Practical Exocompilation for Performance Engineers in User-Schedulable Languages

by

Kevin Qian

B.S. Electrical Engineering and Computer Science, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

© 2024 Kevin Qian. This work is licensed under a CC BY-NC-ND 4.0 license.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Kevin Qian
Department of Electrical Engineering and Computer Science
August 16, 2024

Certified by: Jonathan Ragan-Kelley
Associate Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Practical Exocompilation for Performance Engineers in User-Schedulable Languages

by

Kevin Qian

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

High performance computing libraries provide efficient implementations of common computational kernels. Traditionally, such libraries are written in C or assembly. User-schedulable languages provide performance engineers a productive way to optimize these kernels with well-designed interfaces which provide users control over performance-relevant decisions and automate unnecessary concerns. Often, this is a tradeoff: too much control with too little automation is tedious to program, and too much automation with too little control will hinder obtaining peak performance. The principle of *exocompilation* advocates for one end of the extreme: to give performance engineers maximal control over code execution so they can maximize performance, its current implementation in existing systems is impractical to use. This thesis broadly explores ways to make exocompilation a practical solution for performance engineers. We show that providing more control does not necessitate sacrificing automation, as long as the language is designed so that users can build their own automation. We explore the necessary design features to enable such a system, demonstrate the types of automation users can build in the system, and brainstorm ways to further push the amount of control user-schedule languages expose to the user.

Thesis supervisor: Jonathan Ragan-Kelley
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank my advisor, Jonathan Ragan-Kelley. Without a doubt, working with Jonathan has greatly shaped my views on performance engineering and compilers. As an advisor, Jonathan has an innate talent for pinpointing the crux of problems and has always steered me in the right direction. He is also an amazing story teller, able to clearly articulate my key points back to me better than I could have said it myself. At the end of my MIT journey, I am undoubtedly a significantly better researcher, writer, and presenter.

This project would never have made it this far without the support of my Exo collaborators: Yuka Ikarashi, Samir Droubi, Gilbert Bernstein, Alex Reinking, and William Brandon. Yuka was my UROP mentor when I first joined the project, and was always available and excited to teach, advise, and help me throughout the past three years. Samir, my fellow MEng student on the project, has been a wonderful friend and teammate. I will miss all our bantering and debating. Gilbert, despite being remote most of the time, always found time to advise the project, patiently teaching us to be better researchers and encouraging us to push the project's limits. Alex and I collaborated on the cursors project, working together to flesh out what ultimately became the final system design. William helped immensely in brainstorming the ideas for the project on scheduling over parallel processors.

I have been blessed to be a part of a wonderful research community at MIT. To my fellow members of the VCLS lab, I loved both the research discussions and our spontaneous lab shenanigans. Visual computing tea time was a lovely weekly break for me to relax and hang out with other researchers in the community. The MIT ML Systems reading group hosted many thoughtful discussions (and dinners) weekly. My MEng was partially funded through TAing for computer graphics with Professor Mina Luković. Computer graphics was one of my favorite classes as as student, and I am grateful for the opportunity to teach it.

Outside of research, I thank my amazing friends for all the wonderful times we shared. I would especially like to acknowledge Ellie Feng, Marco Nocito, and Jessica Ding for being the people I could rely on most during my Master's degree. In addition, the MIT Lion Dance, Asian Dance Team, and Gymnastics clubs gave me unforgettable experiences. Though I had zero experience with all prior to college, these clubs welcomed me with open arms and taught me from the basics.

Finally, I would to thank my parents Minglei Cui and Feng Qian, my brother Timothy Qian, and my grandmother Hong Zhu for supporting me throughout my whole life. They never hesitated when it came to dedicating time and resources to help me grow and achieve my goals. I was only privileged to get this far thanks to decades of their effort, and I will forever appreciate it. This thesis belongs to all of us.

Contents

Title page 1

Abstract 3

Acknowledgments 5

List of Figures 9

List of Tables 11

1 Introduction 13

2 Related Work 15
- 2.1 User-schedulable Languages (USLs) 15
- 2.2 Exo Overview 16
- 2.3 GPU Programming Models 17

3 Implementing Stable References in a USL 19
- 3.1 Cursor Representation 20
- 3.2 Stable References via Cursor Forwarding 20
  - 3.2.1 Forwarding Behavior of Atomic Edits 21
  - 3.2.2 Composing Atomic Edits to Build Scheduling Primitives 23
- 3.3 Simplifying Implementation of Scheduling Operations 24

4 Reproducing Halide as an Exo Library 26
- 4.1 Overview of Halide by Example 26
- 4.2 Implementing Halide Scheduling Operations in Exo 27
  - 4.2.1 Ordering Halide-like Scheduling Operations in Exo 29
  - 4.2.2 Bounds Inference 30
  - 4.2.3 Compute At 31
  - 4.2.4 Store At 34
- 4.3 Evaluation 36
  - 4.3.1 3×3 Box Blur 36
  - 4.3.2 Unsharp Masking 37
- 4.4 Areas for improvement 41
  - 4.4.1 Generation of Prologue Loops 41

4.4.2 Handling Tail Cases . . . 42
4.4.3 Instruction Selection with Constants . . . 43

5 Scheduling for Parallel Processors **44**
5.1 Scheduling Simple Parallel For Loops Without Races . . . 44
5.2 Scheduling for GPUs . . . 45
5.2.1 Parallel Hierarchy . . . 45
5.2.2 Memories annotated with scopes . . . 47
5.2.3 Scheduling Synchronization Operations . . . 48
5.2.4 Scheduling Asynchronous Operations . . . 48

6 Conclusion and Future Work **49**

References **50**

List of Figures

2.1 Schedule for tiling the blur algorithm into 32×256 tiles. 17
2.2 Exo object code after applying the tile schedule from Figure 2.1 17

3.1 Example illustration of a cursor pointing to the read of `y` in the code on the left. Its corresponding path in the AST is shown on the right. 20
3.2 Forwarding behavior when inserting an IR fragment into a gap. 21
3.3 Forwarding behavior when deleting a block. 22
3.4 Forwarding behavior when replacing a block of statements. 22
3.5 Forwarding behavior when moving a block of statements. The red highlighted code on the left is moved to the gap, resulting in the green highlighted code on the right. 22
3.6 Forwarding behavior when moving a block of statements. The red highlighted code on the left is moved to the gap, resulting in the green highlighted code on the right. 23
3.7 Desired forwarding behavior for `reorder_loops`. 23
3.8 Implementation of `reorder_loops` as a composition of four atomic edits in order to achieve the desired forwarding behavior shown in Figure 3.7. 24

4.1 While both schedules invoke `producer.compute_at(consumer, y)` to fuse the computations at the `y` loop level, they store the result at different locations. The left schedule stores the producer at the same loop level as `compute_at`, while the right schedule calls `store_root()`, which stores the producer above all loop levels. This results in different generated code. 29
4.2 Decomposing Halide's `compute_at` scheduling operation into Exo's finer-grained primitive scheduling operations for the blur example's `blur_x.compute_at(blur_y, x)`. 32
4.3 Performance comparison between code generated by Exo schedules and code generated by expert-written Halide schedules. The Exo schedules exhibit competitive performance across a range of image sizes. 41
4.4 Different possibilities for generated prologue loops. Ideally, we would generate the code on the left, both to be consistent with the assumptions made at the beginning of Section 4.2 and for easier design of the scheduling operations. However, with current Exo's capabilities, we are only able to generate the code on the right. 42
4.5 Examples of different tail strategies that are necessary for efficient parallelization. Neither one can currently be supported in Exo. 42

4.6 Example of invalid unification. The body of the procedure on the left does not unify with the body of the instruction on the right because unification does not equate a constant float with a scalar variable. 43

5.1 Parallelizing a for loop in Exo. 44
5.2 Managing the parallel hierarchy in CUDA vs in Exo 46
5.3 Managing local memory usage in CUDA vs in Exo 47

List of Tables

Chapter 1

Introduction

Modern machine learning and visual computing systems demand high-performance code. Faster code not only enables more advanced neural network models and image processing pipelines, but also improves response time delay in real-time, latency-sensitive systems. High-performance code must be specialized to the hardware it executes on. In recent years, there has been an explosion of specialized accelerators such as Apple's Neural Engine, Intel's AMX Engine, or Qualcomm's Hexagon DSP which offer faster execution and higher energy efficiency than traditional, general-purpose hardware. Between the many applications and hardware targets, there is a huge demand for high-performance code.

Performance engineering, the process of optimizing a program, consists of repeatedly modifying it into new programs which compute the *same result* more efficiently. Traditionally, performance engineering is done in low-level languages such as C or assembly, which allow programmers to finely control the execution of their code. However, performance engineering in these languages is tedious and error-prone. For example, matrix-multiply, a core algorithm behind many machine learning computing applications, can be expressed as three nested loops in only a few lines of code. However, a peak performance implementation of matrix-multiply in libraries such as IntelMKL or OpenBLAS can take ten thousands lines of hand-written code.

User-schedulable languages (USLs) such as Halide, TVM, TACO, and Taichi offer a promising solution to increase programmer productivity when doing such optimization [1]–[4]. USLs reify this optimization process into an explicit *scheduling meta-program* that transforms an underlying *object program* into a better-performing one. They also guarantee that scheduling transformations preserve the equivalence of the object program, allowing performance engineers to focus on optimization strategies instead of painstakingly ensuring correctness [5].

Most existing USLs are designed with a fixed set of scheduling operations that target a particular application domain (e.g., image processing) and/or class of optimization (e.g., loop transformations). The goal of these fixed sets of scheduling operations is to provide explicit *control* over key optimization choices while abstracting away tedious details. Well designed USLs carefully choose a delineation between *automated* optimizations and *user-scheduled* choices exposed as scheduling operations. For instance, the Halide language automates bounds inference, register allocation, and instruction selection (to name a few) but gives users explicit control over loop tiling, fusion, and work vs. locality tradeoffs. When the design works, it shields performance engineers from unnecessary concerns, improving productivity.

In practice, however, it is difficult to design this automation-control boundary perfectly. When

the design breaks down, performance engineers have no way to cross the boundary and must drop down to C or assembly code (or even modify the compiler) to regain control. These failures can happen either because an optimization is not expressible with the control provided, or because of the limited automation capability. For example, even in a decade old system such as Halide, automated fixed-point vector instruction selection has seen significant improvement in recent years [6]–[8]. Furthermore, users have no way to leverage Halide's well-designed scheduling interface to target novel hardware accelerators without extending the compiler.

Even though existing USLs strive to provide sufficient control to performance engineers, having a fixed set of control operations is inherently brittle and cannot accommodate rapidly evolving application and hardware changes. Thus, the Exo USL explored the idea of *exocompilation* – externalizing components of the compiler to further push the boundary towards more control [9] – by giving performance engineers control over target-specific code generation for matrix accelerators such as Gemmini [10] or vector architectures such as AVX2 and AVX512. In order to maximize control, Exo exposes *primitive* scheduling operations which perform small, local transformations to the code such as "divide this loop into an inner and outer loop". While the resulting programming interface is able to target a broad range of hardware architectures, it was also tedious to program, requiring a scheduling operation for each minor transformation on the code. The resulting exocompilation interface was impractical because automation was sacrificed for control.

However, this need not be the case. In an unpublished manuscript, we propose that the automation-control tradeoff exists because USLs were not designed for growth [11]. While most general purpose programming languages provide users the ability to encapsulate useful, shared code into library functions, existing USLs lack such facilities. Thus, it is difficult to encapsulate reusable schedule fragments and extend the level of automation beyond the interface exposed by the USL designers. To design a USL for such growth, we augmented the Exo USL and empowered users to build new scheduling operations from Exo's set of *primitive* scheduling operations. By composing these safe primitives, users can automate common optimizations beyond the automation exposed in Exo's scheduling interface. When more control is required, users can fall back to using the primitives scheduling operations. This design allows users to choose whether they want to use more automation or take more control, rather than confining them to a particular automation-control boundary.

This thesis broadly describes contributions towards making exocompilation more practical for performance engineers, either by enabling users to extend automation within the language to improve productivity or extending exocompilation to encompass parallel computing architectures to give users even more control. In Chapter 2, we provide background on the development of USLs and discuss key design issues which prevent users from extending automation within those languages. In addition, we also present characteristics of existing GPU programming models. In Chapter 3, we describe the new implementation of Exo which enable users to build reusable schedule fragments within the USL [11]. In Chapter 4, we demonstrate an example of an automation library built in our USL by reproducing the Halide language's scheduling operations. In Chapter 5, we further push the idea of exocompilation in USLs by externalizing control over code generation for parallel computing architectures, with a particular emphasis on GPUs.

Chapter 2

Related Work

2.1 User-schedulable Languages (USLs)

Historically, high-performance computing kernels in machine learning and image processing were often hand-written by expert performance engineers in low-level languages such as C or assembly. During optimization, users must make careful decisions about how to best exploit parallelism, organize the computation, and allocate memory to intermediate results to maximize performance. We call this choice of how to structure the computation as the *schedule* of the computation. Performance engineers must tailor the schedule of the computation to best optimize for each potential hardware target. This process of optimally scheduling is very complex, time-consuming, and error-prone. While compilers offer modest performance improvements, decades of compilers research has yet to surpass hand-written code for many standard kernels such as matrix multiplication.

User schedulable languages (USLs) were introduced as a way to productively navigate the performance tradeoffs when optimizing such kernels, and was first popularized by the Halide language in 2012 [12]. USLs make a clear delineation between the *algorithm*, which defines the mathematical computation that should be computed, from the *schedule*, which defines how this computation should be structured. The schedule is composed of a sequence of *scheduling operations* which dictate how to restructure the computation. USLs guarantee that scheduling decisions do not deviate from the algorithm by checking functional equivalence. Since Halide's inception, many other USLs have been inspired, such as Lift [13], Elevate [14], TVM [2], Taco [3], Taichi [4], and Exo [9], the central language in this paper.

Existing USLs were designed for users to easily write one schedule per algorithm and hardware target. They would design their scheduling operations to be a suitable balance between performance decisions under user control and those automated by the language compiler. The Exo USL introduced the concept of *exocompilation*: externalizing as many performance-relevant decisions (such as hardware instruction selection) to be under the user's control. As a result, the scheduling operations in Exo are small, local transformations, and often referred to as scheduling *primitives*. These scheduling primitives give users fine-grained control over the code's execution. In the next section, we will give a brief overview of Exo's syntax.

2.2 Exo Overview

Exo Algorithm We introduce Exo by example through a 3×3 box blur, which we will revisit again in Chapter 4.

```
@proc
def exo_blur(W: size, H: size, blur_y: ui16[H, W] @ DRAM, inp: ui16[H+2, W+2] @ DRAM):
  assert H % 32 == 0
  assert W % 256 == 0
  blur_x: ui16[H + 2, W] @ DRAM
  for y in seq(0, H + 2):
    for x in seq(0, W):
      blur_x[y, x] = inp[y, x] + inp[y, x+1] + inp[y, x+2]
  for y in seq(0, H):
    for x in seq(0, W):
      blur_y[y, x] = blur_x[y, x] + blur_x[y+1, x] + blur_x[y+2, x]
```

The `exo_blur` function is annotated with `@proc`, marking it as an Exo Procedure. Both procedure arguments and variable declarations follow the syntax

$$\langle name \rangle : \langle type \rangle [\langle size \rangle] \ @ \ \langle memory \rangle.$$

For example, the procedure argument `inp` has type `ui16`, short for unsigned 16-bit integers The $\langle size \rangle$s may be constants, or refer dependently to other arguments (e.g., `[H, W]`) as they do here. The `@` symbol precedes a memory space identifier, specifying in what memory the variable or argument resides (e.g., `@DRAM`). The **`assert`**s guarantee that all input arrays are even multiples of 32×256 tiles, which can be exploited during scheduling. Finally, `for x in seq(0, W)` is a *sequential* for loop that iterates from 0 to $W - 1$, inclusive.

Exo Schedule In Exo, schedules are Python meta-programs that take a procedure (e.g., `exo_blur`) as input and return a functionally equivalent, rewritten procedure as output. For example, the `divide_loop(p, loop, factor, new_vars, ...)` scheduling operator divides a single loop of n iterations into a pair of outer and inner loops of $n/factor$ and $factor$ iterations. It takes the procedure, the loop to be divided, the division factor, new iterator variable names, and optional "tail strategy" arguments for handling cases where n does not divide evenly by $factor$, and returns the modified procedure.

This raises a natural question: how can we specify which loop we want to divide? As we will explore in Section 3, the problem of *referencing* is a fundamental issue with many complexities in scheduling languages. For now, we will only refer to loops by name using strings.

Suppose we want to write a schedule which will tile the blur computation, a common optimization for improving data locality. Schedules can often be expressed as a series of rewrites, and these rewrites can be composed sequentially. Each scheduling operation returns a new procedure that can be further scheduled by the next operation. For example, we can tile `exo_blur` by sequentially composing `divide_loop` and `lift_scope` primitives as follows.

```
p = divide_loop(exo_blur, 'y', 32, ['y','yi'], perfect=True)
p = divide_loop(p, 'x', 256, ['x','xi'], perfect=True)
tiled_exo_blur = lift_scope(p, 'x')
```

Figure 2.1: Schedule for tiling the blur algorithm into 32×256 tiles.

Since we know `H` and `W` are perfectly divisible by 32 and 256, respectively, we can omit code for handling tail cases by passing `perfect=True` to `divide_loop`. `lift_scope` takes either a **for** loop or an **if** statement and interchanges it with the surrounding **for** or **if**. This simple composition of primitives yields the `tiled_exo_blur` object code as shown below:

```
@proc
def tiled_exo_blur(W: size, H: size, blur_y: ui16[H, W] @ DRAM,
                                      inp: ui16[H+2, W+2] @ DRAM):
  assert H % 32 == 0
  assert W % 256 == 0
  blur_x: ui16[H + 2, W] @ DRAM
  for y in seq(0, H + 2):
    for x in seq(0, W):
      blur_x[y, x] = inp[y, x] + inp[y, x+1] + inp[y, x+2]
  for y in seq(0, H/32):
    for x in seq(0, W/256):
      for yi in seq(0, 32):
        for xi in seq(0, 256):
          blur_y[32*y+yi, 256*x+xi] = ...
```

Figure 2.2: Exo object code after applying the tile schedule from Figure 2.1

2.3 GPU Programming Models

In this section, we discuss several popular programming models for writing high-performance GPU code. Of all of the parallel programming models we will discuss, CUDA is most closely aligned with the principle of exocompilation because it maximizes user control. Consequently, most of our design of a parallel programming model in Chapter 5 will be based on supporting features available in CUDA. Nonetheless, in the same way that many of the following programming models are extensions of CUDA, we can design extensions of the Exo system based on the other programming models.

CUDA Developed by NVIDIA specifically to provide programmability to their GPUs, CUDA has seen immense success in the past decade. CUDA exposes a single instruction, multiple thread (SIMT) programming model: users write thread-level programs, and the processor executes them

in groups of 32 known as warps [15]. CUDA gives the user an extremely high degree of control over the execution of their programs on GPUs. To give a few examples particularly relevant to writing high-performance computing kernels, CUDA offers a Cooperative Groups API for easily organizing the subdivision of work across parallel groups, specialized accelerated instructions such as warp-group matrix multiply accumulate and asynchronous memory copies, and various synchronization primitives such as barriers and pipelines. Many state-of-the-art performing kernels are written in CUDA.

However, writing a high-performance kernel in CUDA is difficult: it is easy to introduce bugs. As a consequence, several new programming languages have been designed to facilitate GPU programming.

Triton Triton is another popular GPU programming language [16] which exposes a block-level programming model. Triton aims to abstracts away many complex issues related to concurrency within CUDA thread blocks (e.g., memory coalescing, shared memory synchronization/conflicts, tensor core scheduling). Users simply write an algorithm describing the blockwise operation, and Triton will automatically organizes the computations and data movement to generate a high-performance program. As as result, it is significantly easier to write high performance code in Triton. For instance, Pytorch's TorchInductor uses Triton to automatically generate high-performance kernels for GPUs [17]. Triton's well-designed automation could be an interesting design target for an Exo automation library.

ThunderKittens ThunderKittens is a C++ framework [18] that facilitates writing fast CUDA code. In terms of abstraction, it lies between CUDA and Triton. Users still need to work with small tile sizes and perform computations on them. However, ThunderKittens automates many of the performance-relevant aspects of these tile operations, including utilizing asynchronous tensor core and memory operations and automatically formatting data so as to minimize bank conflicts. Unlike Triton, ThunderKittens will not automatically compile an arbitrary block program – if the block size is larger or smaller than ThunderKitten's tile primitives, it is up to the user to write the program in terms of the tiles. While ThunderKittens is likely not the long-term vision for Exo's parallel programming model, ThunderKittens can be thought of as a simplified instruction set for GPU programming. Exo users could potentially leverage the well-built abstractions of ThunderKittens by writing an Exo hardware library that targets ThunderKittens functions.

Halide Halide was the first user-schedulable language for programming GPUs [12]. In addition to its core scheduling operations, Halide has `gpu_tile`, `gpu_threads`, and `store_in` scheduling operations to manage how computations are organized in the GPUs parallel hierarchy and memories. Extending the Halide library from Chapter 5 to also target GPUs would be an interesting future project.

Chapter 3

Implementing Stable References in a USL

All USLs must have a reference mechanism by which the schedule can specify where to perform the transformations, For instance, the Halide USL uses a nominal reference scheme where each computation is uniquely identified by a named buffer to store the result and each computation's loop nest consists of unique iterator variables. The user may subdivide loop levels further to introduce new loop levels, but the user cannot create new intermediate buffers. The Halide referencing scheme is well-suited to its goal of allowing users to easily navigate tradeoffs between different granularities of fusion: users can easily specify two computations by name and the loop level at which they should be fused.

However, existing USL referencing schemes are insufficient for implementing extensible USLs: USLs in which users can encapsulate reusable schedule fragments as new scheduling operations. For example, Halide faces two main limitations which prevent extensibility by the user: (1) it uses a nominal referencing scheme; and (2) it uses a monolithic-lowering scheduling model. Together, this means that all references must be statically known at compile time. It is impossible to define a reusable schedule fragment in Halide which applies on a generic reference. In Halide, references with descriptions such as "the innermost loop" or "the next statement" are not possible – that reference must be specified by a name.

In contrast, Exo was designed to be an extensible USL from first principles. Exo uses rewrite-based scheduling rather than using a monolithic-lowering scheduling, allowing scheduling to be done step-by-step. Furthermore, we argue that a good scheduling language should have an immutable IR. In a rewrite-based USL with an immutable IR, each scheduling operation results in the creation of a new, separate IR.

This introduces a new challenge of maintaining a *stable reference* which can be updated after a scheduling operation is applied. While a nominal referencing scheme with globally unique names such as Halide's easily maintains stable references, it cannot represent generic references. We seek to implement a more generic referencing scheme. In the following section, we describe our implementation of references in the Exo system, which we term *cursors*. Cursors allow for both globally unique references and generic references, which make it suitable for building reusable schedules. We then present our solution to the stable referencing problem, *cursor forwarding*. Cursor forwarding provides users a means to update their references after scheduling operations have transformed the object program.

3.1 Cursor Representation

We implement a code reference mechanism called a *cursor*, analogous to the blinking cursor found in text editors. Internally, all cursors store two values: a weak reference to the procedure it's pointing at (i.e. a *time* coordinate) and a path defining its relative location inside that procedure's AST (a *spatial* coordinate). The path describes navigation in an AST as a downward traversal. In an AST, all children are labeled (e.g. the *rhs* of a binary operation) and are either a node or list of nodes (e.g. the *"body"* of a for loop). Thus, each downward step in the AST may be represented as a label-index pair, where the index is null if the child is not a list. Figure 3.1 illustrates an example of a cursor and its corresponding path in the AST.

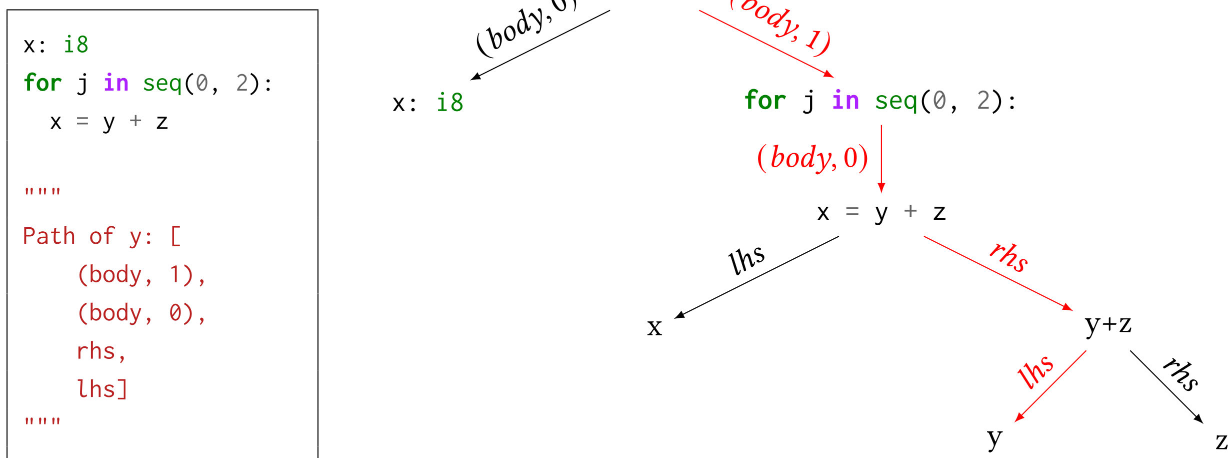


Figure 3.1: Example illustration of a cursor pointing to the read of y in the code on the left. Its corresponding path in the AST is shown on the right.

In addition to standard cursors to nodes in the AST, we also implement two additional types of cursors: blocks and gaps. Block cursors refer to a list of nodes (e.g. the body of a for loop is a list of statement nodes). Representation-wise, cursors to blocks of statements simply store a *range* at the final path index, instead of a single value. Gaps are defined as coming before or after a node, and their path is effectively the path of the anchor node appended with either a "before" or "after" at the final index.

Spatial navigation operations such as `.parent()`, `.rhs()`, `.body()[idx]`, or `.next()` are straightforward to implement by modifying this path representation. Various tree properties (e.g. whether one cursor is a subtree of another) can also be determined by comparing path prefixes. The main complexity in the design addresses the problem of how references and actions interact.

3.2 Stable References via Cursor Forwarding

When a scheduling operation transforms an old procedure into a new procedure, we need a way to update the old cursor to its corresponding location in the new procedure. We term this cursor update policy as *cursor forwarding*, and each scheduling operation has an associated *forwarding*

function. When a scheduling operation transforms a procedure, cursors are not automatically updated. Users must explicitly update the cursors by invoking the forwarding function.

Exo currently has over 40 scheduling operations, and will inevitably add more as the system continues to grow. The task of maintaining stable references is challenging because each scheduling operation's forwarding function can be quite complex. To tackle this problem, we implement cursor forwarding in two steps:

1. We define cursor forwarding through simple transformations, which we term *atomic edits*.
2. We re-implement Exo's scheduling operations as compositions of these atomic edits.

This approach allows us to only implement relatively simple, well-defined forwarding functions for the atomic edits. Each scheduling operation's transformation can then be decomposed into a sequence of atomic edits, and its forwarding function is simply the composition of the forwarding functions of its constituent atomic edits.

3.2.1 Forwarding Behavior of Atomic Edits

There are five atomic edits: `insert`, `delete`, `replace`, `move`, and `wrap`. Each atomic edit takes input cursors to determine where to perform the transformation, and returns the updated IR as well as a forwarding function. It is important to note that these atomic edits *do not necessarily preserve the functional equivalence* of the original procedure. It is up to the USL designer to perform the proper functional equivalence analysis prior to using these atomic edits to transform the underling code.

Insertion Inserts an IR fragment into a gap. This operation preserves all existing cursors. The forwarding function simply adjusts paths through the insertion point by incrementing pre-existing paths at the appropriate tree level. Cursors that do not contain the gap forward naturally. Cursors that contain the gap forward to cursors pointing at the updated AST containing the inserted IR fragment, as exemplified by the orange block cursor in Figure 3.2.

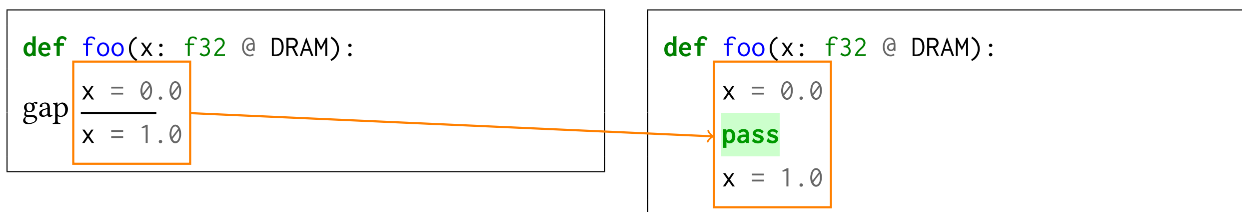


Figure 3.2: Forwarding behavior when inserting an IR fragment into a gap.

Deletion Deletes subtrees of the AST. This invalidates paths within the deleted subtrees, while paths outside remain valid. The forwarding function decrements pre-existing paths through the deletion point at the appropriate tree level. Cursors which do not contain the deleted subtree forward naturally. Cursors which contain the entire deleted subtree forward to updated ASTs without the deleted subtree, as exemplified by the orange block cursor in Figure 3.3. All other cursors, such as those entirely within the deleted subtree or those which partially intersect the deleted subtree, are invalidated by the forwarding function. This is exemplified by the violet block cursor in Figure 3.3.

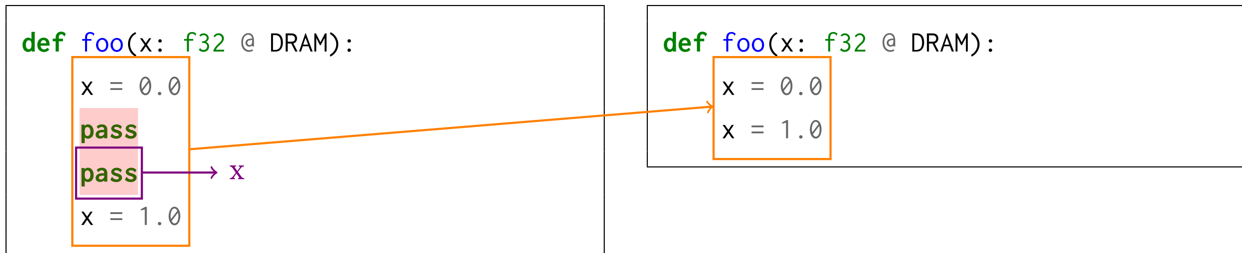


Figure 3.3: Forwarding behavior when deleting a block.

Replacement Replaces subtrees of the AST with a new subtrees. The forwarding behavior is almost identical to inserting the new subtree and deleting the old, except the unique path to the replaced subtree remains valid and forwards to the newly inserted subtree, as exemplified by the orange block cursor in Figure 3.4.

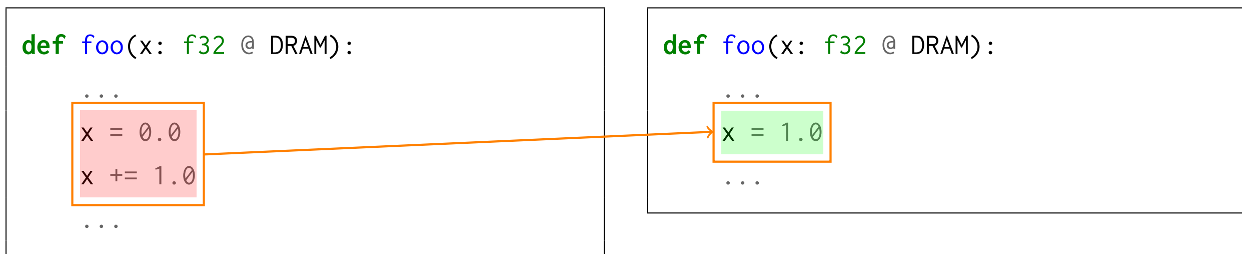


Figure 3.4: Forwarding behavior when replacing a block of statements.

Movement Moves a subtree to a new location specified by a gap. To avoid ambiguity, the gap cannot be within the moved subtree. This preserves all previous node identities, so the forwarding function maps all node cursors to their natural correspondences. Most blocks forward to natural correspondences. Blocks entirely within the moved subtree forward to their new locations in the AST, as exemplified by the orange block cursor in Figure 3.5. Blocks containing the gap or the entire moved subtree forward to the updated ASTs with/without the moved subtree. However, blocks which only partially intersect the moved subtree are invalidated, as exemplified by the violet block cursor in Figure 3.5

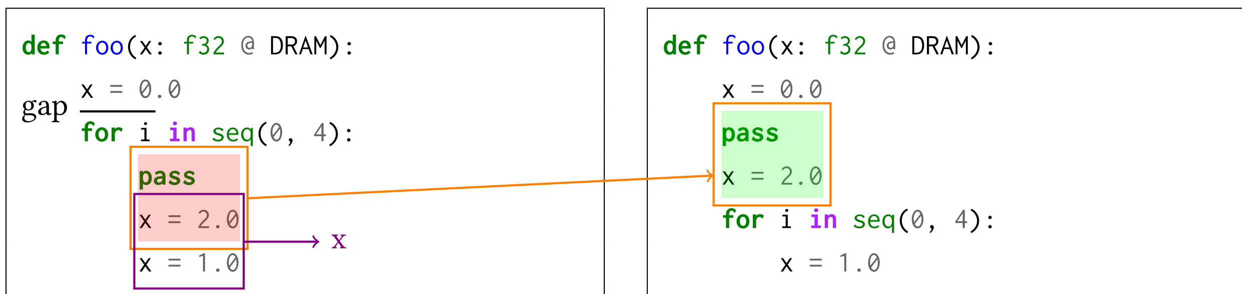


Figure 3.5: Forwarding behavior when moving a block of statements. The red highlighted code on the left is moved to the gap, resulting in the green highlighted code on the right.

Wrapping Wrapping an existing subtree with a one-hole IR fragment. When the subtree we are wrapping is a block of statements, this is equivalent to insertion with movement. However, this is not always the case for lists of expressions. For example, when wrapping plus one around an expression like ($\square + 1$), there's no (easy) way to insert the partial plus one IR fragment into the code. In terms of forwarding behavior, wrapping is very similar to movement. Blocks entirely within the wrapped subtree forward to their new locations in the AST, as exemplified by the orange block cursor in Figure 3.6. Blocks containing the entire moved subtree forward to the updated ASTs with the wrapped subtree. However, blocks which only partially intersect the wrapped subtree are invalidated, as exemplified by the violet block cursor in Figure 3.5

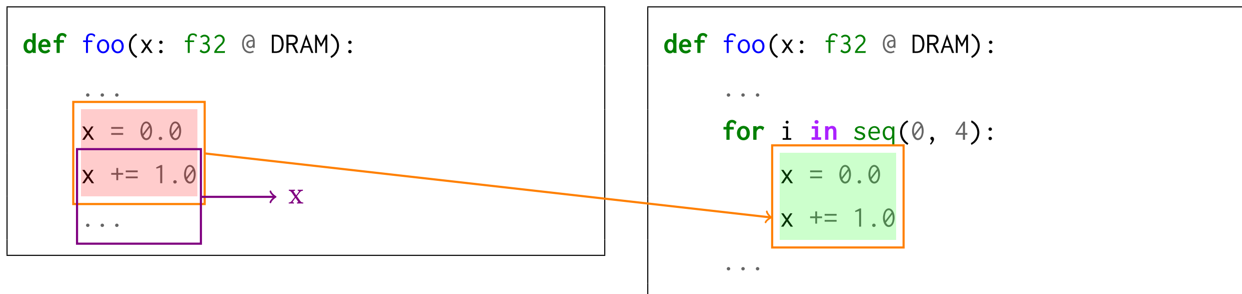


Figure 3.6: Forwarding behavior when moving a block of statements. The red highlighted code on the left is moved to the gap, resulting in the green highlighted code on the right.

Each of the five atomic edits is designed so that it must return a valid AST. While this generally holds true most of the time, there are a few pathological cases that require special care. For instance, Exo is an embedded language within Python, but Python does not allow empty for loop bodies. If a `delete` removes a loop's entire body, the atomic edit will automatically insert a **pass** statement as the loop body to align with Python syntax.

3.2.2 Composing Atomic Edits to Build Scheduling Primitives

While `insert` and `delete` are sufficient to perform arbitrary AST transformations, they will not necessarily yield the desired forwarding behavior needed to maintain stable references. We found that `replace`, `move`, and `wrap` were also necessary to express the desired forwarding behavior. Depending on the desired forwarding behavior of the scheduling operations, we might implement them using a different composition of atomic edits.

For instance, consider the `reorder_loops` scheduling operation. Figure 3.7 shows the desired forwarding behavior for cursors before and after the scheduling operation's transformation. A naive implementation of `reorder_loops` might call the `replace` atomic edit to substitute the entire subtree directly, but this would invalidate all cursors within the replaced subtree.

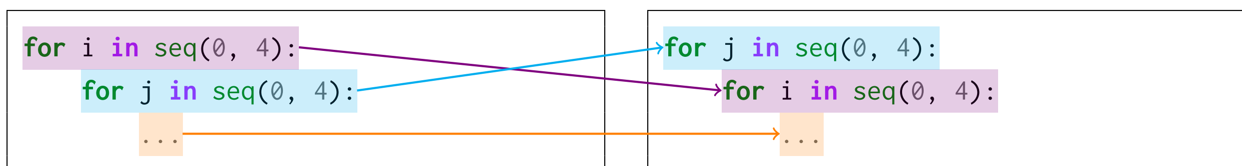


Figure 3.7: Desired forwarding behavior for `reorder_loops`.

Rather than using a replace atomic edit which invalidates many existing cursors, the implementation of `reorder_loops` consists primarily of move atomic edits which preserve most existing cursors. As each atomic edit is being performed, we are keeping track of the composed forwarding function in the `fwd` variable. Note that cursors do not automatically forward after an atomic edit is performed, so prior to applying each atomic edit function, the language designer must call `fwd` on the cursors to update them.

```
# Move the inner loop out of the outer loop, auto-generating a pass in the latter.
ir, fwd_move_1 = inner_loop._move(outer_loop.after())
fwd = fwd_move_1
# Move the old outer loop inside the inner loop
ir, fwd_move_2 = fwd(outer_loop)._move(fwd(body).before())
fwd = _compose(fwd_move_2, fwd)
# Move the body inside the old outer loop (now the inner loop)
ir, fwd_move_3 = fwd(body)._move(fwd(outer_loop).body().after())
fwd = _compose(fwd_move_3, fwd)
# Delete the pass statement auto-generated in step 1
ir, fwd_del = fwd(outer_c).body()[0]._delete()
fwd = _compose(fwd_del, fwd)
```

Figure 3.8: Implementation of `reorder_loops` as a composition of four atomic edits in order to achieve the desired forwarding behavior shown in Figure 3.7.

In order to achieve the desired forwarding behavior, Exo language designers must carefully think about how to compose the atomic edits. However, as long as language designers understand the forwarding behavior of the five atomic edits, they do not have to worry about implementing their own forwarding functions. As seen above, we can simply use function composition to obtain the desired forwarding function for the scheduling primitive.

3.3 Simplifying Implementation of Scheduling Operations

Our cursor and forwarding designs make it easier for Exo language designers to extend the scheduling language. When defining a new scheduling operation, we must implement four core components:

1. Where do we modify the AST?
2. What AST transformations need to be performed?
3. Does the transformation preserve functional equivalence?
4. How do we update existing references to the old procedure?

Our cursor design greatly simplifies the implementation of (1) and (4). For (1), the previous Exo design required each scheduling operation to implement AST navigation to determine where

to perform the transformation. By implementing cursors as a first-class reference mechanism, Exo language designers no longer need to concern themselves with this problem. While (4) is a new requirement introduced by implementing a first-class reference mechanism, our two step design to cursor forwarding makes this easy. Exo langauge designers do not need to define a new forwarding function for each new scheduling operation. They simply need to determine how to decompose the transformation into the five atomic edits, and the forwarding behavior is provided by composing the forwarding functions of the atomic edits.

Chapter 4

Reproducing Halide as an Exo Library

Leveraging the cursors and cursor forwarding defined in the previous chapter, we will demonstrate how users can build libraries of automated scheduling strategies within Exo. We built an Exo library which implements the Halide scheduling operations as compositions of Exo primitive scheduling operations. Halide's well chosen scheduling operations enable performance engineers to rapidly explore the trade-off space. Compared to Exo, Halide's scheduling operations operate at a much higher level of automation, making it a productive system for optimizing many real-world image processing pipelines. In this chapter, our goal is *not* to replicate the entire Halide system, but rather the well-chosen automation level provided by Halide's scheduling operations.

To start off, we introduce the Halide language via the canonical 3×3 box blur example.

4.1 Overview of Halide by Example

Halide Blur Algorithm We divide the computation into two computation stages: a vertical blur and a horizontal blur. These two computations have a producer-consumer relationship: the *consumer* `blur_y` depends on the results of the *producer* `blur_x`. Each `Func` corresponds to an array of intermediate values that must be computed. Although for loops are not explicitly written in Halide, each `Var` implicitly corresponds to a loop iteration variable. Halide has a default loop iteration order of `y` and then `x`.

```
Input<Buffer<uint16_t, 2>> input{"input"};
Output<Buffer<uint16_t, 2>> blur_y{"blur_y"};

Func blur_x("blur_x");
Var x("x"), y("y");

blur_x(x, y) = (inp(x, y) + inp(x+1, y) + inp(x+2, y)) / 3;
blur_y(x, y) = (blur_x(x, y) + blur_x(x, y+1) + blur_x(x, y+2)) / 3;
```

Halide Blur Schedule The expert hand-written Halide schedule for this algorithm, taken from [12], is shown below. The `tile`, `vectorize`, `parallel`, and `compute_at` are Halide's scheduling

operations. Lines 1 and 2 introduce new inner loops to subdivide the computation into 32×256 tiles, making the new loop iteration order (from outermost to innermost) `y`, `x`, `yi`, `xi`. Lines 3 and 6 tell the Halide instruction selector to utilize vector instructions during code generation of the innermost `xi` loops. Line 4 parallelizes the problem across the outermost `y` loop.

```
Var xi("xi"), yi("yi")
blur_y.tile(x, y, xi, yi, 256, 32)
      .vectorize(xi, 16)
      .parallel(y);
blur_x.compute_at(blur_y, x)
      .store_at(blur_y, x)
      .vectorize(x, 16);
```

Line 5 is the most interesting line of the schedule. The `blur_x.compute_at(blur_y, x)` scheduling operation fuses the `blur_y` and `blur_x` computations at the `x` loop level. This means that for each 32×256 `blur_x`, we compute all the inputs needed to evaluate this tile (e.g. a 34×256 tile of `blur_y`). Note that the Halide schedule automatically determines what the set of necessary inputs. The `store_at(blur_y, x)` scheduling operation drops the allocation within the `x` loop iteration, resulting in an allocation of size $(32 + 2) \times 256$ since each row of `blur_y` depends on three rows of `blur_x`. So this schedule implicitly allocate `blur_x` within the `x` loop. Ultimately, the Halide schedule results in the following pseudocode:

```
parallel for y:
  for x:
    alloc blur_x
    for yi in [0, 33]:
      vectorized for xi in [0, 15]:
        blur_x(...) = ...
    for yi in [0, 31]:
      vectorized for xi in [0, 15]:
        blur_y(...) = ...
```

In practice, even if no `store_at` command were provided, Halide would generate the same code because by default, it will store at the same loop as `compute_at`. If neither a `compute_at` or `store_at` scheduling operation were provided for the intermediate value `blur_x`, then the default Halide behavior is to inline it into the computation of `blur_y`. Thus, while the schedule may not always explicitly specify a compute or store location, Halide will always fill in default values at compile time.

4.2 Implementing Halide Scheduling Operations in Exo

In this section, we aim to reproduce the core Halide scheduling operations `tile`, `split`, `reorder`, `compute_at`, `store_at`, `vectorize`, `store_in`, and `parallel` within Exo. However, Exo and Halide

object code are very different: Halide expresses computations as sequences of pure functions over integer domains, while Exo express computations as sequence of loops, conditionals, and statements. We restrict our Exo-implemented `compute_at` and `store_at` scheduling operations to a subset of Exo code satisfying certain assumptions:

1. The code only consists of for loops, if statements, buffer allocations, buffer assignments, and asserts.
2. For each buffer, all of its writes are performed by a single assign statement within loops. This reflects the implicit assumptions of Halide's object code when writing statements such as `blur_x(x, y) = ...`.
3. For each dimension of each buffer assignment, the surrounding loops are arranged from top to bottom in decreasing stride order (e.g. the `x` loop contains the `xi` loop).
4. Each buffer's computation is dependent on a contiguous subset of a previous buffer.

For example, the Exo 3×3 box blur algorithm from Section 2.2 satisfies these properties. The Exo-implemented `compute_at` and `store_at` operations preserve these assumptions throughout their transformations. Note that while we make these assumptions, applying our Exo-implemented `compute_at` and `store_at` scheduling operations to programs which violate the assumptions will never generate incorrect code. Instead, violations of the assumptions will be caught and Exo will throw errors during scheduling. These assumptions simplify the implementation, but also introduce limitations. As we will discuss in Section 4.4.2, the second assumption combined with Exo's current limitations restrict the tail cases we can handle[1]. However, the second assumption is essential for code organization: it prohibits generating an exponential number of separate loop nests for tail cases. This makes it possible to easily chain `compute_at` operations together.

Inevitably, there are also differences between the Halide and Exo scheduling interfaces. Halide is a scheduling language which monolithically lowers code based on the provided schedule. As a result, Halide automatically reorders the scheduling operations at compile time and applies them in a prescribed order based on the type of transformation: specialization, loops, compute, store, and bounds [5]. The user does not have to worry about ordering the scheduling operations. However, in a rewrite scheduling system such as Exo, the user must determine the scheduling operations order because earlier parts of schedules will influence how later parts of the schedule should transform the code. In Section 4.2.1, we explain the order in which the Exo Halide-like scheduling operations should be applied. Additionally, Halide postpones determining loop bounds until the very end. Postponing bounds inference is not possible in Exo because all intermediate code between rewrites must be valid. Therefore, the Exo implementations of `compute_at` and `store_at` must internally perform bounds inference, which we explain in Section 4.2.2.

In Section 4.2.3 and Section 4.2.4, we will utilize bounds inference to implement `compute_at` and `store_at` in Exo. Rather than supplying `compute_at` and `store_at` as built-in scheduling operations, we provide users with the ability to implement them as compositions of Exo's primitive scheduling operations. We only focus on explaining the implementations of `compute_at` and `store_at` since `tile`, `split`, `reorder`, and `store_in` are straight forward to implement, `parallel` is explained in greater detail later in Section 5.1, and `vectorize` was previously explained in [19]. Other scheduling operations were not deemed as essential parts of the Halide scheduling interface, so we did not reproduce them for now.

[1] For innermost loop tail cases, the `vectorize` scheduling operation works.

4.2.1 Ordering Halide-like Scheduling Operations in Exo

In Exo, users must manually order the scheduling operations. The ordering matters because previously-applied scheduling operations change the inputs to future scheduling operations. In Exo, we generally first apply loop transformations such as `tile`, `reorder`, `split`. Then, we apply `compute_at` and `store_at` operations. Lastly, we apply `vectorize` and `parallel`.

The `compute_at` scheduling operation is responsible for moving the buffer assignment while the `store_at` scheduling operation is responsible for moving the memory allocation around. For each intermediate buffer, the proper scheduling operation ordering is compute, store, and then possibly compute again. The second compute phase is necessary because the exact code transformation we want to perform for `compute_at` *depends* on what loop level we pass to `store_at`. To illustrate this point, consider the Halide algorithm below. Even between two schedules with the same `compute_at` command, the generated code for computing producer values depends on the buffer's storage location.

Algorithm

```
producer(x, y) = sin(x * y)
consumer(x, y) = producer(x, y) + producer(x, y + 1)
```

Schedule

```
producer.compute_at(consumer, y)
        .store_at(consumer, y)
```

Generated pseudocode

```
for y:
  alloc producer
  for py in [0, 1]:
    for x:
      producer(...) = ...
  for x:
    consumer(...) = ...
```

(a) Compute and store at the same loop

Schedule

```
producer.compute_at(consumer, y)
        .store_root()
```

Generated pseudocode

```
alloc producer
for y:
  for py in [0, 1]:
    if y == 0 or py == 1:
      for x:
        producer(...) = ...
  for x:
    consumer(...) = ...
```

(b) Store at higher loop than compute

Figure 4.1: While both schedules invoke `producer.compute_at(consumer, y)` to fuse the computations at the `y` loop level, they store the result at different locations. The left schedule stores the producer at the same loop level as `compute_at`, while the right schedule calls `store_root()`, which stores the producer above all loop levels. This results in different generated code.

If we compute and store at the same loop level (Figure 4.1a), then the generated code will recompute some producer values across x loop iterations because the memory is freshly reallocated each iteration. However, if we are store at a higher loop than we are compute at (Figure 4.1b), then there is an opportunity to share work across each loop below allocation level. Thus, the generated code will precompute a prologue of producer values, and then compute the rest of the producer values on demand at the loop level specified by `compute_at`.

To enable users to adequately navigate the Halide trade-off space, we provide users with two different versions of `compute_at`, modulated by setting the `with_prologue` flag to either **`True`** or **`False`**. When scheduling the compute and storage levels for a particular intermediate buffer, Exo users should apply the scheduling operations in the following order:

1. The `compute_at` operation with `with_prologue=False` at the same loop level that the user plans to store at.
2. The `store_at` operation at the desired loop level.
3. (Optional) The `compute_at` operation with `with_prologue=True` at the desired compute loop level. This step is only necessary if computing at a loop level below the storage loop level.

In Section 4.2.3 and Section 4.2.4, we will explain the implementation of each of these scheduling operations in greater detail.

4.2.2 Bounds Inference

Bounds inference is the problem of determining the indices of all possible accesses to an array within a given scope of code. We do not claim to have reproduced Halide's entire bounds inference system, which is more general. Our bounds inference only applies to Exo's object language, which is restricted to affine iteration spaces, making the bounds inference problem simpler than it would be in more general languages which allow data dependent access patterns.

```
for io in seq(0, N / 32):
    # arr is accessed within [32 * io : 32 * io + 34]
    for ii in seq(0, 32):
        x = arr[32 * io + ii] + arr[32 * io + ii + 1] + arr[32 * io + ii + 2]
```

To implement bounds inference, consider the example of inferring access bounds for `arr` within the `io` loop in the code fragment above. Determining the bounds for the array `arr` can be reduced to unioning bounds for each array access. In ordinary Python code, we define a representation of intervals with support for unioning intervals of expressions in arithmetic operations. We will find all accesses to the array, and call `infer_range` (defined below) on each access's index expression to bound it. To determine intervals for each index expression, we decompose the expression into an linear combination of constants and variables which may be either free or bound in the scope. For example, in the index expression `32 * io + ii + 1` above, `ii` is a bound variable and `io` is a free variable. Knowing that $0 \leq$ `ii` < 32 implies that this index expression spans the window `[32 * io + 1 : 32 * io + 32]`. Leveraging cursors' ability to return properties of the object code, our implementation obtains loop bounds (lines 6–9) and maintains

an environment dictionary (line 10) to track the free and bound variables. We then call `index_range_analysis` in line 12, which recursively traverses the AST of `idx_expr` and consults the constructed environment to determine a bound on the index expression.

```
def infer_range(idx_expr: Cursor, scope: Cursor) -> IndexRange:
    # Only add bound variables to the env (which excludes scope)
    env = dict()
    ancestors = list(get_parents(idx_expr.proc(), idx_expr, up_to=scope))[:-1]
    for c in filter(lambda x: isinstance(x, ForCursor), ancestors):
        lo, _ = constant_bound(c.lo(), env)
        _, hi = constant_bound(c.hi(), env)
        if hi is not None:
            hi -= 1  # loop upper bound is exclusive
        env[c.name()] = (lo, hi)

    bounds = index_range_analysis(idx_expr, env)
    return bounds
```

4.2.3 Compute At

We will explain the implementation of `compute_at`, which is provided at the end of this subsection. To ground our discussion, let's start from the 3×3 box blur algorithm after tiling, shown in Figure 2.2.

To perform the code transformation for `blur_x.compute_at(blur_y, x)`, we follow the sequence of transformations shown in Figure 4.2. We fuse the computation of the producer `blur_x` with the computation of the consumer `blur_y` by sinking the computation `blur_x` into the `y` and `x` consumer loops. For each consumer loop we want to sink the compute into, our `compute_at` needs to make decisions based on the code it is being applied to. The two "Inspection" labels in Figure 4.2 are user-implemented analyses which guide the application of scheduling operations within the `compute_at` implementation. The first determines whether or not the outermost loop of the producer loop nest corresponds to the same array dimension as the outermost loop of the consumer loop nest. Since Exo has to explicitly deal with loops, we may also have to reorder loops and statements to make the producer and consumer loops adjacent. For the blur example, we only need to reorder the loops once, as indicated by the red highlighted code in Figure 4.2. The second "Inspection" utilizes the bounds inference described in Section 4.2.2 to determine loop bounds when fusing two successive computation stages.

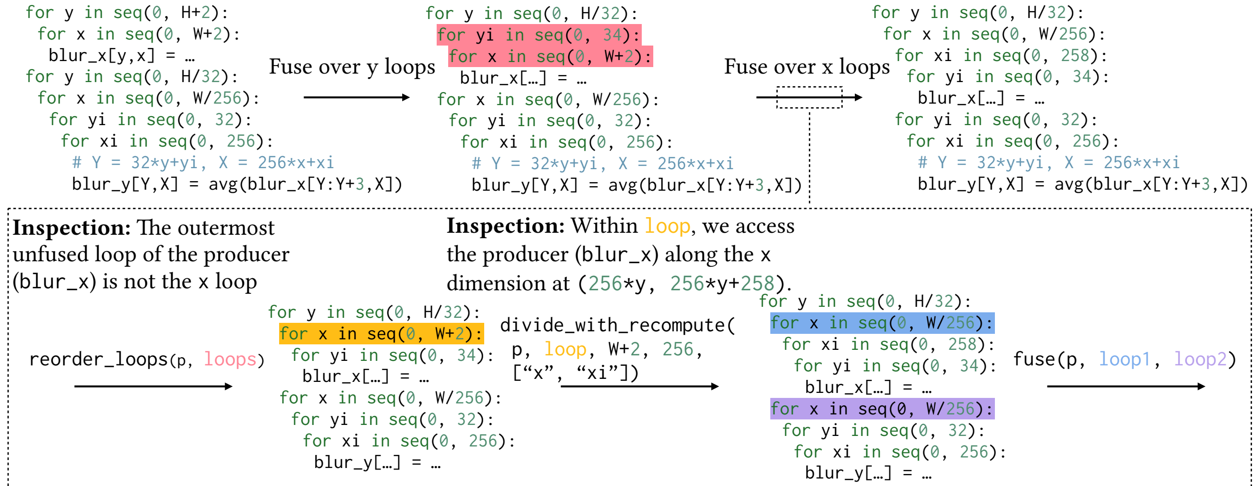


Figure 4.2: Decomposing Halide's `compute_at` scheduling operation into Exo's finer-grained primitive scheduling operations for the blur example's `blur_x.compute_at(blur_y, x)`.

Recall that in Section 4.2.1, we distinguished between two different modes of `compute_at`, depending on whether we are computing and storing at the same loop level. The main difference lies in the precise code transformations that are applied after inspecting the code and reordering loops/statements. The `with_prologue=False` mode corresponds to lines 44–51 in the implementation below. For each consumer loop we want to sink the compute into, we divide corresponding loop around the producer `blur_x` and fusing the resulting outer loop with the consumer's loop. This is the transformation being shown in Figure 4.2 because the blur schedule computes and stores at the same loop level.

In the second case where we want to compute at a lower level then we store at, we use the mode with `with_prologue = True`, which corresponds to lines 52–64 in the implementation below. For each consumer loop we want to sink the compute into, rather than dividing the producer loop, it separates the producer loop into a prologue loop and a main loop via the `cut_loop` scheduling operation. Then, it fuses the main producer loop with the consumer loop.

Full compute_at Implementation

```
def compute_at(
    proc: Procedure,
    producer_assign: AssignCursor,
    target_loop: ForCursor,
    with_prologue: bool = False,
):
    producer_assign = proc.forward(producer_assign)
    target_loop = proc.forward(target_loop)
    producer = producer_assign.name()

    # Identify all the consumer loops we need to fuse into
    p_loop, c_loop = match_parent(producer_assign, target_loop)
    c_loops = [target_loop] + list(get_parents(proc, target_loop, up_to=c_loop))
    for c_loop in reversed(c_loops):
        # Cursor forwarding: enables reusing existing cursors
        producer_assign = proc.forward(producer_assign)
        c_loop = proc.forward(c_loop)

        # Inspection: bounds inference to determine which producer values are consumed
        buffer_dim = get_affected_read_dim(producer, c_loop)
        bounds = bounds_inference(c_loop, producer, buffer_dim, include=["R"])
        N_c = c_loop.hi()._impl._node

        # Inspection: identify the corresponding producer loop based on dimension affected
        dim_vars = _get_reads_of_expr(producer_assign.idx()[buffer_dim])
        p_loop, _ = match_parent(producer_assign, c_loop)
        while p_loop.name() not in dim_vars:
            p_loop = p_loop.body()[0]

        # Surface the corresponding producer loop level
        while p_loop.parent() != c_loop.parent():
            proc = reorder_loops(proc, p_loop.parent())
            # Cursor forwarding: enables reusing existing cursors
            p_loop = proc.forward(p_loop)
            c_loop = proc.forward(c_loop)

        # Reorder producer loop nest to be directly before consumer loop nest
        while p_loop.next() != c_loop:
            proc = reorder_stmts(proc, p_loop.expand(0, 1))
            # Cursor forwarding: enables reusing existing cursors
            p_loop = proc.forward(p_loop)
            c_loop = proc.forward(c_loop)
```

Full `compute_at` Implementation (continued)

```
43        # Different transformations depending on [with_prologue] flag
44        if not with_prologue:
45            # Divide the loop with recomputation
46            w_c = bounds.get_stride_of(c_loop._impl._node.iter)
47            new_iters = [f"{c_loop.name()}", f"{c_loop.name()}i"]
48            proc = _divide_with_recompute(proc, p_loop, f"{N_c}", w_c, new_iters)
49
50            # Fuse producer loop with consumer loop
51            proc = fuse(proc, p_loop, c_loop, unsafe_disable_check=True)
52        else:
53            # Separate out prologue loop
54            w_p = bounds.get_size()
55            proc = cut_loop(proc, p_loop, w_p - 1)
56
57            # Cursor forwarding: enables reusing existing cursors
58            prologue_loop = proc.forward(p_loop)
59            main_loop = prologue_loop.next()
60            producer_assign = proc.forward(producer_assign)._reroute_through(main_loop)
61
62            # Fuse main producer loop with consumer loop
63            proc = shift_loop(proc, main_loop, 0)
64            proc = fuse(proc, main_loop, main_loop.next())
65
66        proc = _simplify_with_preds(proc)
67
68    return proc
```

4.2.4 Store At

The implementation of `store_at` is simpler. This scheduling operation repeatedly sinks an allocation into a for loop and shrinks the size of the allocation to reduce memory footprint. The implementation can be found on the following page. We first gather all the loops we need to move the allocation via the `loops_between` function defined on line 11. For each loop, we use `bounds_inference` (line 28) to determine how much we can shrink the allocation size by. We then reorder the allocation to be right before the loop (lines 32–35), and use `sink_alloc` (line 37) to move the allocation inside the loop. We then resize the allocation using `resize_dim` (line 38) and apply arithmetic simplification (line 39) to the newly generated size expressions. Note that once again, cursor forwarding is the key mechanism which enables composition of scheduling operations operating on a generic reference. Without cursor forwarding, we would have to repeatedly re-obtain a reference to the `producer_alloc` after each transformation.

store_at Implementation

```python
def store_at(proc: Procedure, producer_alloc: AllocCursor, target_loop: ForCursor):
    """
    Moves [producer]'s allocation into [target_loop] and reduces the dimensions
    as necessary.
    """
    producer_alloc = proc.forward(producer_alloc)
    target_loop = proc.forward(target_loop)

    producer = producer_alloc.name()

    def loops_between(producer_alloc, target_loop):
        """
        producer_alloc
        ...
        for i in _:
            for j in _:
                for k in _: <- target_loop
        loops_between(producer_alloc, target_loop) -> [i, j, k]
        """
        top_loop, _ = match_parent(target_loop, producer_alloc)
        return reversed(get_parents(proc, target_loop, up_to=top_loop))

    for loop in loops_between(producer_alloc, target_loop) + [target_loop]:
        loop = proc.forward(loop)
        producer_alloc = proc.forward(producer_alloc)

        buffer_dim = get_affected_read_dim(producer, loop)
        bounds = bounds_inference(loop, producer, buffer_dim)
        lo, _ = bounds.get_bounds()
        size = bounds.get_size()

        while producer_alloc.next() != loop:
            proc = reorder_stmts(proc, producer_alloc.expand(0, 1))
            producer_alloc = proc.forward(producer_alloc)
            loop = proc.forward(loop)

        proc = sink_alloc(proc, producer_alloc)
        proc = resize_dim(proc, producer_alloc, buffer_dim, size, lo)
        proc = simplify(proc)

    return simplify(proc)
```

4.3 Evaluation

We apply our Exo Halide-like scheduling operations to optimize several image processing algorithms. We use our Exo Halide-like scheduling operation to schedule the 3×3 box blur and unsharp masking algorithms, achieving performance competitive with state-of-the-art, handwritten Halide schedules.

For ease of comparison, we defined `halide_`-prefixed wrapper functions around our Halide-like scheduling operations to bridge the gap between Halide's and Exo's referencing schemes. These `halide_`-prefixed wrapper functions expect nominal string references and internally convert to Exo's cursor references. Halide's nominal referencing scheme uses action-specialized built-in navigation. For example, `blur_x.compute_at(blur_y, x)` refers to the `x` loop enclosing the `blur_y` computation. Since cursors generalize nominal referencing, this simply involves translating the built-in navigation performed by Halide's scheduling operations.

Furthermore, to shield users from the ordering of scheduling operations described in Section 4.2.1, we define the following compound scheduling operations:

- `halide_compute_and_store_at_same(proc, producer, consumer, loop)`: first calls `compute_at`, then `store_at`, both at the same loop level
- `halide_compute_and_store_at(proc, producer, consumer, compute_loop, store_loop)`: first calls `compute_at` at `store_loop`, then calls `store_at` at `store_loop`, and finally calls `compute_at_with_prologue` at `compute_loop`
- `halide_fully_inline(proc, producer, consumer)`: calls `halide_compute_and_store_at_same` on the lowest loop level. Then, it inlines the producer into the consumer and deletes the unnecessary intermediate buffer for the producer.

4.3.1 3×3 Box Blur

Exo Schedule using Halide-like Operations We previously provided the Halide algorithm and schedule in Section 4.1 as well as the initial Exo 3×3 box blur algorithm in the beginning of Section 4.2, so we'll dive directly into the Exo schedule. With the exception of the last line, every line of the Exo schedule has a natural corresponding line in the Halide schedule. Furthermore, the code generated by the Exo schedule has competitive performance with the code generated by the Halide schedule, as show in Figure 4.3a.

```
1  p = exo_blur
2  p = halide_tile(p, "blur_y", "y", "x", "yi", "xi", 32, 256)
3  p = halide_compute_and_store_at_same(p, "blur_x", "blur_y", "x")
4  p = halide_parallel(p, "y")
5  p = halide_vectorize(p, "blur_x", "xi", 16)
6  p = halide_vectorize(p, "blur_y", "xi", 16)
7  p = halide_store_in(p, "blur_x", DRAM_STACK)
```

4.3.2 Unsharp Masking

Halide Algorithm The unsharp masking algorithm is an algorithm that enhances the contrast between adjacent pixels. Below, is the Halide algorithm for unsharp masking. It first converts an RGB image into gray scale. Then, it applies a 7×7 Gaussian blur to the gray image in two steps, a vertical blur followed by a horizontal blur, to form a blurry image. The sharpened image is computed as the gray image masked by the blurry image. Finally, the sharpening factor, the ratio between the sharpened image and the gray image, is used to scale the original image. All together, the unsharp masking algorithm consists of 7 computation stages.

```
Input<Buffer<float, 3>> input{"input"};
Output<Buffer<float, 3>> output{"output"};
Var x("x"), y("y"), c("c");

Func kernel("kernel");
const float kPi = 3.14159265358979310000f;
GeneratorParam<float> sigma{"sigma", 1.5f};
kernel(x) = exp(-x * x / (2 * sigma * sigma)) / (sqrtf(2 * kPi) * sigma);

Func gray("gray");
gray(x, y) = (0.299f * input(x, y, 0) +
              0.587f * input(x, y, 1) +
              0.114f * input(x, y, 2));

Func blur_y("blur_y");
blur_y(x, y) = (kernel(0) * gray(x, y + 3) +
                + kernel(1) * (gray(x, y + 2) + gray(x, y + 4))
                + kernel(2) * (gray(x, y + 1) + gray(x, y + 5))
                + kernel(3) * (gray(x, y + 0) + gray(x, y + 6)));

Func blur_x("blur_x");
blur_x(x, y) = (kernel(0) * blur_y(x + 3, y)
                + kernel(1) * (blur_y(x + 2, y) + blur_y(x + 4, y))
                + kernel(2) * (blur_y(x + 1, y) + blur_y(x + 5, y))
                + kernel(3) * (blur_y(x + 0, y) + blur_y(x + 6, y)));

Func sharpen("sharpen");
sharpen(x, y) = 2 * gray(x + 3, y + 3) - blur_x(x, y);

Func ratio("ratio");
ratio(x, y) = sharpen(x, y) / gray(x + 3, y + 3);

output(x, y, c) = ratio(x, y) * input(x + 3, y + 3, c);
```

Exo Algorithm With the exception of explicit loop bounds, the Exo algorithm closely matches the Halide algorithm.

```
from math import exp, pi, sqrt
sigma = 1.5
k1, k2, k3, k4 = [exp(-x * x / (2 * sigma ** 2)) / (sqrt(2 * pi) * sigma) for x in range(4)]
r_to_gray, g_to_gray, b_to_gray = 0.299, 0.587, 0.114

@proc
def exo_unsharp(W: size, H: size, output: f32[3, H, W], input: f32[3, H + 6, W + 6]):
    assert H % 32 == 0

    gray: f32[H + 6, W + 6]
    for y in seq(0, H + 6):
        for x in seq(0, W + 6):
            gray[y, x] = (
                r_to_gray * input[0, y, x]
                + g_to_gray * input[1, y, x]
                + b_to_gray * input[2, y, x]
            )

    blur_y: f32[H, W + 6]
    for y in seq(0, H):
        for x in seq(0, W + 6):
            blur_y[y, x] = (
                k0 * gray[y + 3, x]
                + k1 * (gray[y + 2, x] + gray[y + 4, x])
                + k2 * (gray[y + 1, x] + gray[y + 5, x])
                + k3 * (gray[y + 0, x] + gray[y + 6, x])
            )

    blur_x: f32[H, W]
    for y in seq(0, H):
        for x in seq(0, W):
            blur_x[y, x] = (
                k0 * blur_y[y, x + 3]
                + k1 * (blur_y[y, x + 2] + blur_y[y, x + 4])
                + k2 * (blur_y[y, x + 1] + blur_y[y, x + 5])
                + k3 * (blur_y[y, x + 0] + blur_y[y, x + 6])
            )

    sharpen: f32[H, W]
    for y in seq(0, H):
        for x in seq(0, W):
            sharpen[y, x] = 2.0 * gray[y + 3, x + 3] - blur_x[y, x]
```

Exo Algorithm (continued)

```
    ratio: f32[H, W]
    for y in seq(0, H):
        for x in seq(0, W):
            ratio[y, x] = sharpen[y, x] / gray[y + 3, x + 3]

    for y in seq(0, H):
        for c in seq(0, 3):
            for x in seq(0, W):
                output[c, y, x] = ratio[y, x] * input[c, y + 3, x + 3]
```

Halide Schedule Before we schedule unsharp masking in Exo, it is worth understanding the details of the the Halide schedule, particularly which decisions Halide makes *automatically*.

```
1  Var yo, yi;
2  const int vec = natural_vector_size<float>();
3
4  output.split(y, yo, yi, 32)
5      .parallel(yo)
6      .reorder(x, c, yi, yo);
7      .vectorize(x, vec)
8  gray.compute_at(output, yi)
9      .store_at(output, yo)
10     .vectorize(x, vec);
11 blur_y.compute_at(output, yi)
12     .store_at(output, yo)
13     .vectorize(x, vec);
14 ratio.compute_at(output, yi)
15     .store_at(output, yo)
16     .vectorize(x, vec);
```

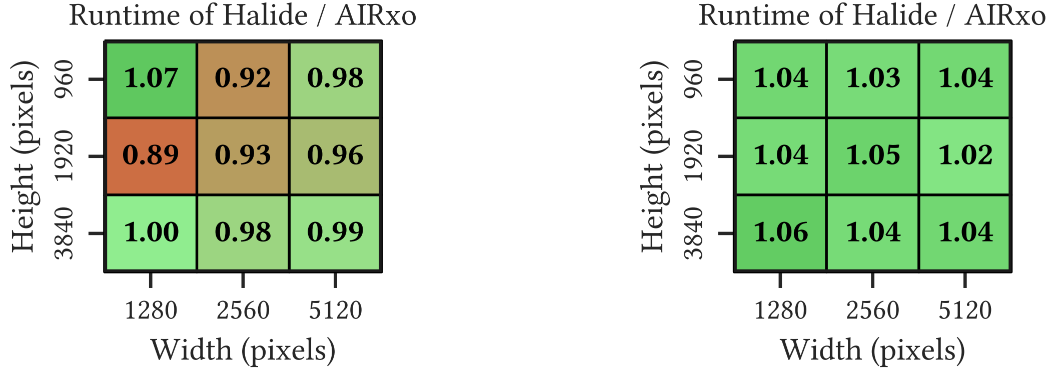
Lines 4–5 divide the computation into groups of 32 output rows and parallelizes the groups across all available cores. Line 6 specifies the loop iteration order. Since no schedule was specified for `blur_x` and `sharpen`, Halide's default behavior is to inline those computations into the computation of `ratio`. For `gray`, `blur_y`, and `ratio`, we apply the same schedule of `.compute_at(output, yi).store_at(output, yo)`. This indicates that we should fuse the computation of these buffers with the final output at the granularity of rows, but only perform the allocation once for each parallel set of 32 rows. Finally, we vectorize all computations along the `x` loop dimension to leverage SIMD parallelism.

Implicit in the schedule above, Halide automatically performs circular buffer optimizations in order to minimize memory usage. For example, consider the fact that we are storing `gray` at the `yo` loop level, but computing it at the `yi` loop level. A naive approach would allocate the `gray` buffer

to store each of the 38 rows needed to compute 32 rows of output. However, we compute the output row-by-row and each row only depends on 7 rows of gray, it suffices to store the 7 most recent rows of gray, and index into the array with modular arithmetic. In practice, Halide rounds the minimum number of rows up to the nearest power of two to enable fast modular arithmetic when indexing. Similarly, Halide also applies the circular buffer optimization to blur_y and ratio. The generated code allocates 1 row of memory for blur_y and ratio, and 8 rows worth of memory for gray.

Exo Schedule using Halide-like Operations To write the same schedule in Exo, we need to order the scheduling operations appropriately. With the exception of lines 3, 8–9, and 14–16, all other lines in the Exo schedule have a natural correspondence to a line in the Halide schedule. In line 3, we defined a custom function which binds all of the constants in the Exo unsharp masking algorithm (e.g. r_to_gray, k0, etc.) into Exo variables. This is due to a limitation in Exo's design which is elaborated in Section 4.4.3. Lines 8–9 perform the Halide's assumed default schedule of fully inlining the computations of sharpen and blur_x into the computation of ratio. Lines 14–16 reflect how Halide automatically performs the circular buffer optimization to reduce the kernel's memory footprint. When comparing the performance of the code generated by the Exo schedule to the code generated by the Halide schedule, our Exo-scheduled kernel shows competitive performance in Figure 4.3b.

```
1  p = exo_unsharp
2
3  p = bind_all_consts_for_unsharp(p)
4  p = halide_split(p, "output", "y", "y", "yi", 32, tail="perfect")
5  p = halide_parallel(p, "y")
6
7  p = halide_compute_and_store_at(p, "ratio", "output", "yi", "y")
8  p = halide_fully_inline(p, "sharpen", "ratio")
9  p = halide_fully_inline(p, "blur_x", "ratio")
10 p = halide_compute_and_store_at(p, "blur_y", "output", "yi", "y")
11 p = halide_compute_and_store_at(p, "gray", "output", "yi", "y")
12
13 # Circular buffer optimization, which Halide does automatically
14 p = resize_dim(p, p.find("ratio: _"), 0, 1, 0, fold=True)
15 p = resize_dim(p, p.find("blur_y: _"), 0, 1, 0, fold=True)
16 p = resize_dim(p, p.find("gray: _"), 0, 8, 0, fold=True)
17
18 p = halide_vectorize(p, "gray", "x", 8)
19 p = halide_vectorize(p, "gray", "x", 8)
20 p = halide_vectorize(p, "blur_y", "x", 8)
21 p = halide_vectorize(p, "ratio", "x", 8)
22 p = halide_vectorize(p, "output", "x", 8)
```



(a) Performance comparison for 3×3 box blur (b) Performance comparison for unsharp masking

Figure 4.3: Performance comparison between code generated by Exo schedules and code generated by expert-written Halide schedules. The Exo schedules exhibit competitive performance across a range of image sizes.

4.4 Areas for improvement

This implementation is far from the production-grade quality of the Halide system. In this section, we highlight Exo's current limitations which either prevent implementing some features present in Halide (Section 4.4.1 and Section 4.4.2), or make the implementation of the automation less convenient (Section 4.4.3). We hope that these initial explorations will be helpful for guiding future improvements to the Exo language.

4.4.1 Generation of Prologue Loops

In the Exo schedule for unsharp masking above, a careful reader may have noticed that lines 16 and 17 are identical. This is not a typo. Rather, this is an artifact of an Exo limitation when generating prologue loops. Recall that at the beginning of Section 4.2, we said that our Halide scheduling operations should only apply to code where every buffer is written to by exactly one assign statement. This artifact is because our scheduling operation technically violates that condition with the generation of prologue loops, as shown in Figure 4.4b. Ideally, we would generate the code shown in Figure 4.4a, but current analysis limitations make it impossible to perform such a transformation. We are currently working on generalizing our analysis to be able to handle such transformations so we can avoid violating our initial assumptions. In the case of unsharp masking, the violation did not result in a substantial change to scheduling. However, the introduction of additional loops could potentially complicate scheduling of earlier pipeline stages because we would now have to fuse into both the prologue and main loop. This problem could potentially create a combinatorial explosion of loops, which is detrimental for designing a scalable `compute_at` scheduling operation in Exo.

```
for y:
  for py in [0, 1]:
    if y == 0 or py == 1:
      for x:
        producer(y + py, x) = ...
```

(a) Ideal form of computing prologues using a single assign statement

```
for x:
  producer(0, x) = ...
for y:
  for x:
    producer(y + 1, x) = ...
```

(b) The current form of computing prologues generated by Exo, which uses multiple assign statements

Figure 4.4: Different possibilities for generated prologue loops. Ideally, we would generate the code on the left, both to be consistent with the assumptions made at the beginning of Section 4.2 and for easier design of the scheduling operations. However, with current Exo's capabilities, we are only able to generate the code on the right.

4.4.2 Handling Tail Cases

There are limitations of the Exo language which currently make it difficult for handling tail cases, particularly for simple loop parallelization strategies such as OpenMP's parallel for. When using a parallel for loop, the body must be shared between both the full main iterations and the partial tail iteration. Ideally, this body should not have predication such as if statements, as that would add a significant runtime overhead to the main iterations. In Halide, two important tail strategies which do not introduce significant overhead to the main iterations are RoundUp and ShiftInwards.

One of the main reasons we cannot handle tail cases for these types of parallel code is that Exo lacks the expressivity to represent these two important tail strategies. The former requries the language to support overcompute analysis to determine when it is safe to compute garbage values, as indicated in Figure 4.5a. The latter requires careful management of index expressions using `min`, as illustrated in Figure 4.5b, which is not possible in current Exo. Furthermore, Exo's current notion of parallel safety has no way to deal with "safe write races", as we will discuss later in Section 5.1, which is required for the ShiftInwards tail case strategy.

```
x: i8[N]
for io in par(0, N / 32):
    for ii in seq(0, 32):
        x[32 * io + ii] = ...
```

(a) RoundUp tail strategy: the access to `x` may be out-of-bounds when `N` is not a multiple of 32, so Exo would need to reason about overcompute.

```
x: i8[N]
for io in par(0, N / 32):
    base = min(32 * io, N - 32)
    for ii in seq(0, 32):
        x[base + ii] = ...
```

(b) ShiftInwards tail strategy: invalid Exo code since Exo lacks support for `min` in index expressions and also cannot reason about safe write races.

Figure 4.5: Examples of different tail strategies that are necessary for efficient parallelization. Neither one can currently be supported in Exo.

4.4.3 Instruction Selection with Constants

In the unsharp masking schedule, we needed to write a custom function which bound all of the constants into an Exo variable due to an Exo limitation. The Exo system uses Unification to perform instruction selection. Unfortunately, there is no way to directly unify the body of Figure 4.6a with the corresponding Exo instruction in Figure 4.6b. As a consequence, we must manual bind all of the constants in the body of the procedure into variables first before we can perform the instruction selection. This is mostly an inconvenience than a limitation, but it would be nice to fix for future use.

```
@proc
def foo(x: f32[8]:
    for i in seq(0, 8):
        x = 1.0
```

(a) Example `Proc` whose body we want to replace with a vectorized load instruction.

```
@instr("{out_data} = _mm256_broadcast_ss(&{val_data});")
def mm256_broadcast_ss(out: [f32][8] @ AVX2, val: [f32][1]):
    assert stride(out, 0) == 1

    for i in seq(0, 8):
        out[i] = val[0]
```

(b) The Exo instruction corresponding to a vectorized load

Figure 4.6: Example of invalid unification. The body of the procedure on the left does not unify with the body of the instruction on the right because unification does not equate a constant float with a scalar variable.

Chapter 5

Scheduling for Parallel Processors

In the original publication, Exo was only scheduled sequentially executed code for single-threaded processors. However, many high-performance computing kernels benefit immensely from parallel execution due to the abundant parallelism inherent in the underlying algorithms. In this section, we explore how we extended the existing scheduling system so that programmers can control parallelism without breaking functional equivalence with the original algorithm.

We aim to design a system which can schedule both simple parallel for loops (e.g. OpenMP) and more sophisticated parallel execution models such as that of CUDA, which has a memory hierarchy and a variety of synchronization primitives. In the first section, we will explain how we support simple parallel for loop *without races*. We describe safety checks which verify that the serial execution and parallel execution always produce the same results. In the second section, we will discuss ideas for how to extend the system to support more complex parallel processors such as NVIDIA GPUs. This includes design choices for how to represent the different memory hierarchies and synchronization primitives exposed by CUDA. This work was ultimately not implemented, but we hope that these preliminary discussions will inform future work.

5.1 Scheduling Simple Parallel For Loops Without Races

We introduce a new scheduling operator, `parallelize_loop`, which converts a sequentially for loop into a parallel for loop (Figure 5.1). Each iteration of the loop is be executed by an independent parallel thread.

```
for in seq(0, N):
    ...
```

```
for in par(0, N):
    ...
```

Figure 5.1: Parallelizing a for loop in Exo.

While the operation's transformation is simple, the associated safety check is anything but simple. Under the parallel operational semantics, there are no guarantees on order of execution between two concurrently executing threads – they could be arbitrarily interleaved. The Exo system needs to verify the functional equivalence between the sequential and parallel operational

semantics of the code. The parallel semantics will always yield a superset of the sequential semantics (which is a singleton set), so our safety check needs to verify that the parallel semantics also yield a singleton set.

Rather than checking parallel functional equivalence before and after each scheduling operation, we opt to do the check right before compilation. Otherwise, every existing scheduling operation would need to verify parallel functional equivalence, which is a significant amount of implementation work. We implement a more conservative check which potentially rejects functionally equivalent versions because this conservative check has a simpler implementation. We check that each iteration of the for loop is *non-interfering* with the other iterations. Two sections of code are considered *non-interfering* if all memory locations written by one of them are neither read nor written by the other. Non-interfering threads will avoid write-after-read, read-after-write, and write-after-write data hazards while still allowing for read-after-read access patterns. When the check passes, the scheduling operator simply replaces the sequential for loop with a parallel for loop, as show in Figure 5.1.

This simple modeling of parallel for loops is sufficient to schedule image processing algorithms such as blur and unsharp masking with state-of-the-art performance on multi-core processors, as previously shown in Section 4.3.1 and Section 4.3.2. However, for a GPU, which has a much more complicated parallel execution model, this approach is far from sufficient. In the next sections, through a series of case studies, we will highlight features that we believe are necessary for a well-designed scheduling system for GPU programming.

5.2 Scheduling for GPUs

CUDA is the programming model for NVIDIA GPUs. We are not interested in reproducing CUDA programming model where users write thread-level programs. This programming model often shields users from many performance-relevant details about how the code is executed. However, since scheduling languages guarantee functional equivalence, the intermediate object code does not need to prioritize clear expression of the *algorithm*: what is being computed. Rather, the intermediate object language should focus on clearly expressing the *schedule*: how the program is being executed. Throughout this section, we propose augmentations of the Exo object language so as to better represent execution of parallel programs beyond simple parallel for loops, using GPUs as our primary target of interest.

While we did not implement the ideas in this section, we make note that since our proposed programming model diverges from CUDA's programming model, we will need a backend compiler which translates from our Exo object language to CUDA code.

5.2.1 Parallel Hierarchy

An individual GPU's parallel hierarchy ranges from individual threads to warps, thread blocks, and thread block clusters. In CUDA's thread-level programming model, users can organize computation at different levels of the hierarchy by predicating on index variables, as shown in Figure 5.2a. Alternatively, for less standard subdivisions of the parallel hierarchy, CUDA offers Cooperative Groups.

```
if (blockIdx.x < 5) {
  if (threadIdx.x < 32) {
    ... // warp 1, step 1
    __syncthreads()
    ... // warp 1, step 2
  }
  else if (threadIdx.x < 64) {
    ... // warp 2, step 1
    __syncthreads()
    ... // warp 2, step 2
  }
}
```

(a) CUDA

```
scope: _[5, 64]
for blk_x, blk_s in scope.par(0, 5, dim='blk_x'):
    warp1, warp2 = blk_s[0:32], blk_s[32:64]
    match fork:
        case warp1:
            for i, _ in warp1.par(0, 32, dim='thd_x'):
                ... # warp 1, step 1
        case warp2:
            for i, _ in warp2.par(0, 32, dim='thd_x'):
                ... # warp 2, step 1
    syncthreads()
    match fork:
        case warp1:
            for i, _ in warp1.par(0, 32, dim='thd_x'):
                ... # warp 1, step 2
        case warp2:
            for i, _ in warp2.par(0, 32, dim='thd_x'):
                ... # warp 2, step 2
```

(b) Sketch of proposed Exo interface (Unfortunately, this switch statement is not valid Python syntax).

Figure 5.2: Managing the parallel hierarchy in CUDA vs in Exo

We propose an interface similar to CUDA's cooperative groups. Users should manage an explicit representation of all the parallel threads which are active in each section of the code, as illustrated in Figure 5.2b. Initially, we have a variable scope which captures all of the available parallelism at the start of the kernel. When Exo users want to subdivide into smaller parallel partitions which execute different code, they must explicitly subdivide the scope and invoke a *fork*. At the end of a fork, the program should join back together.

In terms of top-level program organization, rather than organizing the instructions based on the *warp (or other parallel group)* that executes them, our design organizes based on *execution order*. We believe that prioritizing clear representation of execution will make it easier for performance engineers to understand the performance characteristics of their CUDA programs. While this design obfuscates the algorithm, this is not an issue for scheduling languages which automatically guarantee functional equivalence to the initial algorithm. Thus, at scheduling time, users should focus on performance concerns.

Arguably, the fork/join syntax above has too much indentation and verbosity. This can be a problem since CUDA programs often have many levels in their parallel hierarchies. One way to potentially reduce the amount of new scopes introduced is to omit the `match`/`case` statements, and implicitly parse the fork/join groups based solely on the `for` loops. However, this would require the Exo compiler to automatically parse the different fork/join groups, and may not accurately reflect the intent of the user. We prioritized clarity over concision in this design.

5.2.2 Memories annotated with scopes

As a motivating example, suppose we augment the previous example by adding local memory allocations, as shown below in Figure 5.3a. When the executing thread is in the first warp, we allocate a float, but otherwise, we allocate a double. In general, our programming model should expose a way for users to control memory allocations under each parallel scope.

```
if (threadIdx.x < 32) {
    float x;
    // warp 1, step 1
    __syncthreads();
    // warp 1, step 2
} else {
    double y;
    // warp 2, step 1
    __syncthreads();
    // warp 2, step 2
}
```

(a) CUDA

```
scope: _[5, 64]
for blk_x, blk_s in scope.par(0, 5, dim='blk_x'):
    warp1, warp2 = blk_s[0:32], blk_s[32:64]

    with warp1:
        x: f32
    with warp2:
        y: f64

    match fork:
        case warp1:
            for i, _ in warp1.par(0, 32, dim='thd_x'):
                ... # warp 1, step 1
        case warp2:
            for i, _ in warp2.par(0, 32, dim='thd_x'):
                ... # warp 2, step 1
    syncthreads()
    match fork:
        case warp1:
            for i, _ in warp1.par(0, 32, dim='thd_x'):
                ... # warp 1, step 2
        case warp2:
            for i, _ in warp2.par(0, 32, dim='thd_x'):
                ... # warp 2, step 2

```

(b) Sketch of proposed Exo interface (Unfortunately, this switch statement is not valid Python syntax).

Figure 5.3: Managing local memory usage in CUDA vs in Exo

As shown in Figure 5.3b, we can use `with warp1` to tie the memory allocation to a particular parallel scope. This enables us to continue using our fork/join programming model without introducing extraneous memory usage in scopes that do not use them. At compile time, we will still generate the CUDA code in Figure 5.3a.

5.2.3 Scheduling Synchronization Operations

In order to guarantee safe parallel code, users must insert the proper synchronization instructions into their kernel. We propose a design where users arbitrarily insert synchronization into their programs during scheduling. We propose that parallel functional equivalence be checked after scheduling, during the phase in which Exo compiles and generate CUDA code.

In this design, the Exo system does not need any checks when inserting synchronization into code. Under sequential semantics, synchronization is equivalent to a no-op. Users can insert synchronization into programs is via the `insert_noop_call` scheduling operation, which can arbitrarily inject procedure calls to procedures which are equivalent to a no-op (which is **`pass`** in Exo's object language). This scheduling operation is already used for inserting prefetch instructions into kernels.

In this work, we do not devise an analysis which checks parallel functional equivalence under the various synchronization primitives. However, there is concurrent work which aims to improve the power of Exo's analysis to perform such checks.

5.2.4 Scheduling Asynchronous Operations

One unfinished aspect of this design is how to model the execution of asynchronous instructions. For instance, CUDA offers asynchronous memory copies via the `async_memcpy` command, which is essential for pipelined kernels which overlap data movement with compute. A thread will initiate the command, which is then executed asynchronously, as if by another thread. Likewise, the new tensor cores offer a warp-group matrix multiply accumulate operation which asynchronously operates on tensor cores.

These asynchronous instructions pose several challenges to our current design. Firstly, our current notion of parallel scopes does not represent these other parallel execution pathways. Secondly, the backend parallel functional equivalence check would need to understand how each of these asynchronous operations synchronize with the threads that launch the operation.

Chapter 6

Conclusion and Future Work

Performance Bottlenecks of Composing Atomic Edits The current implementation has some flaws which could potentially degrade the compiler's performance. For instance, each atomic edit requires regenerating the IR for all parents of the edited subtree. When many edits are all being performed in the same subtree, this generates excessive intermediate IRs. One possible solution is to enable independent edits to happen simultaneously and avoid generating the unnecessary intermediate IRs. In addition, forwarding function composition is potentially expensive for operations which involve many atomic edits. One remedy is to "squash" the forwarding functions into a single complicated function. Currently, forwarding is not the main performance bottleneck, but it is worth monitoring whether the bottleneck shifts as users write more complex schedules.

Exo Limitations Preventing Features in Halide Library We already elaborated on this at length in Section 4.4. Addressing the limitations of Exo's functional equivalence analysis is ongoing work, which will help improve code generation for prologue loops. However, Exo's limitations on complex indexing expression is more difficult to address, and currently precludes tail strategies required for certain parallelization strategies.

Expanding Scheduling Libraries in Exo Currently, there are two large libraries that have been built in Exo's scheduling language. There is the Halide library that we have descried in great detail in Chapter 4, but there is also library built specifically to optimize the Basic Linear Algebra Subroutines (BLAS), which is described in [19]. However, there are plenty more domains and hardwares that may require bespoke optimizations that could be shared across many algorithms. Building new libraries which can help more productively performance engineer in those settings is an interesting future direction of work.

Improved Design for Scheduling Code for Parallel Processors Firstly, even our implementation for simple parallel for loops has room for improvement. A limitation of our non-interfering safety check is that we cannot represent algorithms which are safe even in spite of data races. For an example, we refer to the example presented in Section 4.4.2.

This thesis has presented our initial brainstorming for such a scheduling system. However, there is room for substantial iteration on refining the design and solving new challenges that may arise in the future. Furthermore, as discussed in Chapter 2.3, there are many possible extensions of GPU work once the initial design has been finalized.

References

[1] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "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*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530, ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. [Online]. Available: https://doi.org/10.1145/2491956.2462176.

[2] T. Chen, T. Moreau, Z. Jiang, *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594, ISBN: 978-1-931971-47-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291168.3291211.

[3] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, Oct. 2017. DOI: 10.1145/3133901. [Online]. Available: https://doi.org/10.1145/3133901.

[4] Y. Hu, T. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: A language for high-performance computation on spatially sparse data structures," *ACM Trans. Graph.*, vol. 38, no. 6, 201:1–201:16, 2019. DOI: 10.1145/3355089.3356506. [Online]. Available: https://doi.org/10.1145/3355089.3356506.

[5] A. Reinking, G. Bernstein, and J. Ragan-Kelley, *Formal semantics for the halide language*, Oct. 2022. arXiv: 2210.15740 [cs.PL].

[6] M. B. S. Ahmad, A. J. Root, A. Adams, S. Kamil, and A. Cheung, "Vector instruction selection for digital signal processors using program synthesis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 1004–1016, ISBN: 9781450392051. DOI: 10.1145/3503222.3507714. [Online]. Available: https://doi.org/10.1145/3503222.3507714.

[7] A. J. Root, M. B. S. Ahmad, D. Sharlet, A. Adams, S. Kamil, and J. Ragan-Kelley, "Fast instruction selection for fast digital signal processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 125–137. DOI: 10.1145/3623278.3624768. [Online]. Available: https://doi.org/10.1145/3623278.3624768.

[8] A. Kothari, A. R. Noor, M. Xu, H. Uddin, D. Baronia, S. Baziotis, V. Adve, C. Mendis, and S. Sengupta, "Hydride: A retargetable and extensible synthesis-based compiler for modern hardware architectures," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 514–529. DOI: 10.1145/3620665.3640385. [Online]. Available: https://doi.org/10.1145/3620665.3640385.

[9] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718, ISBN: 9781450392655. DOI: 10.1145/3519939.3523446. [Online]. Available: https://doi.org/10.1145/3519939.3523446.

[10] H. Genc, S. Kim, A. Amid, *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[11] Y. Ikarashi, K. Qian, S. Droubi, A. Reinking, G. Bernstein, and J. Ragan-Kelley, *Growing a scheduling language*, In review, 2024.

[12] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, 32:1–32:12, 2012. DOI: 10.1145/2185520.2185528. [Online]. Available: https://doi.org/10.1145/2185520.2185528.

[13] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel ir for high-performance gpu code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[14] B. Hagedorn, J. Lenfers, T. Koehler, S. Gorlatch, and M. Steuwer, *A language for describing optimization strategies*, 2020. arXiv: 2002.02268 [cs.PL].

[15] *Cuda c++ programming guide*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 08/06/2024).

[16] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19, ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. [Online]. Available: https://doi.org/10.1145/3315508.3329973.

[17] *Pytorch 2.0 documentation*. [Online]. Available: https://pytorch.org/get-started/pytorch-2.0/ (visited on 08/06/2024).

[18] *Thunderkittens*. [Online]. Available: https://github.com/HazyResearch/ThunderKittens (visited on 08/06/2024).

[19] S. Droubi, "Exoblas: Meta-programming a high-performance blas via scheduling automations," M.S. thesis, MIT, 2024.