synthesis tools connect to SODA-OPT through progressive intermediate representation lowering. The compiler currently lowers to LLVM IR and then uses the Bambu [Ferrandi et al. (2021)] compiler to generate the final hardware design.

6.3 DSLs

Halide [Ragan-Kelley et al. (2013)] and Polymage [Mullapudi et al. (2015)] DSLs offer HLS backends [Li et al. (2020); Chugh et al. (2016b)] for FPGAs. Darkroom [Hegarty et al. (2014)] and Rigel [Hegarty et al. (2016)] implement a domain-specific high-level synthesis compiler for image processing. SODA [Chi et al. (2018)], StencilFlow [de Fine Licht et al. (2021)] and SASA [Tian et al. (2023)] lower stencil computations described in a domain-specific language to efficient hardware design.

Aetherling [Durst et al. (2020)] introduces sequence types to encode throughput and latency by specifying when sequence elements are produced and consumed. Implemented as a DSL embedded in Haskell, Aetherling compiles data-parallel programs into statically scheduled, streaming hardware circuits. The language attempts to remove the need for synchronization overhead between different stages of the pipeline. Aetherling generates Verilog via the Chisel language. HIR is an ideal fit for such a language. The explicit deterministic scheduling completely removes the need for extra synchronization. HIR can represent designs where the producer and consumer operate in lock steps and there is no back-pressure between the stages and Aetherling's type system ensures that this would always be the case.

6.4 Source-to-source Compilers

The quality of hardware generated by HLS compilers depends heavily on the input program. Many polyhedral-based optimization techniques have been proposed in the past Alias et al. (2010); Pouchet et al. (2013); Zuo et al. (2013b,a) that employ polyhedral techniques to improve the input program for the HLS tool. These pre-processing loop optimization techniques complement our work. For instance, *loop interchange* may enable greater overlap between producer and consumer loops.

AutoSA [Wang et al. (2021)] and **PolySA** [Cong and Wang (2018)] utilize polyhedral

optimization to synthesize efficient systolic-array-based hardware designs. The optimizations in both these works are tailored for kernels that can be represented as systolic arrays in hardware. In contrast, our work focuses on pipelining arbitrary affine kernels with constant loop bounds and is not limited to generating systolic array-based architectures.

ScaleHLS [Ye et al. (2022b,a)] provides an MLIR-based end-to-end compilation framework for high-level synthesis. It takes HLS C (C with HLS-specific directives) as input and after transformations, produces a more optimized implementation in HLS C/C++. It leverages existing HLS compiler [Inc. ([n. d.])] to lower the optimized HLS C/C++ to RTL. Scale HLS is a source-to-source compiler. Its key insight is that the quality of the generated hardware is dependent on how the input HLS program is structured. It transforms the input program in a way that is easier to optimize for the HLS compiler. For instance, Vivado HLS's dataflow optimizations require a single-producer-single-consumer dataflow pattern. It converts the input program to enable this pattern. As ScaleHLS targets Vivado HLS for lowering to RTL, it can utilize the optimizations such as scheduling and pipelining as well as RTL code generation of Vivado HLS, while focusing only on high-level optimizations. The disadvantage of such an approach is that the ScaleHLS compiler is heavily tied to the Vivado HLS compiler. The optimizations and code-generation of ScaleHLS are designed to generate HLS C/C++ that can be easily optimized by Vivado HLS. Additionally, such a compiler can not be used as a testbed to try new scheduling optimizations, since the final schedule is always decided by Vivado HLS. Similarly, the RTL code-generation strategy is also decided by Vivado HLS so it is not possible to generate statically scheduled dataflow circuits until Vivado HLS supports it. On the other hand, our goal in this thesis is to develop an end-to-end compiler in MLIR that can be extended with new optimizations at every level similar to the LLVM compiler. A set of ScaleHLS style pre-optimization passes before scheduling would certainly enhance the capabilities of our compiler in the same way in which ScaleHLS currently improves upon Vivado HLS.

6.5 Intermediate Representations

Static Single Assignment (SSA [Cytron et al. (1991)]) based intermediate representations are standard in software compilation pipelines. Both LLVM [Lattner and Adve (2004)] and GCC [Novillo (2003)] use SSA-based IRs for their compilation pipeline. Some HLS compilers [Canis et al. (2011a); Pilato and Ferrandi (2013)] also reuse these software IRs for high-level synthesis.

LLHD [Schuiki et al. (2020)] is another intermediate representation for hardware description that was later migrated to MLIR and is hosted as part of CIRCT [community (2020)], an LLVM sub-project and an umbrella initiative to adapt and use MLIR for high-level synthesis. 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 a 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 such as SystemVerilog and VHDL and as a low-level IR in the HLS pipeline, it is too low-level for the initial stages of a high-level synthesis compiler.

FIRRTL [Izraelevitz et al. (2017)] IR is designed along with the Chisel [Bachrach et al. (2012a)] 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, FIRRTL is also a low-level IR and a suitable target for HDLs.

μIR [Sharifian et al. (2019)] decouples the micro-architectural representation of the accelerator from its behavioral specification. Microarchitectural optimizations like pipelining and retiming are implemented as transformations of the structural graph.

Calyx [Nigam et al. (2021)] 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. Calyx captures latency-insensitive designs with the ability to utilize optionally provided latency information to simplify the state machine. Calyx is designed as a pre-scheduling IR whereas HIR is designed as a post-scheduling IR. The time variables in HIR allow the scheduler to easily express the desired schedule of the design. Another difference between Calyx and HIR is the function signatures. Calyx allows groups (a unit of encapsulation in Calyx) to specify the group's execution latency using the *static* attribute. HIR's functions capture the exact latency of each argument and result. This helps while interfacing with external Verilog modules. For instance, fused-multiply-add primitives from Xilinx can accept the accumulator input a few cycles after the multiplier inputs since the addition happens after the multiplication. HIR can accurately capture this information in the function declaration corresponding to the external fused-multiply-add Verilog module.

HCL [Pal et al. (2022)] is the MLIR dialect for HeteroCL [Lai et al. (2019)]. It is a high-level unscheduled intermediate representation for accelerator description. Like HeteroCL, it supports compute, data type and memory customizations for efficient hardware design. The HCL dialect lowers to HLS C/C++ and uses existing high-level synthesis compilers such as Vitis HLS to generate the final Verilog design. The generated HLS C/C++ contains pragmas specific to the target HCL tool such as Vitis HLS or Quartus. As HCL and HIR target two distinct phases of the HLS pipeline, namely pre-scheduling and post-scheduling respectively, the optimizations performed by HIR and HLS are complementary to each other. Optimizations on scheduled design, such as *delay coalescing* (Section 3.4.2), do not make sense in the HCL dialect since there will be no extra delays added for pipeline balancing before scheduling is complete. HCL's loop transformations can enable HIR's scheduling optimizations such as loop pipelining and overlapped execution of producer-consumer loops. As the HIR compiler optimizes producer-consumer loops better than Vitis HLS (Section 5.2), using HIR as an alternate backend for affine

workloads with constant memory accesses may improve the overall performance of the designs generated by the HCL compiler pipeline. Also, using HIR's scheduler and back-end to lower directly to SystemVerilog eliminates HCL's dependence on vendor-specific HLS tools outside of the MLIR ecosystem.

6.6 Place & route for HLS

Prior literature has studied how to utilize the physical design process in high-level synthesis. Zheng et al. (2014) proposes to iteratively optimize the design by back-annotating both post-placement and routing information. Their approach could increase the f_{max} compared to a non-iterative flow. Cong et al. (2004) developed the regular-distributed-register microarchitecture to solve the multicycle for multicycle on-chip communication. Cong et al. (2018) proposed the*Latte* microarchitecture to insert additional buffers in critical paths. Others have studied methods to predict logic delay [Tan et al. (2015); Guo et al. (2020)] and routing congestion Zhao et al. (2019). AutoBridge [Guo et al. (2021)] proposes a coarse-grained floorplanning during pipelining in high-level synthesis to improve the timing quality of the design. The HIR compiler currently does not use device properties. Logic delay estimates and floorplanning information can be added in the our compilation pipeline for additional optimizations such as retiming or additional buffer insertion in critical paths.

6.7 Summary

Many of the above DSLs Chi et al. (2018); Wang et al. (2021); Li et al. (2020); Chugh et al. (2016b); Lai et al. (2019) and optimization frameworks Pouchet et al. (2013); Zuo et al. (2013b) use the Vivado HLS (currently known as the Vitis HLS) compiler, justifying our comparison against Vitis HLS in the evaluation. Our work is complementary to these DSLs and optimizations. We choose the HIR compiler due to the ease of representing static schedules in HIR intermediate representation, but our ILP-based scheduling

technique can be incorporated into any HLS compiler as another optimization pass.

Chapter 7

Conclusion

In this thesis, we proposed the HIR compiler for high-level synthesis of affine workloads. Affine kernels are very common in image and audio processing as well as machine learning workloads, making them a very interesting domain for hardware acceleration. Our compiler is based on the MLIR compiler infrastructure. We designed an IR called *HIR* to represent hardware accelerators. HIR combines a high-level functional description of the accelerator with a concrete schedule.

As discussed in Chapter 3, HIR introduces the concept of time variables to represent the schedule of operations in a hardware design. Time variables succinctly represent the schedule while abstracting the exact state of machine-based hardware implementation. HIR's compiler uses this decoupling of schedule from the rest of the logic to perform *delay-coalescing* optimization which reduces the number of shift registers required in the final design. The HIR backend uses the schedule to generate appropriate state machine-based controllers in the final SystemVerilog design. The backend's task is to lower HIR into SystemVerilog. It performs loop unrolling, buffer allocation for memref, and state-machine generation to implement the control-flow operations.

HIR also provides its own *memref* data type to represent multi-ported, banked memories as multi-dimensional arrays. The backend's buffer allocation pass assigns each memory bank to a separate hardware buffer and connects the load/store operations to the correct bank. A multi-banked tensor allows multiple parallel access, thereby allowing

the scheduler to avoid port conflicts and exploit the memory-level parallelism available in the design.

As a part of the Affine-to-HIR lowering process, we implemented an ILP-based scheduler that performs memory dependence analysis on the program to find out a valid parallel schedule, as described in Chapter 4. It ensures that the memory dependencies of the original sequential affine program are not violated in the parallel schedule generated by the ILP solver. We use the parallel schedule to emit HIR from the affine program. The scheduler also ensures that there are no port conflicts in the parallel schedule, i.e. two memory operations are not scheduled on the same port at the same clock cycle. The scheduler also calculates the port assignments as a part of the scheduling process.

Two unique features of our scheduler are that it can perform multi-level loop pipelining, and pipeline across producer-consumer loop nests (task-level-pipelining). We use the multi-level loop pipelining when inner unroll loops are present. We perform the pipelining at both the inner and outer loops without first unrolling the inner loop. Delaying the loop-unrolling until after the scheduling is done, reduces the number of constraints in our ILP formulation allowing the ILP to calculate the results quickly. Pipelining across loop nests helps the benchmarks containing producer-consumer loop nests, such as image processing workloads. Our evaluation in Chapter 5 shows that the HIR compiler's scheduler can pipeline producer-consumer loop nests in the presence of arbitrary affine memory access patterns, while Vitis HLS's dataflow optimization performs poorly due to its limited support for more complex memory access patterns and requirement of single-producer-single-consumer dataflow between the loop nests. As our ILP formulation can only handle constant loop bounds, we evaluate our compiler on affine benchmarks with known loop bounds. We argue that constant loop bounds are not a big constraint for hardware accelerator design. Workloads that are run on hardware accelerators are often tiled and each tile is executed on the hardware accelerator. The accelerator utilized parallelism within each tile for performance. For instance, the tensor cores (a systolic array-based matrix-multiplication accelerator) in NVIDIA GPUs can only perform matrix multiplications of a fixed size. Larger matrices are tiled, and each tile is scheduled

on the tensor core individually. Each tile has a fixed size and hence, the loops in the intra-tile space have constant (known at compile time) loop bounds.

7.1 Future Work

The HIR compiler framework presented in this thesis lays the foundation for an LLVM-like extensible high-level synthesis compiler framework. Built on top of MLIR, it is designed with a modular approach where each component such as the ILP scheduler, the optimization passes and the backend verilog code-generator talk to each other using a well-defined intermediate language. This makes it easy to replace/enhance any of the components for better performance as well as introduce new optimization passes. In this section, we cover some of the potential future directions that we find promising.

Polyhedral optimizations Our compiler uses polyhedral analysis during the scheduling pass. But it does not apply any of the traditional loop optimizations. As our compiler's input is the *affine* dialect which is designed for polyhedral analysis and transformation, new polyhedral-based loop transformation techniques to optimize the hardware design such as loop-interchange, tiling and skewing can be introduced to improve parallelism. Loop interchange can increase the overlap between producer-consumer loop nests by reducing the dependence distance between the producer loop iterations and the consumer loop iterations. For example, if a producer loop nest writes to a matrix in row-major order and the consumer from it in column-major order, the overlap may be very small as the consumer has to wait for the producer to write a complete row before it can read the next column element. Reordering the producer or consumer loops can make the two loop nests completely overlapped/pipelined. *Tiling* can also improve overlap between producer-consumer loop nests. For example, in the case of a producer-consumer relationship between two window-based operations (such as convolution or max-pooling), the consumer has to wait till the producer has processed a few rows (Section 2.3). Tiling reduces the size of each row within the tile, thus increasing the overlap.

Non-affine workloads Another interesting area for extending the current work is to deal with non-affine workloads. The affine constraint is primarily due to our scheduler which does a polyhedral-based analysis. The rest of the optimization passes such as bitwidth reduction and delay coalescing work on arbitrary HIR kernels written in HIR. Novel schedulers for non-affine workloads or even domain-specific scheduling can be incorporated into our compiler pipeline to expand the compiler to new domains. Prior work Hegarty et al. (2014) has used custom schedulers for domain-specific languages.

New frontend languages The HIR compiler infrastructure provides a unique opportunity to design new programming languages and extend current languages for high-level synthesis. For instance, Vitis HLS does not support fork-join parallelism. As HIR can represent arbitrary schedules, modifying the scheduler's ILP constraints to ignore data dependency between memory accesses in different threads can allow the compiler to support fork-join parallelism or C++ style multi-threading. Our compiler can also be used as an alternative to Verilog code-generation for domain-specific languages that require precise control over hardware and schedule Hegarty et al. (2014); Durst et al. (2020).

Retiming Our compiler does not have any information about the FPGA's internal characteristics, such as the routing delay and latency of each combinational unit. In the presence of this information, new optimizations such as retiming can be performed to improve the frequency. Generally retiming can only move registers around but can not add extra delay. But since our compiler performs scheduling as well, we can introduce extra delay and rely on the scheduler to generate the final design that preserves the semantics of the original program. For instance, we can replace a floating-point multiplier with another multiplier that has extra pipeline stages for higher frequency. Such optimizations are beyond the scope of classic retiming which is performed on a pre-scheduled Verilog design.

To conclude, we presented *HIR*, an MLIR-based extensible, modular compiler infrastructure for high-level synthesis. Our evaluation shows that our compiler produces designs with comparable resource usage and performance. In the presence of producer-consumer loop nests, we can outperform Vitis HLS's dataflow optimizations. The HIR compiler is designed to enable research in high-level synthesis optimizations and language design. New optimizations can plug into the compiler while benefiting from existing front-ends, benchmarks and backend code-generators, without compromising the precise control over the hardware generation. Similarly, new HLS-specific languages benefit from an existing compiler. In addition, domain-specific languages can introduce new schedulers to the compiler that are tailored to their workloads.

Appendix A

Source code for benchmarks.

In this appendix we provide the source code of the HLS benchmarks used with Vitis HLS.

A.1 Unsharp mask

```
#define IMG_SIZE 32
#define KERNEL_SIZE 5
static float abs(float v) { return v > 0 ? v : -v; }

void split(float output0[IMG_SIZE][IMG_SIZE], float output1[IMG_SIZE][IMG_SIZE],
           float input[IMG_SIZE][IMG_SIZE]) {
#pragma HLS INLINE
  for (int i = 0; i < IMG_SIZE; i++) {
    for (int j = 0; j < IMG_SIZE; j++) {
#pragma HLS PIPELINE
      output0[i][j] = input[i][j];
      output1[i][j] = input[i][j];
    }
  }
}

void convX(float output[IMG_SIZE][IMG_SIZE], float img[IMG_SIZE][IMG_SIZE],
```

```
          float kernel[KERNEL_SIZE]) {
#pragma HLS INLINE
  for (int i = 0; i < IMG_SIZE - KERNEL_SIZE; i++) {
    for (int j = 0; j < IMG_SIZE - KERNEL_SIZE; j++) {
      output[i][j] = 0;
      for (int kk = 0; kk < KERNEL_SIZE; kk++) {
#pragma HLS PIPELINE
        output[i][j] += kernel[kk] * img[i][j + kk];
      }
    }
  }
}

void convY(float output[IMG_SIZE][IMG_SIZE], float img[IMG_SIZE][IMG_SIZE],
           float kernel[KERNEL_SIZE]) {
#pragma HLS INLINE
  for (int i = 0; i < IMG_SIZE - KERNEL_SIZE; i++) {
    for (int j = 0; j < IMG_SIZE - KERNEL_SIZE; j++) {
      output[i][j] = 0;
      for (int kk = 0; kk < KERNEL_SIZE; kk++) {
#pragma HLS PIPELINE
        output[i][j] += kernel[kk] * img[i + kk][j];
      }
    }
  }
}

void sharpen(float output[IMG_SIZE][IMG_SIZE], float img[IMG_SIZE][IMG_SIZE],
             float blury[IMG_SIZE][IMG_SIZE], float weight) {
#pragma HLS INLINE
  for (int i = 0; i < IMG_SIZE; i++) {
    for (int j = 0; j < IMG_SIZE; j++) {
#pragma HLS PIPELINE
      output[i][j] = (1 + weight) * img[i][j] - weight * blury[i][j];
    }
```

```
  }
}

void mask(float output[IMG_SIZE][IMG_SIZE], float img[IMG_SIZE][IMG_SIZE],
          float blury[IMG_SIZE][IMG_SIZE], float sharp[IMG_SIZE][IMG_SIZE],
          float threshold) {
#pragma HLS INLINE
  for (int i = 0; i < IMG_SIZE; i++) {
    for (int j = 0; j < IMG_SIZE; j++) {
#pragma HLS PIPELINE
      output[i][j] =
          (abs(img[i][j] - blury[i][j]) < threshold ? img[i][j] : sharp[i][j]);
    }
  }
}

void unsharp_mask_hls(float img[IMG_SIZE][IMG_SIZE],
                      float mask_img[IMG_SIZE][IMG_SIZE],
                      float kernelDataX[KERNEL_SIZE],
                      float kernelDataY[KERNEL_SIZE]) {

#pragma HLS DATAFLOW
#pragma HLS INLINE recursive
#pragma HLS INTERFACE mode = ap_memory port = img storage_type = rom_1p
#pragma HLS INTERFACE mode = ap_memory port = mask_img storage_type = ram_1p
#pragma HLS INTERFACE mode = ap_memory port = kernelDataX storage_type = rom_1p
#pragma HLS INTERFACE mode = ap_memory port = kernelDataY storage_type = rom_1p

#pragma HLS INTERFACE mode = ap_ctrl_none

  float blurxData[IMG_SIZE][IMG_SIZE];
  float bluryData[IMG_SIZE][IMG_SIZE];
  float bluryData0[IMG_SIZE][IMG_SIZE];
  float bluryData1[IMG_SIZE][IMG_SIZE];
#pragma HLS STREAM depth = 3 type = pipo variable = bluryData1
```

```
  float imgtemp[IMG_SIZE][IMG_SIZE];
  float img0[IMG_SIZE][IMG_SIZE];

  float img1[IMG_SIZE][IMG_SIZE];
#pragma HLS STREAM depth = 4 type = pipo variable = img1
  float img2[IMG_SIZE][IMG_SIZE];
#pragma HLS STREAM depth = 5 type = pipo variable = img2

  float sharpImgData[IMG_SIZE][IMG_SIZE];

  split(imgtemp, img0, img);
  split(img2, img1, imgtemp);
  convX(blurxData, img0, kernelDataX);
  convY(bluryData, blurxData, kernelDataY);
  split(bluryData0, bluryData1, bluryData);
  sharpen(sharpImgData, img1, bluryData0, 3);
  mask(mask_img, img2, bluryData1, sharpImgData, 0.001);
}
```

A.2 Harris

```
#define R 32
#define C 32
//#include <stdio.h>
void split(float output1[R][C], float output2[R][C], float input[R][C]) {
  for (int r = 0; r < R; r++) {
    for (int c = 0; c < C; c++) {
#pragma HLS pipeline
      float v = input[r][c];
      output1[r][c] = v;
      output2[r][c] = v;
    }
  }
}

void funcIx(float ix[R][C], float img[R][C]) {
  float w[2][3];
  w[0][0] = -1 / 12.0;
  w[1][0] = 1 / 12.0;
  w[0][1] = -2 / 12.0;
  w[1][1] = 2 / 12.0;
  w[0][2] = -1 / 12.0;
  w[1][2] = 1 / 12.0;
  float acc;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
      acc = 0;
      for (int u = 0; u < 2; u++) {
        for (int v = 0; v < 3; v++) {
#pragma HLS pipeline
          acc += img[r + 2 * u - 1][c + v - 1] * w[u][v];
        }
      }
      ix[r][c] = acc;
```

```
    }
  }
}

void funcIy(float iy[R][C], float img[R][C]) {
  float w[2][3];
  w[0][0] = -1 / 12.0;
  w[1][0] = 1 / 12.0;
  w[0][1] = -2 / 12.0;
  w[1][1] = 2 / 12.0;
  w[0][2] = -1 / 12.0;
  w[1][2] = 1 / 12.0;
  float acc;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
      acc = 0;
      for (int u = 0; u < 2; u++) {
        for (int v = 0; v < 3; v++) {
#pragma HLS pipeline
          acc += img[r + v - 1][c + 2 * u - 1] * w[u][v];
        }
      }
      iy[r][c] = acc;
    }
  }
}

void funcIxx(float ixx[R][C], float ix[R][C]) {
  float v;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
#pragma HLS pipeline
      v = ix[r][c];
      ixx[r][c] = v * v;
    }
```

```
  }
}

void funcIyy(float iyy[R][C], float iy[R][C]) {
  float v;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
#pragma HLS pipeline
      v = iy[r][c];
      iyy[r][c] = v * v;
    }
  }
}

void funcIxy(float ixy[R][C], float ix[R][C], float iy[R][C]) {
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
#pragma HLS pipeline
      ixy[r][c] = ix[r][c] * iy[r][c];
    }
  }
}

void funcS(float sxx[R][C], float ixx[R][C]) {
  float acc;
  for (int r = 2; r < R - 2; r++) {
    for (int c = 2; c < C - 2; c++) {
      acc = 0;
      for (int u = 0; u < 3; u++) {
        for (int v = 0; v < 3; v++) {
#pragma HLS pipeline
          acc += ixx[r + u - 1][c + v - 1];
        }
      }
      sxx[r][c] = acc;
```

```
    }
  }
}

void funcDet(float det[R][C], float sxx[R][C], float syy[R][C],
             float sxy[R][C]) {
  float v;
  for (int r = 2; r < R - 2; r++) {
    for (int c = 2; c < C - 2; c++) {
#pragma HLS pipeline
      v = sxy[r][c];
      det[r][c] = sxx[r][c] * syy[r][c] - v * v;
    }
  }
}

void funcTrace(float trace[R][C], float sxx[R][C], float syy[R][C]) {
  for (int r = 2; r < R - 2; r++) {
    for (int c = 2; c < C - 2; c++) {
#pragma HLS pipeline
      trace[r][c] = sxx[r][c] + syy[r][c];
    }
  }
}

void funcHarris(float harris[R][C], float det[R][C], float trace[R][C]) {
  float v;
  for (int r = 2; r < R - 2; r++) {
    for (int c = 2; c < C - 2; c++) {
#pragma HLS pipeline
      v = trace[r][c];
      harris[r][c] = det[r][c] - (float)0.04 * v * v;
    }
  }
}
```

```
void harris_hls(float harris[R][C], float img[R][C]) {
#pragma HLS DATAFLOW
#pragma HLS INLINE recursive
#pragma HLS INTERFACE mode = ap_memory port = img storage_type = ram_1p
#pragma HLS INTERFACE mode = ap_memory port = harris storage_type = ram_1p

  float ix[R][C];
  float iy[R][C];
  float ixx[R][C];
  float iyy[R][C];
  float ixy[R][C];
  float sxx[R][C];
  float syy[R][C];
  float sxy[R][C];
#pragma HLS STREAM depth = 3 type = pipo variable = sxy
  float det[R][C];
  float trace[R][C];

  funcIx(ix, img);
  funcIy(iy, img);
  funcIxx(ixx, ix);
  funcIyy(iyy, iy);
  funcIxy(ixy, ix, iy);
  funcS(sxx, ixx);
  funcS(syy, iyy);
  funcS(sxy, ixy);
  funcDet(det, sxx, syy, sxy);
  funcTrace(trace, sxx, syy);
  funcHarris(harris, det, trace);
}
```

A.3 DUS

```
#define IMG_SIZE 32
void downsample(float out[IMG_SIZE / 2][IMG_SIZE / 2],
                float img[IMG_SIZE][IMG_SIZE]) {
  float temp[IMG_SIZE][IMG_SIZE / 2];
  #pragma HLS bind_storage variable = temp type = ram_s2p impl = bram

  for (int i = 1; i < IMG_SIZE; i++) {
    for (int j = 1; j < IMG_SIZE / 2; j++) {
#pragma HLS pipeline
      temp[i][j] = (img[i][2 * j - 1] + 2 * img[i][2 * j] + img[i][2 * j + 1]) *
                   (float)0.25;
    }
  }

  for (int i = 1; i < IMG_SIZE / 2; i++) {
    for (int j = 1; j < IMG_SIZE / 2; j++) {
#pragma HLS pipeline
      out[i][j] = (temp[2 * i - 1][j] + (float)2.0 * temp[2 * i][j] +
                   temp[2 * i + 1][j]) *
                  (float)0.25;
    }
  }
}

void upsample(float out[IMG_SIZE][IMG_SIZE],
              float img[IMG_SIZE / 2][IMG_SIZE / 2]) {
  float temp[IMG_SIZE / 2][IMG_SIZE];
  float v;
  #pragma HLS bind_storage variable = temp type = ram_s2p impl = bram

  for (int i = 1; i < IMG_SIZE / 2; i++) {
    for (int j = 1; j < IMG_SIZE / 2 - 1; j++) {
#pragma HLS pipeline
```

```
      v = img[i][j];
      temp[i][2 * j] = v;
      temp[i][2 * j + 1] = v + img[i][j + 1] * (float)0.5;
    }
  }

  for (int i = 1; i < IMG_SIZE / 2 - 1; i++) {
    for (int j = 1; j < IMG_SIZE; j++) {
#pragma HLS pipeline
      out[2 * i][j] = temp[i][j];
      out[2 * i + 1][j] = (temp[i][j] + temp[i + 1][j]) * (float)0.5;
    }
  }
}

void dus_hls(float dus[IMG_SIZE][IMG_SIZE], float img[IMG_SIZE][IMG_SIZE]) {
  float down[IMG_SIZE / 2][IMG_SIZE / 2];
#pragma HLS DATAFLOW
#pragma HLS INLINE recursive
#pragma HLS interface mode = ap_memory port = img storage_type = rom_1p
#pragma HLS interface mode = ap_memory port = dus storage_type = ram_1p
#pragma HLS bind_storage variable = down type = ram_2p impl = bram

  downsample(down, img);
  upsample(dus, down);
}
```

A.4 Optical flow

```
#define R 32
#define C 32
//#include <stdio.h>
void split(float output1[R][C], float output2[R][C], float input[R][C]) {
  for (int r = 0; r < R; r++) {
    for (int c = 0; c < C; c++) {
#pragma HLS pipeline
      float v = input[r][c];
      output1[r][c] = v;
      output2[r][c] = v;
    }
  }
}

void funcIx(float ix[R][C], float img[R][C]) {
  float w[2][3];
  w[0][0] = -1 / 12.0;
  w[1][0] = 1 / 12.0;
  w[0][1] = -2 / 12.0;
  w[1][1] = 2 / 12.0;
  w[0][2] = -1 / 12.0;
  w[1][2] = 1 / 12.0;
  float acc;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
      acc = 0;
      for (int u = 0; u < 2; u++) {
        for (int v = 0; v < 3; v++) {
#pragma HLS pipeline
          acc += img[r + 2 * u - 1][c + v - 1] * w[u][v];
        }
      }
      ix[r][c] = acc;
```

```
    }
  }
}

void funcIy(float iy[R][C], float img[R][C]) {
  float w[2][3];
  w[0][0] = -1 / 12.0;
  w[1][0] = 1 / 12.0;
  w[0][1] = -2 / 12.0;
  w[1][1] = 2 / 12.0;
  w[0][2] = -1 / 12.0;
  w[1][2] = 1 / 12.0;
  float acc;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
      acc = 0;
      for (int u = 0; u < 2; u++) {
        for (int v = 0; v < 3; v++) {
#pragma HLS pipeline
          acc += img[r + v - 1][c + 2 * u - 1] * w[u][v];
        }
      }
      iy[r][c] = acc;
    }
  }
}

void funcIt(float it[R][C], float img[R][C], float prev[R][C]) {

  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
      it[r][c] = img[r][c] - prev[r][c];
    }
  }
}
```

```
void funcIaa(float iaa[R][C], float ia[R][C]) {
  float v;
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
#pragma HLS pipeline
      v = ia[r][c];
      iaa[r][c] = v * v;
    }
  }
}

void funcIab(float iab[R][C], float ia[R][C], float ib[R][C]) {
  for (int r = 1; r < R - 1; r++) {
    for (int c = 1; c < C - 1; c++) {
#pragma HLS pipeline
      iab[r][c] = ia[r][c] * ib[r][c];
    }
  }
}

void funcS(float sxx[R][C], float ixx[R][C]) {
  float acc;
  for (int r = 2; r < R - 2; r++) {
    for (int c = 2; c < C - 2; c++) {
      acc = 0;
      for (int u = 0; u < 3; u++) {
        for (int v = 0; v < 3; v++) {
#pragma HLS pipeline
          acc += ixx[r + u - 1][c + v - 1];
        }
      }
      sxx[r][c] = acc;
    }
  }
```

```
}

void funcDet(float det[R][C], float sxx[R][C], float syy[R][C],
             float sxy[R][C]) {
  float v;
  for (int r = 2; r < R - 2; r++) {
    for (int c = 2; c < C - 2; c++) {
#pragma HLS pipeline
      v = sxy[r][c];
      det[r][c] = sxx[r][c] * syy[r][c] - v * v + 1e-05;
    }
  }
}

void funcFlowLK(float flow[R][C][2], float det[R][C], float sxx[R][C],
                float syy[R][C], float sxy[R][C], float sxt[R][C],
                float syt[R][C]) {
  for (int r = 4; r < R - 4; r++) {
    for (int c = 4; c < C - 4; c++) {
#pragma HLS pipeline
      float d = 1 / det[r][c];
      flow[r][c][0] = d * (sxy[r][c] * syt[r][c] - syy[r][c] * sxt[r][c]);
      flow[r][c][1] = d * (sxx[r][c] * syt[r][c] - sxy[r][c] * sxt[r][c]);
    }
  }
}

void optical_flow_hls(float flow[R][C][2], float img[R][C], float prev[R][C]) {
#pragma HLS DATAFLOW
#pragma HLS INLINE recursive
#pragma HLS INTERFACE mode = ap_memory port = img storage_type = ram_1p
#pragma HLS INTERFACE mode = ap_memory port = prev storage_type = ram_1p
#pragma HLS INTERFACE mode = ap_memory port = flow storage_type = ram_1p

  float img1[R][C];
```

```
float img2[R][C];
float img3[R][C];
float img4[R][C];
float ix[R][C];
float ix1[R][C];
float ix2[R][C];
float ix3[R][C];
float ix4[R][C];
float iy[R][C];
float iy1[R][C];
float iy2[R][C];
float iy3[R][C];
float iy4[R][C];
float ixx[R][C];
float iyy[R][C];
float ixy[R][C];
float sxx[R][C];
float sxx1[R][C];
float sxx2[R][C];
float syy[R][C];
float syy1[R][C];
float syy2[R][C];
float sxy[R][C];
float sxy1[R][C];
float sxy2[R][C];
float it[R][C];
float it1[R][C];
float it2[R][C];
float ixt[R][C];
float iyt[R][C];
float sxt[R][C];
float syt[R][C];
float det[R][C];

split(img1, img2, img);
```

```
  split(img3, img4, img2);
  funcIx(ix, img1);
  split(ix1, ix2, ix);
  split(ix3, ix4, ix2);
  funcIy(iy, img3);
  split(iy1, iy2, iy);
  split(iy3, iy4, iy2);
  funcIaa(ixx, ix1);
  funcIaa(iyy, iy1);
  funcIab(ixy, ix3, iy3);
  funcS(sxx, ixx);
  split(sxx1, sxx2, sxx);
  funcS(syy, iyy);
  split(syy1, syy2, syy);
  funcS(sxy, ixy);
  split(sxy1, sxy2, sxy);
  funcDet(det, sxx1, syy1, sxy1);
  funcIt(it, img4, prev);
  split(it1, it2, it);
  funcIab(ixt, ix4, it1);
  funcIab(iyt, iy4, it2);
  funcS(sxt, ixt);
  funcS(syt, iyt);
  funcFlowLK(flow, det, sxx2, syy2, sxy2, sxt, syt);
}
```

A.5 2mm

```
#define _PB_NI 8
#define _PB_NJ 8
#define _PB_NL 8
#define _PB_NK 8
#define DATA_TYPE float

void kernel_2mm_hls(DATA_TYPE alpha, DATA_TYPE beta,
                    DATA_TYPE tmp[_PB_NI][_PB_NJ], DATA_TYPE A[_PB_NI][_PB_NK],
                    DATA_TYPE B[_PB_NK][_PB_NJ], DATA_TYPE C[_PB_NJ][_PB_NL],
                    DATA_TYPE D[_PB_NI][_PB_NL]) {
//Dataflow can't handle read/write to external rams.

#pragma HLS INTERFACE mode = ap_memory port = A storage_type = rom_1p
#pragma HLS INTERFACE mode = ap_memory port = B storage_type = rom_1p
#pragma HLS INTERFACE mode = ap_memory port = C storage_type = rom_1p
#pragma HLS INTERFACE mode = ap_memory port = D storage_type = ram_s2p
#pragma HLS INTERFACE mode = ap_memory port = tmp storage_type = ram_s2p
  int i, j, k;

  /* D := alpha*A*B*C + beta*D */
  DATA_TYPE acc;
  for (i = 0; i < _PB_NI; i++)
    for (j = 0; j < _PB_NJ; j++) {
      acc = 0;
      for (k = 0; k < _PB_NK; ++k) {
#pragma HLS pipeline
        acc += alpha * A[i][k] * B[k][j];
      }
      tmp[i][j] = acc;
    }

  DATA_TYPE acc2;
  for (i = 0; i < _PB_NI; i++)
```

```
    for (j = 0; j < _PB_NL; j++) {
      acc2 = D[i][j] * beta;
      for (k = 0; k < _PB_NJ; ++k) {
#pragma HLS pipeline
        acc2 += tmp[i][k] * C[k][j];
      }
      D[i][j] = acc2;
    }
}
```

Bibliography

Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-Based Compiler Flow for System-Level Design and Hardware Acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/3508352.3549424

Christophe Alias, Alain Darte, and Alexandra Plesco. 2010. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators an experience with the Altera C2H HLS tool. In *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. 329–332. https://doi.org/10.1109/ASAP.2010.5540967

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.

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012a. 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

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012b. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

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

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

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011a. 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

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011b. 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

Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake

City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*. 262–263.

Y. Chi, J. Cong, P. Wei, and P. Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *ICCAD*.

Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

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

Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016b. A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 327–338. https://doi.org/10.1145/2967938.2967969

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

J. Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang. 2004. Architecture and synthesis for on-chip multicycle communication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 4 (2004), 550–564. https://doi.org/10.1109/TCAD.2004.825872

Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (San Francisco Airport, CA, USA) *(ISLPED '16)*. Association for Computing Machinery, New York, NY, USA, 154–155. https://doi.org/10.1145/2934583.2953984

Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 51 (aug 2022), 42 pages. https://doi.org/10.1145/3530775

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

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

Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality Aware Transformation for High-Level Synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 125–128. https://doi.org/10.1109/FCCM.2018.00028

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.

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.

Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 315–326. https://doi.org/10.1109/CGO51591.2021.9370315

Jan Decaluwe. 2004. MyHDL: A Python-Based Hardware Description Language. *Linux J.* 2004, 127 (nov 2004), 5.

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

Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

Jean-Philippe Fricker. 2022. The Cerebras CS-2: Designing an AI Accelerator around the World's Largest 2.6 Trillion Transistor Chip. In *Proceedings of the 2022 International Symposium on Physical Design* (Virtual Event, Canada) *(ISPD '22)*. Association for Computing Machinery, New York, NY, USA, 71. https://doi.org/10.1145/3505170.3511036

Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*

(Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 84–93. https://doi.org/10.1145/3289602.3293906

Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. 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, 81–92. https://doi.org/10.1145/3431920.3439289

Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218718

Christopher G. Harris and M. J. Stephens. 1988. A Combined Corner and Edge Detector. In *Alvey Vision Conference*.

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

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

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

Intel. [n. d.]. Intel FPGA SDK for OpenCL. https://www.intel.in/content/www/in/en/software/program for-opencl/overview.html.

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

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

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

Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 461–475. https://doi.org/10.1145/3173162.3173176

Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

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.

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)*.

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]

Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. 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, 51–57. https://doi.org/10.1145/3373087.3375320

Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 369–381. https://doi.org/10.1145/2694344.2694358

Yanqiang Liu, Yao Li, Weilun Xiong, Meng Lai, Cheng Chen, Zhengwei Qi, and Haibing Guan. 2017. Scala Based FPGA Design Flow (Abstract Only). In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 286. https://doi.org/10.1145/3020078.3021762

Bruce D. Lucas and Takeo Kanade. 1981. An Iterative Image Registration Technique with an Application to Stereo Vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2* (Vancouver, BC, Canada) *(IJCAI'81)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 674–679.

Kingshuk Majumder and Uday Bondhugula. 2022. HIR source code. https://github.com/mcl-csa/hir-dev

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *CoRR* abs/1803.04014 (2018). arXiv:1803.04014 http://arxiv.org/abs/1803.04014

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

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.

Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 429–443. https://doi.org/10.1145/2694344.2694364

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.

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

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

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

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

Debjit Pal, Yi-Hsiang Lai, Shaojie Xiang, Niansong Zhang, Hongzheng Chen, Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Louis-Noël Pouchet, and Zhiru Zhang. 2022. Accelerator Design with Decoupled Hardware Customizations: Benefits and Challenges: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1351–1354. https://doi.org/10.1145/3489517.3530681

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

Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

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.

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]

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

Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. 2015. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 190–199. https://doi.org/10.1145/2684746.2689063

Xingyu Tian, Zhifan Ye, Alec Lu, Licheng Guo, Yuze Chi, and Zhenman Fang. 2023. SASA: A Scalable and Automatic Stencil Acceleration Framework for Optimized Hybrid Spatial and Temporal Parallelism on HBM-Based FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* (jan 2023). https://doi.org/10.1145/3572547 Just Accepted.

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

Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022a. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 741–755. https://doi.org/10.1109/HPCA53966.2022.00060

Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022b. ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1355–1358. https://doi.org/10.1145/3489517.3530631

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.

Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine Learning Based Routing Congestion Prediction in FPGA High-Level Synthesis. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, Jürgen Teich and Franco Fummi (Eds.). IEEE, 1130–1135. https://doi.org/10.23919/DATE.2019.8714724

Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. 2014. Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2554688.2554775

Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013a. Improving polyhedral code generation for high-level synthesis. In

2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6659002

Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013b. Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2435264.2435271