Lightweight and Locality-Aware Composition of Black-Box Subroutines

MANYA BANSAL, Massachusetts Institute of Technology, USA
DILLON SHARLET, Google, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA

Subroutines are essential building blocks in software design: users encapsulate common functionality in libraries and write applications by composing calls to subroutines. Unfortunately, performance may be lost at subroutine boundaries due to reduced locality and increased memory consumption. Operator fusion helps recover the performance lost at composition boundaries. Previous solutions fuse operators by manually rewriting code into monolithic fused subroutines, or by relying on heavy-weight compilers to generate code that performs fusion. Both approaches require a semantic understanding of the entire computation, breaking the decoupling necessary for modularity and reusability of subroutines.

In this work, we attempt to identify the minimal ingredients required to fuse computations, enabling composition of subroutines without sacrificing performance or modularity. We find that, unlike previous approaches that require a semantic understanding of the computation, most opportunities for fusion require understanding only data production and consumption patterns. Exploiting this insight, we add fusion on top of black-box subroutines by proposing a lightweight enrichment of subroutine declarations to expose data-dependence patterns. We implement our approach in a system called Fern, and demonstrate Fern's benefits by showing that it is competitive with state-of-the-art, high-performance libraries with manually fused operators, can fuse across library and domain boundaries for unforeseen workloads, and can deliver speedups of up to 5× over unfused code.

CCS Concepts: • **Software and its engineering** → **Source code generation**; • **Mathematics of computing** → **Mathematical software performance**.

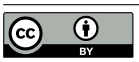
Additional Key Words and Phrases: Operator Fusion, High-Performance Composition, External Function Calls

ACM Reference Format:
Manya Bansal, Dillon Sharlet, Jonathan Ragan-Kelley, and Saman Amarasinghe. 2025. Lightweight and Locality-Aware Composition of Black-Box Subroutines. *Proc. ACM Program. Lang.* 9, PLDI, Article 189 (June 2025), 25 pages. https://doi.org/10.1145/3729292

1 Introduction

Subroutines make it possible to write complex and scalable software. The modularity subroutines offer is due to a division of responsibilities: the caller and callee are agnostic to each other's implementation, and, at the point of the subroutine call, the compiler is only tasked with meeting the contract set up by the calling convention. High-performance applications, too, are written as a composition of subroutines from popular libraries [3, 20, 22, 23, 28, 30, 33].

Authors' Contact Information: Manya Bansal, Massachusetts Institute of Technology, Cambridge, USA, manya227@mit.edu; Dillon Sharlet, Google, Menlo Park, USA, dsharlet@google.com; Jonathan Ragan-Kelley, Massachusetts Institute of Technology, Cambridge, USA, jrk@mit.edu; Saman Amarasinghe, Massachusetts Institute of Technology, Cambridge, USA, saman@csail.mit.edu.



ACM 2475-1421/2025/6-ART189
https://doi.org/10.1145/3729292

While the division of responsibilities makes subroutines a powerful abstraction, it also makes their composition slow. Each subroutine introduces an explicit boundary, forcing materialization of large intermediate results, which leads to poor cache utilization and increased data copying. For example, one can compute a linear layer in a neural network (a matrix multiplication and an addition) by composing subroutines, producing complete intermediates at each step. Or, one can compute the output at a granularity of tiles, immediately applying all operators for a given tile. For a matrix of size 1024×1024, a tile size of 64×64, and an L3 cache of 9.0 MiB, the tiled program has a 28% higher cache hit rate, 63% faster runtime, and smaller memory allocations for intermediates. This technique of evaluating operations back-to-back on small chunks of data as opposed to entire outputs is a well-known optimization called **operator fusion**.

Fusion involves manipulating iteration spaces to produce computationally equivalent, and often faster, programs. Many libraries and compilers implement fusion as an optimization, but make different decisions about how they manipulate iteration spaces and their resident computations. These systems can be categorized as operating on the iteration language (which defines looping over operands and loop-carry dependencies — *how to compute?*), or the computation language (which defines operations and computes expressions — *what to compute?*), or both.

Many compilers operate on the *iteration language* and *computation language* simultaneously. Both the earliest instances of fusion [17, 48] and recent technologies like LLVM loop fusion [5], the polyhedral framework [1], and MLIR loop fusion [36] work directly on pre-existing source code. Since they act on the source language, their ability to perform fusion is limited by the strength of the compiler's analysis. High-performance code makes rampant use of inline assembly, aliasing, and complex control flow. This code, where fusion is most important, is unfortunately also an adversarial case for the compiler's analysis machinery. For example, no general-purpose compiler today can fuse two hand-optimized matrix multiplications.

In contrast, some high-level, domain-specific languages (e.g., Halide [38], TACO [27], TVM [10] and Mosaic [4]) expose *a computation language*. These compilers can fuse through complicated dependence patterns by restricting their language to a particular domain. They derive iteration spaces present implicitly in scalar representations, and generate high-performance fused code all the way from outermost loops down to the leaf computations. Therefore, in addition to their fusion machinery, they also require infrastructure to generate high-performance code that rivals handwritten implementations. Even so, fusion opportunities are limited to exactly the computations that the DSL can express. To fuse computations across domains (e.g., databases and scientific computation), a new DSL (SciDB [7]) must be invented.

Finally, programmers can write fused code manually, exercising complete control over the *iteration language* and *computation language*. In the best case, it is laborious to modify handwritten high-performance code. In the worst case, it is impossible: programmers may not have the source code for libraries that target undocumented accelerators (e.g., Apple's Accelerate library [22]).

In all three cases — general-purpose compilers, DSL compilers, or programmers hand-fusing code — the fusion machinery understands the complete semantics of the computation, breaking the decoupling of implementations that subroutines enable.

In this work, we propose an approach that understands only the *iteration language*. Our core insight is that fused code tends to divide responsibility between inner, independent loops that perform the computation and contain hardware-specific logic, and outer loops that orchestrate data movement. We rely on black-box subroutines to perform the computation in the inner loops, while generating code for the outer loops using simple dataflow analysis of the subroutines' dependencies. Because we reuse high-performance inner-loop implementations, we do not need a compiler capable of automatically generating high-performance code. By operating at the level of subroutine interfaces, we can fuse computations across domains and libraries.

Our work relies on two key ideas:

(1) Fusion requires understanding the data production and consumption patterns of subroutines, but not understanding the functional semantics of the computation itself.
(2) Inner-loop kernels used in fused code often operate on subsets of data that are naturally encoded in the same type of data structure as the original unfused operations (e.g., a matrix tile is itself a matrix), enabling reuse of the same kernel for both fused and unfused code.

Based on these ideas, we develop Fern, a simple, domain-independent system that fuses black-box subroutines. Fern relies on enriched subroutine interfaces that communicate data production and consumption patterns for each subroutine. Using these patterns, Fern calls the original black-box subroutines with subsets of the data and stitches them together by generating an outer-loop nest. To work with Fern, subroutines must meet the following criteria:

(1) The subroutine must produce a single output.
(2) Values of the output must depend only on values of the input arguments. The subroutine can enact side effects that have no impact on the value of the final output.
(3) The size of the subroutine's output should not depend on values of the data.
(4) Outputs are produced in destination-passing style.

In our experience, subroutines over dense, multi-dimensional arrays often satisfy these criteria out of the box, whereas irregular data may require modification or fail to meet them.

We show that Fern can (1) match high-performance libraries with manually fused operators (Section 9.2) (2) fuse nontrivial dependence patterns (Section 9.3), and (3) fuse computations for different data structures like trees, graphs, and databases (Section 9.5). In our evaluation, we are able to liberally reuse functions from libraries across multiple domains (Intel OneDNN, several BLAS implementations, GEOS, and TileDB), including those to which we have no source code access (ArmPL and Apple's Accelerate library), and even implementations from existing compilers (Halide and ISPC). We also release Fern as open-source software (https://fern-lang.com/).

2 Motivating Example and Workflow

Consider a user who wants to compute a General Vector Rank Update (GER) operation expressed as $A = \alpha(x \times y^T) + A$, where A is a matrix, x and y are vectors, α is a scalar, and all datatypes are floats. GER is a basic building block of linear algebra, and the BLAS [28] specification includes an interface (**`cblas_sger`**) for it.

The most performant BLAS implementations on Apple's M3 chip that uses the AArch64 instruction set reside in two proprietary libraries: the ARM Performance Library (ArmPL) [30] and Apple's Accelerate Framework [22]. Although proprietary, both implementations are known to take advantage of classic optimizations. They use SIMD instructions to parallelize computation, tiling to promote data-cache reuse, special microkernels to increase instruction-cache hit rate, and software pipelining to hide latency. Each implementation, even for a simple GER calculation, likely consists of hundreds of lines of intricate, architecture-specific code. Our user can directly use these implementations, and both are competitive in performance (Figure 1).

On the other hand, if the user wants to compute a variant of GER, called GERB, where A is scaled by an additional parameter β (i.e. $A = \alpha(x \times y^T) + \beta A$), then the user has two possible options. First, they can use ArmPL's fused BLAS extension interfaces that include an **`sgerb_`** subroutine. Or, as Accelerate has no fused subroutine, they can compose Accelerate's **`cblas_sger`** and **`cblas_saxpy`** subroutines. The fused ArmPL subroutine is 1.24× (geomean) faster than the composed Accelerate implementation (Figure 2). Since Accelerate already matched ArmPL's performance on GER (Figure 1), the performance difference between the two implementations on GERB is not due to an under-optimized inner loop or poor instruction selection, but is rather introduced at the point of

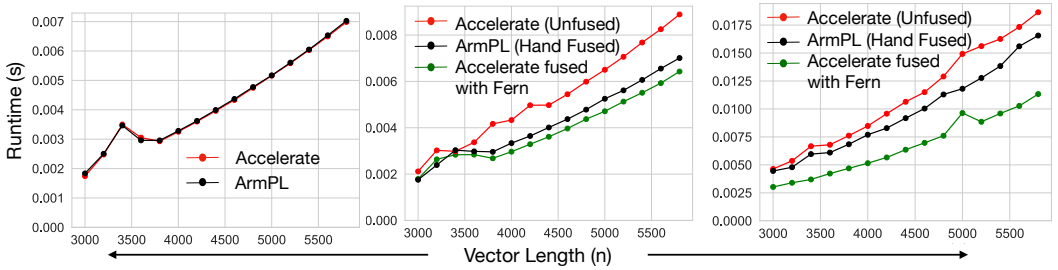


Fig. 1. Performance of **sger** ($A = \alpha(x \times y^T) + A$) as the vector dimension of $|x| = n \times 1$, $|y| = n \times 1$ and $|A| = n \times n$ increases.

Fig. 2. Performance of **sgerb_** ($A = \alpha(x \times y^T) + \beta A$) as the vector dimension of $|x| = n \times 1$, $|y| = n \times 1$ and $|A| = n \times n$ increases.

Fig. 3. Performance of $A = \alpha(x \times y^T) + \beta A + B + C$ as the vector dimension of $x = |n \times 1|$, $y = |n \times 1|$ and $|A| = |B| = |C| = n \times n$ increases.

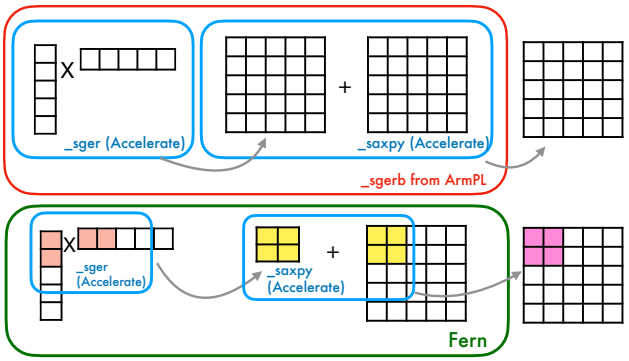


Fig. 4. To compute SGERB ($A = \alpha(x \times y^T) + \beta A$), users can use one subroutine from ArmPL or compose two subroutines from the Accelerate Library. Fern generates code that computes tiles of outputs using the same subroutines from Accelerate.

```
/*<Fern Annotation>
l_u: Symbolic
l_v: Symbolic
for u in [out.x,out.x+out.len_x,l_u]{
 for v in [out.y,out.y+out.len_y,l_v]{
    produces{
     out{ x: u,
          y: v,
          len_x: l_u,
          len_y: l_v }}
    when consumes{
      a { start: u,
          len: l_u },
      b { start: v,
          len: l_v }}}}
</Fern Annotation>*/
void sger(float alpha,const Matrix& a,
  const Matrix& b, Matrix& out)
```

Fig. 5. Data-dependence pattern for the **sger** subroutine added to a header file.

composition itself. By the time **cblas_saxpy** is ready to consume the intermediate result produced by **cblas_sger**, the first element of the intermediate has fallen out of the cache.

Manually recovering performance lost at subroutine boundaries by rewriting code is not only tedious and error-prone, in this case, it is impossible. Both ArmPL and Accelerate are propriety frameworks. In fact, the Accelerate library is the only way to target undocumented, Apple-specific hardware. If our user wants to use this special hardware, they are tied to the Accelerate library, which does not have extended BLAS kernels like **sgerb_**. But, if our user wants to use fused implementations provided in the BLAS extension interface, they must use ArmPL.

Instead of using naive subroutine calls for Accelerate, the user can use code generated by Fern, which continues to call Accelerate subroutines, but does so for small tiles of the output at a time (Figure 4). This not only ensures that the intermediate result is in the cache by the time it is ready to be reused, but also asymptotically decreases memory footprint for intermediate storage, allocating only a single tile's worth of memory as opposed to a large intermediate. Figure 2 shows that Fern recovers the performance lost at the composition boundary.

To use Fern, users write a list of subroutine calls they would like to compose (Figure 6). Each subroutine is a special Fern object, and the **()** operator has been overloaded to mimic subroutine call syntax. The Fern objects have been automatically generated from header files containing data-dependence annotations. Figure 5 shows the data-dependence pattern for SGER.

In the pattern, each sub-array is described using a collection of two metadata fields: the start of the sub-array, **`start`**, and the length of the sub-array, **`len`**. Meanwhile, for a matrix, metadata fields **`x`** and **`y`** point to the beginning of a sub-matrix and **`len_x`** and **`len_y`** describe the length and the width of a sub-matrix respectively. The annotation specifies that to produce a sub-matrix **`out(x=u,y=v,len_x=l_u,len_y=l_v)`**, the subroutine must consume sub-arrays **`a(start=u,len=l_u)`** and **`b(start=v,len=l_v)`**. Finally, the **`for`** loops specify all the subsets the function must compute to generate a complete output. Given a list of subroutine calls and their corresponding annotations, Fern determines the subset relationships that need to be satisfied across subroutine calls, and generates a fused implementation (Figure 4).

Now, assume that the user wants another variant of the above computation that is not available in any library. Consider the computation $A = \alpha(x \times y^T) + \beta A + B + C$ where B and C are matrices. Neither ArmPL nor Accelerate provide a fused implementation for this computation. Fern, however, can fuse this computation and is 1.45× faster than the ArmPL implementation and 1.65× faster than the Accelerate implementation (Figure 3). Fern delivers this performance by continuing to reuse high-performance implementations within these libraries and by simply partitioning data into small chunks that continue to stay resident in the cache.

```
// Data structures x, y, A are symbolic,
// and used to specify the composition.
Matrix A, R; // R is an intermediate.
Vector x, y;
// Saxpy and Sgerb are Fern objects
// generated from .h files.
fern::Saxpy saxpy;
fern::Sger sger;

fern::Composition program {
    sger(α, x, y, R),
    saxpy(R, β, A)};
// Generate fused code (will be called
// with actual data structures).
program.compile(compile_options);
```

Fig. 6. Fern program to fuse SGERB.

3 Design of Fern

Fern's design builds on four key ingredients:

- The data production and consumption patterns of subroutines *do not depend on values of the data*. At compile time, given a subset of the output, it must be possible to compute a symbolic expression for the required subsets of the input. Subroutines must also produce the same output if run on the same set of inputs (*idempotence*). By restricting the type of subroutines allowed in Fern, we are able to manipulate the order of computation without accessing the source code or building heavy-weight analysis machinery.
- An *output-first* view of the computation: Fern understands data-dependence patterns from the perspective of the output. An output-first perspective enables Fern to run a demand-driven analysis [16] to compute subsets of the intermediates that will be ultimately used.
- A *logical* understanding of subsets of a data structure: As opposed to operating on concrete data structures that are owned and managed by Fern, users can bring their choice of data structure that Fern will understand logically through a collection of metadata fields and a set of small interfaces. By introducing a logical interface, Fern is able to fuse computations across domains and for different data structure implementations.
- *Parametrization* of subroutine implementations: In Fern-generated code, the compute subroutines ultimately get called over subsets of data. Therefore, the same subroutine implementation must be able to operate on subsets as on the whole input and output. We exploit the fact that fused code often operates on subsets that are self-similar to the overall data structure to recursively break the problem down into smaller sub-problems for better locality.

Previous work has exploited similar ideas, but Fern is the first to combine all four into a lightweight system for fusing existing black-box subroutines. Halide [39] adopts a functional algorithm language (*idempotence*), but is restricted to dense arrays that it understands how to physically index into. Legion [6] and Sequoia [11] adopt an abstract view of tasks and can operate on irregular data structures (*logical view*), but cannot automatically rearrange the granularities of these tasks since

Variable $v \in$ String, $x \in \mathbb{N}$

⟨dependence⟩	::=	⟨Vardecl⟩ ∗ ⟨Interval⟩
⟨Vardecl⟩	::=	⟨Var⟩ `: Symbolic`
⟨interval⟩	::=	`for` v `in (` ⟨Expr⟩`,`⟨Expr⟩`,`⟨Expr⟩ `){`⟨Fire⟩`}` |
		`for` v `in (` ⟨Expr⟩`,`⟨Expr⟩`,`⟨expr⟩ `){`⟨Interval⟩`}`
⟨fire⟩	::=	`produces {`⟨Subset⟩`} when consumes {[`⟨Subset⟩`] ∗ }`
⟨Subset⟩	::=	⟨Data⟩`{`⟨Fields⟩`}`
⟨Data⟩	::=	⟨String⟩ *(Name of data structure)*
⟨Fields⟩	::=	⟨String⟩ `:` ⟨Expr⟩`[ , `⟨Fields⟩`] ∗`
⟨Expr⟩	::=	x | v | ⟨Data⟩`.`⟨String⟩ |
		⟨Expr⟩`{+,-,∗}`⟨Expr⟩ | ⟨Expr⟩`{%,/}`x

Fig. 7. The syntax of the data-dependence language.

```
1  /*<Fern Annotation>
2  l_u: Symbolic
3  l_v: Symbolic
4  for u in [out.x,out.x+out.len_x,l_u]{
5   for v in [out.y,out.y+out.len_y,l_v]{
6      produces {
7       out { x: u,
8             y: v,
9             len_x: l_u,
10            len_y: l_v }}
11     when consumes {
12       a { start: u,
13           len: l_u},
14       b { start: v,
15           len: l_v }}}}
16 </Fern Annotation>*/
17 void sger(float alpha,const Matrix& a,
18   const Matrix& b, Matrix& out)
```

Fig. 8. Data-dependence description for **sger** described in Section 2.

they lack data-dependence information. Split Annotations [35] exploits *parametrization* and exposes data relationships, but struggles to generalize to computations beyond elementwise, "embarrassingly" fusable patterns. Mosaic [4] can call black-box subroutines (*idempotence*) and automatically generate surrounding code (*parametrization*), but is restricted to einsum-based computations. Fern is inspired from the collective experience of building such systems, and combines aspects of their design in a single system. Section 10 discusses related work in more detail.

Now, we shift our discussion to how these design decisions are reflected in Fern. Sections 4 and 5 concretize the design of the language used to define data consumption and production patterns. Section 6 describes how a Fern program is lowered through an example, Section 7 discusses the semantics of Fern programs, and Section 8 discusses a scheduling language.

4 Data-Dependence Language

Fern converts a list of subroutines, where each subroutine computes its output completely, into code that computes the output subset-by-subset using the same subroutines. To do so, Fern needs to understand the data production and consumption patterns of subroutines. In this section, we describe the design of the data-dependence language that is used to express this relationship. Figure 7 shows the complete grammar of the dependence language.

Users or library authors add data-dependence descriptions to subroutine declarations in header files (Figure 8). A description is written once per subroutine, and the cost of writing it is amortized over multiple uses of the library. As the description is tied to subroutine *declarations* (not definitions), Fern can fuse subroutines without source-code access.

The dependence language describes subsets using metadata fields. For example, to refer to a sub-block of a matrix (Figure 8), we can use four metadata fields **x**, **y**, **len_x** and **len_y**, where **x** and **y** indicate the starting coordinate of the block, and **len_x** and **len_y** denote its height and width. Similarly, access into a graph can be decomposed using two fields, **node** and **ring**, where **node** represents a vertex ID, and **ring** specifies the distance from **node** up to which we want to gather a neighborhood of points.

Users describe dependence relationships by referencing metadata fields. The dependence relationship consists of a **produces** node, and a corresponding list of zero or more **consumes** nodes. Each node is tied to a parent data structure whose subsets it describes. Lines 6-15 in Figure 8 show an example for the **sger** subroutine introduced in Section 2. The underlying data structure is still owned by user code, and Fern understands the data structure logically (i.e., in terms of metadata fields), not physically (i.e., in terms of how the data is laid out in memory). The descriptions are

written from the perspective of the output, answering the the following question: *To produce a given subset of the output, what subsets of the input does the subroutine need?*

The dependence relationship can also be wrapped in intervals that have a start, end, and step parameter. While the **`produces`** and **`consumes`** nodes provide information to successfully fire a subroutine to produce a subset of the output, the intervals enumerate the combination of metadata fields that the subroutine must be called on to produce a complete output. An interval introduces a bound variable that can be used within the description. The extent of the interval can be an output field (e.g., **`out.len_x`**), or integer arguments used in the subroutine interface. The decomposition can use any number of fields and the physical subsets themselves need not be rectangular; however, the choice of subsets must be able to tile the entire output data structure.

```
sger(a, b, out);
```

↓

```
for u in range(out.x, ... , l_u):
  for v in range(out.y, ... , l_v):
    // Query a subset.
    out_q = out.query(u,v,l_u,l_v)
    //  Call with subsets.
    sger(a.query(u, l_u),
         b.query(v, l_v), out_q)
    // Insert the subset.
    out.insert(u,v,l_u,l_v,out_q)
```

Fig. 9. The annotation presented in Figure 8 make the above rewrite legal. The **`query`** and **`insert`** interfaces are described in Section 5.

The description can also use symbolic variables to describe dependencies. In the example shown in Figure 8, the start, end, and step of the interval are all symbolic. Integer arguments can be used within dependence descriptions even if they cannot be statically determined. The annotation lets Fern know what decompositions of the output data structures are legal and result in equivalent programs (Figure 9).

During lowering (Section 6), Fern generates expressions that compute the precise values of these variables at runtime, or, if possible, at compile time. Fern ultimately generates a C++ object from the header files' data-dependence descriptions. As opposed to the preprocessor generating these classes, it is also possible for users to construct this object directly. Since the class can contain arbitrary C++ logic, it is possible to metaprogram the annotations.

5 Abstract Data-Structure Interfaces

In Section 4, we described how Fern understands a subroutine's data production and consumption requirements. Central to this strategy was the decision to describe subsets of data structures using some collection of metadata fields. While Fern can resolve logical dependencies using only these dependence descriptions, during code generation, it needs to query, allocate, insert, and free the corresponding subsets of the *physical* data structures. To enable this, a data structure with type **`Data`** used by a subroutine must support the following interface:

(1) A **`query :: Int`** $\to$ **`Int`** $\ldots$ $\to$ **`Data`** method that takes values of metadata fields and returns an object representing the corresponding subset of the data structure. Depending on the library's data structure implementation, this can either be a fully materialized subset or a lightweight view.
(2) An **`insert :: Int`** $\to$ **`Int`** $\ldots$ $\to$ **`Data`** $\to$ **`()`** method that inserts a subset into an existing data structure at the location referred to by the values of the fields.
(3) An **`allocate :: Int`** $\to$ **`Int`** $\ldots$ $\to$ **`Data`** function that allocates a fresh object of type **`Data`** with the provided values of the fields.
(4) A **`free :: ()`** $\to$ **`()`** method that frees the data structure.
(5) An optional **`free_query :: ()`** $\to$ **`Bool`** method that returns whether the output of a query needs to be freed (assumed **`false`** by default).
(6) An optional **`insert_query :: ()`** $\to$ **`Bool`** method that returns whether the subset needs to be explicitly inserted, or is modeled by a view (assumed **`true`** by default).

6 Lifecycle of a Fern Program

Fern accepts as input a list of subroutine calls and produces as output fused imperative code. The generated code completely evaluates the output of the last subroutine, and computes only the subsets of the intermediates relevant for the final output. Users can optionally control properties of the generated code using a scheduling language (Section 8).

In this section, we describe how a Fern program is lowered to produce a C++ file that contains the fused code. The fused code continues to call the the original subroutines at smaller granularities, letting users take advantage of previously-written high-performance subroutines. After lowering, Fern-generated code can also be auto-tuned to determine the granularity at which the final output should be computed. In practice, this amounts to a straightforward enumeration of possible combinations of tiling parameters.

```
1  // The input and output.
2  annot::Matrix input, output;
3  // All the temporaries.
4  annot::Vector t1, t4;
5  annot::Matrix T2, T3;
6  Variable width(Datatype::Int32);
7  // Annotated functions.
8  annot::MaxRow max_row;
9  annot::SubtractVec sub_vec;
10 annot::Exp2d exp_2d;
11 annot::SumRow sum_row;
12 annot::DivideVec divide_vec;
13
14 fern::Composition program {
15     max_row(input, width, t1),
16     sub_vec(t1, input, T2),
17     exp_2d(T2, T3),
18     sum_row(T3, width, t4),
19     divide_vec(t4, T3, output),
20 };
21 // Options contains compiler flags.
22 program.compile(options);
```

Fig. 10. Fern program for the decomposed softmax operator.

6.1 Our Running Example

To guide readers through the internals of our compiler, we will focus our discussion on a Fern program that computes the softmax function, but decomposes it into smaller operations (Figure 10). The decomposition of the softmax computation itself is novel [14], and, to the best of our knowledge, current compilers — domain-specific or otherwise — cannot discover this decomposition automatically. While Fern cannot generate the optimal decomposition either (it does not even know what the leaf computations are), it is able programmatically generate a fused program *once* the user has specified the decomposition.

The discussion that follows is centered on the softmax program. The appendix presents imperative algorithms for lowering arbitrary Fern programs.

6.2 Understanding the Softmax Program

As Lines 14-20 in Figure 10 show, the input to Fern is a list of "subroutine calls". Each subroutine object (Line 8-12) is an instance of the **`AbstractSubroutine`** class. These classes do not need to be manually written, and can be automatically generated from the data-dependence annotations written in header files. Any subclass that implements the interfaces specified by the **`AbstractSubroutine`** class can operate within the Fern ecosystem.
First, let's look at the data-dependence annotations for the subroutines in the program.

Figure 11 contains the annotation for the **`max_row`** subroutine. **`max_row`** takes as input a matrix and a reduction parameter **`width`**. It then computes a vector that stores the maximum element up to **`width`** per row of the matrix. Therefore, to compute the subset of the output vector that starts at index **`x`** and is of length **`x_tile`**, **`max_row`** requires the sub-matrix of input starting at index **`x, 0`** that has height **`x_tile`** and width **`width`**. The annotation for **`sum_row`** is identical.

The annotation for **`sub_vec`** (Figure 12) is similar. It takes as input a vector and a matrix, and computes a matrix. The subroutine computes a broadcasted subtraction i.e., it subtracts each element in the row of the input matrix by the corresponding element in the input vector. To compute a subset of the output matrix that starts at index **`x, y`** and has width **`x_tile`** and height **`y_tile`**, **`subtract_vec`** requires the sub-matrix of the input starting at **`x, y`** that has width **`x_tile`** and height **`y_tile`**, and the input vector starting at **`x`** and with length **`y_tile`**. The annotation for **`divide_vec`** is identical to that of **`sub_vec`**.

```
/* <Fern Annotation>


x_tile : Symbolic
y_tile : Symbolic
for x in [0, output.len, x_tile]{
produces {
 output { start: x,
          len: x_tile}}
 when consumes {
 input { x: x,
         y: 0
         width: width
         height: x_tile
   }
 }
}
</Fern Annotation> */
void max_row(const Matrix& input,
 const int& width, Vector& output);
```

Fig. 11. Annotation for **max_row**.

```
/* <Fern Annotation>
x_tile : Symbolic
y_tile : Symbolic
for x in [0, output.width, x_tile]{
for y in [0,output.height,y_tile]{
produces {
 output { x: x,
          y: y,
          width: x_tile,
          height: y_tile }}
 when consumes {
 input { x: x,
         y: y,
         width: x_tile,
         height: y_tile },
   v  { start: x,
        len: y_tile }}}}
</Fern Annotation> */
void sub_vec(const Vector& v,const
  Matrix& input, Matrix& output);
```

Fig. 12. Annotation for **sub_vec**.

```
/* <Fern Annotation>
x_tile : Symbolic
y_tile : Symbolic
for x in [0, output.width, x_tile]{
for y in [0,output.height,y_tile]{
produces {
 output { x: x,
          y: y,
          width: x_tile,
          height: y_tile }}
 when consumes {
 input { x: x,
         y: y,
         width: x_tile,
         height: y_tile}
   }
 }}
</Fern Annotation> */
void exp_2d(const Matrix& input,å
     Matrix& output);
```

Fig. 13. Annotation for **exp_2d**.

Finally, Figure 13 presents the annotation for **exp_2d**, an elementwise subroutine. **exp_2d** takes a matrix as input and computes a matrix whose elements are the exponentials of the corresponding input elements. To compute a subset of the output matrix that starts at index **x, y** and has width **x_tile** and height **y_tile**, **exp_2d** requires the sub-matrix of the input starting at **x, y** that has width **x_tile** and height **y_tile**.

The **()** operator of the **AbstractSubroutine** class is overloaded to mimic function call syntax. The **()** operator produces a **ConcreteSubroutineCall** object that captures the arguments of the subroutine call. The **ConcreteSubroutineCall** models a distinct callsite. Since dependencies of distinct callsites need to be solved separately, each time Fern produces a **ConcreteSubroutineCall**, it also mangles the variable names introduced in the data-dependence description, generating a fresh name for each variable.

6.3 Lifting a Computation Graph

From the list of subroutine calls and the implicit dependencies between them, Fern lifts a computation graph. A computation graph is a directed-acyclic graph (DAG) where nodes represent subroutine calls, and an edge indicates a producer-consumer relationship between two subroutines (Figure 14). Each computation graph computes one final output, in particular, the output produced by the last subroutine call. At this stage, a fresh output data structure is used for each subroutine call. Fern will automatically optimize reuse of intermediate allocations at a later step. Because the input is simply an ordered list of subroutine calls with fresh outputs, the computation graph is guaranteed to be a DAG by construction.

A computation graph can contain three types of connections (Figure 14):

- A *straight* connection where one node produces data for one node.
- A *merge* connection where two or more nodes produce data for one node.
- A *fork* connection where one node produces data for two or more nodes.

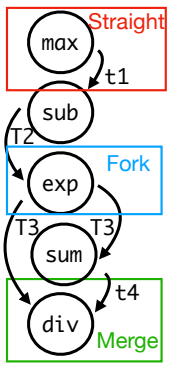


Fig. 14. The computation graph for the softmax program.

6.4 Resolving Dependencies across Call-sites

For all nodes, Fern needs to compute data dependencies that satisfy the callsite. To do so, Fern begins by traversing the computation graph bottom up. In our example, Fern will first visit the **divide_vec** node that consumes data structures **t4** and **T3**. Both **t4** and **T3** are outputs of other

subroutine calls. Let's follow the dependencies of **T3**. Figure 15 shows the dependencies of **T3** as a function of **divide_vec**'s output fields. Next, to compute **T3**, we need to call **exp_2d** on **T2**. Figure 16 shows the relationship between **T2** and **T3**. **T2** itself is an output of **sub_vec**, which consumes **t1**, and dependencies will be chained in a similar manner. **t1**, in turn, is an output of **max_row**, which consumes **input**, a data structure that the user provides as input. When Fern reaches a user-provided data structure, it stops traversing the computation graph.

```
divide_vec.input.x = divide_vec.output.x
divide_vec.input.y = divide_vec.output.y
divide_vec.input.width = divide_vec.output.width
divide_vec.input.height = divide_vec.output.height
```

Fig. 15. Relationship between **T3** (**divide_vec**'s input) and **output** (**divide_vec**'s output)

```
exp_2d.input.x = divide_vec.input.x
exp_2d.input.y = divide_vec.input.y
exp_2d.input.width = divide_vec.input.width
exp_2d.input.height = divide_vec.input.height
```

Fig. 16. Relationship between **T2** (**exp_2d**'s input) and **T3** (**divide_vec**'s input & **exp_2d**'s output).

Lowering straight and merged connections is straightforward using the above strategy. However, lowering fork connections poses a problem. Since multiple subroutines are consuming the output of a single subroutine, the producer subroutine must produce an output that satisfies the dependencies of *all* the consumer subroutines. To resolve fork nodes, Fern replaces the consumer subroutine nodes with a dummy subroutine node that artificially represents the superset of the consumer subroutines' requirements. The dummy subroutine is used to resolve subset requirements for the producer subroutine. Fern generates queries on the produced superset for individual consumer functions and passes the queried subset to corresponding consumer functions. In our example, **exp_2d** is replaced with a dummy subroutine that produces output such that **exp_2d.output.x = min(divide_vec.input.x, sum_row.input.x)**, and **exp_2d.output.width = max(divide_vec.input.x + divide_vec.input.width, sum_row.input.x + sum_row.input.width)**. **exp_2d.output.y** and **exp_2d.output.height** are computed similarly.

In this way, for each producer-consumer pair, Fern solves an equation that requires that the metadata fields of the producer subroutine satisfy the requirements of the consumer subroutine.

6.5 Generating Pipelines

After computing dependencies for each call site, Fern generates a *pipeline*. A pipeline is a recursive data structure that consists of an ordered list of compute, query, insert, allocate, free or other pipeline nodes; an ordered list of intervals; and a list of relationships between metadata fields of inputs and outputs of different subroutines. The pipeline for softmax is shown in Figure 17.

Query nodes request subsets of data structures (Lines 12, 16). For inputs that are intermediate results of another computation, Fern emits an allocate node instead of a query node (Lines 15, 21, 24, 27). Each allocated or queried subset has an associated free node (Lines 33–36). A compute node is then generated for the subroutine call (Lines 19, 22, 25, 28, 30), and if the output data structure requires it, an insert node is added. Finally, Fern iterates over the intervals of the final output and emits intervals corresponding to the final output's domain (Lines 3, 4).

Users can optionally control parameters of the pipeline, such as the tiling strategy, loop order, etc. using the scheduling language described in Section 8.

6.6 Optimizing Pipelines

Two optimizations are applied on pipelines before generating the final C++ code. First, if an allocation nested inside a loop does not depend on the loop variable and and the loop is not

```
1  // ------------------------------------------
2  //  Pipeline nodes for the softmax computation.
3  for ((_gern_x_15_22_33=0); output_final.row; l_x ){
4    for ((_gern_y_16_26=0); output_final.col; l_y){
5      // Definitions of metadata fields.
6      (_gern_y_16=_gern_y_16_26)
7      (_gern_l_y_14=l_y)
8      (_gern_x_15=_gern_x_15_22_33)
9      (_gern_l_x_13=l_x)
10
11     // Query node for user-provided data structures.
12     Query _query_output_final_40 from output_final
13         with {_gern_x_15,_gern_y_16, ...}
14     // Allocate intermediate data structures.
15     Allocate rowMax with {_gern_x_2,_gern_l_x_1}
16     Query _query_input_41 from input with ...
17
18     // Now, compute the intermediate result.
19     Compute max_row {_query_input_41,rowMax,width}
20
21     Allocate subDS with ...
22     Compute sub_vec {rowMax,_query_input_41,subDS}
23
24     Allocate expDS with ...
25     Compute exp_2d {subDS,expDS}
26
27     Allocate rowSum with ...
28     Compute sum_row {expDS,rowSum,width}
29
30     Compute divide_vec {..,_query_output_final_40}
31
32     // Free allocations.
33     Free rowSum
34     Free rowMax
35     Free expDS
36     Free subDS}}
```

Fig. 17. Pipeline nodes for the softmax computation.

```
void function_39(Matrix &input, Matrix &output_final,
        int64_t l_x, int64_t l_y, int32_t width) {
  for (_gern_x_15_22_33 = 0;  _gern_x_15_22_33 < ....){
    for (_gern_y_16_26 = 0;  _gern_y_16_26 < ....){
      // Definitions of metadata fields.
      int64_t _gern_y_16 = _gern_y_16_26;
      int64_t _gern_l_y_14 = l_y;
      int64_t _gern_x_15 = _gern_x_15_22_33;
      int64_t _gern_l_x_13 = l_x;

      // Query using the provided interface.
      auto _query_output_final_40 = output_final.query(
        _gern_x_15, _gern_y_16, _gern_l_x_13, ..);
      // Allocations for intermediates.
      Array rowMax = Array::allocate(_gern_x_2, ...);
      auto _query_input_41 = input.query(...);

      // Call the compute function.
      max_row(_query_input_41, rowMax, width);

      Matrix subDS = Matrix::allocate(...);
      sub_vec(rowMax,_query_input_41,subDS);

      Matrix expDS = Matrix::allocate(...);
      exp_2d(subDS, expDS);

      Array rowSum = Array::allocate(...);
      sum_row(expDS, rowSum, width);

      divide_vec(rowSum ..., _query_output_final_40);

      // Free allocations.
      rowSum.destroy();
      rowMax.destroy();
      expDS.destroy();
      subDS.destroy();}}}
```

Fig. 18. Generated C++ for the softmax computation.

computed in parallel, then the allocation is hoisted outside the loop interval. Second, to reduce the amount of memory allocations, intermediates with the same type and metadata values can be reused across different subroutine calls once they are done being used as inputs. After generating the complete pipeline, Fern optimizes for the reuse of allocated intermediates using liveness analysis.

6.7 Example of Generated Code

Once a pipeline has been generated, Fern is ready to lower it into imperative code (Figure 18). As mentioned in Section 6.5, pipelines contain query, compute, free, allocate or other pipeline nodes. Additionally, they contain intervals over the final output's domain, and a list of metadata relationships. For a pipeline, Fern first lowers the interval nodes into **for** loops (Lines 3, 4), and then proceedes to generate the list of dependence relationships nested inside the loop. It then recursively lowers each node, generating calls to allocate (Lines 15, 21, 24, 27), query (Lines 12, 16), compute (Lines 19, 22, 25, 28, 30), or free (Lines 33–36) functions, or nesting another pipeline inside the original pipeline when lowering a pipeline node. Finally, Fern wraps this code in a function interface (Line 1), which users can use in existing applications. In this way, Fern pieces together pre-compiled high-performance code without understanding their implementations.

In the example shown above, while most of the metadata relationships have been completely resolved, the tiling parameters, **l_x** and **l_y** (Line 2), have not been set. Fern exposes these unresolved variables in the function interface as well. Users can set values for these variables, and since the equations were solved symbolically, the function implementation is already parametric over different choices of the symbolic variable. These variables can also be exposed as template

parameters, retaining flexibility while taking advantage of compile-time specialization. As a simplication, Fern assumes that the loop step divides the loop extent exactly. We expect users to pad data structures to handle boundary conditions. This simplication is not fundamental to our design.

While we have shown a C++ function, our approach is language independent. In addition to C++, Fern can generate Python code. The code generator can be extended further to another backend.

7 Denotational Semantics

A Fern program is a list of subroutines $f_1, \ldots, f_n$ invoked using the `()` operator with corresponding data-dependence patterns $p_1, \ldots, p_n$. Each subroutine f_i is an object of type **`AbstractSubroutine`**, and each annotation p_i is of type **`Annotation`**. Each subroutine call produces a distinct output $\textbf{out}_i$ of type **`Data`** which supports the required interfaces (Section 5).

```
class Annotation:
 -- List of [start, end, step]
 -- of the annotation's intervals.
 range :: List[[Expr,Expr,Expr]]
 -- Variables that index the intervals.
 loop_var :: List[Variable]
 -- Fields used to point at the subset of
 -- a data structure in the annotation.
 fields :: Data -> List[Expr]
```

```
class AbstractSubroutine:
   -- The data-dependence pattern.
   annot :: Annotation
   -- The C++ function signature
   -- of the asbtract function.
   -- (name, types of parameters,
   -- templated parameters, output
   -- type).
   fs :: FunctionSignature
```

```
class
↪  ConcreteSubroutineCall:
   -- Annotation specialized
   -- to the call.
   annot :: Annotation
    -- Call generated from
    -- AbstractSubroutine's
    -- function signature.
   fc :: FunctionCall
```

The `() :: AbstractSubroutine → a → b ... → ConcreteSubroutineCall` operator invokes an **`AbstractSubroutine`** with a polymorphic list of arguments to produce an object representing a subroutine call of type **`ConcreteSubroutineCall`**. **`ConcreteSubroutineCall`** models a distinct callsite of an **`AbstractSubroutine`**. A Fern program is, therefore, just a list of **`ConcreteSubroutineCall`** objects c_i that produce output out_i.

$$\{\underbrace{f_1(\ldots, out_1)}_{c_1}, \ldots, \underbrace{f_n(\ldots, out_n)}_{c_n}\}$$

We can now discuss the semantics of Fern programs. For the semantics, we maintain some state denoted by Γ and Φ. Γ `:: Data →` **`ConcreteSubroutineCall`** maps each out_i to the **`ConcreteSubroutineCall`** in charge of producing it. Φ `:: Variable →` **`Expr`**, on the other hand, tracks the implicit relationships between different metadata fields in the program.

 # 1. Applying the top-level rule: Fern commits to computing the *final* output in the program. Therefore, the denotation of the program is just the denotation of the *last* concrete call:

$$[\![\{f_1(\ldots, out_1), \ldots, f_n(\ldots, out_n)\}]\!]_{\Gamma,\Phi} = [\![f_n(\ldots, out_n)]\!]_{\Gamma[out_i \to f_i(..)]_{0 \le i \le n-1}, \Phi}$$

Notice how we do not insert the final output out_n into Γ.

 # 2. Computing at smaller granularities: Fern fuses its calls i.e. it calls a C++ sub-routine at smaller granularities. We use the annotation to rewrite the program, and compute the function at smaller subsets (this is the same transformation we applied in Figure 9).

$$[\![\ \underbrace{f(\ldots, out)}_{\text{or, the } \textbf{ConcreteSubroutineCall } c}\ ]\!]_{\Gamma,\Phi} = \textit{for}\ \underbrace{i_0}_{\text{Fresh Variable}}\ \textit{in range } c.annot.range[0]:$$

$$\ldots$$

$$\textit{for } i_p \textit{ in range } c.annot.range[p]: \quad (\textit{where } p = len(c.annot.range) - 1)$$

$$out' = [\![out.query(c.annot.fields(out))]\!]_{\Gamma,\Phi}$$

$$[\![c.fc]\!]_{\Gamma,\Phi[c.annot.loop\_vars[m] \to i_m]_{\forall 0 \le m < len(c.annot.range)}}$$

$$[\![out.insert(c.annot.fields(out), out')]\!]_{\Gamma,\Phi}$$

 # 3. Generating a call: To actually call a function fc of a **`ConcreteSubroutineCall`** c, we need to make sure that all fc's inputs and outputs are ready. In the rule below, the notation $data\{f_1, \ldots, f_n\}$ describes a subset of $data$ indexed by metadata fields $f_1, \ldots, f_n$.

$$
\begin{aligned}
[\![c.fc(arg_1, \ldots, out)]\!]_{\Gamma,\Phi} = {} & arg_1' = [\![arg\_1\{c.annot.fields(arg_1)\}]\!]_{\Gamma,\Phi} && \textit{(Prepare } arg\_1\textit{'s the required subset)} \\
& \ldots \\
& out' = [\![allocate(c.annot.fields(out))]\!]_{\Gamma,\Phi} && \textit{(Allocate the output of the call.)} \\
& fc(arg_1', \ldots, out'); && \textit{(Generate the actual C++ function call)}
\end{aligned}
$$

 # 4. Recursively preparing the data structures: To prepare a subset of data structure $data$ with some associated metadata fields $m_1, \ldots, m_p$:

$$
[\![data\{m_1, \ldots, m_p\}]\!]_{\Gamma,\Phi} =
\begin{cases}
[\![data.query(m_1, \ldots, m_p)]\!]_{\Gamma,\Phi} & \text{if } data \notin \Gamma \\
[\![g.fc]\!]_{\Gamma,\Phi[v \to \textit{solve for } v \text{ in } \text{“}g.annot.fields(data)[j] = m\_j\text{”}]_{v \in V_j, 1 \le j \le p}} & \text{if } data \in \Gamma \text{ with } g = \Gamma[data] \\
 & \text{where } V_j \text{ are non-free variables used in} \\
 & g.annot.fields(data)[j]
\end{cases}
$$

Base Case: If the data is not the output of another call, the user is required to pass it as input. In this case, we simply query the data structure.

Recursive Case: When the data is produced by another call, the call is evaluated first, and Rule #3 is applied again. Before evaluating the call, we record the fact that the call is expected to produce the subset indexed by $\{m_1, \ldots, m_j\}$ by storing the implied relationship in Φ.

 # 5. Generating insert, query and allocate nodes: For the calls described to $query$, $insert$ and $allocate$, we simply call the user-defined function for the datatype (Section 5).

$$
\begin{aligned}
[\![data.query([m_1, \ldots, m_p])]\!] &= data.\texttt{query}([\![m_1]\!]_{\Gamma,\Phi}, \ldots [\![m_p]\!]_{\Gamma,\Phi}); && \textit{(User-defined query function)} \\
[\![allocate([m_1, \ldots, m_p])]\!] &= \texttt{allocate}([\![m_1]\!]_{\Gamma,\Phi}, \ldots [\![m_p]\!]_{\Gamma,\Phi}); && \textit{(User-defined allocate function)} \\
[\![data.insert([m_1, \ldots, m_p], data')]\!] &= data.\texttt{insert}([\![m_1]\!]_{\Gamma,\Phi}, \ldots, data'); && \textit{(User-defined insert function)}
\end{aligned}
$$

For each $query$ and $allocate$ call, Fern maintains a corresponding list of data structures that need to be freed, a detail we elide in the presented semantics.

 # 6. Generating expressions:

$$
\begin{aligned}
[\![e_1 + e_2]\!]_{\Gamma,\Phi} &= [\![e_1]\!]_{\Gamma,\Phi} + [\![e_2]\!]_{\Gamma,\Phi} \\
[\![e_1 - e_2]\!]_{\Gamma,\Phi} &= [\![e_1]\!]_{\Gamma,\Phi} - [\![e_2]\!]_{\Gamma,\Phi} \\
[\![e_1 * e_2]\!]_{\Gamma,\Phi} &= [\![e_1]\!]_{\Gamma,\Phi} * [\![e_2]\!]_{\Gamma,\Phi} \\
[\![e_1 / e_2]\!]_{\Gamma,\Phi} &= [\![e_1]\!]_{\Gamma,\Phi} / [\![e_2]\!]_{\Gamma,\Phi} \\
[\![e_1 \% e_2]\!]_{\Gamma,\Phi} &= [\![e_1]\!]_{\Gamma,\Phi} \% [\![e_2]\!]_{\Gamma,\Phi} \\
[\![v]\!]_{\Gamma,\Phi} &= \begin{cases} [\![\Phi[v]]\!]_{\Gamma,\Phi} & \text{if } v \in \Phi \\ v & \text{Otherwise} \end{cases} \\
[\![\mathbb{N}]\!]_{\Gamma,\Phi} &= \mathsf{N}
\end{aligned}
$$

To evaluate an expression, we first compute the values of its variables and apply the appropriate operators. To determine the value of a variable v, we check whether v has a previously defined value in Φ. If v is not in Φ, it is treated as a free variable whose value must be provided by the user.

Table 1. Scheduling Commands to control pipeline construction in Fern.

Scheduling Commands	Description
`s.parallelize(i)`	Parallelize the interval at nesting level `i`.
`s.reorder(i, j)`	Reorder intervals at nesting level `i` and nesting level `j`.
`s.split(i,x_o,x_i,s_i,s_o)`	Stripmines loop at nesting level `i` by inner and outer iteration variables `x_i` and `x_o`, and inner and outer step variables `s_i` and `s_o`.
`s.substitute(x, val)`	Bind the value of a free variable to concrete value `val`.
`s.reuse(ds, i)`	Reuses values in data structure `ds` (if there is overlap) across iterations of loop at nesting level `i`. Default behavior recomputes values.
`s.break(i)`	Unfuse the pipeline at subroutine at index i and produce the full intermediate.
`s.subpipeline(start,end)`	Compute subroutines between index start and index end at a finer granularity.

8 Scheduling Language

For a fixed computation, there exists a space of fusion strategies. For example, a simple elementwise computation may be fused at the granularity of a single output element or at vector-lane width, exposing parallelism. The best performing strategy depends on the arithmetic intensity of the computation, sizes of the caches, parallelism, and the availability of hardware accelerators.

Modern domain-specific languages like Halide [37], TACO [12], and TVM [10] as well as loop transformation frameworks like CHiLL [9] use scheduling languages to let users explore possible fusion strategies. Scheduling languages provide a set of correct and compositional rewrites that let users sample the space of choices and apply their decisions in a precise and deterministic manner. Scheduling languages often include commands analagous to classical loop optimizations like loop permutation, loop tiling, vectorization and parallelization. More interesting commands like Halide's **`compute_at`** command describe the granularity at which data is to be produced for a consumer.

To control properties of the fused code, we provide a small scheduling language for Fern (Table 1). The design of **`parallelize`** (parallel-for loop using Cilk [42]), **`split`** (loop tiling) and **`reorder`** (loop permutation) follow the tradition of existing scheduling languages. The **`substitute`** command replaces a symbolic variable with a concrete value. During lowering, the value is constant propagated and can help resolve dependencies statically. The **`reuse`** command helps control sliding-window style optimizations by reusing overlapping values over consecutive iterations of some domain. Meanwhile, **`break`** helps unfuse pipelines, and **`subpipeline`** can fuse a subset of subroutines at a finer granularity. In this section, we discuss the **`reuse`**, **`break`** and **`subpipeline`** commands.

Reuse: The **`reuse(ds,i)`** command when applied on a pipeline **`P`** produces a new pipeline **`P'`** that reuses overlapping intermediate values stored in a data structure **`ds`** across iterations of the loop at level i. To apply a reuse command on a data structure, the pipeline **`P`** must satisfy three conditions:

(1) The data structure **`ds`** is an intermediate (not an input or the final output), which is computed in the scope of loop i.
(2) The storage of **`ds`** should not depend on the variable bound by loop i.
(3) Loop i has not been marked as parallel by another scheduling command.

To apply the reuse optimization, Fern hoists the data structure outside loop i, and computes the complete data structure for iteration 0 outside the loop. Fern computes the intersection between different loop iterations of loop i. For the values that do not intersect, Fern generates a fused pipeline, and the output is computed as normal. Finally, a special epilogue is added. Fern generates a query into values computed at the current iteration that overlap with the next iteration, inserting them into the reused data structure.

Break: The break command `P' = P.break(i)` when applied to a pipeline `P`, produces a new pipeline `P'`, where the intermediate produced by the subroutine at index `i` is computed entirely. In this way, the break command unfuses computations. To apply this command, Fern treats subroutines in the pipeline until index `i-1` and the subroutines starting from index `i` as distinct pipelines (and mangles variable names). The lowered pipeline for the first set of subroutines contains an extra allocation for the intermediate of subroutine `i-1`, and both graphs are lowered as usual.

Subpipeline: The subpipeline command `P' = P.subpipeline(i, j)` when applied to a pipeline `P`, produces a new pipeline `P'`, where the intermediates produced by the subroutine at index `i` uptil (but not including) index `j` are fused at a finer granularity. In this way, the subpipeline command introduces a layer of nesting in the generated code. To accomplish subpipelining, Fern lowers the pipeline as normal. And, Fern also lowers the computation for subroutines at index `i` to `j` separately with fresh variable names. Finally, the computation for subroutines at index `i` to `j` is replaced by a pipeline node for subroutines at index `i` to `j`.

9 Evaluation

We evaluate Fern on a range of workloads to show that it can compete with state-of-the-art libraries and compilers that have built-in fusion technology (Section 9.2, Section 9.3), exploit nontrivial fusion patterns (Section 9.3, Section 9.4), exploit fusion patterns that current libraries cannot (Section 9.4, Section 9.5, Section 2), and capture a variety of data structures across domains (Section 9.5). Section 9.7 discusses the impact of different fusion choices on cache utilization. We also show that Fern's end-to-end runtime is small (Section 9.8). Finally, we characterize the overhead of fusing through library interfaces using a synthetically constructed "worst-case" example (Section 9.6).

We extensively reused existing high-performance implementations as microkernels, showcasing that Fern promotes reuse of existing code (Section 9.9).

9.1 Methodology

Experiments are run on an Intel machine and an Apple machine. We use two machines to ensure that our baselines—such as Intel's oneDNN library [33] and Apple's Accelerate library [22]—are evaluated on hardware they were designed for. The Intel machine has an Intel(R) Core(TM) i5-8400 CPU with a 192 KiB instruction cache, 192 KiB L1 data cache, 1.5 MiB L2 cache, 9 MiB L3 cache, and 24.0 GiB of RAM. It is clocked at 2.80 GHz and runs Ubuntu 20.04.6 LTS. The CPU supports the AVX2 [24] vector extension. The Apple machine is a 2020 MacBook Air with an Apple M1 chip. It has four high-performance cores, each with a 192 KiB instruction cache and a 128 KiB L1 data cache, a shared 12 MiB L2 cache, an 8 MiB system-level cache, and 8.0 GiB of RAM. The cores are clocked at 3.23 GHz.

All code is compiled with `-O3 -ffast-math`, and target specific optimizations are enabled when applicable. We use the average of the minimum running time sampled 5 times across 10 iterations. Aggregate speedup refers to the geometric mean of the speedup across different data points.

9.2 Comparison with State-of-the-Art Systems

In the section, we show that Fern can deliver performance that is competitive with state-of-the-art, domain-specific systems and libraries that have built-in fusion machinery.

Experiment 1: (Intel Machine) The first experiment is a convolutional layer which consists of a multi-channel convolution, an addition with a bias vector, and a rectified linear unit (ReLU).

The Intel oneAPI Deep Neural Network Library [33] (oneDNN) is a highly optimized neural network library that implements the convolutional layer as a fused operator. To fuse operators, oneDNN provides two choices. First, the interface of the convolution function is set up such that

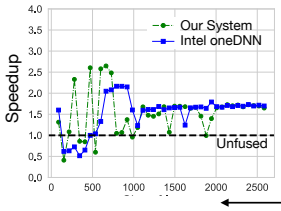


Fig. 19. Speedup of Fern & Intel oneDNN over unfused oneDNN for a convolutional layer as the image size (n × n) increases.

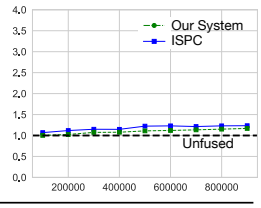


Fig. 20. Speedup of Fern & ISPC over unfused (but vectorized) ISPC for Haversine as the array size (n) increases.

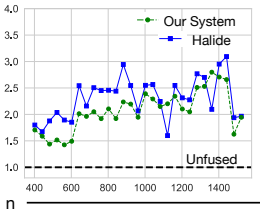


Fig. 21. Speedup of Fern & Halide over unfused (but vectorized & parralel) Blur as image size (n × n) increases.

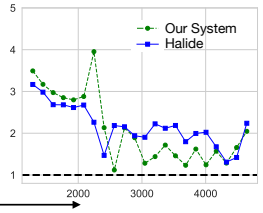


Fig. 22. Speedup of Fern & Halide over unfused (but vectorized & parralel) Unsharp Mask v.s. image size (n×n).

it takes a bias vector and an optional epilogue operator as a parameter. Second, oneDNN offers a graph extension where users describe their computation as a graph that then gets just-in-time compiled, and is pattern matched for a fixed set of fusion patterns.

For Intel's performance numbers, we use the fused API interface. For the Fern implementation, we implement and annotate interfaces for convolution, bias and ReLU. The convolution interface reuses the oneDNN implementation under the hood. Fern is competitive with oneDNN, and achieves oneDNN's performance (1% slowdown on aggregate), while being 1.44× faster than the unfused implementation (Figure 19).

For both the API and the graph extension, the fusion capability of oneDNN is limited and brittle. For the interface, the choice of the epilogue is only limited to elementwise and binary operators. And, for the graph extension, not only does the user need to construct their computation as a graph, but if their computation lies outside the set of fixed fusion patterns that oneDNN can match on, the user is out of luck. In Section 9.3, we show the fusion of the convolutional layer with a maxpool layer, an operation that lies outside the scope of oneDNN's capability.

Experiment 2: (Intel Machine) The Haversine formula calculates the great-circle distance between two points on a sphere. It comprises of 17 basic operations like addition, subtraction, square root, sin, cosin, etc., and is ripe with opportunity for vectorization. Fusion at the granularity of multiples of vector-lane width can be fruitful for such elementwise computations.

We use the ISPC compiler [25] from Intel as the state-of-the-art comparison. ISPC specializes in single program, multiple data (SPMD) workloads, and excels at mapping such programs to single instruction, multiple data (SIMD) instructions.

We also implemented a library of vector functions that compute additions, sin, cosin, etc. over entire arrays in ISPC. We wrote the Haversine computation as a composition of these functions, and directed Fern to fuse exactly at vector-width granularity. We also enable the inlining of the ISPC functions through LLVM to mitigate the overhead of function calls. The arithmetic intensity of the benchmark also hides some of the overhead of making function calls. (Section 9.6 characterizes this overhead). Fern closely follows ISPC's performance, lies within 8% the ISPC implementation on aggregate (Figure 20), and provides a speedup of 1.09× over the unfused implementation.

Experiment 3: (Same as Section 2) Figure 2 presented in Section 2 shows how our system can achieve performance matching ArmPL's fused implementation for the **`sgerb_`** kernel.

Experiments with other state-of-the-art systems are also presented in Section 9.3.

9.3 Exploiting Diverse Fusion Patterns

In this section, we demonstrate that Fern can programmatically exploit nontrivial fusion patterns. Further, we show that Fern can achieve performance similar to a high-performance, domain-specific language (DSL) that offers built-in fusion capabilities for such patterns.

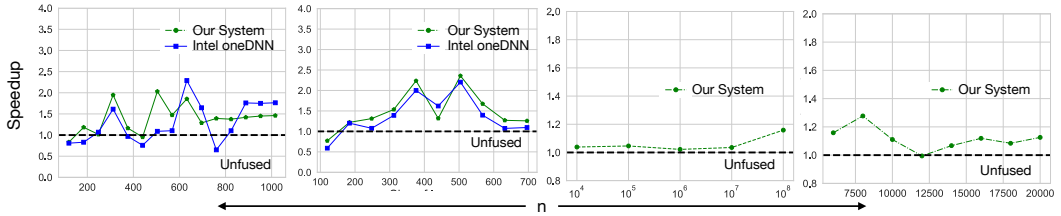


Fig. 23. Speedup of Fern & Intel oneDNN over unfused oneDNN for a Convolutional layer → Maxpool as the image size (n × n) increases.

Fig. 24. Speedup of Fern & Intel oneDNN over unfused oneDNN for a Conv. → Maxpool → Tanh as the image size (n × n) increases.

Fig. 25. Speedup of Fern over unfused GEOS code used by Shapely to query geometries as the # nodes in the tree (n) increase.

Fig. 26. Speedup of Fern (fused database reads and processing) over unfused processing as the size of the input ($n \times n$) increases.

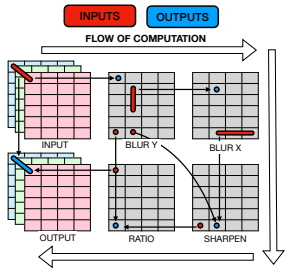


Fig. 27. Unsharp Mask's dependence graph.

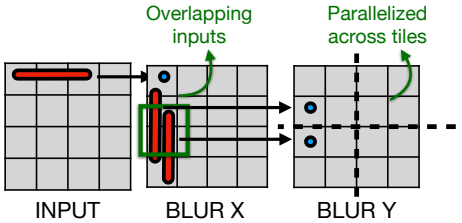


Fig. 28. Overlapping inputs for 2D Blur.

We consider two image processing benchmarks: 2D blur and unsharp mask, and compare the performance of our system to Halide [37]. Halide is a high-performance, image-processing DSL that enables users to fuse their computation through a scheduling command. Both benchmarks have corresponding schedules written by experts.

Experiment 4: (Apple Machine) The 2D blur benchmark consists of a 3×3 separable box filter that is applied on the x-axis and y-axis of a 2D image. The most performant schedule for a 2D blur requires a sliding window and tiling optimization, where tiles of the output are computed in parallel, each tile is fused such that enough output is produced to compute one row of y-axis blur at a time, and the intermediates produced by the x-axis blur are reused across different computations of the y-axis blur as shown in Figure 28.

Fern can tile, parallelize and identify the opportunity for reuse to produce an implementation that is competitive with Halide. Fern fuses vectorized Blur X and Blur Y functions, similar to those generated by Halide, achieving 0.88× of Halide's performance, and is 2.53× faster than the unfused implementation (Figure 21). The difference between Halide's and Fern's performance is due to the fact that Halide is a DSL that owns the storage of underlying data structures. Therefore, Halide can use a ring buffer to store the x-axis blur intermediates that will be used in later iterations. Since Fern does not own the data structures related to the computation, it cannot perform a similar ring buffer optimization, and needs to perform an extra copy.

Experiment 5: (Apple Machine) The unsharp mask computation not only involves the 2D blur computation, but also has multiple intermediates that are fed into multiple subroutines (Figure 27). Fern is competitive with Halide, achieving 0.91× of its performance, and Fern is 1.94× faster than the unfused (but parallelized and vectorized) implementation (Figure 22). The Halide schedule parallelizes, vectorizes and performs a sliding-window optimization. Similar to the previous experiment, Fern needs to perform an extra copy.

9.4 Extending Existing Fusion Machinery

In this section, we show that Fern can add to existing fusion capabilities of libraries by generating fused implementations of computations that have no corresponding equivalent within the library. For the following set of experiments, we will consider the oneDNN [33] library, an industry-standard, high-performance library optimized for almost a decade. Our technique can find and exploit performance opportunities in this highly-optimized library.

Experiment 6: (Intel Machine) In Section 9.2, we compared the performance of oneDNN's fused convolution layer to Fern's as oneDNN provides an interface for a fused covolutional layer. However, a convolutional layer is often followed by a pooling operation and the oneDNN library does not contain a pattern for convolution-bias-reLU-maxpool.

In Fern, fusing convolution-bias-reLU-maxpool is as simple as adding the dependence information for the maxpool subroutine call. We compare Fern's fused implementation to a oneDNN implementation (function composition of fused conv-bias-reLU and maxpool) as shown in Figure 23. Our implementation is 12.6% faster than the oneDNN's implementation, and 34.8% faster than the unfused implementation.

The oneDNN implementation relies on a handwritten fused interface for the convolution-bias-reLU and takes advantage of a memory propagation optimization, where later operands adapt to the data layout of preceding operands [26]. Not only does our implementation provide better data locality, it also does less work, only requesting the convolution output that will eventually be used by the maxpool subroutine. We did not need to write a handwritten fused interface or implement a special memory propagation optimization like the oneDNN library, leading to a much simpler user experience without being domain-specific.

Experiment 7: (Intel Machine) While oneDNN provides an epilogue argument for the convolution interface (Figure 19), it does not do the same for the maxpool operator. Therefore, it cannot fuse convolution-bias-reLU-maxpool-tanh kernel. Fern performs this fusion, resulting in an implementation that is 10.9% faster than the oneDNN implementation, and 42.8% faster than the unfused implementation (Figure 24).

Experiment 8: (Apple Machine) This experiment is the same as Section 2, Figure 3, where we fuse computations across the ArmPL and Accelerate framework, both proprietary libraries. Fern delivers a speed up of 1.45× and 1.65× over ArmPL and Accelerate implementations respectively.

9.5 Adding Fusion to Different Domains

In this section, we show that it is possible to apply the ideas described in this paper to target libraries that use data structures beyond dense n-dimensional arrays.

Experiment 9: (Apple Machine) Shapely [19] is a Python library for geospatial computations. Shapely offers a special data structure called **`STRTree`** that uses the sort-tile-recursive [29] algorithm to insert geometries at construction time. Under the hood, Shapely uses the GEOS library [18], a C/C++ library for computational geometry.

For our benchmark, we consider Shapely's **`query`** function that takes in an **`STRTree`** object, a list of input geometries, and an optional predicate. The function returns a list of geometries that intersect those in the tree and satisfy the optional predicate. This is done within the Shapely codebase as a composition of two GEOS functions: (1) A function that checks for intersection (2) A function that evaluates the predicate. There is no fused implementation to perform both operations. We change the first function's implementation slightly: in the case that it does not find a valid intersection, the function returns an **`None`** type object. This is necessary so that we can define a symbolic expression to compute the size of the input.

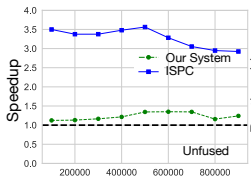


Fig. 29. Speedup of Fern & ISPC over unfused (but vectorized) adversarial example as the size of the array (n) increases.

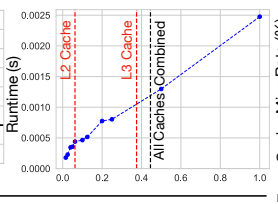


Fig. 30. Runtime of Fern as the fusion granularity ($n \times 10^6$) increases.

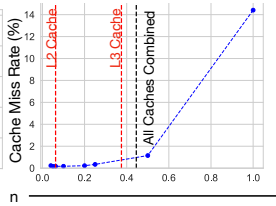


Fig. 31. Cache miss rate (%) of Fern as the fusion granularity ($n \times 10^6$) increases.

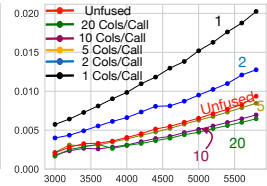


Fig. 32. Runtime of the SGERB program as the tile size changes.

We extend the **STRTree** type with a data query interface and a specialized insertion function that recurses on the newly inserted node. Additionally, we enrich the query and predicate functions to specify that they operate on one node at a time. Fern fuses these computations and generates a C++ interface compatible with libraries such as Shapely. As shown in Figure 25, the fused code achieves speedups of up to 1.15× compared to naive function composition.

Experiment 10: (Apple Machine) Consider the case where some processing is supposed to be performed on data stored in a database. For example, a sigmoid needs to be applied to every entry of a matrix stored in a database (Figure 26).

We use TileDB [44] as our database backend. TileDB is a database management system designed to handle large-scale, multi-dimensional arrays. We use TileDB as an in-memory database, and query TileDB to produce tiles of data at a time.

Fusing the database reads and the scientific computation results in a 1.11× speedup over the unfused implementation. There are more performance opportunities left to be exploited at this boundary. In our implementation, we make repeated queries on tiles of the data and pay the penalty of query parsing and interpretation each time. A more natural streaming interface may result in even better performance. [1]

9.6 Measuring the Overhead of Fern's Design

Since our approach aims to fuse black-box subroutines, there is overhead related with making subroutine calls when compared to handwritten fused code that can simply inline computations. Additionally, since individual fused computations are separated by subroutines, our system cannot fuse at register granularity and cannot expose reuse at that fine-grained a level. In this section, we characterize the overhead of our system by constructing an adversarial "worst-case" example.

Experiment 11: (Intel Machine) For the worst case, we consider a simple, six-operand element-wise addition on vectors. Similar to Section 9.2, we use an ISPC handwritten fused implementation as the state-of-the-art. Since the arithmetic intensity of the benchmark is quite low and the ISPC compiler can fuse at the register level, we expect this benchmark to be the "worst case" and the overheads to dominate the runtime. We also explicitly disable LLVM's inlining pass to prevent optimizations across subroutines. Our system runs 2.6× slower than ISPC (Figure 29), but delivers 1.27× speedup over the unfused case.

[1] Note that operator fusion is beneficial only in the case that the data is stored in memory. If the data is requested over the network, then the network calls become the bottleneck, and benefits seen from operator fusion (which promotes cache reuse) become negligible.

9.7 Impact of Fusion Choices

So far, we have compared fused and unfused code. However, there are several choices available for the granularity at which to fuse. In this section, we compare the impact of fusion granularity on program runtime, cache behavior, and memory footprint. (Figures 30 to 32, and Table 2).

Experiment 12: (Intel Machine) We ran a three-operand elementwise addition, changing the size of the working set (computed by $n * 6$ due to three inputs, one output and two intermediates). At the extreme, the working set size is set to the size of the full array, modeling the unfused case. The increase in runtime is closely related to the increased cache miss rate, giving us confidence that locality benefits correspond to better performance. Table 2 shows how the memory consumption of the program increases with the tile size. The memory allocation of the program scales with the size of the intermediates (the input and output are allocated by the user before calling the Fern function), leading to an asymptotic decrease in memory allocation.

Table 2. Working set size and memory allocations as fusion granularity (tile size) increases.

Tile Size	Working Set Size (Bytes)
1000000	24 MB
500000	12 MB
L1 Cache	9 MB
250000	6 MB
200000	4.8 MB
100000	2.4 MB
L2 Cache	1.5 MB
62500	1.2 MB
50000	1.2 MB
40000	960 KB

Experiment 13: (Same as Section 2) For the SGERB benchmark presented in Figure 2, we had to pick a concrete fusion granularity with which to run the benchmark. To do so, we performed an enumerative search over a set of tile sizes. For our search, we explored computing x number of columns per subroutine call, resulting in a tile size of $x \times n$ where n is the length of the column. Smaller tile sizes result in smaller working-set sizes, making it more likely that the data remains resident in cache, but they also lead to more subroutine calls. Since we do not have source code access to the library, the downstream compiler cannot inline the subroutines, and we must pay the overhead of these calls. Larger tile sizes result in fewer subroutine calls, adding less overhead, but increase the working-set sizes. Figure 32 shows how the runtime of the program changed with different tile sizes.

9.8 Scaling of Analysis

Fern's primary task is to reason about dependence patterns over a chain of subroutine calls, and the majority of compile time is spent resolving these patterns. Fern's compile time scales linearly with the length of the program as dependencies need to only be resolved across consecutive pairs of subroutines. Table 3 shows compile times for a subset of applications. The maximum time taken to compile a fused pipeline was 6.91 ms across all experiments in Section 9, making Fern easy to integrate into a rapid edit-compiler-test loop.

Table 3. Fern's compile time while varying # subroutine calls.

Application	# Calls	Time
2D Blur	2	1.34 ms
Conv, Max, Tanh	4	1.63 ms
Haversine	17	6.91 ms

9.9 Simplicity and Reuse of Existing Code

For the experiments presented above, we reused subroutine implementations across several high-performance libraries and DSLs (Halide, oneDNN, BLAS, GEOS, TileDB, ISPC, ArmPL, Accelerate). Since we reused previously existing interfaces, generating fused code with Fern was simpler than handwriting fused code, or building compilers that produce fused code by default.

To apply the approach presented, data-dependence information needs to be added to subroutine interfaces. This information is easy to derive and write down, and needs to be done once per computation. For example, once the data-dependence information for matrix multiplication has been added for Intel MKL's BLAS library, it can be quickly adapted for ArmPL, Accelerate, etc. Further, writing annotations can be automated for high-level languages (like Halide). We also added data querying and insertion interface to data structures that these libraries and systems operate

on. This amounts to only a few hours of work, and had to be done only once per library. The subroutines may also need to be wrapped in newer interfaces that interoperate with the new data structures that have the added querying and insert interfaces.

10 Related Work

Many compilers and languages perform operator fusion as an optimization. In Section 1, we categorized past approaches to fusion as operating on the iteration language, the computation language, or both. In this section, we discuss past approaches to fusion, and continue categorizing them as such. We also discuss the scope of fusion opportunities that these system can exploit.

General-purpose compiler frameworks operate on both the iteration and computation language. For example, LLVM loop fusion [5], the polyhedral framework [1], and MLIR loop fusion [36] work directly on source code. Meanwhile, several languages like Delite [8] and Weld [34] introduce novel intermediate representations (IR) to which applications must be translated, which can then be analyzed for fusion opportunities. When fusion machinery is retrofitted into pre-existing languages and compilers, its capability is limited by the properties of the source language. For example, aliasing and complex dependence patterns can be hard to analyze in C. On the other hand, novel IRs require rewriting entire codebases, with no reuse of previously written high-performance implementations. In both cases, the entire implementation must be visible to the compiler, and it is not possible to fuse through proprietary subroutines.

In contrast, domain-specific languages (e.g., Halide [38], TACO [27], TVM [10], Mosaic [4], Hector [49], and Orchard [43]) often expose only the computation language. Their languages are limited to a particular domain (sparse tensors, dense tensors, etc.). Their compilers derive iteration spaces implicitly, generating high-performance fused code from outermost loops down to the leaf computations. Since these languages understand the mathematical domain of their computations, they can also expose fusion opportunities by applying mathematical rewrites, a capability that our system lacks. However, these approaches are limited to the domain in which their computation language specializes. Additionally, these languages completely own the physical implementation of data structures, requiring users to pay the penalty of translating data formats into the systems' data structures. Diffuse [50], a domain-specific system built on top of Legion [6], can dynamically fuse functions from the cuNumeric and Legate Sparse libraries at runtime. Languages like Fireiron [21] treat data movement as a first-class citizen, and can recursively decompose the computation to generate performant implementations for a single kernel, but not *across* computations.

Functional approaches model iteration using a set of patterns such as map, zip, and reduce. Examples include MapReduce [15], Spark [51], Haskell's stream fusion [13], deforestation [46], and push arrays [47]. However, many fusion patterns—such as those with overlapping inputs in stencil computations—fall outside these constructs, requiring users to handwrite fused code.

Lighter-weight approaches like Split Annotations [35] and Legion [6] decide how to partition data and own the iteration language. These systems are the closest to our approach. Split Annotations splits data to parallelize across subroutine boundaries by adding annotations to the subroutine interface. However, Split Annotations' approach does not generalize to patterns beyond elementwise fusion of array operations. This is because the proposed split types only describe how to break down operations and data structures into smaller pieces, and not how to chain dependencies. Legion is a programming model for parallel architectures that exploits data locality and task independence. Legion operates on logically independent regions of memory, and uses this independence to dynamically extract parallelism. The notion of data dependence (Section 4) presented in this work closely resembles Legion's idea of logical regions. However, Legion requires a heavy-weight runtime scheduler whose cost must be recovered by the workload. As Fern understands data dependencies, it can statically schedule work and operate on much finer granularities (Section 9.2).

Several neural network optimization systems like PyTorch 2.0, [2], nvFuser [41], XLA [40], TVM's graph optimizer [10], and Intel oneDNN's Graph Extension [33] fuse representations lifted from a chain of subroutine calls. They generate call graphs that can then be pattern matched or rewritten for known fusion opportunities. These approaches are still domain-specific and understand the semantics the computation since they are limited to operators in the library itself. At the extreme, several libraries also provide handwritten fused interfaces for popular sets of subroutine compositions (e.g., Apple BNNS [3], BLAS [28], CUTLASS [32]). However, these are limited to a predetermined set of fused computations. Handwritten libraries like the BLIS framework [45] and the AMD composable library [31] have exploited the notion of a *microkernel* to dispatch inner loop computations to a small, highly optimized kernel. In these cases, programmers must manually orchestrate data movement and reason about dependence patterns, which Fern automates.

11 Non-Critical Design Decisions

Section 3 details decisions that were fundamental to Fern's design. However, we would like to distinguish certain features of Fern that are more inessential artifacts of our implementation:

(1) Currently, Fern generates a C++ file that is linked into a user's application. Alternatively, it would be possible to interpret outer loops present in Fern-generated code and call into precomplied functions. It would also be possible to just-in-time compile Fern code.
(2) While Fern uses abstract data types to interface with data structures, it would be possible to specialize a Fern implementation to a particular choice of data structure and make it domain specific. This can unlock more optimizations, and reduce overhead related with querying and inserting subsets using interfaces presented in Section 5.
(3) Fern's core ideas are language independent. We chose C++ because most high-performance code is written in C++. We also implemented a Fern back end to Python, and believe that our ideas can generalize to other languages.

There may be instances where an alternative choice is more ergonomic or performant, although we believe that the ideas in Section 3 are still key to making these designs work.

12 Conclusion

We have aimed to combine the robustness of fusion approaches that exist in compilers and languages with the simplicity and ease-of-use of subroutines. We presented a lightweight annotation language for subroutine interfaces that enables fusion for nontrivial dependence patterns across many domains, without requiring source code access to their implementations. We showed that our approach is competitive with state-of-the-art systems and can fuse new workloads, even across boundaries of proprietary libraries. We hope that that instead of reinventing infrastructure to perform fusion for new domain-specific compilers and libraries, programmers will be able to reuse our solution to add fusion on top of existing interfaces. Fern is open source and can be accessed at https://fern-lang.com/.

Acknowledgment

We thank Amanda Liu for many discussions about formalizing Fern's semantics. We also thank the anonymous reviewers for their thoughtful comments on the paper and its presentation. We are grateful to Willow Ahrens, William Brandon, Ajay Brahmakshatriya, Filippo Carloni, Kartik Chandra, Tom Chen, Teodoro Collin, Samir Droubi, Olivia Hsu, Yuka Ikarashi, Amanda Liu, Ahmed Mahmoud, AJ Root, and Jaeyeon Won for their insightful comments and feedback. This research was supported by Intel and NSF PPoSS Grant CCF-2217064, NSF Award CCF-2328543, and the DARPA PROWESS Award HR0011-23-C-0101.

References

[1] Corinne Ancourt and François Irigoin. 1991. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Williamsburg, Virginia, USA) *(PPOPP '91)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/109625.109631

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366

[3] Apple Inc. [n. d.]. Apple Developer Documentation: BNNS. https://developer.apple.com/documentation/accelerate/bnns. Accessed: 2024-05-20.

[4] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (jun 2023), 26 pages. https://doi.org/10.1145/3591236

[5] Kit Barton, Johannes Doerfert, Hal Finkel, and Michael Kruse. 2018. Revisiting Loop Fusion and its place in the loop transformation framework. https://llvm.org/devmtg/2018-10/slides/Barton-LoopFusion.pdf LLVM Developers' Meeting, October 2018.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society, USA, 1–11. https://doi.org/10.1109/SC.2012.71

[7] Paul G. Brown. 2010. Overview of sciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 963–968. https://doi.org/10.1145/1807167.1807271

[8] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-Specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/1941553.1941561

[9] Chun Chen, Jacqueline Chame, and Mary W. Hall. 2007. *CHiLL : A Framework for Composing High-Level Loop Transformations.* Technical Report.

[10] 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 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[11] Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024. Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding. arXiv:2402.12374 [cs.CL] https://arxiv.org/abs/2402.12374

[12] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (oct 2018), 30 pages. https://doi.org/10.1145/3276493

[13] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 315–326. https://doi.org/10.1145/1291151.1291199

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[16] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/199448.199461

[17] Galler and Perlis. 1976. *A View of Programming Languages (Addison-Wesley Series in Computer Science and Information Pr)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[18] GEOS contributors. 2021. *GEOS coordinate transformation software library.* Open Source Geospatial Foundation. https://libgeos.org/

[19] Sean Gillies, Casper van der Wel, Joris Van den Bossche, Mike W. Taves, Joshua Arnott, Brendan C. Ward, and others. 2024. *Shapely*. https://doi.org/10.5281/zenodo.5597138

[20] Brian Gough. 2009. *GNU scientific library reference manual*. Network Theory Ltd.

[21] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3410463.3414632

[22] Apple Inc. [n. d.]. Accelerate Framework. https://developer.apple.com/documentation/accelerate. Accessed: 2024-06-10.

[23] 2009. *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara. Accessed: 2024-06-10.

[24] 2011. *Intel Advanced Vector Extensions Programming Reference*. Intel Corporation, Santa Clara, USA. https://www.intel.com/content/dam/develop/external/us/en/documents/36945

[25] Intel Corporation. 2024. Intel Implicit SPMD Program Compiler (ISPC). https://ispc.github.io/ispc.html. Accessed: 2024-06-10.

[26] Intel Corporation. 2024. Memory Format Propagation. https://www.intel.com/content/www/us/en/docs/onednn/developer-guide-reference/2024-1/memory-format-propagation.html. Accessed: 2024-06-10.

[27] Fredrik Kjølstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 180–192. https://doi.org/10.1109/CGO.2019.8661185

[28] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

[29] S.T. Leutenegger, M.A. Lopez, and J. Edgington. 1997. STR: a simple and efficient algorithm for R-tree packing. In *Proceedings 13th International Conference on Data Engineering*. 497–506. https://doi.org/10.1109/ICDE.1997.582015

[30] Arm Limited. [n. d.]. Arm Performance Libraries. https://developer.arm.com/downloads/-/arm-performance-libraries.

[31] Chao Liu, Jing Zhang, Letao Qin, Qianfeng Zhang, Liang Huang, Shaojie Wang, Anthony Chang, Chunyu Lai, Illia Silin, Adam Osewski, Poyen Chen, Rosty Geyyer, Hanwen Chen, Tejash Shah, Xiaoyan Zhou, and Jianfeng Yan. [n. d.]. *Composable Kernel*. https://github.com/ROCm/composable_kernel

[32] NVIDIA. 2022. *CUTLASS*. NVIDIA. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/

[33] oneDNN Contributors. [n. d.]. oneAPI Deep Neural Network Library (oneDNN). https://github.com/oneapi-src/oneDNN Accessed: 2024-06-10.

[34] Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2017. Weld: Rethinking the Interface Between Data-Intensive Applications. *CoRR* abs/1709.06416 (2017). arXiv:1709.06416 http://arxiv.org/abs/1709.06416

[35] Shoumik Palkar and Matei Zaharia. 2019. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 291–305. https://doi.org/10.1145/3341301.3359652

[36] LLVM Project. 2024. Affine Loop Fusion - MLIR Documentation. https://mlir.llvm.org/docs/Passes/#-affine-loop-fusion Accessed: 2024-06-20.

[37] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

[38] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (dec 2017), 106–115. https://doi.org/10.1145/3150211

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

[40] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[41] Christian Sarofeen, Piotr Bialecki, Jie Jiang, Kevin Stephano, Masaki Kozuki, Neal Vaidya, and Stas Bekman. 2022. Introducing nvFuser, A Deep Learning Compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/.

[42] Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 189–203. https://doi.org/10.1145/3572848.3577509

[43] Vidush Singhal, Laith Sakka, Kirshanthan Sundararajah, Ryan Newton, and Milind Kulkarni. 2024. Orchard: Heterogeneous Parallelism and Fine-grained Fusion for Complex Tree Traversals. *ACM Trans. Archit. Code Optim.* 21, 2, Article 41 (May 2024), 25 pages. https://doi.org/10.1145/3652605

[44] TileDB, Inc. 2024. TileDB: The Universal Storage Engine. https://github.com/TileDB-Inc/TileDB. Accessed: 2024-06-10.

[45] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (jun 2015), 33 pages. https://doi.org/10.1145/2764454

[46] Philip Wadler. 1988. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.* 73, 2 (Jan. 1988), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[47] Philip Wadler. 1988. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.* 73, 2 (Jan. 1988), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[48] Robert Alan Wagner. 1968. *Some techniques for algorithm optimization with application to matrix arithmetic expressions.* Ph. D. Dissertation. USA. AAI6907907.

[49] Kun Wu, Mert Hidayetoğlu, Xiang Song, Sitao Huang, Da Zheng, Israt Nisa, and Wen-Mei Hwu. 2024. Hector: An Efficient Programming and Compilation Framework for Implementing Relational Graph Neural Networks in GPU Architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 528–544. https://doi.org/10.1145/3620666.3651322

[50] Rohan Yadav, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. 2025. Composing Distributed Computations Through Task and Kernel Fusion. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 182–197. https://doi.org/10.1145/3669940.3707216

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.

Received 2024-11-13; accepted 2025-03-06