Exocompilation for Productive Programming of Hardware Accelerators

Yuka Ikarashi[*] MIT CSAIL, USA
Gilbert Louis Bernstein[*] UC Berkeley, USA
Alex Reinking UC Berkeley, USA
Hasan Genc UC Berkeley, USA
Jonathan Ragan-Kelley MIT CSAIL, USA

Abstract

High-performance kernel libraries are critical to exploiting accelerators and specialized instructions in many applications. Because compilers are difficult to extend to support diverse and rapidly-evolving hardware targets, and automatic optimization is often insufficient to guarantee state-of-the-art performance, these libraries are commonly still coded and optimized by hand, at great expense, in low-level C and assembly. To better support development of high-performance libraries for specialized hardware, we propose a new programming language, Exo, based on the principle of *exocompilation*: externalizing target-specific code generation support and optimization policies to user-level code. Exo allows custom hardware instructions, specialized memories, and accelerator configuration state to be defined in user libraries. It builds on the idea of user scheduling to externalize hardware mapping and optimization decisions. Schedules are defined as composable rewrites within the language, and we develop a set of effect analyses which guarantee program equivalence and memory safety through these transformations. We show that Exo enables rapid development of state-of-the-art matrix-matrix multiply and convolutional neural network kernels, for both an embedded neural accelerator and x86 with AVX-512 extensions, in a few dozen lines of code each.

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

Keywords: program optimization, hardware accelerators, user-schedulable languages, instruction abstraction, scheduling, user-extensible backend & scheduling

[*]Both authors contributed equally to this paper

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523446

ACM Reference Format:
Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523446

1 Introduction

Modern computers are increasingly comprised of accelerators. From neural and cryptography engines, to image signal processors, to GPUs, a state-of-the-art system-on-chip (SoC) today includes dozens of different specialized accelerators. Even within their main CPUs, performance improvement increasingly comes via new instructions performed by specialized functional units. This specialized hardware is orders of magnitude more efficient than software running on general-purpose hardware, but most applications are only able to realize this performance and efficiency insofar as key low-level libraries of high-performance kernels (e.g., BLAS, cuDNN, MKL, etc.) are optimized to exploit the hardware.

While the role played by high-performance kernel libraries is increasingly critical, there is little programming language support for the performance engineers who write them. Progress continues to be made after decades of effort on fully-automatic compiler optimization, but state-of-the-art kernels—from linear algebra, to deep learning, to signal processing and cryptography—are still predominantly written by hand, directly in low-level C and hardware-specific intrinsics or assembly, or with lightweight metaprogramming (e.g., macros or C++ templates) of such low-level code. As a result, developing and tuning these libraries is enormously labor intensive, limiting the range of accelerated routines and creating barriers to deploying new or improved accelerators.

Developing accelerated high-performance libraries is a unique software engineering task, with several unusual characteristics. First, in contrast to conventional programs on general-purpose processors, the hardware-software interfaces to accelerators are both complex—including specialized memories, exposed configuration state, and complex operations—and highly diverse, with *different* complexities unique to each accelerator. Second, the rates of change at different levels in the stack—from applications to hardware ISA—are inverted: accelerator architectures change *more* rapidly than the essential functions which run on them (e.g., mobile phone SoCs are rebuilt every year, with major revisions to nearly every accelerator block, while the BLAS standard changes much more slowly), and the implementation of these functions to most efficiently use the hardware is iterated more quickly, still. This is especially acute during accelerator development, where target application workloads are often fixed, while both the hardware architecture and kernels mapping to it are iteratively co-designed to maximize performance and efficiency.

In this paper, we propose *exocompilation* as a new approach to programming language and compiler support for developing hardware-accelerated high-performance libraries. The principle of exocompilation is to externalize as much accelerator-specific code-generation logic and optimization policy from the compiler as possible, instead exposing them at the user level to high-performance library writers. Specifically, we externalize accelerator specification to user-level libraries, and we build on the idea of user scheduling, popularized by languages like Halide and TVM [8, 29], to externalize hardware mapping and optimization decisions.

We develop a new language and compiler called Exo based on this principle of exocompilation. Exo allows custom hardware instructions to be user-defined and abstracted as procedures. It also allows specialized memories and accelerator configuration state to be defined in user code, without modifying the core compiler. User scheduling enables a rich space of optimization and hardware mapping choices to be directly explored by the performance engineer, rather than requiring an automated optimizer to always make perfect decisions.

In contrast to optimization by manually rewriting low-level code, scheduling transformations are concise and safe. They elide many details like array and loop re-indexing (which can be automatically inferred), while guaranteeing both functional equivalence and memory safety. Different schedules best optimize the same library function for different hardware, or even for different parameter values, and specialized versions for each case can be generated from a single source algorithm. Arbitrary program fragments can be replaced during scheduling with equivalent user-defined accelerator instructions, or specialized subroutines, using a *unification* procedure that automates the transformation of essential arguments and array indexing. Finally, in contrast to languages like Halide and TVM, Exo implements user scheduling via composable rewrite rules. This allows the scheduling language itself to be easily extended, since each operator defines an independent rewrite, rather than interacting with all others in a monolithic lowering process.

We explore what is required of safety analyses for such a language, and define a set of *effect analyses* which support guarantees of program equivalence and memory safety after scheduling (§5). We make the simplifying assumption of affine loops and array indexing, which has been shown to be sufficient for many kernels of interest in high-performance libraries [12]. Nonetheless, accelerator configuration introduces global mutable state which breaks the classic "static control program" assumption, and requires introducing approximation into the analyses. Our analyses are then defined in a ternary logic, which distinguishes effects which *definitely* occur (necessary for, e.g., eliminating redundant setting of configuration state) from those which *maybe* occur (relevant for reasoning about the statement reorderings which emerge from many loop transformations).

Finally, we perform a series of case studies applying Exo to optimizing high-performance kernels for specialized hardware. We develop user-level backends for the Berkeley Gemmini neural network accelerator [16] (a software-controlled systolic array similar to many TPU-like architectures) and x86-64 with AVX-512. For each target, we focus on optimizing matrix multiply and convolutional neural network layers — among the most highly-optimized kernels in common libraries. Using Exo, we were able to easily develop implementations competitive with state-of-the-art libraries in a few days and a few dozen lines of code.

2 Example

Today's large machine learning models (and scientific computing) rely on highly tuned matrix-matrix multiplication kernels (aka. GEMM). In order to introduce Exo, we will show how to write and optimize such GEMM kernels, targeting one to an accelerator ISA designed to resemble machine learning accelerators. These accelerators all focus on the efficient execution of small (e.g. 16×16), dense matrix-matrix multiplication instructions.

Optimizing these kernels is primarily an exercise in orchestrating data movement, and only secondarily a matter of selecting compute instructions, such as the actual matrix multiplication primitive. Therefore, we need to explicitly schedule loads and stores from custom, explicitly managed accelerator memories. Lastly, much of the behavior of hardware accelerators is controlled by infrequently changing *configuration state*. Instructions to configure such state usually flush the accelerator pipeline.

To model a particular hardware accelerator, users must define custom memories, instructions and configuration state. This work is done once per accelerator, written as a *hardware library*. Throughout the example, we will indicate whether each piece of code lives in the application (GEMM) or can be abstracted out into a reusable description of the hardware.

2.1 Exo Procedures, Compilation, and Scheduling

Consider matrix-matrix multiplication, written in Exo:

```
@proc
def gemm(A: R[128, 128] @ DRAM, B: ..., C: ...):
  for i in seq(0, 128):
    for j in seq(0, 128):               in app.py
      for k in seq(0, 128):
        C[i, j] += A[i, k] * B[k, j]
```

Exo is embedded in Python, and the function decorator `@proc` indicates the beginning of an Exo function. Function arguments are given by the syntax

⟨*name*⟩:⟨*type*⟩[⟨*size*⟩] @ ⟨*memory*⟩

`R` is an abstract type for all numeric data types, which can be specialized to specific precision types such as `f32` and `i8` via scheduling operations. For simplicity, the ⟨*size*⟩ in this example is constant, but usually refers dependently to other function arguments. The `@` symbol is a *memory specification*; `@DRAM` means that the buffer is expected to be in DRAM. Finally, `for i in seq(0, 128)` is a *sequential* for loop that ranges from `0` to `127` (inclusive).

Exo compiles to C source code in the expected way:

```
void gemm(float *A, float *B, float *C) {
  for (int i=0; i<128; i++) {
    for (int j=0; j<128; i++) {
      for (int k=0; i<128; i++) {
        C[128*i + j] += A[128*i + k] * B[128*k + j];
} } } }
```

In order to target our accelerator, we need to expose a 16×16 matrix-multiplication as the inner loop nest. We do this by using *scheduling operations* to rewrite the procedure. In particular, we `split(i,16,io,ii)` (sim. for j, k) and then `reorder()` the loops (see §3.3) to produce the following tiled matrix multiplication:

in app.py

```
def gemm(A: R[128, 128] @ DRAM, B: ..., C: ...):
  for io in seq(0, 8):
    for jo in seq(0, 8):
      for ko in seq(0, 8):
        for ii in seq(0, 16):
          for ji in seq(0, 16):
            for ki in seq(0, 16):
              C[16*io+ii, 16*jo+ji] += A[..] * B[..]
```

2.2 Memories

Many accelerators—including ours in this example—have explicitly-managed memories. Performance critically depends on how data movement to and from these memories is interleaved with other computation. Therefore Exo puts scheduling of data movement in the hands of the programmer. The first step in doing this, is to define *custom memories* on a per-accelerator basis. For example,

in hw_lib.py

```
class ACCUMULATOR(Memory):
    def alloc(...):
        return f"{prim_type} {name} = hw_malloc({sz});"
    def free(...):
        return f"hw_free({name});"
    def read(...):  # also write, reduce
        raise MemGenError('memory is not addressable')
```

If a buffer is annotated with ACCUMULATOR instead of DRAM, then these `alloc` and `free` *macros* will determine the C code that is generated when that buffer is allocated or freed. (see §3) Furthermore, note that the ACCUMULATOR memory explicitly disables code generation for reading, writing and accumulating into individual locations, preventing direct access from C. Instead, we will only allow custom instructions (see below) to access this custom memory.

Supposing we have written custom ACCUMULATOR and SCRATCHPAD memories, we use `stage_mem` scheduling operations to stage C, A, and B into these memories:

in app.py

```
def gemm(...):
    res: R[...] @ ACCUMULATOR
    a  : R[...] @ SCRATCHPAD
    b  : R[...] @ SCRATCHPAD
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Load C to res
            for ko in seq(0, 8):
                # Load A to a
                for ii in seq(0, 16):
                    for ki in seq(0, 16):
                        a[...] = A[...]
                ... # Load B to b
                # Matmul of a and b
                for ii in seq(0, 16):
                    for ji in seq(0, 16):
                        for ki in seq(0, 16):
                            res[..]+=a[..]*b[..]
            ... # Store res to C
```

2.3 Instructions

We can clearly see opportunities in the above code to map loops to semantically equivalent accelerator instructions. However, to do this safely and soundly, the compiler needs definitions of our accelerator instructions in terms of Exo's semantics. The key idea of exocompilation is to provide users with a framework for defining these instructions in libraries, without modifying the compiler itself. Below, we show an example of such a definition for the scratchpad load.

in hw_lib.py

```
@instr("config_ld({src}.strides[0]);\n"
       "mvin({src}.data, {dst}.data, {m}, {n});")
def ld_data(n: size, m: size,
            src: [R][n, m] @ DRAM,
            dst: [R][n, 16] @ SCRATCHPAD):
    assert m <= 16
    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```

Notice that this function has been annotated with `@instr` rather than `@proc`. This indicates that the declaration *asserts* equivalence between the Exo code in the body and the C code template (i.e. macro) in the annotation. The resulting `ld_data` function may be scheduled and called like any other function, but Exo's C code generator will instead emit the C code "`config_ld({src}.strides...)`", with argument placeholders `{src}` and `{dst}` substituted appropriately.

Exo provides a `replace()` scheduling directive (§3.4) for matching code in one procedure with the body of another procedure (including an `@instr` like `ld_data`), then *replacing* the matched code with an appropriate procedure call.

2.4 Configuration State

We could issue this directive now to schedule the accelerator instructions, however, the C code has fused the expensive `config_ld` instruction to the `mvin` instruction we are really interested in scheduling. Since the stride does not actually change during the kernel, this will cause the accelerator pipeline to repeatedly flush and stall. We must somehow schedule the configuration instruction independently of the actual load.

Therefore, we need a way to define hardware state. The following code models the stride configuration state in Exo.

```
@config
class ConfigLoad:                                in hw_lib.py
    src_stride : stride

@instr("config_ld({s});")
def config_ld_def(s : stride):
    ConfigLoad.src_stride = s
```

Here, `ConfigLoad` defines a global struct of configuration variables, here containing a single `src_stride` field that models the state of the stride hardware parameter. We also write an instruction definition, `config_ld_def`, that updates the `src_stride` field. Now we can write a new instruction for the 16×16 load without the `config_ld` setup:

```
@instr("mvin({src}.data, {dst}.data, {m}, {n});")
def real_ld_data(...):                           in hw_lib.py
    assert ConfigLoad.src_stride ==
           stride(src, 0)
    # same as ld_data
```

Using scheduling instructions, we will rewrite the body of `ld_data` into a call to `config_ld_def()`, followed by a call to `real_ld_data()`. First, we use the `configwrite_at()` scheduling operation to rewrite `ld_data` into the following:

```
def ld_data(...):                                in hw_lib.py
    assert m <= 16
    ConfigLoad.src_stride = stride(src, 0)
    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```

Unlike previous scheduling operations, `configwrite_at()` only partially preserves procedure equivalence—the new `ld_data()` is only equivalent *up to* the configuration state `ConfigLoad.src_stride`. In general, Exo needs to reason about this kind of program equivalence modulo configuration state (see definition 4.1 and §6.2).

Since the statement `ConfigLoad.src_stride = ...` is equivalent to the body of `config_ld_def`, and the statement `for i in seq(...):...` is equivalent to the body of `real_ld_data`, we can now `replace()` the body of `ld_data` with the two calls, as desired:

```
def ld_data(...):                                in hw_lib.py
    assert m <= 16
    config_ld_def(stride(src, 0))
    real_ld_data(n, m, src, dst)
```

By following this same procedure, we can create instruction abstractions for our 16x16 matmul and store instructions. At last, we can replace the code in `gemm` with calls to `ld_data` and inline its definition.

```
def gemm(...):
    res: R[...] ...                              in app.py
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Loading C to res
            for ko in seq(0, 8):
                config_ld_def(stride(A, 0))
                real_ld_data(16, 16, A[...], a[...])
            ... # etc. etc.
```

We will hoist the call to `config_ld_def` using scheduling operations `reorder_stmts()`, `fission_after()`, as well as `remove_loop()`. Doing so will require Exo's program analysis to both reason about when different statements *commute* (can be reordered) as well as when they are *idempotent* (allowing the loop to be removed). To further complicate matters, the presence of global, mutable *configuration* state means that fully precise analyses are undecidable, and thus impossible in Exo. By using a ternary logic (§5), Exo can distinguish between memory locations that are *definitely* written to (a necessary condition for idempotency) and locations that are *maybe* written to (the relevant condition for commutativity).

```
def gemm(...):
    config_ld(stride(A, 0))                      in app.py
    res: R[...] ...
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Loading C to res
            for ko in seq(0, 8):
                real_ld_data(16, 16, A[...], a[...])
            ... # etc. etc.
```

All of the above code transformations are achievable using the scheduling primitives discussed in Section 3. Full definitions of the memory, configuration, and load instructions for the Gemmini accelerator can be found in supplemental appendix G.[1]

3 The Exo Language and System

The Exo system consists of an imperative programming language (§3.1), means of defining hardware targets via libraries (§3.2), and a rewrite-based scheduling system (§3.3, 3.4). Figure 1 shows the Exo system from the standpoint of a particular program being compiled. In this section, we explain each part of this process.

3.1 The Exo Language

Exo is a familiar imperative language in the mold of the static control program model [12]. It supports for-loops, if-conditions, arrays and procedures, but not while-loops or recursion. A BNF grammar for its formal core is defined later (Fig. 3). In addition to that grammar, the full language supports `stride` values and expressions, as well as memory annotations, both of which were shown in the example (§2).

[1]Appendices are available as Supplemental Material on the ACM Digital Library.

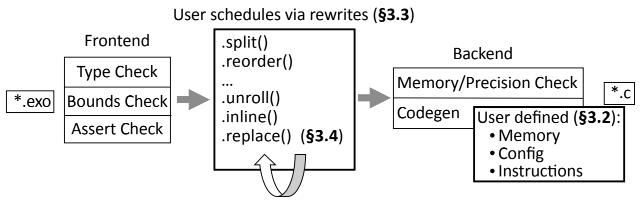


Figure 1. Exo system overview

Six relatively standard (but not universally adopted) features of Exo are worth discussing further: (1) control/data separation, (2) mutable global control state, (3) dependently typed arrays [38], (4) array windowing/slicing, (5) explicit += reduction primitives, and (6) static assertions.

(1) Exo is built around a distinction between control and data values. Control values (types `int`, `bool`, `size`, etc.) are constrained so that they may be analyzed more precisely. Arithmetic on integer control values must be quasi-affine, meaning that values can only be multiplied, divided, or modulo-ed by an integer literal. Expressions inside loop bounds and branches must be control values. Meanwhile, data values (types `R`, `f32`, `i8`, etc.) are floating-point or fixed-point numbers stored in scalars or arrays. There are no restrictions on allowed computations between data values. (2) Configuration state (§2) is introduced via structs of variables using `@config` and modeled formally as global variables (§4). Unlike the other sources of control values, configuration state is mutable. Consistent with the idea of static control programs, Exo currently prohibits any dependence of control values on data-values, regardless of whether those control variables are local or global.

(3) Dependently typed arrays allow sizes to be specified by control value expressions of strictly positive value. Exo then performs static bounds checks, guaranteeing memory safety without incurring any of the costs of dynamic bounds checks. This is made possible by the control/data separation idea. (4) Arrays in Exo are further extended with support for windowing (aka. slicing) via the `x[lo:hi]` syntax. Creating a window does not copy data; instead, reading from and writing to locations in a window accesses the underlying buffer (e.g. if `y = x[3:8]` then `y[2] == x[3+2]`). In particular, note that windows may be lower-dimensional than their underlying buffers by slicing some indices, while point-accessing others. For instance, `x[0:n,j]` creates a 1-dimensional window on column `j` of matrix `x`. (5) In addition to primitive reading and writing, reduction via the += syntax is supported as a special commutative and associative operation from the point of view of program analysis.

(6) Finally, we allow static assertions about control values to be made at the beginning of procedures. These assertions act as *pre-conditions* and not as dynamic tests. Program analysis within a procedure may assume its asserted pre-conditions, whereas a calling procedure is only valid if it ensures that the callee's pre-conditions are true.

3.1.1 Backend Checks: Precision and Memory.

Type-checking, bounds-checking, and assertion checking are all front-end checks on Exo code. By contrast, consistency of data-variable precision types as well as consistency of memory annotations are performed as back-end checks immediately prior to code generation. Exo requires all data-expressions to have consistent precision, (e.g. multiplying an `f32` and `i8` is forbidden) but inserts type-casts as necessary just before writing or reducing data values.

3.1.2 Code Generation.

Exo is designed to generate human-readable C-code that is more or less a syntactic translation of the corresponding Exo code. This enables the programmer to more easily integrate Exo with existing tools and workflows. There are a few non-obvious details with this translation that merit discussion. First, all data values (including scalars, buffers, and windows) are passed by pointer rather than by value. This is necessary even in the case of scalars to allow "returning" modified scalar values to a caller. Second, windows are compiled to structs containing both the data pointer and stride values, since the static size of a window is insufficient to compute a linear address into the underlying buffer. Lastly, we translate static assertions into compiler-specific optimization hints to help improve downstream analyses and optimizations.

3.2 Hardware Targets as Libraries

To add support for a new hardware accelerator to Exo, programmers write a library, rather than a compiler backend. These libraries use three key features of the Exo language: (1) memories, (2) instructions, and (3) configuration state. Using these features, an Exo programmer can hand-write code to target a given accelerator, or use scheduling to rewrite a simple program into one targeting a given accelerator (§3.3).

Defining hardware in libraries has two advantages over defining hardware in compiler backends (as Halide, TVM, LLVM and most compilers do). First, hardware vendors do not need to maintain compiler forks in order to protect proprietary details of their hardware. Second, the cost of adding support for new hardware is significantly reduced. Our experience adding support for new hardware to both Exo and Halide suggests that the library approach requires at least an order of magnitude less development time.

3.2.1 Memories.

By default, all Exo buffers are assumed to reside in system DRAM and are managed using standard `malloc` and `free`. However, hardware accelerators often require modeling buffers that are resident in special accelerator memories, are pinned to special address ranges in the global address space, or otherwise exhibit strange behavior. To support these scenarios, Exo allows users to tag buffer and window types with a memory annotation. For example, `x : f32[n] @ MEM` says that the vector `x` lives in a custom memory `MEM`. These custom memories are defined by subclassing a `Memory` base class (§2) and overloading methods.

Exo allows custom memories to change code generation for buffer `alloc`, `free`, and `windowing` via string interpolation. The author of a custom memory chooses whether to allow standard reading and writing the buffer (e.g., if the memory simply changes the memory management policy) or disable all usual accessing of the memory. The latter option is ideal for modeling hardware scratchpads, which should only be accessed using custom instructions. Such improper accesses are prevented by "backend checks." In general, memory annotations are ignored during scheduling.

3.2.2 Instructions. Instructions in Exo are procedures that are annotated with a macro/string-template. For example, given a vector load procedure with the signature `load(n : size, dst : f32[n], src : f32[n])`, we can make it into an instruction by annotating it with `@instr("hw_ld({src},{dst},{n})")` instead of `@proc`. When code generating calls to instructions, this annotation string is used instead of a sub-procedure call. Arguments are interpolated into the template as strings. This works as well for scheduling fine-grained intrinsics as it does for coarse-grained calls to existing microkernels or library calls.

As a result, the annotated Exo procedure has no effect on code generation, but instead serves as a semantic specification of the instruction for the purposes of checking correctness and program equivalence (for scheduling). This approach to an instruction mechanism has the following benefits and tradeoffs. First, programmers need not learn any additional specification language beyond Exo. Second, Exo entrusts programmers with the responsibility of verifying the link between the Exo procedure and annotation. Third and finally, programmers can use instructions in clever ways, including as an escape hatch. For example, a prefetch instruction can be modeled using a no-op procedure and thereby be inserted anywhere.

3.2.3 Configuration State. As we saw in §2, Exo models hardware configuration state via global structs of control variables annotated by `@config`. When defining configurations, programmers have the choice of realizing them as DRAM-resident data or disabling direct access to the configuration state (similar to disabling direct reading and writing of a memory). In the latter case, no global struct is generated.

3.3 Scheduling via Rewrites

Rather than directly writing code that uses a hardware library, Exo users transform a simple program into an equivalent, but more complex and high-performance version, targeted to the specific hardware accelerator. This transformation is accomplished via successive rewriting of the application—a process called scheduling.

Because Exo is an embedded DSL, schedules are written as meta-programs in the host language (Python). Each primitive scheduling operator (Figure 2) takes a procedure p plus some other arguments as input, and returns an equivalent,

Command	Transform

`p.reorder(i,j)`	`for i:`
	`  for j:` ⇝ `for j:`
	`  for i:`

`p.split(i,c,io,ii)`	`for i<I:` ⇝ `for io<I/c:`
	`  for ii<c:`

`p.unroll(i)`	`for i:` ⇝ `for 0:`
	`...`

`p.inline(foo)`	inline a callsite of foo in p
`p.set_memory(a,MEM')`	`a @ MEM` ⇝ `a @ MEM'`
`p.set_precision(a,typ')`	`a : typ` ⇝ `a : typ'`
`p.call_eqv(foo,foo')`	call foo' at a callsite of foo

`p.bind_expr(a,a')`	`s` ⇝ `a' : R`
	`a' = a`
	`s[a ↦ a']`

`p.stage_mem(a,a',s)`	`s` ⇝ `a' : R[]`
	`for i:`
	`  a' = a`
	`s[a |↦ a']`
	`for i:`
	`  a = a'`

`p.bind_config(config,a)`	`s` ⇝ `config = a`
	`s[a ↦ config]`

`p.lift_alloc(a:R)`	`for i:`
	`  a : R`
	`  s` ⇝ `a : R`
	`for i:`
	`  s`

`p.fission_after(s1)`	`for i:`
	`  s1`
	`  s2` ⇝ `for i:`
	`  s1`
	`for i:`
	`  s2`

`p.reorder_stmts(s1,s2)`	`s1`
	`s2` ⇝ `s2`
	`s1`

`p.configwrite_at(s,config,e)`	`s` ⇝ `s`
	`config = e`

`p.replace(s,foo)`	`s` ⇝ `foo(«inferred»)`

`p.add_guard(s,e)`	`s` ⇝ `if e: s`
	`else: s`

`p.fuse_loop(i)`	`for i:`
	`  s1`
	`for i:`
	`  s2` ⇝ `for i:`
	`  s1`
	`  s2`

`p.lift_if(if c: s)`	`for i:`
	`  if c: s` ⇝ `if c:`
	`  for i: s`

`p.partition_loop(i,c)`	`for i in lo,hi:`
	⇝
	`for i in lo,c:`
	`for i in c,hi:`

`p.remove_loop(i)`	`for i:`
	`  s` ⇝ `s`

Figure 2. Some primitive Exo scheduling operators. Each operator rewrites $s_0 \rightsquigarrow s_1$ within a procedure p. This sort of rewrite based scheduling makes it easier to expand the list of primitive operators, since the correctness of each operator is independent of the correctness of each other operator.

rewritten procedure as output. Most of these operators require *pointing* at a location within the procedure. In our prototype, this is accomplished via simple syntactic pattern matching strings. For instance, `src : _` points at the first allocation of a buffer named `src`, and `for i in _: _ #2` points at the third loop in `p` with an iteration variable named `i`. This API is currently being re-designed, but was sufficient to demonstrate the benefits of rewrite-based scheduling.

Exo advances the idea of user-scheduling in two important ways. First, like Lift and Elevate [19, 31] but unlike Halide and TVM, scheduling operators are rewrites of programs, rather than arguments to a monolithic lowering process. As a result, the implementation and correctness of a scheduling primitive is independent of each other primitive. This makes the Exo implementation much simpler and easier to maintain. Importantly, Exo rewrites imperative rather than functional programs (Lift and Elevate). This makes checking the correctness of primitive rewrites more complex (§5,F).

Second, Exo supports scheduling of programs decomposed into procedures. This happens via the `inline()`, `call_eqv()`, and `replace()` primitives. `inline()` simply inlines a procedure's body at some call site, and `replace()` can be thought of as the inverse of `inline()` (see next section). `call_eqv()` on the other hand replaces a call to some sub-procedure `f` with a call to an equivalent sub-procedure `f'`. This equivalence is tracked by provenance, since the Exo system records the sequence of transformations by which `f` was transformed into `f'`. This concept of an *equivalent* sub-procedure is complicated by those scheduling primitives which pollute configuration state (e.g. `bind_config()`). To handle these, Exo tracks a lattice of different equivalence relations, modulo different parts of the configuration state (§6).

This provenance tracking system also enables an important optimization: when constructing SMT queries we may use the *simplest* equivalent (including configuration) definition of a procedure when constructing SMT queries. This is necessary to keep the cost of calling the solver low as scheduling complicates a procedure.

3.4 Code Replacement & Instruction Selection

The `replace()` scheduling primitive takes a designated statement block `s` and *replaces* it with a call to a designated sub-procedure `foo`. In particular, when `foo` is an `@instr`, this rewriting performs instruction selection. In other cases, it allows Exo programmers to manage code size trade-offs, as well as more neatly abstract and organize their code.

Our implementation of `replace()` is based on a form of unification modulo linear equalities. First, we attempt to unify (i.e. pattern match) the body of the sub-procedure `foo` with the designated statement block `s`. When doing this, the arguments of `foo` are designated as unknowns, the free variables of `s` as known symbols and any symbols introduced/bound in the body of `foo` or within `s` are unified. The ASTs are required to match exactly with respect to statements, and with respect to all expressions which are not simply integer typed control. Equivalences between integer typed control expressions are recorded as a system of linear equations to be solved in a second step.

If Exo did not support windowing, then we could determine expressions for the unknown argument variables by symbolically solving the resulting linear system of equations. However, the possibility of windowing expressions as arguments forces us to make categorical choices between different possible windowing expressions, resulting in disjunctions as well as conjunctions of linear equalities. For example, if replace is asked to infer a 1-dimensional window onto a 2-dimensional buffer `x`, it could infer an expression of the form `x[i,jlo:jhi]` or of the form `x[ilo:ihi,j]`. To handle this complication, we observe that all inferred integer expressions must be affine combinations of the known, free variables. Therefore, we can transform our symbolic linear system problem into a linear system in the unknown coefficients of these affine expressions. Once encoded in this way, we can discharge the problem to an SMT solver.

4 Formal Core Language

In order to define our program analysis, we provide a formal definition of the core of Exo, including a denotational semantics. The core idea is that statements denote store-transforming functions of type $\Sigma \to \Sigma$. Using these semantics we can define equivalence of Exo programs as functional equivalence of their denotations. A scheduling transformation can then be said to be safe when it transforms between equivalent Exo programs.

4.1 Mathematical Model of Exo Programs

The main concept in our mathematical model of Exo programs is the *store*, which represents the program state at any given point during its execution. The simplest model of a store $\sigma \in \Sigma$ would be a partial function from variable names to values. However, we must complicate this naive model in a few ways. Rather than present the full definitions (available in a supplemental appendix), we will focus on a high level gloss of the ideas here.

Control values are modeled as Boolean or integer values (in $\mathbb{B}$ and $\mathbb{Z}$) while data values are modeled as real numbers (in $\mathbb{R}$). Names of variables are drawn from a set of identifiers Name. Additionally, we rely on exceptional values to capture errors ϵ and unknown or uninitialized data $\bot$. For simplicity, we assume that all built in functions on data (basic arithmetic and the math library) are total, so that e.g. $0/0$ is not an error.

The first complication is that we need to model buffers and windows. Buffers can be thought of as maps from coordinate tuples to data $\mathbb{Z}^m \to (\mathbb{R} \uplus \{\bot, \epsilon\})$, where $\bot$ designates uninitialized but allocated memory, whereas ϵ designates out-of-bounds memory. These buffers are placed in the store Σ at special addresses $\ell \in$ Name that are disjoint from names used in the program. Then windows can be modeled as a pair of a buffer address ℓ and affine-indexing function $\phi \in \mathbb{Z}^n \to \mathbb{Z}^m$. For instance, reading a window at coordinates i would translate to the lookup $\sigma(\ell)(\phi(i))$.

Having modeled buffers and windows, we can define stores $\sigma \in \Sigma$ as partial functions from Name to buffers, windows, or control values. In order to further capture the concept of program crashes (which should never happen for well-typed, well-bounded and assertion-satisfying programs) we expand the domain of stores to include the special value ϵ. We may assume that all functions are strict with respect to ϵ, meaning that once a program crashes it remains crashed.

4.2 Syntax, Semantics, and Well-Typed Programs

The syntax for the formal core of Exo is straightforward (Figure 3). The denotation of a statement or procedure s is written $\mathrm{S}\,[\![s]\!]$ and is a function $\Sigma \to \Sigma$. The full definition of denotations for expressions, statements and procedures are deferred to a supplemental appendix (§A). Note again that this core language makes no reference to user-defined instructions or memories. This is because the core program analysis is blind to those features—which only affect code generation. This separation is what allows us to make the program analysis extensible to new hardware backends.

Our focus in this paper is not on basic type-checking (which is standard) or even bounds-checking and assertion-checking (which are straightforward based on prior work and repurposing our later analysis machinery). However, it is worth re-iterating what guarantees all of these front-end checks provide for Exo programs. First, all integer-valued control expressions are constrained to be quasi-affine. Second, all windowing and accessing of buffers and windows is statically guaranteed to be in-bounds. Lastly, any procedure call is guaranteed to satisfy the asserted pre-conditions of the called procedure. Mutation of non-global control values is also prohibited. The quasi-affine restriction in particular is what allows us to translate arbitrary control expressions into SAT queries modulo the Linear Integer Arithmetic (LIA) theory, and thus discharge problems to an SMT solver.

4.3 Program Equivalence

Definition 4.1 (program equivalence)**.** Let s_1, s_2 both be Stmt or Proc. These two programs are equivalent, written $s_1 \cong s_2$ when the store-transforming functions they denote are equivalent $\mathrm{S}\,[\![s_1]\!] = \mathrm{S}\,[\![s_2]\!]$ on valid input stores—i.e. stores which are not in an error state and satisfy any precondition assertions of s_1 and s_2, which are equivalent.

As we discussed earlier (§2), we often want to reason about programs that are equivalent "up-to/excluding a set of globals $\mathcal{L}$" because many transformations end up polluting configuration state. We define a lattice of weaker equivalence relations:

Definition 4.2 (program equivalence modulo globals)**.** Let s_1, s_2 both be Stmt or Proc, and let $\mathcal{L} \subseteq \mathrm{Name}_{global}$ be a set of globals to ignore. The two programs are equivalent "modulo $\mathcal{L}$", written $s_1 \cong_{\mathcal{L}} s_2$ when $\forall \sigma, x \notin \mathcal{L}.\ \mathrm{S}\,[\![s_1]\!]\ \sigma\ x = \mathrm{S}\,[\![s_2]\!]\ \sigma\ x$, with the same caveats about valid input stores.

τ_a : ArgType	::=	`bool` | `int` | `R`[e^*]	
τ_s : SigType	::=	$(x : \tau_a) \to \tau_s$ | `unit`	
τ : Type	::=	τ_a | `R`	
note: we use $\cdot^*$ to mean 0 or more			
e : Expr	::=	x	variables
	|	$op(e^*)$	built-in operations
	|	$e[e^*]$	array read
	|	`win`(e, w^*)	window expression
w : WinCoord	::=	e	point-access
	|	$e\,..\,e$	interval-access
$op \in$		$\left\{ \begin{array}{l} +, -, *, /, \texttt{mod}, \texttt{and}, \texttt{or}, \texttt{not}, \\ ==, <, <=, >, >= \end{array} \right\} \cup Literals$	
s : Stmt	::=	$s\,;s$	sequencing
	|	`if` e `then` s	guards
	|	`for` x `in` $e\,..\,e$ `do` s	sequential loops
	|	`alloc` $x(e^*)$	array allocation
	|	$e[e^*]$`=` e	array write
	|	$e[e^*]$`+=` e	array reduce
	|	$x = e$	global write
	|	$p(e^*)$	sub-procedure call
$pdef$: Proc	::=	`proc` $p : \tau_s$ `assert` e `do` s	
L : Lib	::=	`globals` $(x : \tau)^*$ $pdef^*$	

Figure 3. Abstract Syntax for Exo core language

5 Effect Analysis & Transformation of Programs

Our analysis of Exo programs is based on an *effect* analysis. An effect a extracted from a statement s characterizes which functions $f : \Sigma \to \Sigma$ the statement s could possibly denote $\mathrm{S}\,[\![s]\!]$. This effect analysis allows us to determine when code transformations like $s_1\,;s_2 \rightsquigarrow s_2\,;s_1$ and $s_1\,;s_2 \rightsquigarrow s_2$ are valid.

This analysis will require us to define (1) effect-expressions and environments, (2) a global symbolic data-flow analysis, (3) location sets as a symbolic abstraction of store locations, and finally (4) effects as an abstraction of programs. We can then state safety conditions for various program rewrites using these building blocks.

5.1 Ternary Logic

When extended with $\perp$, $\mathbb{B}$ becomes a ternary logic with the values true (true or T), false (false or F), and unknown ($\perp$). Intuitively, this ternary logic will allow us to distinguish between statements that are *definitely* true, and statements that *may be* true. As detailed in supplemental appendix B, this logic can be encoded in classical logic for the purposes of targeting SMT solvers.

We define two additional operators for collapsing back down from ternary to classical logic. $D\ p$ ("definitely p") is defined by $DT = T$, $D\perp = F$, and $DF = F$; $M\ p$ ("maybe p") is defined by $MT = T$, $M\perp = T$, and $MF = F$.

5.2 Effect Expressions

Effect Expressions both give us a way of expressing symbolic values and of encoding sentences in a first-order logic, for discharging to an SMT solver.

Definition 5.1 (Effect Expressions)**.** We define the following grammar of effect-expressions

$$ee : \mathsf{EffExpr} ::= x \mid c \mid \bot \mid op(ee^*) \mid ee?\ ee\ \mathsf{else}\ ee \mid \forall x.ee$$

where every expression either has sort bool or sort int. The operators are the same as the bool and int operators from Figure 3. Recall that in the case of int operators, the pseudo-affine condition means that the quotient for / and mod must be a constant, and one side of $*$ must be a constant.

Definition 5.2 (Effect Environments)**.**

$$\gamma : \mathsf{EffEnv} = (\mathsf{Name}_{global} \uplus \mathsf{Name}_{local}) \rightarrow \mathsf{EffExpr}$$

are partial functions that default to mapping x to x, not $\bot$.

Effect environments abstract functions $\Sigma \rightarrow \Sigma$ with respect to control values, not stores Σ. This is why they may appear to be impredicative (mapping x to x by default). We define substitution $\gamma(ee)$ in the usual way. Using this, we can define composition of two effect environments $(\gamma \cdot \gamma')x = \gamma(\gamma'(x))$, which may also be resolved by substituting with γ inside the expressions bound by γ'. This definition of substitution extends naturally up to our later definitions of location sets LocSet, and effects a.

5.3 Global Dataflow

The major complication in our program analyses is handling mutable, global control state—which makes precise analysis of program control logic undecidable. Our dataflow analysis is symbolic (producing effect environments as a result) and control-sensitive (symbolic values reflect guards wrapped around statements). However we must make some kind of approximation to force convergence on loops. We use a very simple heuristic, expressed symbolically: If every loop iteration does not change the value of a global variable x, then the loop behaves as an identity function. Otherwise, the loop drives x to the uncertain value $\bot$. This usually suffices because configuration state that depends on the loop iteration is usually meaningless outside of the loop.

We define global dataflow analysis **ValG** : Stmt → EffEnv precisely in supplemental appendix C, along with lifting of expressions to effect expressions *Lift* : Expr → EffExpr.

5.4 Location Sets

Definition 5.3 (Location Set)**.**

$$\begin{array}{rcl} \mathcal{L} : \mathsf{LocSet} & ::= & \emptyset \mid \{x, ee^*\} \mid \mathcal{L} \cup \mathcal{L} \mid \bigcup_x \mathcal{L} \\ & \mid & \mathcal{L} \cap \mathcal{L} \mid \mathcal{L} - \mathcal{L} \mid \mathsf{filter}(ee, \mathcal{L}) \end{array}$$

Location sets symbolically abstract sets of global and heap locations in the store.

These sets support a set membership predicate $(\_ \in \_) : (\mathsf{Name} \times \mathsf{EffExpr}^n) \rightarrow \mathsf{LocSet} \rightarrow \mathsf{EffExpr}$ and an is-empty predicate $(\_ = \emptyset) : \mathsf{LocSet} \rightarrow \mathsf{EffExpr}$, both in the expected way (see supplemental appendix D for details)

Note that because effect expressions are a ternary logic, these location sets express upper and lower bounds on a set of locations: points definitely not in the set, points definitely in the set, and a penumbra of points ambiguously in the set. We collapse these sets down to "classical sets" using the aforementioned operators: $D\mathcal{L}$ meaning points definitely in the set, and $M\mathcal{L}$ meaning points that might be in the set. Thus $x \in D\mathcal{L}$ means $D(x \in Ls)$ and $x \notin M\mathcal{L}$ means $D(x \notin \mathcal{L})$.

5.5 Effects

Definition 5.4 (Effects)**.**

$$\begin{array}{rcl} a : \mathsf{Effect} & ::= & a\,;a \mid \emptyset \mid \mathsf{Guard}(ee, a) \mid \mathsf{Loop}(x, a) \\ & \mid & \mathsf{GlobalRead}(x) \mid \mathsf{GlobalWrite}(x) \\ & \mid & \mathsf{Read}(x, ee^*) \mid \mathsf{Write}(x, ee^*) \\ & \mid & \mathsf{Reduce}(x, ee^*) \mid \mathsf{Alloc}(x) \end{array}$$

This definition allows us to define the obvious translation of expressions (Eff_e : Expr → Effect) and statements (Eff : Stmt → Effect) into effects (see supplemental appendix E). Effects then allow us to define read, write, and reduce location sets.

To start, we define the set of buffers allocated by and visible to subsequent statements/effects:

$$\begin{array}{ll} \mathbf{A} : \mathsf{Effect} \rightarrow \mathsf{LocSet} & \\ \mathbf{A}\ \mathsf{Alloc}(x) & = \{x\} \\ \mathbf{A}\ (a_1\,;a_2) & = \mathbf{A}(a_1) \cup \mathbf{A}(a_2) \\ \mathbf{A}\ \_ & = \emptyset \end{array}$$

Definition 5.5 (Locations of an Effect)**.** Let $\mathbf{Rd}_\mathrm{G}$, $\mathbf{Wr}_\mathrm{G}$, $\mathbf{Rd}_\mathrm{H}$, $\mathbf{Wr}_\mathrm{H}$, and $\mathbf{R{+}}_\mathrm{H}$, be functions Effect → LocSet. To avoid redundancy, define common cases for all such functions $\mathcal{F}$:

$$\begin{array}{ll} \mathcal{F} : \mathsf{Effect} \rightarrow \mathsf{LocSet} & \\ \mathcal{F}\ \mathsf{Guard}(ee, a) & = \mathsf{filter}(ee, \mathcal{F}\ a) \\ \mathcal{F}\ \mathsf{Loop}(x, a) & = \bigcup_x \mathcal{F}\ a' \end{array}$$

Sequencing is defined differently for read and write sets:

$$\begin{array}{ll} \mathbf{Rd}_\mathrm{G}\ (a_1\,;a_2) & = \mathbf{Rd}_\mathrm{G}(a_1) \cup (\mathbf{Rd}_\mathrm{G}(a_2') - \mathbf{Wr}_\mathrm{G}(a_1) - \mathbf{A}(a_1)) \\ \mathbf{Wr}_\mathrm{G}\ (a_1\,;a_2) & = \mathbf{Wr}_\mathrm{G}(a_1) \cup (\mathbf{Wr}_\mathrm{G}(a_2') - \mathbf{A}(a_1)) \\ \mathbf{Rd}_\mathrm{H}\ (a_1\,;a_2) & = \mathbf{Rd}_\mathrm{H}(a_1) \cup (\mathbf{Rd}_\mathrm{H}(a_2') - \mathbf{Wr}_\mathrm{H}(a_1) - \mathbf{A}(a_1)) \\ \mathbf{Wr}_\mathrm{H}\ (a_1\,;a_2) & = \mathbf{Wr}_\mathrm{H}(a_1) \cup (\mathbf{Wr}_\mathrm{H}(a_2') - \mathbf{A}(a_1)) \\ \mathbf{R{+}}_\mathrm{H}\ (a_1\,;a_2) & = \mathbf{R{+}}_\mathrm{H}(a_1) \cup (\mathbf{R{+}}_\mathrm{H}(a_2') - \mathbf{A}(a_1)) \end{array}$$

Each function detects its corresponding leaf-node:

$$\begin{array}{ll} \mathbf{Rd}_\mathrm{G}\ \mathsf{GlobalRead}(x) & = \{x\} \\ \mathbf{Wr}_\mathrm{G}\ \mathsf{GlobalWrite}(x) & = \{x\} \\ \mathbf{Rd}_\mathrm{H}\ \mathsf{Read}(x, ee_1, \ldots, ee_n) & = \{x, ee_1, \ldots, ee_n\} \\ \mathbf{Wr}_\mathrm{H}\ \mathsf{Write}(x, ee_1, \ldots, ee_n) & = \{x, ee_1, \ldots, ee_n\} \\ \mathbf{R{+}}_\mathrm{H}\ \mathsf{Reduce}(x, ee_1, \ldots, ee_n) & = \{x, ee_1, \ldots, ee_n\} \\ \mathcal{F}\ \_ & = \emptyset \end{array}$$

From these five primitive sets we can define six other useful sets:

$$\begin{aligned}
\mathbf{Rd}\ a &= \mathbf{Rd}_\mathrm{G}\ a \cup \mathbf{Rd}_\mathrm{H}\ a \\
\mathbf{Wr}\ a &= \mathbf{Wr}_\mathrm{G}\ a \cup \mathbf{Wr}_\mathrm{H}\ a \\
\mathrm{R+}\ a &= \mathrm{R+}_\mathrm{H}\ a - \mathbf{Wr}_\mathrm{H}\ a \\
\mathbf{All}\ a &= \mathbf{Rd}\ a \cup \mathbf{Wr}\ a \cup \mathrm{R+}_\mathrm{H}\ a \\
\mathbf{Mod}\ a &= \mathbf{Wr}\ a \cup \mathrm{R+}_\mathrm{H}\ a \\
\mathbf{RW}\ a &= \mathbf{Rd}\ a \cup \mathbf{Wr}\ a
\end{aligned}$$

5.6 Effects as Abstraction

The different objects we have talked about so far each abstract some part of the program. For instance, the dataflow analysis of a statement $\mathbf{ValG}\ [\![s]\!]$ is an abstraction of its denotation $\mathrm{S}\ [\![s]\!]$ with respect to global values. Similarly, the effect extracted from an expression $\mathit{Eff}_e\ [\![e]\!]$ abstracts its denotation $\mathrm{E}\ [\![e]\!]$, and the effect extracted from a statement $\mathit{Eff}\ [\![s]\!]$ abstracts its denotation $\mathrm{S}\ [\![s]\!]$. But what do we mean by this?

The effect abstraction a for a statement s with denotation f guarantees a few properties. First, it provides an analogue of the "frame axiom" from separation logic. If a location x lies outside of $M\mathbf{Mod}(a)$, then it is unmodified: $f\sigma x = \sigma x$. Second, if a location is in the write set $x \in D\mathbf{Wr}(a)$, then the post-hoc value at that location $f\sigma x$ is determined solely by the values at read locations $y \in M\mathbf{Rd}(a)$. Third, if a location is reduced to $x \in D\mathrm{R+}(a)$, then the difference between the initial and final value at that location $f\sigma x - \sigma x$ is determined solely by values at read locations $y \in M\mathbf{Rd}(a)$. Finally, so long as the values at read locations $y \in M\mathbf{Rd}(a)$ are determined, then one of the three previous cases applies to every store location, even if we can't be certain which set(s) the location is in.

Even more simply in the case of expression abstraction, the effect a of an expression e with denotation $f : \Sigma \to \mathsf{Val}$ guarantees one property: The value $f\sigma$ is solely determined by the values at read locations $y \in M\mathbf{Rd}(a)$.

5.7 Basic Program Rewrites

The preceding analysis objects allow us to turn program equivalence checks into SMT queries.

Reorder statements. The rewrite $s_1 ; s_2 \rightsquigarrow s_2 ; s_1$ is safe when $\mathsf{Commutes}\ \mathit{Eff}\ [\![s_1]\!]\ \mathit{Eff}\ [\![s_2]\!]$ holds. Commutativity of statements is defined as non-interference of effects. A special exception must be made for locations that are reduced.

Definition 5.6 (Commutativity)**.**

$$\begin{aligned}
&\mathsf{Commutes}\ a_1\ a_2 = \\
&D\begin{pmatrix} \mathbf{Wr}(a_1) \cap \mathbf{All}(a_2) = \emptyset \ \wedge\ \mathbf{Wr}(a_2) \cap \mathbf{All}(a_1) = \emptyset \\ \mathrm{R+}(a_1) \cap \mathbf{Rd}(a_2) = \emptyset \ \wedge\ \mathrm{R+}(a_2) \cap \mathbf{Rd}(a_1) = \emptyset \end{pmatrix}
\end{aligned}$$

Shadow statement. The rewrite $s_1 ; s_2 \rightsquigarrow s_2$ is safe when $\mathsf{Shadows}\ \mathit{Eff}\ [\![s_1]\!]\ \mathit{Eff}\ [\![s_2]\!]$ holds. Whereas commutativity requires reasoning about what definitely doesn't intersect (and hence what memory might be touched), shadowing requires reasoning positively about what definitely is overwritten—which is why a one-sided approximation sets is insufficient.

Definition 5.7 (Shadowing)**.**

$$\begin{aligned}
&\mathsf{Shadows}\ a_1\ a_2 = \\
&\forall x \in M\mathbf{Mod}(a_1) \implies (x \notin M\mathbf{Rd}(a_2)\ \wedge\ x \in D\mathbf{Wr}(a_2))
\end{aligned}$$

New config write. The rewrite $s \rightsquigarrow s ; x_g{=}e$ is always safe, but only results in code that is equivalent modulo $\{x_g\}$. As we will soon see (§6.2), performing this rewrite in a context requires satisfying additional conditions, but in isolation it is very simple.

5.8 Loop Rewrites

When working with rewrites of loops, it is convenient to abbreviate notation for an iteration variable being in bounds. If the variable x occurs in `for` x `in` $e_{lo} .. e_{hi}$ `do` , then let $\mathsf{Bd}(x) = \mathit{Lift}\ [\![e_{lo}]\!] \le x < \mathit{Lift}\ [\![e_{hi}]\!]$ in the following.

Loop reordering. One of the most basic non-trivial loop transformations is loop-reordering. When can we rewrite `for` x `do` `for` y `do` s into `for` y `do` `for` x `do` s? This transformation is valid when the loop bounds commute with the body, and when any loop iterations that are moved past each other commute. To formulate these conditions, let a_x be the effect of the x-loop's bound-expressions and a_y similarly for the y-loop. Let x', y' be copies of these iteration variables s.t. $s' = [x \mapsto x'][y \mapsto y']s$. Let $a = \mathit{Eff}\ [\![s]\!]$ and $a' = \mathit{Eff}\ [\![s']\!]$. Then the reordering condition may be precisely stated as

$$\begin{aligned}
&\big(\forall x, y.\ M\mathsf{Bd}(x, y) \implies \mathsf{Commutes}((a_x ; a_y), a)\big) \\
&\wedge \begin{pmatrix} \forall x, y, x', y'.\ M(\mathsf{Bd}(x, y, x', y') \wedge x < x' \wedge y' < y) \\ \implies \mathsf{Commutes}(a, a') \end{pmatrix}
\end{aligned}$$

Loop fusion & fission. Another basic loop transformation is to fuse two loops together, or in reverse to fission one loop in two. When can we rewrite `for` x `do` $s_1 ; s_2$ into (`for` x `do` s_1) ; `for` x `do` s_2? This is possible when the loop bound commutes with the first statement, and when the statements that get reordered commute with each other. Letting a_x be the effect of the loop bounds, $a_1 = \mathit{Eff}\ [\![s_1]\!]$, $s_2' = [x \mapsto x']s_2$ and $a_2' = \mathit{Eff}\ [\![s_2']\!]$ we can state fission conditions precisely as

$$\begin{aligned}
&(\forall x.\ M\mathsf{Bd}(x) \implies \mathsf{Commutes}(a_x, a_1))\ \wedge \\
&\big(\forall x, x'.\ M(\mathsf{Bd}(x, x') \wedge x' < x) \implies \mathsf{Commutes}(a_1, a_2')\big)
\end{aligned}$$

Loop removal. In order for the rewrite `for` x `do` $s \rightsquigarrow s$ to be safe, the variable x must not be free in s, s must be idempotent, and the loop must run for at least one iteration. If $a = \mathit{Eff}\ [\![s]\!]$, then these conditions are precisely

$$(\exists x.\ D\mathsf{Bd}(x))\ \wedge\ \mathsf{Shadows}(a, a)$$

6 Contextual Analyses

In order to make our program rewriting primitives useful, we must be able to modify some fragment of a procedure in a context. In this section, we define one-holed statement contexts, define how to process them, and extend equivalences between statements to account for context.

6.1 Contexts & Derived Quantities

Definition 6.1 (Contexts).

$$C : \mathsf{Ctxt} \;::=\; \bullet \mid C; s \mid s; C \mid \texttt{for}\ x\ \texttt{in}\ e\,..\,e\ \mathsf{do}\ C \;\mid\; \texttt{if}\ e\ \mathsf{then}\ C$$

The expression $C[s]$ means a statement resulting from substituting the hole ($\bullet$) in context C with statement s. Similarly, we can have a Proc context: $\texttt{proc}\ p : \tau_s\ \texttt{assert}\ e\ \mathsf{do}\ C$.

We define three derived quantities from a context/statement pair C/s: (1) CtrlPred $[\![C]\!]\,s$: EffExpr, a predicate expressing under what conditions the statement s will execute; (2) PreValG $[\![C]\!]\,s$: EffEnv, capturing the dataflow values right before executing s; and (3) PostEff $[\![C]\!]\,s$: Effect, telling us the effect of context code that executes *after* s. (See supplemental appendix F for details.)

6.2 Context Extension

Using these tools we can get from an argument of the form $s_1 \cong s_2$ back up to an argument of the form $C[s_1] \cong C[s_2]$. Thus, we can reach into the body of some procedure and perform a local rewrite, while maintaining equivalence of the overall procedure.

Consider a context C with statements s_1 and s_2, as well as a set of global names $\mathcal{L}$ to consider equivalence "up to."

$$\begin{aligned} \text{Let } p &= \mathsf{CtrlPred}\ C\ s_1 \\ \gamma &= \mathsf{PreValG}\ C\ s_1 \\ a &= \mathsf{PostEff}\ C\ s_1 \\ \mathcal{L}' &= M(\mathcal{L} - \mathbf{Wr}_{\mathrm{G}}\ a) \\ s_1', s_2' &= \gamma(s_1), \gamma(s_2) \end{aligned}$$

$$\text{If} \quad (Mp \implies s_1' \cong_{\mathcal{L}} s_2') \wedge D(\mathbf{Rd}_{\mathrm{G}}\ a \cap \mathcal{L} = \emptyset)$$
$$\text{Then } C[s_1] \cong_{\mathcal{L}'} C[s_2]$$

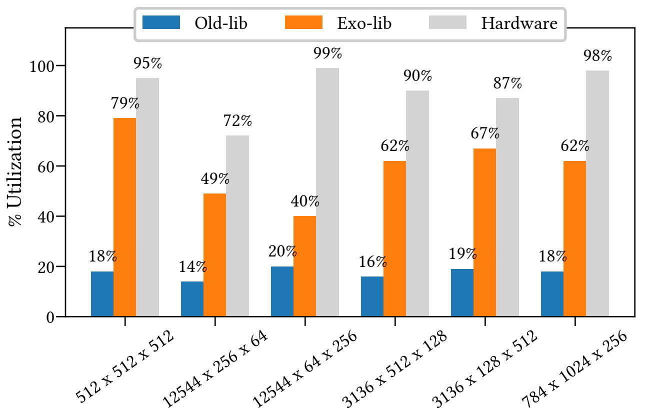
7 Case Studies

7.1 Gemmini

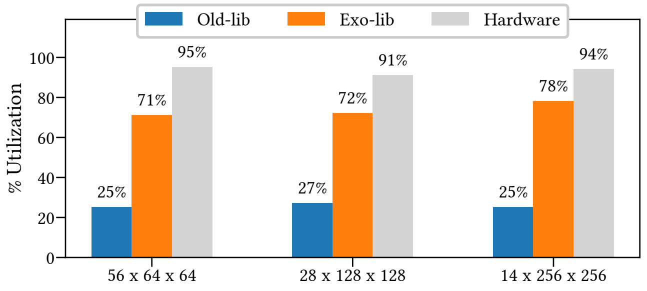
Using Exo, we developed highly-optimized schedules for Gemmini [16], a DNN accelerator, which significantly outperformed DNN kernel implementations that had been hand-written by Gemmini's designers.

We targeted Gemmini's default architectural instantiation, which include a 16x16 systolic array that performs block matrix multiplications, a 256KB scratchpad for quantized inputs and weights, and a 64KB accumulator for partial sums. Gemmini's instruction set architecture (ISA) includes low-level instructions to move strided matrices to and from the scratchpad, as well as instructions to calculate dot products and perform non-linear activations on this data.

Gemmini also ships with a hand-written C library for common DNN kernels. This library wraps calls to Gemmini's low-level ISA in statically-scheduled, hand-tuned loops. However, Gemmini can also be built with hardware loop unrollers that dynamically schedule these kernels to maximize overlap between data loads, data stores, and matrix multiply operations.



(a) MATMUL utilization (as a percentage of peak FLOPS). X axis labels are the size of matrices in N x M x K.



(b) CONV utilization (as a percentage of peak FLOPS). X axis labels are the shape of convolution in *output dimension* x *output channel* x *input channel*.

Figure 4. Performance of Exo-generated code on the Gemmini DNN accelerator. Exo-generated code achieves much higher performance than the DNN kernels hand-written by the designers of Gemmini (Old-lib). Gemmini's dynamically-scheduled hardware loop unrollers (Hardware) outperform Exo by using additional hardware resources, but therefore require additional chip area and power consumption.

The hardware implementations typically run much faster than the software implementations at the cost of hardware complexity, area, power consumption, and reduced scheduling flexibility. The hardware kernels also have fixed loop orders and dataflows, while the software can adapt these to different tensor shapes.

We implemented kernels for matrix multiply (MATMUL) and convolutional (CONV) layers in Exo and compared their performance against Gemmini's handwritten C library and hardware loop unrollers. The results are shown in Figures 4a and 4b, respectively. The tensor shapes in both are selected from those in a ResNet-50 DNN with a batch size of 4.

On average, Exo-generated code outperforms Gemmini's handwritten C library by 3.5× on the MATMUL sizes listed above, and achieves 67% of the performance of the hardware loop unrollers. For the convolutions listed, it runs 2.9× faster than the handwritten library, and is competitive with the hardware loop unroller, achieving 79% of its performance.

Note that the hardware loop unrollers use optional hardware resources (increasing area and power consumption) which are not available to Exo or the handwritten C library. However, we expect that changing Gemmini's ISA to support coarser-granularity instructions and better schedules may be able to close this performance gap in the future, providing software-programmable performance comparable to the inflexible hardware loop-unrollers.

Finally, Exo enabled faster co-design of Gemmini's hardware-software interface. When we started targeting Gemmini, its low-level hardware configuration instructions had many side effects which made optimizations difficult to reason about, limiting the performance we could achieve. We worked with the Gemmini hardware designers to disaggregate these configuration instructions into more orthogonal components; e.g. instructions which configured Gemmini's memory units would no longer have any side effects on the arithmetic units. 46 lines in Gemmini's handwritten C library had to be updated after this change, compared to only 5 in Exo's implementation. Exo made it easier for programmers to target fluid and changing hardware targets, which is common when developing new accelerators.

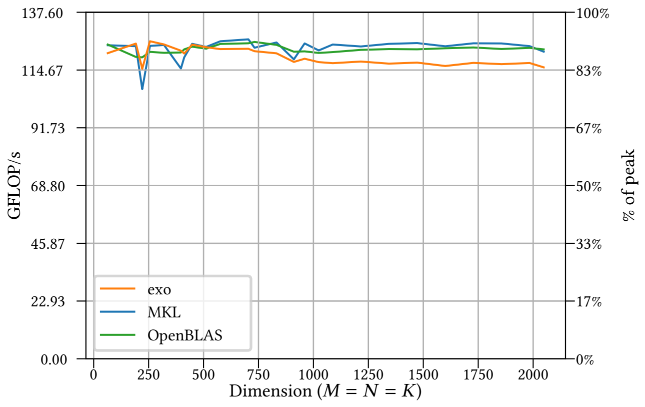
7.2 x86

As an acid test of the language design, we optimized matrix-matrix multiplication (SGEMM) for x86, where we can compare against state-of-the-art libraries that run near theoretical peak compute throughput. We chose to target single-core x86 with AVX512 extensions.[2]

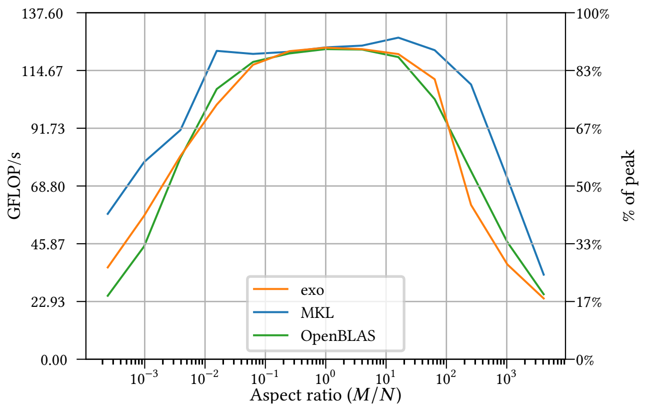
Recall that the computation is given by $C \mathrel{+}= A \cdot B$ where C is $M \times N$, A is $M \times K$, and B is $K \times N$. Our Exo implementation decomposes the problem as follows: at the deepest level of blocking, a register-blocked micro-kernel accumulates the inner dimension into a 6×64 panel of C, the output matrix. The level above the micro-kernel handles edge cases by dispatching to *specialized* versions of the micro-kernel for each edge case. Along the bottom, five distinct kernels are needed as they are always 64 elements wide and never 0 or 6 tall; similarly, four distinct kernels are needed along the right. The variable tail on the right edge is handled by masked loads. Finally, one level above this handles staging memory and blocking.

Every one of these routines was produced by scheduling and specializing a single, naive implementation of SGEMM consisting of three nested loops. Unification and equivalent-call replacement were crucial for avoiding any sort of error-prone, manual optimization.

[2]Although multi-core implementations are valuable, single-core workloads are representative of practice (ML inference in interactive web services is often run batch-parallel on single-core kernels), and the baselines are highly-optimized.



(a) SGEMM performance on square matrices. We approximately match other systems on square matrices.



(b) SGEMM performance with fixed workload and variable output aspect ratio. $K = 512$ and $M \times N = 512^2$, with the ratio of M to N varying. We match OpenBLAS performance across aspect ratios.

Figure 5. SGEMM performance compared to state-of-the-art libraries on x86. Benchmarks were run on one core of an Intel i7-1185G7 running at 4.3GHz.

The performance results are shown in Figure 5. All benchmarks were run on an Intel i7-1185G7 at 4.3 GHz, a Tiger Lake CPU with AVX-512 instructions and peak single-precision floating-point performance of 137.60 GFLOPs. We tested our SGEMM against the hand-optimized implementations in Intel's MKL and the open-source OpenBLAS in two experiments. First (Fig. 5a), we tested square matrices , so $M = N = K$. Each implementation performs quite closely (within measurement noise), between 80-95% of theoretical peak FLOPS across the parameter range.

Second (Fig. 5b), we tested our SGEMM on a fixed workload, but with a variable aspect ratio for C. Specifically, we fix the inner dimension $K = 512$ and the product $MN = 512^2$, then we sweep across the ratio M/N keeping the total FLOP count identical across experiments. Here, Exo matches OpenBLAS almost exactly, but MKL pulls ahead of both implementations when the aspect ratio is very far from square. MKL includes

Impl.	N	W	H	IC	OC	% of peak
Exo	5	82	102	128	128	40.50%
Halide	5	82	102	128	128	40.59%
oneDNN	5	82	102	128	128	40.55%

Figure 6. Summary of x86 CONV performance results. Single-threaded performance of various implementations with no padding and unit stride. A ReLU activation is applied. Benchmarks were run on an Intel i7-1185G7 running at 4.3GHz on a single core. The size was chosen to match the previously-published hand-scheduled Halide implementation. All three specialize or JIT to tune their code to specific sizes.

more specialized kernels for these extreme aspect ratios, which would be natural to do with further scheduling in Exo, as well.

For a final experiment, we tried to replicate the convolutional layer performance of a highly-tuned implementation provided by the Halide project. State of the art convolutions specialize or JIT-compile code templates to particular input, output, and kernel sizes. In Halide's case, it specialized to a batch size of 5, a kernel size of 3×3, an output size of 80×100, and 128 channels for both input and output. There is no padding and unit stride is used. We configured Intel's oneDNN convolution to use these parameters and scheduled a basic description of convolution in Exo to these parameters, too. The results are shown in Figure 6. Our CONV performs almost identically to the optimized baselines.

Overall, we believe these results show that Exo can be used to achieve performance competitive with state-of-the-art, highly hand-tuned libraries on x86.

7.3 Code Size

Figure 7 summarizes some statistics regarding the size of Exo programs relative to hand-written C baselines.

On x86, our SGEMM schedule instantiates many specialized micro-kernels for handling loop tail cases at higher levels. Unlike Gemmini, it does not have SGEMM-specific hardware to utilize that might reduce the scheduling burden. Even so, the basic algorithm is expressed in 11 statements (the function signature, three loops, an accumulation statement, and a handful of size assertions) and 162 scheduling directives. The generated C code totals 831 source lines of code This already constitutes a nearly 5x code size reduction, but a comparison to OpenBLAS (an established open-source implementation) is even more favorable: at least 1690 source lines of code[3] make up that implementation. MKL is more complex, still.

Although the x86 conv implementation is "only" half the size of the equivalent generated C, it is much more flexible

[3]Summing the source line counts of the files mentioned in `kernel/-x86_64/KERNEL.SKYLAKEX` for non-transposed SGEMM gives a very loose lower bound

App.	Platform	C (gen)	C (ref)	Alg.	Sched.
MATMUL	Gemmini	462	313	23	43
CONV	Gemmini	8317	450	26	44
SGEMM	x86	846	>1,690	11	162
CONV	x86	102	>5,400	23	39

Figure 7. Source code sizes for matrix multiplication and convolutional layer on Gemmini and x86. Gemmini implements a fixed-point matrix multiply neural network layer (with fused ReLU activation), while x86 implements the BLAS SGEMM kernel. Both implement a standard 2D convolutional layer with ReLU activation. The Exo sources are counted in lines of code for the algorithm and number of directives for the schedule. This is compared to the size of both the Exo-generated C and state-of-the-art reference implementations (Gemmini standard library, OpenBLAS, and oneDNN, respectively) in source lines of code.

since other specialized versions can be quickly instantiated by meta-programming the schedule in Python. The size of the most comparable open-source implementation, Intel's oneDNN, is difficult to measure; just one file in the implementation measures well over 5000 source lines of code[4]. The size of the Halide code and schedule was nearly identical to ours: 64 relevant lines, compared to 62.

The story is similar for our Gemmini kernels. Both the matmul and conv Exo implementations are an order of magnitude smaller than the original, handwritten C implementations. The large generated code sizes reflect the high degree of loop unrolling in the generated schedules. A real application would likely either resort to the C preprocessor to manage this complexity, or not attempt the transformation at all (or as aggressively) beyond whatever the C compiler might choose to do automatically.

8 Related Work

User-Schedulable Languages Exo builds on the idea of programmer-visible scheduling languages, popularized in part by Halide and TVM, and used in many recent languages and systems [5, 8, 18, 21, 24, 28, 29, 34, 35, 41]. The explicit control over compiler transformations offered by user-schedulable languages was foreshadowed earlier in many script- or pragma-based compiler tools in HPC [7, 10, 11, 20, 39], and the definition of parametric optimization spaces in SPIRAL [15], all of which have been applied to matrix-matrix multiply and related kernels. The polyhedral loop optimization community has simultaneously explored similar ideas in its own context [3, 4, 32, 36, 37, 40].

Exo builds on attempts to formalize guarantees of safety and equivalence under scheduling in Halide [30]. In sharp contrast to Halide, Exo adopts the approach of implementing scheduling via algebraic rewrites within a core language.

[4]`src/cpu/x64/jit_avx512_common_conv_kernel.cpp`

While prior systems which follow this approach work mostly on restricted functional languages, where equivalence before and after rewrites is straightforward (and often not formally checked) [19, 25, 31], Exo rewrites *imperative* code, and relies on effect analyses which reduce to SMT for verification.

Instruction Selection Exo's instruction/procedure mapping mechanism is related to the classic problem of instruction selection [2]. Traditional instruction selection applies local pattern matching rules to replace small IR fragments with equivalent instructions, but this struggles to effectively exploit accelerator instructions which correspond to large, complex program fragments. Recent work applies more powerful search techniques to target more complex SIMD instructions using program synthesis [27] and equality saturation [33]. Exo allows substitution of much larger program fragments with arbitrary equivalent procedures, under explicit programmer control, and allows these substitutions to be interleaved with further scheduling transformations rather than confined to the compiler backend. TVM provides a related "tensorization" directive for replacing loop fragments with instructions asserted as equivalent [8], but it lacks the combination of automation and checking provided by Exo's unification procedure.

Program Analysis Our framework for verifying equivalence and safety of Exo programs builds on several threads from type systems and dependence analysis. Dependently-typed arrays, especially as adapted in the formalization of Halide, inform our treatment of memory safety [22, 23, 30, 38]. Dependence analysis, especially on static control programs, forms a common basis for reasoning about the safety of loop transformations [12, 14]. When combined with reasoning about affine indexing, this is the basis of polyhedral compilation [13]. In contrast, our approach builds on effect types, as proposed by Gifford and Lucassen [17]. While these approaches are distinct, the earliest foundations of dependencies for program parallelization define conditions on read and write sets closely related to our effect analyses [6].

Despite this difference, Exo can be seen as a polyhedral compiler, in the sense that it is built on linear integer arithmetic and static control programs. However, the program analysis used in Exo goes beyond what is normally called "polyhedral analysis" in two respects: mutable control state (for which we must rely on an approximating symbolic dataflow analysis §5.3), and justifying code deletion/insertion (§5.7, §6.2). Both of these phenomena are necessary to support scheduling of hardware accelerators that make use of configuration state. They also forced us to adopt ternary logic at the base of our program analysis in order to safely propagate the dataflow approximations. If configuration state were eliminated, Exo would more closely resemble traditional polyhedral compilers focused purely on reordering statement instances.

9 Limitations & Future Work

Multi-Core Semantics Although the instruction replacement directive (§3.4) enables users to access fine-grained intra-instruction or SIMD parallelism, Exo does not currently model multi-core parallelism. Naïvely, we could introduce a parallel for-loop with OpenMP-like semantics. Our effect analysis is powerful enough to conservatively check that different loop iterations touch strictly disjoint regions of memory. However, there is no single platform independent approach to threading—which clashes with our design goal of externalizing hardware backends. A more ambitious solution would find some way to externalize both the semantics and primitives associated with different kinds of threading. (e.g. pthreads, CUDA, MPI, etc.)

Alternatively, the `.replace()` directive applied to a no-op instruction can serve an escape hatch to, for example, inject OpenMP pragmas around a given loop. We tested this on our conv implementation and observed that our new implementation still matches Halide, while both pull ahead of oneDNN by 25% (flops) on 8 or more threads.

Automatic Scheduling We have not yet written any autoschedulers [1, 9, 26, 42] for Exo, but plan to. We expect Exo autoscheduling to differ from prior systems in two essential ways. First, because hardware targets are externalized, idiosyncratic, and frequently proprietary, we do not expect any one single autoscheduling strategy to work across all accelerators. Second, because Exo schedules are composable (as successive rewrites) rather than monolithic, Exo autoschedulers can also be developed compositionally. This opens up the possibility of developing libraries of re-usable mid-level scheduling operators built from semi-automated combinations of primitive scheduling operators. With time, whole suites of optimization passes could be written—entirely external to the Exo compiler.

Acknowledgments

We would like to thank the first Exo users, especially Grace Dinh, Abhijit Davare, and Kevin Qian, for providing feedback. We would also like to thank the PLDI reviewers for their suggestions. This work was partially supported by the Applications Driving Architectures (ADA) center, one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. It was also funded in part by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006) and NSF awards CCF-1723445 & CCF-1846502. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Yuka Ikarashi was supported by a Funai Overseas Scholarship, Masason Foundation fellowship, and Great Educators fellowship.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12. https://doi.org/10.1145/3306346.3322967

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2 ed.). Pearson Education.

[3] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *PACT*. IEEE Computer Society, San Francisco, CA, USA, 138–149. https://doi.org/10.1109/PACT.2015.17

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019* (Washington, DC, USA). IEEE, Piscataway, NJ, USA, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12* (Salt Lake City, UT, USA). IEEE, Piscataway, NJ, USA, 66. https://doi.org/10.1109/SC.2012.71

[6] A. J. Bernstein. 1966. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers* EC-15, 5 (1966), 757–763. https://doi.org/10.1109/PGEC.1966.264565

[7] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. University of Southern California.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[9] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 3393–3404. http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs

[10] Sébastien Donadio, James C. Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David A. Padua, and Keshav Pingali. 2005. A Language for the Compact Representation of Multiple Program Versions. In *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005*. Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151. https://doi.org/10.1007/978-3-540-69330-7_10

[11] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (Tampa, Florida) *(SC '06)*. Association for Computing Machinery, New York, NY, USA, 83–es. https://doi.org/10.1145/1188455.1188543

[12] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[13] Paul Feautrier and Christian Lengauer. 2011. The Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[15] Franz Franchetti, Tze Meng Low, Doru-Thom Popovici, Richard Michael Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[16] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[17] David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

[18] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs. arXiv:2003.06324 [cs.PL]

[19] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. 2020. A Language for Describing Optimization Strategies. arXiv:2002.02268 [cs.PL]

[20] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009* (Rome, Italy). IEEE, Piscataway, NJ, USA, 1–11. https://doi.org/10.1109/IPDPS.2009.5161004

[21] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* 38, 6 (2019), 201:1–201:16. https://doi.org/10.1145/3355089.3356506

[22] C. Barry Jay and Milan Sekanina. 1997. Shape Checking of Array Programs. In *Computing: the Australasian Theory Symposium*. Sydney, Australia.

[23] C. B. Jay and P. A. Steckler. 1998. The functional imperative: Shape!. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer, Berlin, Heidelberg, 139–153. https://doi.org/10.1007/BFb0053568

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–29. https://doi.org/10.1145/3133901

[25] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization Via High-level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[26] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (2016), 83:1–83:11. https://doi.org/10.1145/2897824.2925952

[27] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019.

Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3297858.3304059

[28] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

[29] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018), 106–115. https://doi.org/10.1145/3150211

[30] Alex Reinking, Gilbert Bernstein, and Jonathan Ragan-Kelley. 2020. *Formal Semantics for the Halide Language.* Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-40.html

[31] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 74–85. https://doi.org/10.1109/CGO.2017.7863730

[32] Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2018. Meta-Programming for Cross-Domain Tensor Optimizations. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE 2018)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3278122.3278131

[33] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[34] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL]

[35] Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. 2019. SWIRL: High-performance many-core CPU code generation for deep neural networks. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1275–1289. https://doi.org/10.1177/1094342019866247

[36] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, Berlin, Heidelberg, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[37] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule Trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). INRIA, Vienna, Austria, 1–9.

[38] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

[39] Qing Yi, Keith Seymour, Haihang You, Richard W. Vuduc, and Daniel J. Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)* (Rome, Italy). IEEE, Piscataway, NJ, USA, 1–8. https://doi.org/10.1109/IPDPS.2007.370637

[40] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Springer, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-642-37658-0_2

[41] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491

[42] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. Article 49, 17 pages. https://doi.org/10.5555/3488766.3488815