Exo 2: Growing a Scheduling Language

Yuka Ikarashi
MIT CSAIL
Cambridge, USA

Kevin Qian
MIT CSAIL
Cambridge, USA

Samir Droubi
MIT CSAIL
Cambridge, USA

Alex Reinking
Adobe
Cambridge, USA

Gilbert Louis Bernstein
University of Washington
Seattle, USA

Jonathan Ragan-Kelley
MIT CSAIL
Cambridge, USA

Abstract

User-schedulable languages (USLs) help programmers productively optimize programs by providing safe means of transforming them. Current USLs are designed to give programmers *exactly* the control they want, while automating all other concerns. However, there is no universal answer for what performance-conscious programmers want to control, how they want to control it, and what they want to automate, even in relatively narrow domains. We claim that USLs should, instead, be designed to grow. We present Exo 2, a scheduling language that enables users to define *new scheduling operations* externally to the compiler. By composing a set of trusted, fine-grained primitives, users can safely write their own scheduling library to build up desired automation. We identify *actions* (ways of modifying code), *inspection* (ways of interrogating code), and *references* (ways of pointing to code) as essential for any user-extensible USL. We fuse these ideas into a new mechanism called *Cursors* that enables the creation of scheduling libraries in user code. We demonstrate libraries that amortize scheduling effort across more than 80 high-performance kernels, reducing total scheduling code by an order of magnitude and delivering performance competitive with state-of-the-art implementations on three different platforms.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: user-schedulable languages, meta-programming, performance engineering, high-performance computing

ACM Reference Format:
Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 35 pages. https://doi.org/10.1145/3669940.3707218

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03.
https://doi.org/10.1145/3669940.3707218

1 Introduction

The process of optimizing a program consists of repeatedly modifying it into new programs which compute the *same result* more efficiently. User-schedulable languages (USLs) such as Halide, TVM, TACO, and Taichi offer a promising solution to increase programmer productivity when doing such optimization [10, 28, 39, 54, 56]. USLs reify this optimization process into an explicit *scheduling meta-program* that transforms an underlying *object program* into a better-performing one. By providing formal or informal guarantees that scheduling transformations preserve the equivalence of the object program, USLs enable performance engineers to focus on optimization strategies rather than painstakingly ensuring correctness [57].

Some non-user schedulable languages, such as C/C++ and Python, are designed to "grow" to large projects through the development and composition of libraries built on top of them [63]. These libraries add richer interfaces and functionality to the language without needing modifications to the compiler. Current 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). Similar to how scripting languages like bash, sed, and awk excel at writing small scripts, USLs excel at optimizing small, individual kernels when the task aligns with the language's capabilities. However, similar to specialized scripting languages, existing USLs are not designed to grow by implementing libraries.

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 must therefore 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 USL compiler) to regain control. These failures can happen either because an optimization is not expressible with the provided controls, or because of imperfect automation. For example, prefetching was initially automatic in Halide but later became user-schedulable, and fixed-point vector instruction selection still requires substantial research and engineering efforts to improve its automation, more than a decade after Halide's initial release [2, 40, 58]. Targeting novel hardware accelerators introduces additional challenges. Halide has yet to support tensor accelerators (e.g., Gemmini, Tensor Core, TPU) because they pose many non-trivial questions about the separation of control and automation. For example, should Halide automatically handle instruction selection for accelerators as well? If so, to what extent can the existing vector instruction selection logic be reused? If not, how should Halide expose control over instruction selection to users? Such design failures are inevitable and generate pressure on USL implementers both to extend the scheduling language and to improve automation.

Furthermore, even in an ideal world where USLs provide users ample control and automation in their scheduling operations, users would still face challenges when implementing realistic HPC kernels. Leading HPC libraries, such as OpenBLAS, cuDNN, and MKL, provide a wide range of configurations through their APIs. With existing USLs, users must laboriously write different schedules for each of these configurations—a cross-product of operations, data types, operational parameters, storage formats, and target architectures. This approach quickly becomes unwieldy and cannot scale effectively. Thus, providing users with means to abstract and automate schedule generation is crucial to effectively manage this vast configuration space.

Therefore, we believe we should design USLs that allow users to (i) build automation *external* to the compiler and (ii) decide to use more or less automation in different contexts. We propose Exo 2, a scheduling language that empowers users to safely build *new scheduling operations* from a set of *primitive*, equivalence-preserving operations. Exo 2 is a new version of Exo [31] and is publicly available under the MIT license[1]. By composing safe primitives, users can write libraries external to the compiler and automate common optimizations, allowing schedule reuse across different kernels while retaining safety guarantees. When users require more control, they can always fall back to using lower-level primitives, satisfying both aforementioned criteria.

Drawing an analogy from the classic perception-control model in robotics and cognition, we conceptualize scheduling languages through the **actions** they take on object code (Section 3), their **inspection** or perception of that code (Section 4), and the **references** pointing to the object code (Section 5). Exo 2's reference mechanism, *Cursors*, binds these

[1]https://github.com/exo-lang/exo

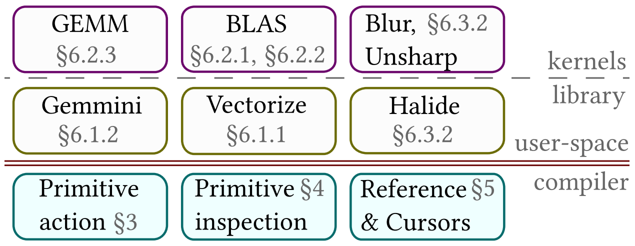


Figure 1. Paper overview

three pillars together and enables the creation of scheduling libraries within user code (Figure 1, middle). With this, we demonstrate libraries which amortize scheduling effort across more than 80 high-performance kernels, including BLAS level 1, 2, GEMM, blur, and unsharp mask (Figure 1, top). These libraries reduce total scheduling code by an order of magnitude and deliver performance comparable to or better than MKL, OpenBLAS, BLIS, and Halide on AVX2, AVX512, and Gemmini platforms.

The full version of this paper, including appendices with a list of all Exo 2 primitives, library and kernel code for GEMM, and performance graphs for BLAS, can be found in [32].

2 Preliminaries

```
def gemv(M: size, N: size,
         A: f32[M, N] @DRAM,
         x: f32[N] @DRAM,
         y: f32[M] @DRAM):
  assert M % 8 == 0
  assert N % 8 == 0
  for i in seq(0, M):
    for j in seq(0, N):
      y[i] += A[i, j] * x[j]
```

We implemented Exo 2 on top of the Exo *object* language [31]. Exo object code will be framed in boxes to distinguish it from Exo 2 *scheduling* code (without the frame) throughout the paper. We introduce Exo with a matrix-vector multiply example (gemv, inset above), although the specifics of Exo are mostly incidental to this paper. Both procedure arguments and variable declarations follow the syntax $\langle name\rangle$:$\langle type\rangle$[$\langle size\rangle$] @ $\langle memory\rangle$. In gemv, f32 is a numeric type for 32-bit floating point. The $\langle size\rangle$s may be constants, or refer dependently to other arguments (e.g., [M,N]) as they do here. The @ symbol, followed by a memory space identifier, specifies the memory space where the variable or argument resides (e.g., @DRAM). The asserts guarantee that all sizes are even multiples of 8, which can be exploited during scheduling. Finally, `for i in seq(0, M)` is a *sequential* for loop that iterates from 0 to $M - 1$, inclusive.

In Exo 2, schedules are Python meta-programs that take a procedure (e.g., gemv) 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 5, the problem of *reference* is a fundamental issue in scheduling languages. To get us off the ground, we will introduce two mechanisms for identifying object code. First, we can refer to parts of the object code by *name*. For example, we can divide the `'i'` loop of `gemv` simply by naming it: `divide_loop(gemv, 'i', ...)`. Second, we can refer to more complex or specific pieces of code structurally using more general *patterns*. For example, we could refer to that same `'i'` loop in `gemv` with the pattern `'for i in _: _'`. This pattern matches a loop with the iteration variable `i`. The wildcards (`_`) match any range specification and any loop body. Finally, we reify these references in objects we call *Cursors*, drawing an analogy to cursors in text editors. Procedures expose `find_loop` and `find` methods that take loop names or patterns and return matching cursors:

```
cur_0 = gemv.find_loop('i')          # loop name
cur_1 = gemv.find('for i in _:_')    # pattern
assert(cur_0 == cur_1)  # points to the same loop
```

Most scheduling operators take cursors for any reference arguments, though they often also accept pattern strings as an optional convenience. For example, `divide_loop(p, 'i', ...)` is shorthand for `divide_loop(p, p.find_loop('i'), ...)`.

3 Action

The core of any USL is a set of scheduling operations tailored to specific applications, optimization classes, and target programmers. In order to design a USL for growth, the set of scheduling operations should enable users to (i) develop libraries for automation while (ii) retaining the ability to resort to low-level control when necessary. While many USLs' scheduling operation sets are designed to fit specific use cases, Exo 2's design revolves around composing low-level primitives. Instead of building in scheduling operations closely matched to what we think users will want, we argue that scheduling languages should focus on providing the essential composable building blocks to ultimately be able to achieve those same transformations. We call these *scheduling primitives*. Primitives should be fine-grained, offering control over low-level details and enable a wide range of transformations through composition. Although composition can always provide abstraction, it cannot provide *finer-grained* control than the underlying primitives themselves. Primitives should also be *safe*: the language implementation should verify that the program remains functionally equivalent after applying a primitive. `divide_loop` and `lift_scope` are examples of such primitives used in the following sections. We implemented a rich set of 46 scheduling primitives for Exo 2, which is shown in Appendix A [32]. These low-level primitives provide users with precise control but reduce the degree of automation provided. In the following sections, we show various ways of composing these scheduling primitives to build automation.

3.1 Sequential Composition of Primitives

Suppose we want to write a schedule that tiles a loop nest, 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 `gemv` by sequentially composing `divide_loop` and `lift_scope` primitives as follows. For brevity, we abbreviate `gemv` to just `g`.

```
g = divide_loop(g,'i',8,['io','ii'],perfect=True)
g = divide_loop(g,'j',8,['jo','ji'],perfect=True)
g = lift_scope(g,'jo')
```

```
for io in seq(0, M/8):
  for jo in seq(0, N/8):
    for ii in seq(0, 8):
      for ji in seq(0, 8):
        y[8*io+ii] += A[...]*x[...]
```

Since we know that the factor 8 perfectly divides both `M` and `N`, 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 `gemv` object code as shown above.

3.2 Reusable Schedule Fragments as Functions

Since tiling is a common optimization, many existing USLs provide built-in tile scheduling operations. However, there is no need to provide tiling as a built-in: function abstraction offers a natural way to encapsulate such scheduling patterns as a reusable, user-level scheduling function.

```
def tile2D(p,               # procedure
           i_lp, j_lp,      # loop names
           i_itrs, j_itrs,  # list of new names
           i_sz, j_sz):     # tile sizes
  p = divide_loop(p,i_lp,i_sz,i_itrs,perfect=True)
  p = divide_loop(p,j_lp,j_sz,j_itrs,perfect=True)
  p = lift_scope(p, j_itrs[0])
  return p
```

Now, instead of calling scheduling primitives directly, we achieve the same transformation with a single call to `tile2D`:

```
g = tile2D(g,'i','j',['io','ii'], ['jo','ji'], 8,8)
```

Functionally, `tile2D` has the same behavior as if it were a built-in scheduling operation. Each Exo 2 primitive has the type $\text{Op} = \text{Proc} \times \text{Cursor} \times \ldots \rightarrow \text{Proc}$, taking a procedure, a cursor, and other arguments and returning a procedure. This design contrasts with the more common method chaining style used in Halide [55], TVM [10], and the original scheduling language design of Exo [31], where scheduling operators are methods on a program object. Our design enables users to create user-defined scheduling operations with the same type, allowing for seamless integration of primitives and user-defined functions. With the expected type Op, `tile2D` is indistinguishable from a built-in scheduling primitive.

3.3 Schedules with Control Flow

Traditional control flow constructs are useful when developing new scheduling operations and automation. Conditionals enable schedules to perform different actions based on the situation, such as applying different schedules depending on the precision or hardware target. Loops allow us to parameterize the notion of repeating an action. For example, the `tile2D` function could be generalized to work on an arbitrary number of dimensions:

```
def tilenD(p, loops, new_iters, tile_sizes):
    for i, loop in enumerate(loops):
        p = divide_loop(p, loop, tile_sizes[i],
                        new_iters[i], perfect=True)
    for i, _ in enumerate(loops):
        for j in range(0, i):
            p = lift_scope(p, new_iters[i][0])
    return p
```

Since Exo 2 scheduling primitives are guaranteed to raise an error when the transformation breaks functional equivalence, exception handling mechanisms like `try`/`except` can thus implement different behaviors based on the safety of scheduling primitives. This approach generalizes `tile2D` to non-perfectly divisible loops as shown below.

```
def general_tile2D(...): # same signature as tile2D
  orig_p = p
  try:
    p = tile2D(p, ...) # call to tile2D by default
  except:
    p = divide_loop(orig_p, ..., tail="guard")
    p = divide_loop(p, ..., tail="guard")
    p = lift_scope(p, new_j_iters[0])
    p = lift_scope(p, new_j_iters[0])
  return p
```

This schedule first tries to perfectly tile the code by calling `tile2D`, and if unsafe (i.e. it raises an exception), it defaults to a general tiling schedule by re-starting the process using the tail strategy (`tail="guard"`) which adds a guard to avoid out-of-bounds accesses. Since two calls to `divide_loop` generate two `if` guards, we need to lift the outer loop (`new_j_iters[0]`) twice to achieve the desired tiled loop ordering.

In practice, there are three distinct types of user-facing errors that can occur, and it is beneficial to differentiate between them. The first type of error is the *SchedulingError*, which is raised by the compiler analysis when user code attempts to apply transformations that do not preserve functional equivalence. The second type is the *InvalidCursorError*, which is triggered when navigating to invalid locations (see Section 5.2). Lastly, there may be internal compiler errors caused by compiler implementation bugs or other internal issues. It is preferable to specify the type of error being caught, for example, by using `except SchedulingError`, rather than catching all exceptions indiscriminately.

3.4 Higher-order Scheduling Functions

We define operations of type $\widehat{\text{Op}} = \text{Proc} \times \text{Cursor} \times \ldots \to \text{Proc} \times \text{Cursor}$. Any Op, defined in Section 3.2, can be lifted to an $\widehat{\text{Op}}$ by lift $op = \lambda(p, c).(op(p), c)$. This allows us to define higher-order scheduling combinators that take $\widehat{\text{Op}}$s as arguments and produce a $\widehat{\text{Op}}$ as output.

```
def seq(*ops):
  def func(p, c, *args):
    for op in ops:
      p,c = op(p,c,*args)
    return p, c
  return func
```

```
def repeat(op):
  def func(p, c, *args):
    try:
      while True:
        p,c = op(p,c,*args)
    except:
      return p, c
  return func
```

```
def try_else(op, opelse):
  def func(p,c,*args):
    try:
      p,c = op(p,c,*args)
    except:
      p,c = opelse(p,c,*args)
    return p, c
  return func
```

```
def reduce(op, top):
  def func(p,cur,*args):
    for c in top(cur):
      p,c = op(p,c,*args)
    return p, c
  return func
```

Higher-order scheduling functions are implemented using these combinators. For example, `seq(lift_alloc, lift_alloc)` is a scheduling function that lifts an allocation twice, while `repeat(lift_alloc)` lifts an allocation as much as possible. Both functions retain the type $\widehat{\text{Op}}$, and so can be further combined as expected.

4 Inspection

Type reflection, also known as introspection for homogeneous systems and inspection for heterogeneous systems, is a meta-programming feature that enables programs to dynamically examine object code through built-in functions (e.g., `typeOf` in Haskell) or pattern matching. To the best of our knowledge, no existing USL exposes inspection of object code to users, making operations such as "unroll all loops with bounds less than 5" inexpressible. We believe that inspection is essential for effective meta-programming in user-extensible USLs. For example, when creating a vectorizer library, identifying properties such as reductions, buffer precisions, and loop-invariant vectors in the object code is crucial for determining parallelization strategies and choosing vector instructions. Figure 4 and Section 6.1.1 demonstrate how the presence of FMA instructions affects optimal staging, necessitating inspection to identify them.

Exo 2 provides type reflection through cursors, allowing users to examine standard AST properties like variable names, literal expression values, and annotations (e.g., memory spaces and precisions) at scheduling time. Cursors also support local AST navigation, accessing loop bounds (`loop.hi()`) and bodies (`loop.body()`). More complex inspection operations can be built by combining inspection and navigation primitives. For instance, bounds inference, which

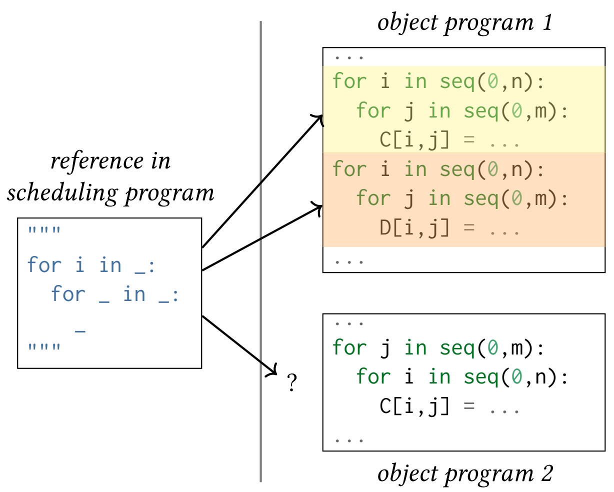


Figure 2. Schedules need mechanisms to *refer* to the object code they wish to modify. The meaning of a given reference expression is often dependent on context (*frame of reference*), and may refer to zero, one, or many parts of the program.

determines all possible index accesses to an array within a given scope, is provided as a *built-in* feature in Halide. In contrast, Exo 2 provides users with facilities to implement bounds inference as a new inspection operator *external* to the language, which we can then use as a building block of the Exo 2 Halide library (Section 6.3.2). As an example, consider inferring access bounds for `arr` within the `io` loop:

```
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]
```

Determining the bounds for the array `arr` can be reduced to unioning bounds for each array access. Each index expression is an affine expression consisting of constants and variables which may be either free or bound in the scope. In the index expression `32 * io + ii + 1`, `ii` is a bound variable and `io` is a free variable. Knowing that $0 \leq$ `ii` < 32, the index expression spans the window `[32 * io + 1 : 32 * io + 33]`. Our implementation combines primitive cursor inspections, which query the loop bounds expressions, with ordinary Python code which maintains the environment of free/bound variables and unions the inspected bounds.

5 Reference

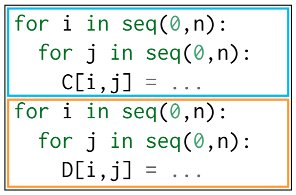
5.1 References in USLs

Nominal v.s. relative reference. Suppose we want to specify the loop nest to which we want to apply `tile2D`. Given a pattern (Figure 2, left), there is no guarantee that this reference is unique (Figure 2, program 1), or that we are referring to anything at all (Figure 2, program 2), because resolving references depends on the context, or *frame of reference*. Therefore, USLs must address questions of ambiguity and invalidity for object program references. Halide, for example, opts for a *nominal* reference system that eliminates ambiguity by design. In Halide, each statement is uniquely identified by the buffer it writes to (e.g., `blur_x`; see Figure 11 for sample code), and each loop within loop nest has a unique iterator variable (`x`, `y`, and `xi`). This allows for globally unambiguous reference of loops using the pair (`blur_x`, `y`).

Global nominal references prevent ambiguity but introduce significant limitations, requiring users to manage *globally distinct* names for each object code fragment. Without context-dependent references, it is impossible to specify transformations such as "parallelize the second most outer loop" or "unroll three stages back". Instead, schedules would be hard-coded with nominal references for specific object code. We propose that *relative* references that depend on the frame of reference are essential for building growable scheduling languages. The "nominal" and "relative" reference distinction is analogous to "proper nouns" and "noun phrases" – proper nouns must refer by name (e.g., Boston, NY), while noun phrases enable context-based references (e.g., cities on the east coast), potentially matching zero, one, or many objects. If we change the context from the US to Canada, relative references remain applicable (though they refer to different objects), while nominal references do not.

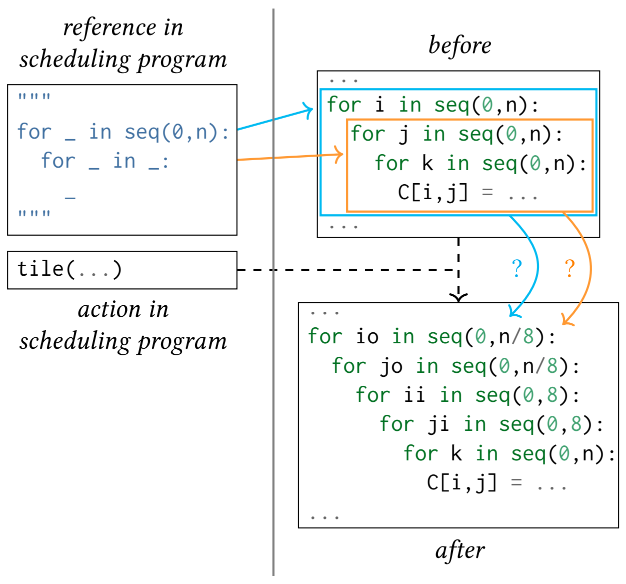
Single v.s. multiple references. In contrast to Halide, ELEVATE allows users to systematically define traversal strategies to disambiguate references with respect to a tree traversal order. For instance, `topDown` applies a rewrite at the *first* successful application, while `tryAll` applies the rewrite wherever possible. However, ELEVATE only allows having a single reference at a time. This single-frame limitation in ELEVATE can be inflexible and inconvenient because users always have to navigate relative to a single point of reference. Halide, on the other hand, allows multiple references to coexist. This means that users can refer to multiple specific points in the code simultaneously, using their unique names.

Stable v.s. one-time references. Scheduling is a programming process that evolves over time, meaning that frames of reference have both temporal and spatial dimensions. Suppose there are two references to the object code: one pointing to the loop with iterator `i` (Figure 3 light blue) and the other to the loop with iterator `j` (Figure 3 orange). If we tile loops `i` and `j`, what happens to the second reference? Has the object it's pointing to changed, or has it ceased to exist?



```
for i in seq(0,n):
  for j in seq(0,n):
    C[i,j] = ...
for i in seq(0,n):
  for j in seq(0,n):
    D[i,j] = ...
```

Now consider the same schedule applied to a different program (inset left). In this case, there is no complication because the loop nest to which the second cursor points is unaffected by tiling the first.



```
"""
for _ in seq(0,n):
  for _ in _:
    _
"""
```

```
tile(...)
```

```
...
for i in seq(0,n):
  for j in seq(0,n):
    for k in seq(0,n):
      C[i,j] = ...
...
```

```
...
for io in seq(0,n/8):
  for jo in seq(0,n/8):
    for ii in seq(0,8):
      for ji in seq(0,8):
        for k in seq(0,n):
          C[i,j] = ...
...
```

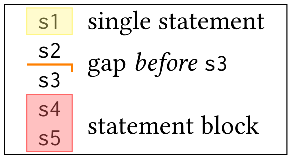
Figure 3. The process of transforming code raises the question: where should references to the original code point after a transformation is applied, or should they be invalidated?

These examples demonstrate how references are always implicitly made at a specific moment in the scheduling process. When we resolve a reference, we are essentially asking, "What does this reference denote, *right now*?" To answer this, we identify another taxonomy for USLs: *stable* references that can be reused after applying an action, and *one-time* references that get invalidated after an action. Nominal references, like those in Halide, are effective at providing stable references because they are globally unambiguous. In contrast, ELEVATE's references are one-time, which is similar to other transformative meta-programming systems like jQuery. Pattern matching, as used in these systems, is a brittle mechanism for the scheduling process. A pattern that existed at one point in the scheduling process might not exist or may point to completely different object code after applying an action (e.g., how the `for i` loop ceases to exist after tiling in Figure 3). Furthermore, pattern matching requires exact knowledge of the code structure and its potential transformations, hindering the encapsulation of scheduling operations.

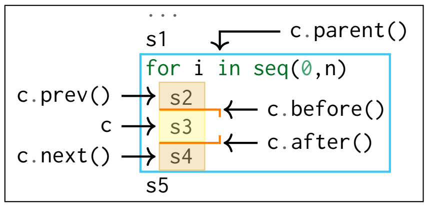
$$\begin{tikzcd}
\text{fixed :} & \bullet_p \\
\text{linear :} & \bullet_p \arrow[r] & \bullet_p \arrow[r] & \bullet_p \arrow[r] \arrow[l, bend right, "\text{rewind}"'] & \bullet_p \arrow[r] & \bullet_p \\
\text{branching :} & \bullet_p \arrow[r] & \bullet_p \arrow[r] \arrow[dr] & \bullet_p \arrow[r] & \bullet_p \\
& & & \bullet_{p'} \arrow[r] & \bullet_{p'} \arrow[r] & \bullet_{p'}
\end{tikzcd}$$

Scheduling time models in USLs. The choice of aforementioned reference designs affects the fundamental design of USLs, and consequently affects the scheduling time models for USLs: (i) The *fixed* time model (inset top) uses a single, static reference frame, largely achieved by Halide's *multiple, stable, nominal* references. (ii) The *linear* time model (inset middle) advances the reference frame for each action and allows it to "rewind" after encountering an error. ELEVATE's *single, one-time, relative* reference supports this model. (iii) The *branching* time model (inset bottom), adopted by Exo 2's *multiple, stable, relative references*, treats time as branching into parallel versions of the procedure, with cursors living at specific versions and scheduling creating different cursors for each branch. The branching time model encompasses the other time models, as we will demonstrate by recreating Halide and ELEVATE-style references in Section 6.3.

5.2 Cursors

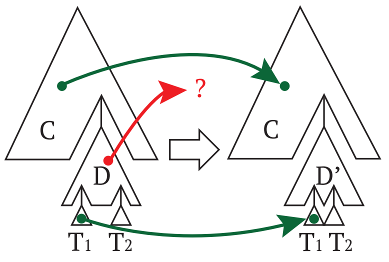


Exo 2's reference mechanism, embodied by *Cursors*, maintains *multiple, stable, relative* references. Cursors, inspired by the blinking vertical bar in text editors, allow users to select and refer to parts of the code such as expressions, blocks of statements, and gaps between statements (inset above).



Multiple, relative cursors require modulating the spatial reference frame (navigation) and querying cursors in context. Cursors enable spatial navigation within a procedure (i.e. "after _," "before _," "parent of _," etc.) to proximate locations (see inset above). Moreover, instead of invoking `proc.find(...)`, we can invoke `cursor.find(...)` to restrict our pattern match to the sub-AST identified by the `cursor`. *InvalidCursorError* is raised when navigation is invalid. For instance, `c.parent()` is invalid when `c` is already pointing to the top-level statement.

Forwarding. *Cursor forwarding* is the mechanism by which Exo 2 supports stable references. Consider a generic action which rewrites the inset left AST to the inset right AST. We can decompose the AST of the object code as $P = C[S]$, where C is the unmodified subtree and S is the modified subtree. Furthermore, we can usually identify some *invariant* sub-trees $T_1, \ldots, T_k$ of the modified sub-tree S which are not



changed by the action. Therefore, we have the further decomposition $P = C[S] = C[D[T_1, \ldots, T_n]]$. Now, we can describe the procedure after the action as $P' = C[D'[T_1, \ldots, T_n]]$ — all changes are isolated to the transformation of $D \rightarrow D'$. Given the decomposition of a program into an outer context C, a sub-tree D that is being transformed, and sub-trees $T_1, \ldots, T_n$ that replace D, any cursor pointing to a statement in C or T_i can be unambiguously forwarded after the action is completed. Cursors to gaps and statement blocks can be unambiguously forwarded if attached to statements or within C or T_i. For composite actions, if a cursor forwarding policy handles the unambiguous primitive action cases correctly, it will correctly forward cursors to any AST part untouched by all scheduling actions.

The ambiguous cases occur when the cursor points to a statement in D. There are two high-level design decisions for handling such ambiguous cases: (i) invalidate any ambiguous cursor, which ensures unambiguous behavior but makes scheduling programs more laborious to write, or (ii) attempt to produce a valid cursor whenever possible, which maximizes convenience but may lead to unintuitive behavior. Exo 2 has chosen the second approach and defines heuristic forwarding rules for each scheduling primitive. This can lead to ambiguous forwarding behavior, especially when calling scheduling library functions that compose many forwarding effects. However, since forwarding is deterministic even in ambiguous cases, we find that simply printing cursors after the function call is usually sufficient to understand the behavior in practice.

When a primitive action takes in a procedure and input cursors, Exo 2 will *implicitly* forward the cursors to the procedure's reference frame, since cursors should, by definition, always point to the procedure. Therefore, `expand_dim(p, c, ...)` becomes a shorthand for `expand_dim(p, p.forward(c), ...)`. However, navigation and forwarding do not commute in general. If `c.next()` is passed to a primitive action, the implicit forwarding behavior is equivalent to `p.forward(c.next())`, not `p.forward(c).next()`.

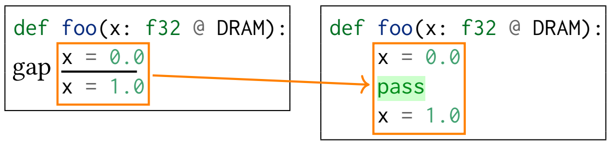
Implementation. Internally, a cursor stores two values: a reference to the procedure it's pointing at (i.e. a *time* coordinate) and a path defining its relative location inside that procedure's AST (a *spatial* coordinate). The path describes navigation in an AST as a downward traversal. In an AST, 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.

```
x: i8
for j in seq(0, 2):
  x = y + z
```

For example, in this code block, the `y` variable has path $(body, 1)$, $(body, 0)$, $(rhs, \emptyset)$, $(lhs, \emptyset)$. Traverse to the second statement of the procedure body $(body, 1)$, the first statement of the loop body $(body, 0)$, the right-hand-side of the assignment $(rhs, \emptyset)$, and then the left-hand-side of the addition $(lhs, \emptyset)$. Spatial navigation operations such as `.parent()`, `.rhs()`, `.body()[idx]`, or `.next()` are straightforward to implement by modifying this path representation. Cursors to gaps are relative to the statement nodes, and cursors to statement blocks simply store a *range* at the final path index, instead of a single value.

Whenever a scheduling action is applied to procedure p to produce a new procedure p', the Exo 2 runtime records this provenance along with an internal forwarding function. This internal forwarding function defines how to forward from a cursor c into p to a cursor c' into p'. This is done by decomposing the effect of every primitive action into atomic AST edits (insert, delete, replace, move, and wrap; as shown below), each with its canonical forwarding function.

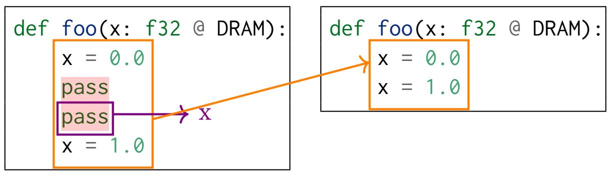
Insertion. Inserting an IR fragment into the AST preserves existing paths. The forwarding function adjusts paths through the insertion point by incrementing pre-existing paths at the appropriate tree level. The orange block cursor illustrates how existing cursors are forwarded when a new `pass` is inserted into a gap.



```
def foo(x: f32 @ DRAM):
gap x = 0.0
    x = 1.0
```

```
def foo(x: f32 @ DRAM):
    x = 0.0
    pass
    x = 1.0
```

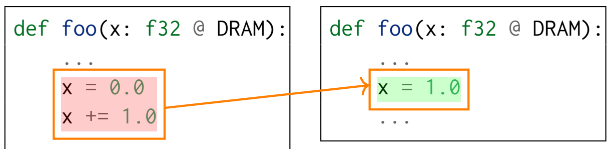
Deletion. Deleting a subtree from the AST invalidates paths within the subtree (shown as a violet cursor below), while paths outside remain valid (orange cursor). The forwarding function decrements pre-existing paths past the deletion point at the appropriate tree level.



```
def foo(x: f32 @ DRAM):
    x = 0.0
    pass
    pass
    x = 1.0
```

```
def foo(x: f32 @ DRAM):
    x = 0.0
    x = 1.0
```

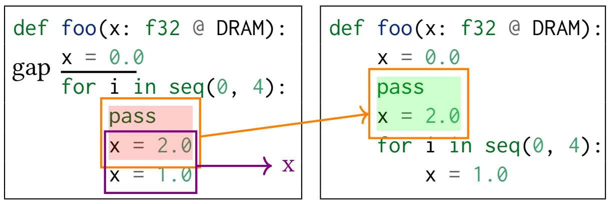
Replacement. Replacement supports higher-level operations like algebraic simplification by replacing an existing subtree (two statements with a red background below) with a new one (`x = 1.0`). The forwarding behavior is similar to inserting the new subtree and deleting the old, except the unique path to the old subtree remains valid.



```
def foo(x: f32 @ DRAM):
    ...
    x = 0.0
    x += 1.0
    ...
```

```
def foo(x: f32 @ DRAM):
    ...
    x = 1.0
    ...
```

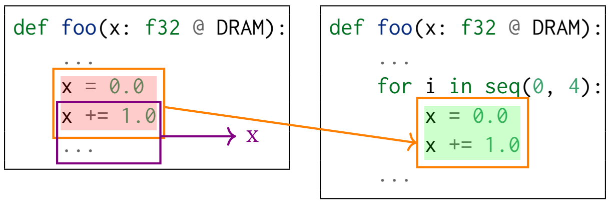
Movement. Moving a subtree (orange block cursor) to a gap within the AST preserves its node identity but invalidates cursors across the subtree boundary (violet block cursor). The forwarding function maps paths to their natural correspondences, handling cases of moving to higher or lower levels in the AST.



```
def foo(x: f32 @ DRAM):
    x = 0.0
    for i in seq(0, 4):
        pass
        x = 2.0
        x = 1.0
```

```
def foo(x: f32 @ DRAM):
    x = 0.0
    pass
    x = 2.0
    for i in seq(0, 4):
        x = 1.0
```

Wrapping. Wrapping an existing subtree (orange cursor) with a one-hole IR fragment (e.g. `for i in seq(0, 4): _`, where the body of the for loop is the hole) is equivalent to insertion with movement for blocks, but not for expressions. Consequently, we separate this function as its own atomic edit. Paths into the wrapped subtree are adjusted by inserting a step at the appropriate level to navigate into the wrapping fragment. Paths through the wrapping point are decremented at the level by the length of the replaced subtree minus one.



```
def foo(x: f32 @ DRAM):
    ...
    x = 0.0
    x += 1.0
    ...
```

```
def foo(x: f32 @ DRAM):
    ...
    for i in seq(0, 4):
        x = 0.0
        x += 1.0
    ...
```

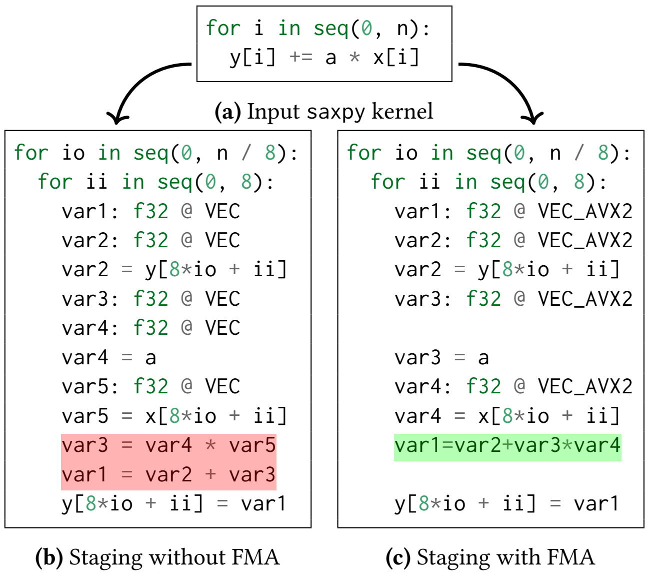
The internal forwarding function for a primitive action is the sequential composition of forwarding through its constituent atomic edits. For example, suppose `p` is the result of applying a primitive action to cursor `c`, and this primitive action was comprised of n atomic edits. When `p.forward(c)` is called, Exo 2 composes the forwarding functions (f_1, f_2, ..., f_n) for each primitive action between `p_0 = c.proc()` and `p_n = p` (the output procedure). This composition produces a single function $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$ that maps `c_0` from its location in the original procedure to the corresponding location in p_n. Therefore the forwarded cursor can be obtained by computing $f(c)$. If any forwarding function returns an invalid cursor, invalidity is maintained by each subsequent forwarding function.

6 Building Scheduling Libraries

6.1 Target-specific Functions

Exo 2's action, inspection, and reference empower users to create scheduling libraries external to the compiler. This section shows how Exo 2 enables users to automate hardware architecture-specific optimization strategies while maintaining control over low-level performance considerations.

6.1.1 Vector Architectures. We define a new `vectorize` scheduling operator in user code, which is parameterized over vector width, precision, memory type, and vector instructions, so it can be instantiated for many different vector machines. By leveraging the safety guarantees of the `fission` primitive, which checks for loop-carry dependencies, we eliminate the need for separate dependency analyses, allowing for the concise implementation of transforming loop



```
for i in seq(0, n):
    y[i] += a * x[i]
```

(a) Input saxpy kernel

```
for io in seq(0, n / 8):
  for ii in seq(0, 8):
    var1: f32 @ VEC
    var2: f32 @ VEC
    var2 = y[8*io + ii]
    var3: f32 @ VEC
    var4: f32 @ VEC
    var4 = a
    var5: f32 @ VEC
    var5 = x[8*io + ii]
    var3 = var4 * var5
    var1 = var2 + var3
    y[8*io + ii] = var1
```

(b) Staging without FMA

```
for io in seq(0, n / 8):
  for ii in seq(0, 8):
    var1: f32 @ VEC_AVX2
    var2: f32 @ VEC_AVX2
    var2 = y[8*io + ii]
    var3: f32 @ VEC_AVX2

    var3 = a
    var4: f32 @ VEC_AVX2
    var4 = x[8*io + ii]
    var1=var2+var3*var4

    y[8*io + ii] = var1
```

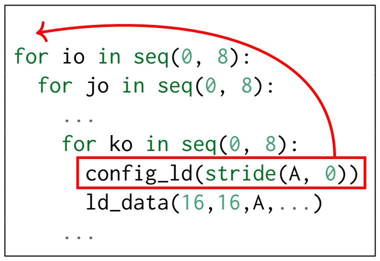
(c) Staging with FMA

Figure 4. Code transformations for staging the computation (step 3 of `vectorize`), depending on the presence of FMAs.

operations into their SIMD equivalents. Programmers can customize the vectorization process by scheduling deviations as a prologue (e.g., Common Subexpression Elimination) or epilogue (e.g., Loop Invariant Code Motion), or by incorporating hooks to modify behavior in certain cases (e.g., detecting a Fused Multiply-Add, or FMA).

```
def vectorize(p, loop, vw, precision,
              mem_type, instrs, rules=[]):
    p = divide_loop(p, loop, vw, tail="cut")
    p = parallelize_reductions(p, loop, ...)
    inner = p.forward(loop).body()[0]
    p = stage_compute(p, inner, ..., rules)
    p = fission_into_singles(p, inner)
    return replace_all_stmts(p, instrs)
```

The signature of `vectorize` follows the type Op = Proc × Cursor × ... → Proc (Section 3.2), which takes a procedure and a cursor pointing to the loop to be vectorized. The implementation follows these steps: (1) Expose parallelism by dividing the loop. (2) Parallelize all reductions in the loop. (3) Get a cursor `inner` to the innermost loop, and stage the computation into temporary assign statements, representing unary (e.g., load) or binary (e.g., addition) operations. Figure 4 shows an example object code of this step, depicting the starting loop in Figure 4a and applying the default staging in Figure 4b. Users can overwrite the default automation when necessary. For instance, Figure 4c shows more efficient staging when an FMA instruction is available. Staging behavior can be customized by `rules`, which scan the expression and specify which sub-expressions in the subtree should be recursively staged. (4) Finally, we fission between the statements to generate a loop-based representation of the SIMD operations, which are replaced with the equivalent hardware target instructions.



```
for io in seq(0, 8):
  for jo in seq(0, 8):
    ...
    for ko in seq(0, 8):
      config_ld(stride(A, 0))
      ld_data(16,16,A,...)
    ...
```

(a) The `ld_data` instruction, which reads from the configuration state, is prepended with the `config_ld` instruction to set the stride. The goal in this object code example is to hoist an expensive `config_ld` instruction out of the loops.

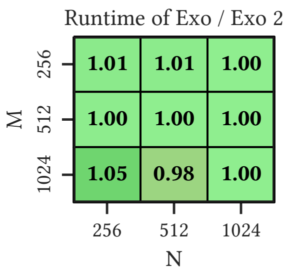
```
while True:
  try:
    try:
      while True:
        try:
          p = reorder_stmts(p, ...)
        except:
          raise Exception("...")
    except:
      p = fission(p, ...)
      p = remove_loop(p, ...)
  except:
    break
```

(b) Schedule for hoisting a single statement to the top of an object program.

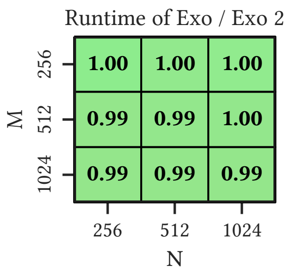
```
repeat(
  try_else(
    seq(
      fission_after,
      remove_parent_loop
    ),
    reorder_before
  )
)(p, c)
```

(c) Alternatively, the same schedule as in Figure 5b can be implemented using higher order functions defined in Section 3.4. Section 6.3.1 defines the scheduling operations combined with relative references.

Figure 5. (a) Gemmini object code and (b,c) two possible schedules for hoisting a Gemmini configuration instruction.



(a) Gemmini



(b) AVX512

	Gemmini	AVX512
Library ref	313	>1,690
Exo [31] schedule	61	180
Exo 2 schedule	29	97

(c) Lines of code comparison. Exo 2 and Exo schedules are compared against state-of-the-art reference implementations (Gemmini standard library and OpenBLAS).

Figure 6. Performance evaluation of Exo 2 against Exo on matmul (higher is better). K is set to 512.

6.1.2 Gemmini.

Gemmini is a machine learning accelerator developed at UC Berkeley that supports matrix-matrix multiplication (tensor) operations [21]. The GEMM optimization strategy for tensor operations inherently depends on the specificity of the accelerator, and Exo 2 allows users to implement such accelerator-specific optimization passes in a library, which is fully explained in Appendix B [32]. This section focuses on one of the Gemmini library functions, configuration hoisting, which is a unique and specific requirement of stateful accelerators.

Gemmini programmers must program configuration registers to influence the behavior of instructions, such as stride, activation, and quantization. The simplest approach for a compiler is to naively reset this configuration state before every operation, resulting in patterns like `config_ld` followed by `ld_data` (Figure 5a). However, these configuration instructions must be hoisted outside the inner loops to avoid expensive, redundant reissuing. Figures 5b and 5c show two schedules that hoist a single statement as much as possible: one using Python's loop and exception constructs, and the other using higher-order scheduling operations defined in Section 3.4. The statement hoisting schedule repeatedly attempts to reorder the statement to the beginning of the loop, fission the loop, and remove the enclosing loop. This schedule is then used to implement a function that hoists all Gemmini configuration statements.

We implemented matrix-matrix multiply (matmul, gemm) using Exo 2 and compared its performance with Exo. As shown in Figure 6, our implementation achieved similar or faster runtime compared to Exo, which itself is already 3.5x faster than the original Gemmini library [31], while achieving less lines of code than Exo. Performance was measured on the Chipyard version 1.9.1 running Firesim simulator on FPGA, using Gemmini version 0.7.1 [3, 37].

6.2 Linear Algebra Library

Unlike existing USLs, Exo 2 allows users to encapsulate application-specific shared scheduling strategies in libraries. This greatly simplifies the optimization of HPC kernels over various configurations—a cross-product of operations, data types, operational parameters, storage formats, and target architectures. We built a linear algebra scheduling library and used it to optimize most of the kernels in BLAS levels 1 and 2, as well as GEMM.

6.2.1 BLAS Level 1 Operations.

$O(n)$ BLAS operations can be implemented using a single loop. The optimization process involves two main strategies. First, SIMD vector parallelism can be exploited by interleaving loop iterations (modulo the vector width) and autovectorizing the resulting code. Second, the amount of work can be reduced through common subexpression elimination (CSE) and loop-invariant code motion (LICM). Increased register pressure is generally not a concern due to the low arithmetic intensity of level 1 operations. For reductions (dot, asum), computing partial sums in each vector lane exposes data parallelism for vectorization. The degree of loop interleaving is tuned to balance performance gains with register pressure. In practice, CSE and LICM do not typically cause register spilling issues or limit the amount of beneficial loop interleaving that can be performed for BLAS level 1 operations.

We developed a scheduling operator (`optimize_level_1`) that optimizes the entire BLAS level 1 kernels, including asum, axpy, dot, rot, rotm, scale, swap, copy, and dsdot, for a total of 24 kernel variants. However, due to limitations in Exo's object code, which lacks support for value-dependent control, we were unable to support nrm2 and iamax kernels. Appendix D.1 [32] offers the implementation of `optimize_level_1` and performance graphs comparing our results against OpenBLAS, MKL, and BLIS using AVX2 and AVX512 instructions. All results presented in this paper were collected on an AWS m7i.xlarge instance equipped with an Intel(R) Xeon(R) Platinum 8488C processor running at 3.2GHz. Our optimized kernels match the performance of the aforementioned libraries across the board.

6.2.2 BLAS Level 2 Operations.

We show how optimization for $O(n^2)$ BLAS level 2 operations can be shared across different kernels, precisions, and operational parameters.

General Matrix. The common optimization approach for general matrices is turning the loop into a batched program of multiple dot products, allowing vector reuse and reducing memory loads (unroll-and-jam). In contrast to normal loop unrolling, this transformation will jam c iterations from an outer loop to an inner loop and then unroll the iterations (it will also do normal unrolling for any statements around the inner loop). The following is the result of applying this to `sgmev_n` (see Figure 7a top for the starting object code) with $c = 2$:

```
def sgemv_n(...):
    for io in seq(0, M / 2):
        for j in seq(0, N): # <--- main inner loop
            y[0+2*io] += x[j] * A[0+2*io, j]
            y[1+2*io] += x[j] * A[1+2*io, j]
    for ii in seq(0, M % 2):
        for j in seq(0, N): # <--- tail inner loop
            y[ii+M/2*2] += x[j] * A[ii+M/2*2, j]
```

In the first inner loop, `x[j]` is loaded from memory twice. To optimize this, we can load `x[j]` once and store it in a register for both updates, which is equivalent to applying CSE. Our implementation treats the resulting inner loop as a level-1 problem and calls `optimize_level_1` from Section 6.2.1.

Triangular Matrix. In triangular matrices, the inner loop bounds depend on the outer loop variable, preventing direct loop reordering for data reuse. To address this, the inner loop bound is rounded up or down to a multiple of the blocking factor, removing the dependence on the outer loop variable. This allows the loops to be reordered using unroll-and-jam to enable data reuse optimizations. The exact rounding strategy depends on whether the diagonal is included and if the target machine supports predicated vector memory operations.

The combined scheduling code for general and triangular matrices, along with their performance graphs, can be found in Appendix D.2 [32] and in more detail in [19]. We optimized 50 kernel variants, supporting most BLAS level-2 operations (excluding banded operations and packed-triangular formats) across all configurations. This includes operations (gemv, ger, symv, syr, syr2, trmv, trsv), precisions (float (`s`), double (`d`)), operational parameters (transpose (`t`), non-transpose (`n`), lower (`l`), upper (`u`) triangular, unit (`u`), non-unit (`n`)), and target architectures (AVX2, AVX512). Our implementation achieved competitive performance with OpenBLAS, MKL, and BLIS on both AVX2 and AVX512 platforms.

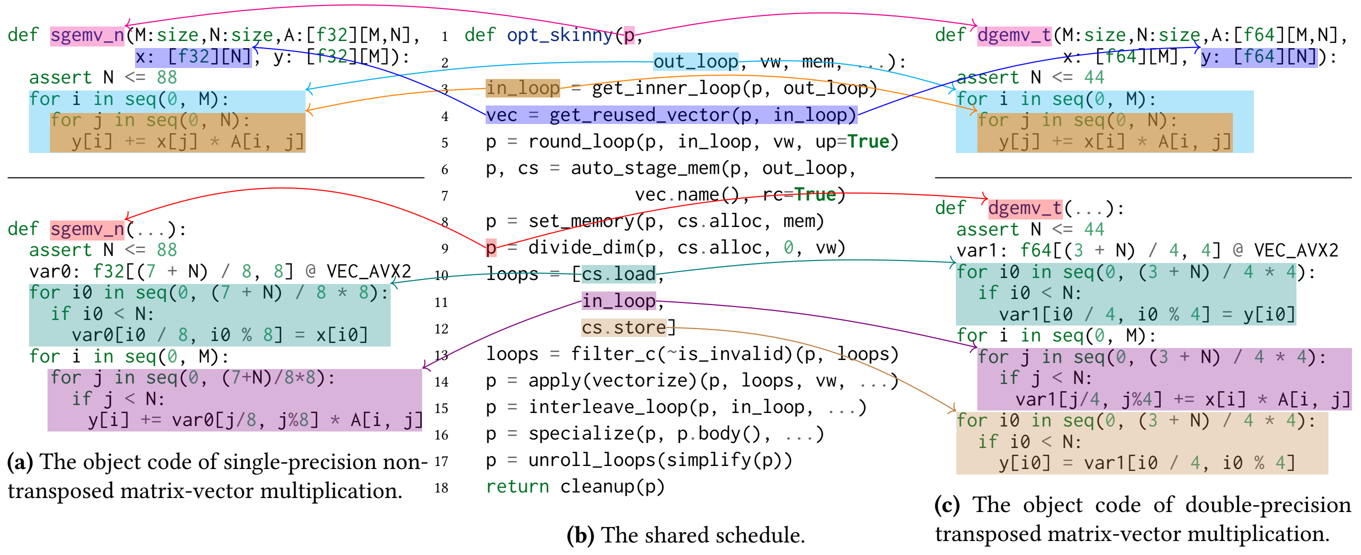
Skinny Matrix. A more efficient strategy for short vectors and skinny matrices is to load the entire vector into registers before the quadratic math loop. Since the vector is stored in registers, the vector length must be statically determined as the closest multiple of the register width, and specialized cases must be generated for each multiple.

Figure 7 shows the implementation of this shared schedule (7b) and its application on two programs (7a top) and (7c top). We developed this shared schedule for CPUs with vector extensions that support predicated vector loads/stores (e.g. x86 AVX2 and AVX512). The schedule follows four steps: (1) Inspect the program to obtain cursors to the inner loop (line 3) and the reused vector (line 4). (2) Stage the reused vector into registers by rounding up the loop iteration bound to a multiple of the vector width (line 5) and staging the vector around the doubly nested loops (line 6). The bottom of Figures 7a and 7c show the resulting object codes after this step. (3) Optimize the generated loops (load, inner math loop, and store) by applying vectorization (line 14) and interleaving the inner loop to increase ILP. (4) Specialize the program to ensure static information about the number of registers and their accesses is known for efficient code generation.

We compared the performance of the generated programs against Intel's MKL, OpenBLAS, and BLIS on problems with $N = 40$ and M having powers of 2 and 3. Figure 8 shows the results, where each heatmap cell represents the geometric mean over problems in the respective x-axis bucket. The evaluated programs also handle scaling, which was omitted for brevity in Figure 7.

6.2.3 Matrix-Matrix multiply.

The leading gemm literature, such as GotoBLAS [23] and BLIS [69], suggests building a general high-performance GEMM out of highly performant register-level tiles (micro-kernels). In addition to the main micro-kernel that fully utilizes registers and L1 cache size, we



```
def sgemv_n(M:size,N:size,A:[f32][M,N],
            x: [f32][N], y: [f32][M]):
  assert N <= 88
  for i in seq(0, M):
    for j in seq(0, N):
      y[i] += x[j] * A[i, j]
```

```
def sgemv_n(...):
  assert N <= 88
  var0: f32[(7 + N) / 8, 8] @ VEC_AVX2
  for i0 in seq(0, (7 + N) / 8 * 8):
    if i0 < N:
      var0[i0 / 8, i0 % 8] = x[i0]
  for i in seq(0, M):
    for j in seq(0, (7+N)/8*8):
      if j < N:
        y[i] += var0[j/8, j%8] * A[i, j]
```

(a) The object code of single-precision non-transposed matrix-vector multiplication.

```
def opt_skinny(p,
               out_loop, vw, mem, ...):
  in_loop = get_inner_loop(p, out_loop)
  vec = get_reused_vector(p, in_loop)
  p = round_loop(p, in_loop, vw, up=True)
  p, cs = auto_stage_mem(p, out_loop,
                         vec.name(), rc=True)
  p = set_memory(p, cs.alloc, mem)
  p = divide_dim(p, cs.alloc, 0, vw)
  loops = [cs.load,
           in_loop,
           cs.store]
  loops = filter_c(~is_invalid)(p, loops)
  p = apply(vectorize)(p, loops, vw, ...)
  p = interleave_loop(p, in_loop, ...)
  p = specialize(p, p.body(), ...)
  p = unroll_loops(simplify(p))
  return cleanup(p)
```

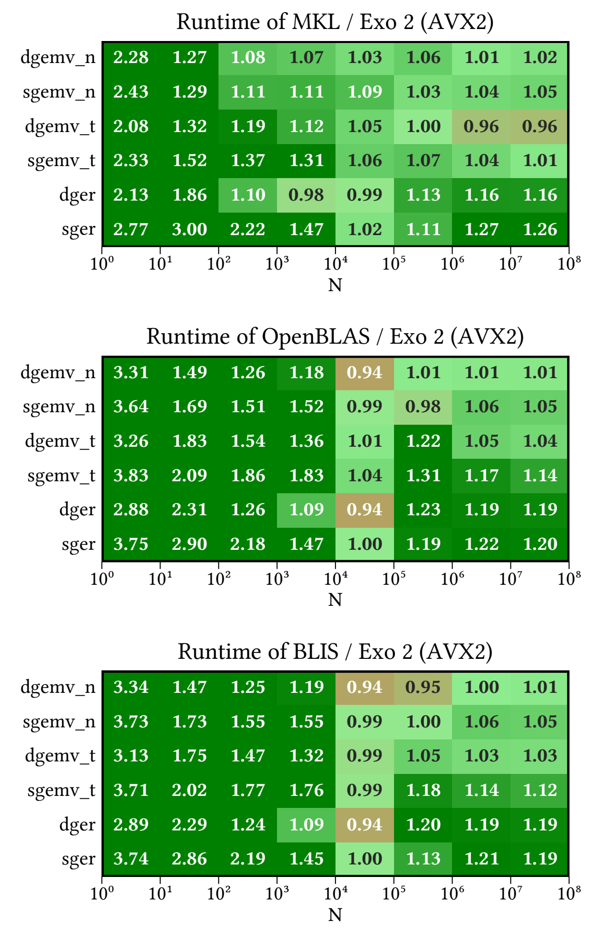
(b) The shared schedule.

```
def dgemv_t(M:size,N:size,A:[f64][M,N],
            x: [f64][M], y: [f64][N]):
  assert N <= 44
  for i in seq(0, M):
    for j in seq(0, N):
      y[j] += x[i] * A[i, j]
```

```
def dgemv_t(...):
  assert N <= 44
  var1: f64[(3 + N) / 4, 4] @ VEC_AVX2
  for i0 in seq(0, (3 + N) / 4 * 4):
    if i0 < N:
      var1[i0 / 4, i0 % 4] = y[i0]
  for i in seq(0, M):
    for j in seq(0, (3 + N) / 4 * 4):
      if j < N:
        var1[j/4, j%4] += x[i] * A[i, j]
  for i0 in seq(0, (3 + N) / 4 * 4):
    if i0 < N:
      y[i0] = var1[i0 / 4, i0 % 4]
```

(c) The object code of double-precision transposed matrix-vector multiplication.

Figure 7. 7b encapsulates BLAS level 2, skinny matrix-specific optimization in a single library function, used to optimize both transpose (7c) and non-transpose (7a) gemv and ger, amortizing the cost of writing schedules across kernel variants.



Runtime of MKL / Exo 2 (AVX2)

dgemv_n	2.28	1.27	1.08	1.07	1.03	1.06	1.01	1.02
sgemv_n	2.43	1.29	1.11	1.11	1.09	1.03	1.04	1.05
dgemv_t	2.08	1.32	1.19	1.12	1.05	1.00	0.96	0.96
sgemv_t	2.33	1.52	1.37	1.31	1.06	1.07	1.04	1.01
dger	2.13	1.86	1.10	0.98	0.99	1.13	1.16	1.16
sger	2.77	3.00	2.22	1.47	1.02	1.11	1.27	1.26

Runtime of OpenBLAS / Exo 2 (AVX2)

dgemv_n	3.31	1.49	1.26	1.18	0.94	1.01	1.01	1.01
sgemv_n	3.64	1.69	1.51	1.52	0.99	0.98	1.06	1.05
dgemv_t	3.26	1.83	1.54	1.36	1.01	1.22	1.05	1.04
sgemv_t	3.83	2.09	1.86	1.83	1.04	1.31	1.17	1.14
dger	2.88	2.31	1.26	1.09	0.94	1.23	1.19	1.19
sger	3.75	2.90	2.18	1.47	1.00	1.19	1.22	1.20

Runtime of BLIS / Exo 2 (AVX2)

dgemv_n	3.34	1.47	1.25	1.19	0.94	0.95	1.00	1.01
sgemv_n	3.73	1.73	1.55	1.55	0.99	1.00	1.06	1.05
dgemv_t	3.13	1.75	1.47	1.32	0.99	1.05	1.03	1.03
sgemv_t	3.71	2.02	1.77	1.76	0.99	1.18	1.14	1.12
dger	2.89	2.29	1.24	1.09	0.94	1.20	1.19	1.19
sger	3.74	2.86	2.19	1.45	1.00	1.13	1.21	1.19

Figure 8. Performance evaluation of Exo 2 against Intel MKL, OpenBLAS, and BLIS on gemv and ger variants. _n suffix means non-transpose, and _t means transpose.

	BLAS-lib	std-lib	ins-lib	Level 1	Level 2	sgemm
Obj.	0	0	0	90	139	11
Schd.	241	1089	449	18	34	97
Gen. C	N/A	N/A	N/A	3196	11040	521

(a) Breakdown of the lines of code distribution in our library and kernels. std-lib refers to higher-order and target-specific functions like `vectorize`, BLAS-lib represents BLAS-specific optimizations such as `opt_skinny` and `optimize_level1`, and ins-lib refers to the inspection library functions.

kernel	rewrites	kernel	rewrites	kernel	rewrites
asum	1760	axpy	1826	dot	1208
rot	1716	rotm	3372	scal	1308
swap	1484	gemv	3996	ger	2848
symv	3310	syr2	7832	syr	4046
trmv	4828	trsv	5370	sgemm	756

(b) Number of primitive rewrites required for optimizing each kernel. This includes the generation of all configurations, such as single and double precision, transpose and non-transpose operations, stride-1 and stride-any cases, and other variations (e.g., upper or lower triangular, and unit or non-unit options).

Figure 9. (a) Lines of code and (b) the number of primitive rewrites for library functions and kernels from Section 6.2. sgemm refers to the AVX512 sgemm kernel.

also need to generate smaller micro-kernels of size $m \times 16n$, where either $m = 6$ and $n \leq 4$, or $m \leq 6$ and $n = 4$. Generating these micro-kernels by hand is laborious and typically written in assembly in the aforementioned literature.

We implemented a single scheduling function, `gen_ukernel`, for all the micro-kernel generation, parameterized by precision, vector width, vector predicates, and the micro-kernel sizes N and M. The `gen_ukernel` function stages memory for registers, applies `vectorize` (see Section 6.1.1), and inserts calls to AVX512 intrinsics. This function is used to generate all the micro-kernels in the main, right, and bottom panels (see Appendix C [32]).

Figure 9 shows the lines of code in the scheduling library and the number of primitive rewrites performed to optimize each kernel. Exo 2 reduces scheduling complexity by allowing users to implement new scheduling operators and reuse them for the optimization of multiple kernels rather than hand-writing primitives for each kernel one by one. We show the number of primitive rewrites required for scheduling kernels as it reflects what users would directly write in Exo. This may slightly overstate the difference compared to an equivalent best-effort Exo schedule, but accurately reflects the order of magnitude improvement we observe with Exo 2. Moreover, we compare the performance of the Exo 2 generated GEMM against the Exo generated GEMM. As shown in Figure 6, our implementation achieved similar or faster runtime compared to Exo's implementation (which is comparable to MKL's) while requiring fewer lines of code.

6.3 Reproducing Existing Scheduling Languages

Existing USLs hardcode actions and referencing schemes, limiting users to specific design choices. For instance, ELEVATE limits users to a linear time model and actions controlled by traversal functions. Halide limits users to a well-chosen set of scheduling primitives and a fixed-time, nominal referencing scheme. In contrast, Exo 2 allows users to choose referencing schemes and the level of automation in scheduling operators. By reproducing ELEVATE-style traversal functions, Halide's scheduling operations, and their respective referencing schemes within the user-level library code, we demonstrate that Exo 2 can recreate even complex actions such as `compute_at` and `store_at`, and show how Exo 2's branching time model encompasses the linear and fixed time reference models employed by other USLs. Our approach gives users not only the choice of operators and references but also enables interoperability between different user-defined operators and referencing schemes within the same program as needed.

6.3.1 Reproducing ELEVATE.

Traversal functions. Traversal functions specify a traversal order starting from a cursor and have type Top = Cursor $\rightarrow$ Stream[Cursor]. In rewrite-based USLs, users must explicitly control the order in which scheduling actions are applied to the object code. This is because the sequence of actions does not commute in general, even when the same scheduling primitive is repeatedly applied, similar to the phase-ordering problem in general compilers. For example, suppose there is a three-nested loop of `i, j, k`. Applying `lift_scope` on `j` and then `k` will yield a loop nest of `j, k, i`, while applying it on `k` and then `j` will yield a loop nest of `i, j, k`. Therefore, it is necessary to define rules for different traversal strategies.

Strategy languages [43, 68] and USLs like ELEVATE separate traversal strategies from rewrite rules. By leveraging navigation and introspection, we can support traversal strategies over cursors, such as the postorder traversal (`lrn`).

```
def lrn(c):
  for c in c.body():
    if isinstance(c,
        (ForCursor,IfCursor)):
      yield from lrn(c)
    yield c
```

Linear time. Moreover, we can reproduce a functional-style linear-time referencing model using higher-order functions. Cursors were implicitly forwarded and returned in all the higher-order scheduling functions we discussed earlier (see Section 3.4). However, we can also define higher-order scheduling functions that return other cursors by using the `nav` function, which navigates the frame of reference:

```
def nav(move):
  return lambda p,c: p, move(p.forward(c))
def savec(op):
  return lambda p,c: op(p,c)[0], c
```

Let `move` : Cursor $\rightarrow$ Cursor be a function that transforms a cursor. `savec` allows us to navigate the reference frame arbitrarily within the argument `op`, but restore it afterwards. We can compose these combinators into a useful pattern where the cursor is navigated to take some action and then restored afterward. This pattern allows us to avoid ambiguity in our references by staging all operations relative to a stable point within the AST.

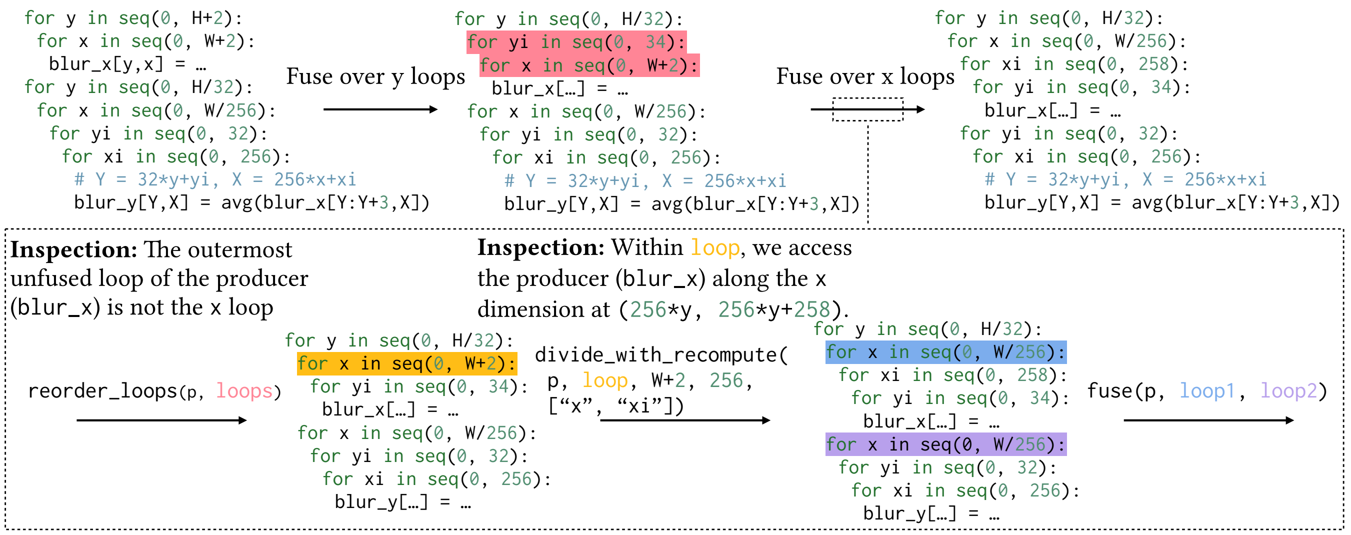
```
reframe = lambda move, op: savec(seq(nav(move), op))
```

Because `nav` forwards the cursor before performing any spatial navigation, these combinators have now recreated linear-time reference frames (Section 5.1) used by strategy languages and ELEVATE. Furthermore, as shown below, Exo 2 concisely recreates many Exo scheduling primitives that combined actions with relative references, in a single line.

```
reorder_before     = reframe(lambda c:c.expand(1, 0),
                             lift(reorder_stmts))
remove_parent_loop = reframe(lambda c:c.parent(),
                             lift(remove_loop))
fission_after      = reframe(lambda c:c.after(),
                             lift(fission))
```

6.3.2 Reproducing Halide.

Blur in Halide and Exo. As shown in Figure 11, the 3x3 blur algorithm and schedule from the Halide paper [56] uses built-in actions (`compute_at`, `tile`, `vectorize`, `parallel`) and nominal referencing. When `compute_at` is called without a corresponding `store_at`, Halide automatically stores the buffer at the same loop level as the computation. The equivalent Exo starting object code (Figure 10 upper left, after tiling to `y` and `x` has already been applied) is longer than the Halide version because Exo's lower-level IR explicitly expresses loops, allocations, and read/write/reduce statements, while Halide is restricted to pure functions of integer coordinates. For simplicity, we restrict input images to whole multiples of the tile size.



```
for y in seq(0, H+2):
 for x in seq(0, W+2):
  blur_x[y,x] = …
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):
    # Y = 32*y+yi, X = 256*x+xi
    blur_y[Y,X] = avg(blur_x[Y:Y+3,X])
```

```
for y in seq(0, H/32):
 for yi in seq(0, 34):
  for x in seq(0, W+2):
   blur_x[…] = …
 for x in seq(0, W/256):
  for yi in seq(0, 32):
   for xi in seq(0, 256):
    # Y = 32*y+yi, X = 256*x+xi
    blur_y[Y,X] = avg(blur_x[Y:Y+3,X])
```

```
for y in seq(0, H/32):
 for x in seq(0, W/256):
  for xi in seq(0, 258):
   for yi in seq(0, 34):
    blur_x[…] = …
  for yi in seq(0, 32):
   for xi in seq(0, 256):
    # Y = 32*y+yi, X = 256*x+xi
    blur_y[Y,X] = avg(blur_x[Y:Y+3,X])
```

Inspection: The outermost unfused loop of the producer (blur_x) is not the x loop

Inspection: Within loop, we access the producer (blur_x) along the x dimension at (256*y, 256*y+258).

```
reorder_loops(p, loops)
```

```
for y in seq(0, H/32):
 for x in seq(0, W+2):
  for yi in seq(0, 34):
   blur_x[…] = …
 for x in seq(0, W/256):
  for yi in seq(0, 32):
   for xi in seq(0, 256):
    blur_y[…] = …
```

```
divide_with_recompute(
p, loop, W+2, 256,
["x", "xi"])
```

```
for y in seq(0, H/32):
 for x in seq(0, W/256):
  for xi in seq(0, 258):
   for yi in seq(0, 34):
    blur_x[…] = …
 for x in seq(0, W/256):
  for yi in seq(0, 32):
   for xi in seq(0, 256):
    blur_y[…] = …
```

```
fuse(p, loop1, loop2)
```

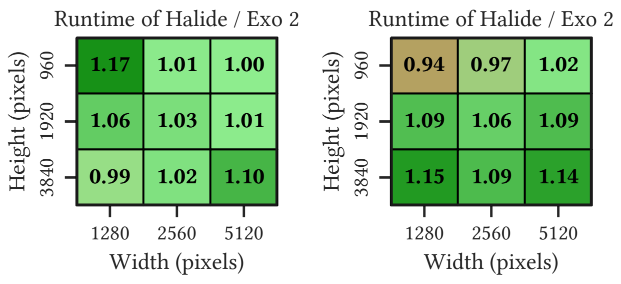
Figure 10. Exo 2 implementation of Halide's `blur_x.compute_at(blur_y, x)` operation. The starting code is a tiled version of the blur algorithm. We first fuse the `blur_x` and `blur_y` computations at the `y` loop level, and then fuse at the `x` loop level. Focusing on the second fusion, the bounds inference inspection described in Section 4 determines the producer (`blur_x`) accesses in the `x` loop. Then, a sequence of primitive actions is applied to the object code to yield code fused at the `x` loop level.

```
// ALGORITHM
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;
// SCHEDULE
blur_y.tile(x, y, xi, yi, 256, 32)
      .vectorize(xi, 16)
      .parallel(y);
blur_x.compute_at(blur_y, x)
      .vectorize(x, 16);
```

Figure 11. Halide's blur algorithm and schedule.

```
p = H_tile(p,"blur_y","y","x","yi","xi",32,256)
p = H_compute_store_at(p,"blur_x","blur_y","x")
p = H_parallel(p, "y")
p = H_vectorize(p, "blur_x", "xi", 16)
p = H_vectorize(p, "blur_y", "xi", 16)
p = H_store_in(p, "blur_x", DRAM_STACK)
```

Figure 12. Exo 2's blur schedule.



(a) 2D 3×3 box blur. **(b)** Unsharp masking.

	Rewrites	Exo 2 Schd.	Halide Schd.
blur	116	6	5
unsharp	975	26	13

(c) Lines of code for the Exo 2 schedule, Halide schedule, and the number of rewrites performed during the Exo 2 scheduling process.

Figure 13. Exo 2 schedules show competitive performance to corresponding expert-written Halide schedules on a variety of image sizes.

Exo 2 Blur Schedule. Halide's `compute_at` and `store_at` scheduling operations use automated bounds inference to determine loop bounds and buffer sizes. We leverage the previously discussed bounds inference inspection (Section 4) to reproduce these Halide operations. Figure 10 summarizes our `compute_at` implementation, and similar ideas apply to `store_at`. To bridge the gap between Halide's and Exo 2's referencing schemes, we defined `H_`-prefixed functions that expect nominal references and internally convert to Exo 2'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. As shown in Figure 12, using the Halide-like scheduling operations with nominal referencing, Exo 2 can reproduce Halide-style schedules. `H_compute_store_at` is called to implement Halide's aforementioned compile-time decision of calling both `compute_at` and `store_at`.