Splitability Annotations: Optimizing Black-Box Function Composition in Existing Libraries

Shoumik Palkar and Matei Zaharia
Stanford University

Abstract

Data movement is a major bottleneck in parallel data-intensive applications. In response to this problem, researchers have proposed new runtimes and intermediate representations (IRs) that apply optimizations such as loop fusion under existing library APIs. Even though these runtimes generally do not require changes to user code, they require intrusive changes to the library itself: often, all the library functions need to be rewritten for a new IR or virtual machine. In this paper, we propose a new abstraction called splitability annotations (SAs) that enables key data movement optimizations on black-box library functions. SAs only require that users add an annotation for existing, unmodified functions and implement a small API to split data values in the library. Together, this interface describes how to partition values that are passed among functions to enable data pipelining and automatic parallelization while respecting each library’s correctness constraints. We implement SAs in a system called Mozart. Without modifying any library function, on workloads using NumPy and Pandas in Python and Intel MKL in C, Mozart provides performance competitive with intrusive solutions that require rewriting libraries in many cases, can sometimes improve performance over past systems by up to 2×, and accelerates workloads by up to 30×.

1 Introduction

Developers write software by combining functions from existing libraries. Unfortunately, even when the library functions are highly optimized, this composition can be expensive, especially on modern parallel hardware. As the gap between memory bandwidth and processing speeds has grown [19, 40], researchers have observed that data movement between functions that exchange values using pointers to in-memory buffers dominates running time in many data-intensive workloads (e.g., data analytics) [5, 29, 37, 41]. For example, even though Intel’s MKL [24] library contains optimized, multithreaded vector math functions written in assembly, the interface to these functions is to take pointers to in-memory buffers: repeated calls to the functions thus causes repeated passes through memory, which we found can be 8× slower than custom code that pipelines these operations.

In recognition of these performance gaps, researchers have proposed redesigning existing libraries either by using specialized DSLs [22, 31, 36, 37] or runtimes [1, 20, 29, 33]. As one recent example, Weld [29] proposes rewriting library functions to target a common runtime using an intermediate representation (IR), and JIT-compiles parallel code for optimized execution. In addition to the benefits of generating machine code, this approach reduces data movement by applying optimizations such as loop fusion, and parallelizes computations automatically without modifying the libraries’ user-facing APIs.

Unfortunately, obtaining the benefits of these systems requires highly intrusive changes to the library itself. For example, in order to analyze code and apply optimizations such as loop fusion, Weld asks developers to throw out library functions that developers have already optimized entirely and rewrite them using its IR. Other systems similarly replace or re-architect existing library code using IRs [10, 29, 31] or custom virtual machines [20]. For meticulously optimized libraries such as MKL, these proposed systems might not even support all the existing hand-coded optimizations (e.g., Weld currently does not vectorize element-wise trigonometric functions, unlike MKL), so developers face a tough choice between expanding these already complex compilers, dropping their own optimizations, or forgoing optimization under composition.

To address this problem, we propose a new abstraction called splitability annotations (SAs) that allows a system to perform data pipelining and automatic parallelization under black-box library functions for the first time. Splitability annotations not only keep the user-facing APIs of libraries intact but also greatly reduce the developer effort required to enable the main data movement optimizations provided by existing systems. We show that SAs yield competitive performance with prior systems on data-intensive workloads without requiring any modifications to existing library functions. Of course, SAs cannot provide all the optimizations possible with a compiler (e.g., rewriting a library function), but past work has shown that data movement and parallelization are often the most impactful optimizations in data-intensive code [29], explaining why our performance comes close to these systems in many cases.

Splitability annotations are annotations on side-effect-free functions that define how inputs can be split (i.e., partitioned) and fed into the function, as well as how different functions can safely be composed. Fundamentally, SAs are used to divide data passed into a function call into multiple pieces in order to enable pipelining the pieces through a composition of functions. Pipelining small pieces reduces data movement by precluding a read from RAM for every operation by keeping data in fast...
memory (e.g., the L1 cache). SAs also enable parallelism by processing multiple pieces in parallel. This type of L1-level data pipelining has already been used successfully in some systems, such as vectorized databases [43], but our contribution is a novel abstraction to safely use this mechanism under diverse, existing libraries.

To safely compose functions over split data values, SAs need to ensure that the functions’ parameters have been split in a compatible way. For example, 1D vectors can be split into pieces of varying lengths, but some functions, such as the add function on vectors, require that the two arguments be split in the same way in order to process the corresponding input pieces safely together. Other functions, such as filtering a vector to remove negative values, might change the size of the underlying pieces and then be pipelined with a function like add. For more complex data types, such as 2D matrices, there might be multiple ways of splitting the data (e.g., by row or by column). To specify and reason about these constraints, splitability annotations use a type system to assign a split type to the parameters and return values of individual functions, and then automatically infer how to safely split, pipeline and parallelize a computation. We use a simple parameterized type system that we show supports many existing libraries, but in principle any type system can be used to represent split types.

SAs require a computation DAG representing the functions called by the user to make cross-function optimizations. SAs can be used with any existing method to capture such a DAG, such as compiler passes [37] or techniques that make an existing library API lazy [29]. To demonstrate SAs, we implemented a system called Mozart that can captures DAGs through easy-to-inject lazy evaluation techniques in Python and C (decorators in Python [14] and macros plus memory protection in C [23]). Mozart thus requires no code changes to existing library functions: users or developers only need to provide the annotations for the functions they wish to support and small helper functions to split the library’s underlying data types.

We evaluate SAs using Mozart and show that they can improve the performance of workloads that compose functions from existing libraries such as Pandas and NumPy in Python and even the closed-source Intel MKL in C. We show that, on analytics workloads such as cleaning data using Pandas (taken from Weld [29]) and vector algebra using MKL, Mozart-enabled libraries accelerate the native libraries by 5–30× on 32 threads. Mozart-enabled libraries provide performance within 1.1–6× of Weld, a JIT-compiling optimizing runtime in a multi-threaded setting, and can even improve performance over Weld by up to 2× by leveraging optimizations that existing libraries perform that Weld does not. Mozart also reduces memory usage by up to 2.5× over the original library by preventing allocation of intermediate values. Finally, we show that integration effort for Mozart is low, requiring on average 100 lines of code total to implement the data splitting logic for each library and add SAs to each function (10× less than Weld [29]).

To summarize, our contributions are:

1. Splitability annotations (SAs), a new abstraction that enables automatic pipelining and parallelization of black-box library functions while respecting the library’s correctness constraints.
2. Mozart, an implementation of SAs in C and Python that can accelerate existing libraries without requiring any code changes or access to source code.
3. An evaluation of Mozart to show that it accelerates existing libraries by up to 30×, an improvement within 1.1–6× of runtimes that require replacing code.

2 Overview

To motivate splitability annotations, consider the VAdd function shown in Listing 1 based on the MKL vector-add function. This function has been meticulously tuned by the library’s authors and makes use of low-level optimizations such as SIMD vectorization and instruction scheduling to saturate the CPU execution units.

```c
struct vec_t { double *data; size_t length; }; // Computes res = a + b element-wise
void VAdd(vec_t *a, vec_t *b, vec_t *res);
```

Listing 1: A vector add function based on MKL [24].

Listing 2 shows an artificial workload that chains calls to this function. Even though the VAdd function is highly optimized, because the interface of this function is to pass it pointers to memory buffers, repeated calls to it will repeatedly scan the full array. As a result, if the arrays are large, data will always be loaded from main memory rather than a fast core-local memory (e.g., the L1 cache). This data movement issue quickly becomes a bottleneck in data-intensive applications, and becomes even worse if the library function is parallelized. With more threads, the gap between compute cycles and memory bandwidth grows wider, and the workload spends increasingly more time stalled on memory than on computation. Runtimes like Weld propose new interfaces for these data-intensive functions [28] by expressing the computation in a special IR to enable fusing individual calls, but changing the interface requires deeply intrusive modifications: in this case, it requires rewriting an optimized function completely.

```c
// res = inp1 + inp2 + ... + inp10
VAdd(inp1, inp2, res);
VAdd(res, inp3, res);
...
VAdd(res, inp10, res);
```

Listing 2: An example program using the vector library.

Splitability annotations (SAs) solve this problem by enabling data movement optimizations such as pipelined...
execution for black-box functions. An SA is a per-function annotation that users specify for library functions: SAs do not require any changes to the functions themselves. As an example, Listing 3 shows an SA that a user could provide for the $\text{VAdd}$ function:

```c
// @splittable
// (a: S, b: S, mut res: S) -> void
void VAdd(vec_t *a, vec_t *b, vec_t *res);
```

**Listing 3**: A simple splitability annotation for the $\text{VAdd}$ function shown in Listing 3.

SAs specify how to split function inputs into smaller pieces in order to pipeline values and parallelize the computation. Specifically, SAs assign each function parameter and return value a split type that defines how a value is split. If two values have the same split type, then they have equivalent partitions. For arrays, one way to partition data is to divide it into fixed sized chunks: two arrays have the same partitions if they are split into the same number of pieces, and corresponding pieces have the same length. In the annotation for $\text{VAdd}$, we assign the three inputs the same placeholder split type $S$, so $\text{VAdd}$ accepts any set of inputs with equivalent partitions. We discuss split types in detail in §3.

To illustrate the benefits of using SAs, Figure 1 shows the performance of the workload from Listing 2 with and without an SA on the $\text{VAdd}$ function. Each input vector contains $2^{30}$ double elements. Without SAs, MKL internally parallelizes each function individually. SAs not only improve the raw performance of this workload but also improve scalability: by splitting and pipelining data through the functions, the SA runtime keeps data in core-local caches rather than stalling on main memory access. MKL exhibits diminishing speedups after only 2 threads, while SAs enable linear scaling to 32 threads.

**2.1 System Architecture**

**Annotations.** Splitability annotations are per-function annotations that specify how function arguments are split into multiple, smaller pieces by assigning each argument a split type. A split type defines a specific partitioning for a value: two values have the same split type if and only if they are partitioned in the same way. The split type of a value thus depends both on the value’s original data type (e.g., an array) and how the data type is split (e.g., splitting arrays into fixed-sized pieces, where corresponding pieces have the same length).

The $\text{VAdd}$ SA in Listing 3 required that each input have the same split type because split inputs to $\text{VAdd}$ must have the length and be “lined up.” SAs formally use a small type system to describe split types, which we discuss in detail in §3. To split data types in existing libraries into multiple pieces, users provide user-defined splitter functions for each data type, and also provide merger functions to merge split pieces into a full value again.

**Computation DAG.** In addition to the annotations themselves, the SA runtime requires viewing a full computation in order to pipeline and parallelize functions. For example, in Listing 2 we can only pipeline the calls to $\text{VAdd}$ by seeing the full computation up-front. The SA runtime thus requires a computation DAG in order to apply optimizations. The computation DAG captures dependencies among the inputs and outputs of a computation: each node is a function call, and each edge is a data dependency (e.g., we draw an edge from the return value of one function to the argument of another if the second function consumes the return value of the first). The DAG also specifies which values are outputs, i.e., values that are visible outside the DAG’s scope. The SA runtime can reduce the memory footprint of the intermediate values internal to the DAG.

Many existing libraries already contain mechanisms for capturing a computation DAG lazily (e.g., TensorFlow). Another technique for constructing a DAG is to write a compiler pass (e.g., for LLVM) that analyzes functions with annotations. Our runtime, Mozart, also provides a proof-of-concept API for constructing a computation DAG at runtime using function decorators in Python and virtual memory protection in C without changing library functions.

**Using SAs in Existing Libraries.** Overall, users can integrate SAs into existing software libraries as follows:

1. First, users add SAs to each library function that they want to support optimizing. They also provide user-defined splitter functions for partitioning data types passed to the library (e.g., arrays) into smaller pieces.
2. Users choose a mechanism to construct computation DAGs representing multiple function calls in their library. Our system Mozart contains APIs that wrap existing Python and C libraries for constructing a lazy computation DAG.
3. Users submit the constructed computation DAG to the SA runtime (e.g., by using the wrapped APIs above). The runtime uses the annotations and splitter functions to optimize the computation.
Once users submit a computation DAG to the runtime, SAs reduce data movement by pipelining split pieces through the computation so each function in the DAG operates over a small working set that is served from a fast memory. SAs parallelize libraries by processing disjoint pieces of data in parallel. Unlike systems that make use of a compiler, functions that developers have hand-optimized do not need to be rewritten, and programs still benefit from existing low-level single-thread optimizations (e.g., vectorization or instruction scheduling).

2.2 Limitations and Non-Goals
SAs have a number of restrictions and non-goals. First, SAs only focus on data movement optimizations such as pipelining single- and multi-threaded computations, so they do not apply the compute-based optimizations that systems like Weld [29] and TensorFlow XLA [41] can provide by introspecting library functions. Second, since SAs make repeated calls to black-box functions to achieve pipelining and parallelism, they are limited to functions that do not cause undesirable side effects.

3 Splitability Annotation Abstraction
This section describes the splitability annotation abstraction. We first provide a simple motivating example for the components of our abstraction. We then describe the split types and the annotation type system, as well as the other features of SAs. Finally, we show some additional examples of using SAs to represent more complex workloads.

3.1 Motivating Example: Filtering Data
To motivate the split types and type system, consider the program in Listing 3, which introduces a new library function VFilter. The workload adds two vectors using the VAdd function defined in §2, filters the result using VFilter, and then adds a third vector with the filtered one. Like the first example from Listing 2, this workload can benefit from SAs because it makes repeated library functions calls and heavily reuses data: SAs can enable pipelining and parallelization to improve performance.

```c
// Returns a new vector with elements from a that // are less than max.
vec_t *VFilter(vec_t *a, double max);

// tmp = inp1 + inp2 + inp3
VAdd(inp1, inp2, tmp);
VAdd(tmp, inp3, tmp);
vec_t *filtered = VFilter(tmp, 10.0);
VAdd(filtered, inp4, result);
```

Listing 4: An example program using the vector library.

To execute the program in Listing 4 using SAs, we first need to add an SA for the VFilter function. Recall that we defined an SA for VAdd in Listing 3 already: the SA for VAdd accepts three inputs that are split in the same way (i.e., have the same placeholder split type S), because VAdd requires inputs and outputs to be of the same length. In this example, we will assume that the arrays passed to VAdd are all split into regularly sized pieces.

The VFilter function differs from VAdd in one major aspect: VFilter returns an array that is not necessarily the same length as its inputs. To understand why this is significant for writing an SA, consider what happens when we pass split pieces through the computation. We first split inp1, inp2, inp3, and tmp into regularly sized pieces, and then call the first two VAdd functions on the pieces. After calling these functions, the input to VFilter, tmp, is still a regularly-sized piece. However, we then call VFilter, which returns a value filtered with an unknown size. Note that filtered is still split in some way, since it was produced by calling VFilter on a split piece.

Because filtered is still split, before we call the last VAdd in the computation, we need to split the arrays inp4 and result. However, we cannot simply split these arrays into regularly-sized pieces, like we did for the other input arrays. The reason for this is that filtered could have a different length, so passing it into VAdd with regularly-sized pieces of inp4 and result could lead to incorrect results. Equivalently, we can say that filtered has a different split type. Runtimes can choose how to address this issue after detecting it: our implementation merges values before calling a function with a different split type.

Critically, this mismatch is only an issue when we split data: we assume that in the original program the user writes (over un-split data), the user knows that the full filtered and result arrays will have the same length. We cannot make the same guarantee on the individual partitions of the data. In short, the splits for the inputs and the splits for the output values of VFilter are incompatible, so the annotation need some way of detecting this mismatch.

The split type system encodes this information. Listing 5 shows the SA for VFilter. This annotation states that VFilter takes an input array a with split type S—so the input can be split in any way—and returns an array that is split in a different way by using a different placeholder T. The max parameter is not assigned a split type, so it is broadcast (i.e., we copy it instead of splitting it).

```c
//@splittable
//(a: S) -> T
vec_t *VFilter(vec_t *a, double max);
```

Listing 5: A splitability annotation for VFilter.

Split types can be used to capture pipelining constraints for a variety of data types. For example, split types can represent matrix functions that iterate matrices by-row.
vs. by-column, or string processing functions that operate on a character-by-character basis vs. splitting strings by newline. We describe the split type system formally in the following subsection, and show more examples of their use in §3.4.

### 3.2 Split Types

A split type defines a unique partitioning for some data value. Split types allow checking the *compatibility* between two functions after their values have been split. If a function’s output has the same split type as another function’s input, we can safely pipeline split values between the output and input. If the output and input have different split types, then we need to *merge* split pieces before passing them to the next function, effectively causing a break in a pipeline.

The precise definition of a *split* (i.e., partition) is data- and library-dependent, but generally captures some information about the *shape* of a data type. For example, an array can be split into multiple regularly-sized pieces and is uniquely defined by the initial length of the array and the number of split pieces. An array with 10 elements can be split into two pieces with five elements or five pieces with two elements: these splits have *different* split types, even if the underlying data refers to the same array.

Split types follow some arbitrary type system: in our implementation, we use a small *dependent* type system [16] where we represent each split type as a *name* and a set of *parameter values*. We represent a split type using the notation `Name<Parameters>`. As an example of a split type, we can represent an array split into regularly-sized pieces using the type `RegularSplit<L, P>`, where `L` represents the length of the original array, and `P` represents the number of pieces. Note that the parameters `L` and `P` are *values*: two `RegularSplit` types with *different* values are different split types, so:

```plaintext
RegularSplit<A, B> != RegularSplit<C, D>
```

Split types may or may not have an associated user-defined *splitter function* and *merger function*. A splitter function takes a value in the program and splits it into multiple pieces with some split type, while a merger function merges those pieces back into a full value. As an example, we can define a splitter function that creates values with the split type `RegularSplit<L, P>` trivially, using the length and number of pieces to compute offsets into the array. We discuss how users define splitter and merger functions in §4.

Split types may not have any splitter function. As an example, consider the `VFilter` function from Listing 4, which has an annotation that takes a value with some split type `S` and returns some split type `T`. We could have equivalently defined `VFilter` as shown below:

```plaintext
//@splittable
```
// Only accept length-10 arrays that have been
// split into 10 regularly-sized pieces.
VAdd(a: RegularSplit<10, 10>,
    b: RegularSplit<10, 10>,
    c: RegularSplit<10, 10>)

// Accept any arrays split regularly, as long as
// the splits are the same.
VAdd(a: RegularSplit<A, B>,
    b: RegularSplit<A, B>,
    c: RegularSplit<A, B>)

// Accept arrays split in any way, as long
// as the splits are the same.
VAdd(a: S, b: S, c: S)

// Accept any arrays, each split in any way.
// NOTE: incorrect annotation for VAdd!
VAdd(a: A, b: B, c: C)

Listing 6: Sample annotations for VAdd. The first two
annotations are overly restrictive and the last annotation
is not restrictive enough.

ops. Listing 7 gives several examples.

// Concatenates two vectors of any length.
// @splittable
// (a: S, b: T) -> U
vec_t *VConcat(vec_t *a, vec_t *b);

// Unary in-place exponent on each element.
// @splittable
// (mut a: S) -> void
void VExp(vec_t a);

// Compute the sum of a vector.
// @splittable
// (a: S) -> Sum<double>
double VSum(vec_t a);

Listing 7: Several array functions expressed using SAs.
The Sum<double> is a split type that we can merge by
summing partial results.

In the VSum function, the Sum<double> split type simply
wraps a double. The reason for this is that we need to
implement a custom merger function which sums the
partial results of split pieces. §4 describes the interface
for this in more detail.

Ex. 2: Arrays with Separate Length Parameter. In our
original definition of VAdd, we defined the function over
a struct vec_t, which bundled the length and data pointer
as a single argument. However, many C functions pass
an array’s length and its data pointer separately, including
MKL. Listing 8 shows the VAdd definition as specified by
MKL, and also shows its corresponding SA.

Listing 8: SA for VAdd with a separate len field like in MKL.

Note that we are no longer using a placeholder parameter
S to define this function. Instead, we use two split types: a
RegularSplit<L, P> as we defined earlier to split the array,
and a new split type SizeSplit<L, P> that returns the cor-
responding length. Since these are two separate parameters
(and the two to map to separate splitter function implement-
ations), the author of the splitter function must ensure that
the correct array length is returned for a given <L, P> pair.

Ex 3: Splitting Matrices. Matrices can be accessed either
in a row-major or column-major access pattern. Matrix
functions that mix these access patterns generally cannot
be pipelined: if we pipeline a row-major function with a
column-major one, the second function could read stale or
incorrect values. We can thus use SAs to enable pipelining
matrix functions with the same access pattern, and prevent
pipelining different access patterns. Listing 9 shows SAs
for two functions, incrementByRow and incrementByCol,
that respectively increment elements in a matrix by
traversing them in row-major and column-major order.

Listing 9: SAs for functions that iterates over matrices.

The incrementByRow function uses a new split type
RowMajorMatrixSplit<R,C,P> to split the matrix. The
split type takes three parameters, representing the number
of rows, the number of columns, and the number of
pieces to split the matrix into. These three parameters
uniquely define how to split a matrix by row. In particular,
RowMajorMatrixSplit represents splits with a single row
of the matrix at a time. We thus set the rows argument
to a constant value of 1 (a split matrix will always have
The remaining string functions operate over characters.

string *to_upper(string *s);

tuple_t *read_csv(string *s);

int strchr(string *s, char c);

// (s: S) -> S
@splittable
// (s: LineSplitter<L,P>) -> S
tuple_t *read_csv(string *s);

// Convert string to uppercase.
@splittable
// (s: S) -> S
string *to_upper(string *s);

// Find a character in a string, returning
// the index of the character or -1 if it
// is not found.
@splittable
// (s: S, c: c) -> Index<int>
int strchr(string *s, char c);

Listing 10: SAs for string functions.

One row) and the cols value to cols, which represents the original function argument.

The function that iterates over columns is symmetrical: the splitter function splits the matrix into columns, and adjusts the remaining arguments accordingly: a matrix split by column will always have a single column and an unmodified number of rows. Since we assigned these two functions different split types, the SA runtime will pipeline across repeated calls to the same function, but merge and pass full values across different functions.

Ex 4: Splitting Strings. Different string functions require different splitting schemes when pipelining. For example, a function that takes a string buffer and parses CSV splits a string by newline, while others iterate over individual characters. Listing[10] show a variety of string functions and their SAs.

The read_csv function specifies a specific split type LineSplitter<L,P> that splits strings by newline, since the function needs to see entire lines. L and P represent the length of the string and the number of pieces, which we can use to compute offsets (similar to the array splitters). The function returns an array that is split in some unknown way. The remaining string functions operate over characters.

For the strchr function, we split a string in some way and search for c in each substring. If we want to find the first character in the string, we need a custom function to merge the partial results (i.e., to get the minimum non-negative index)—we thus assign the output of strchr a custom split type. The next section describes how users define custom splitter functions to split functions and to merge partial results into a full value.

4 Splitter Functions

Users provide splitter functions and merger functions to split and merge data types in their libraries. Concretely, a splitter function takes a value and returns a collection, where the collection represents partitions of the original value. The returned collection must supported indexed access to retrieve partitioned values to support parallelism. A merger function takes an array of split values and merges them into a result.

Each splitter function has a corresponding split type (every split type does not necessarily have a splitter function, however). Recall that in our type system, split types can have a set of parameters: these parameters represent parameters passed into a splitter function when splitting a value. As an example, consider the RegularSplit<L,P> split type, which has two parameters L (array length) and P (number of pieces). A splitter function that splits an array into RegularSplit<L,P> takes the array length and the number of pieces as input parameters.

Merger functions are similarly associated with split types, and can be used to define custom aggregation operators. As an example, the VSum function from Listing[7] returned the split type VSum<double>: we can write a custom merger function that sums values for this split type. The Index<int> type defined in Listing[10] can be handled similarly.

To summarize, for each split type in the program, users can optionally implement the following splitter and merger function interface for the underlying library data type:

1. Provide a splitter function that returns a collection of split values. In our implementation, collections are lazy (i.e., we do not materialize an array of values: when we index into the collection, we either return a view into an existing object (e.g., by returning an offset pointer) or create one if necessary). Collections must support random-access indexing to support parallelism.

2. Provide a merger function that takes an array of values and merges them into a full value. If a function needs a particular merger to be used (e.g., summing partial results), users can give the return value of the function signature a special split type in the SA and write a merger for that split type (e.g., VSum<double>).

5 Mozart Runtime and Optimizations

Mozart is a runtime we have designed for C and Python programs that uses SAs to optimize function compositions. We describe the optimizations SAs enable in in this section, and describe our implementation of Mozart in §6.

SAs enable various optimizations over a computation DAG. Recall that a computation DAG in our system represents annotated function calls and their arguments as nodes and data dependencies among arguments as edges. The DAG also captures which variables are outputs and which ones are intermediate values that exist only within the scope of the DAG.

Libraries can capture computation DAGs in a variety of ways, such as a compiler pass that analyzes calls to annotated functions. As a proof-of-concept, Mozart contains a
The pipeline optimization enables passing split values between functions calls directly to improve memory locality. We enable pipelining by finding the largest pipelined components in the DAG—this is similar to how existing compilers fuse loops greedily [6, 29]. A pipelined component is a subgraph in the DAG where each root node in the subgraph splits data, each leaf node merges the data into a full value, and each intermediate node operates directly on split values.

The SAs and split types enable pipelining. Recall that we can pipeline data between two DAG nodes \( f_1 \) and \( f_2 \) if the output type of \( f_1 \) matches the input type of \( f_2 \). In particular, each node will have \( n \) incoming edges—where \( n \) is the number of arguments—so we can pipeline a node with its predecessors only if every edge has matching types (this is because we cannot pass a combination of split and un-split values to a function at once).

To determine which values can be pipelined, Mozart applies type inference to propagate split types through the computation DAG. Starting at the inputs of the DAG, we choose a splitter function for each argument value based on its split type. If the SA declares a placeholder split type (e.g., \( \text{Vec} \)), Mozart can choose any splitter for that data type (e.g., for \( \text{VAdd} \), Mozart will pick any function available for \( \text{Vec} \), \( t \)). The placeholder is then replaced with the split type produced by the chosen splitter function. Placeholders with the same name in an annotation are assigned the same split types, and different placeholder names are assigned different split types. Note that placeholder names are local to a single SA.

Mozart then walks the DAG and \( \text{syncs} \) the split types of inputs and outputs: for each edge, we match the split types of the edge source and destination. If the edges cannot be matched (i.e., because they have different split types), Mozart applies the \text{merger function} of the source edge to coalesce values. The algorithm then recurses starting at the next node (Mozart chooses a new splitter function, assigns split types to placeholders, etc.). After type inference, nodes that pass split values without calling a merger function are pipelined. Figure 2 shows the code example from Listing 4 after type inference.

5.2 Automatic Parallelization

Mozart parallelizes both individual functions and pipelines by processing multiple split pieces in parallel, as long as they are not marked with the \text{noparallel} tag in the SA (§3). To parallelize a pipeline, every function in the pipeline must be parallelizable. Our implementation statically partitions pieces and runs them in parallel: if we split inputs into \( n \) pieces, we run \( n \) concurrent tasks in parallel. Note that, to support efficient parallelism, collections holding split pieces should not hold locks or mutable state.

5.3 Buffer Reuse

Finally, Mozart leverages knowledge of which variables are intermediate values to reduce memory usage. These values are not accessed outside of the DAG: this means that they do not need to be fully materialized or even contain “correct” results when the DAG execution finishes.

To reduce memory usage, for each intermediate variable used within a pipeline, Mozart always \text{reuses the same piece} for temporary (in a parallel setting, Mozart uses a unique piece per thread). This keeps the piece in fast memory, and, in many cases, also prevents excess memory from being allocated (e.g., if memory for a temporary value is allocated via demand paging [15] or created by copying data into a buffer). We found this scenario was common especially for Python workloads that created large intermediate buffers for every operation.

6 Mozart Implementation

Mozart implements a parallel runtime for SAs and also provides an API to inject lazy evaluation into existing programs. We describe two implementations of Mozart: a dynamically linkable library for C programs, and a Python module for Python-based libraries. Both implementations do not require any changes to existing library functions.

6.1 C Library Implementation

Our C-linkable implementation is a dynamic library written in Rust. We describe the key components below.

Providing Annotations. Users write annotations over C function headers and then use a provided Python script that parses annotations and generates two wrapper functions. Users of Mozart do not interact with these wrapper functions directly. The first wrapper function has the same argument list as the library function. It intercepts
calls to the function, packages arguments into a packed
struct, and passes them to our Rust runtime. The runtime
then records the function call in a graph data structure.
The second wrapper function accepts the packed struct
as an argument and calls the original library function.

Runtime and Capturing a DAG. To inject lazy evalua-
tion and capture a DAG, our runtime provides a custom
malloc and free implementation that protects memory
(similar to [20, 32]). Memory protection is implemented
by calling mmap with PROT_NONE (unreadable, unwritable)
permissions. The runtime then registers a SIGSEGV signal
handler to intercept protection violations: if the faulting
address was allocated using our allocator, we execute the
task graph. We implement the optimizations in §5 over our
task graph and use Rust’s threading library for parallelism.

To use the C Mozart implementation, users thus need to
(1) write annotations for their functions, (2) implement
splitters for the data types, (3) call our Python script
to generate function wrappers (this can be done in, e.g., a
Makefile) and (4) link libmozart to access our runtime.

6.2 Python Module

We implemented our Python runtime natively in Python
to avoid the complexity of managing objects in another
language.

Providing Annotations. Developers provide annotations
by using function decorators that take a string as an input:
the string is parsed into an annotation object for a function.
No other changes are required to provide annotations.

Runtime and Capturing a DAG. Our Python implemen-
tation also uses the same function decorator to enable
lazy evaluation. The decorator wraps the original Python
function in one that records it and its arguments in a task
graph when called. The wrapper function then returns a
placeholder object. When users access the placeholder
object, we evaluate the task graph. In Python, we can
detect when an object is accessed by overriding its
special methods (e.g., __getattr__ and __str__ for printing the object, etc.).

We implement parallelism in Python using the
multiprocessing module (Python’s GIL prevents parallel-
ism, except for IO). One limitation of this approach is
that developers need to put their data in shared memory
to prevent large copies. For array-like data, we wrap
data in shared memory using the sharedmem module
for this purpose. For general Python objects, data needs
to be serialized for IPC. To use the Python Mozart
implementation, users again need to annotate functions,
write splitters, and import the mozart module.

7 Evaluation

We evaluate splitability annotations and Mozart on a
variety of analytics workloads using existing software
libraries to show that (1) they can produce performance
competitive with code generation systems like Weld and
specialized systems like Bohrium, and (2) that the effort
required for integrating them into existing libraries is low.

Summary of Results. We measured performance on
seven benchmarks: five in Python using NumPy and
Pandas (popular numerical and data science libraries) to
evaluate our Python runtime (§7.1, §7.3), and two in Intel
MKL to evaluate our C runtime (§7.2). We also ran one
micro-benchmark using MKL to evaluate the tradeoff
between data movement optimizations and computational
intensity. Our evaluation has the following takeaways:

1. Mozart consistently outperforms unmodified Pan-
das, NumPy, and MKL with its data movement
optimizations and consumes less memory.

2. Mozart can produce performance within 6× of Weld
(a JITing parallel runtime that requires rewriting
library code) in the worst case, and out-performs Weld
in the best case by 2.5× by leveraging optimizations
in existing libraries that Weld did not apply.

3. Integrating SAs into existing libraries required little
effort (§7.5) compared to systems that use an IR.

Experimental Setup. We evaluate our Python runtime
using NumPy v1.14.2 and Pandas v0.22.0. Both NumPy
and Pandas implement their core operators in C or Cython
and are optimized for a single thread. We evaluate our
C runtime using Intel MKL 2018 [24], a closed-source
vector math library optimized for x86 CPUs. We ran
experiments on an Amazon EC2 m4.10xlarge instance
with Intel Xeon E5-2676 v3 CPUs (40 hyperthreads) and
160GB of RAM. Results are an average of five runs.

7.1 Python Data Science Workloads

We evaluate Mozart’s single- and multi-threaded perfor-
mance against the native NumPy and Pandas libraries
and Weld. Weld takes the approach of replacing existing
library code, while Mozart takes the black-box approach
of annotating functions with SAs. We also evaluate workloads that use NumPy against Bohrium [4], a virtual
machine for NumPy that fuses loops. Like Weld, Bohrium
implements an entirely new backend for NumPy.

We use the following data science workloads, taken
from the Weld [28, 29] evaluation:

1. Black Scholes [2], a computationally expensive
workload that uses NumPy to perform several
element-wise vector operations on a set of five input
arrays. We ran on arrays with 2^30 64-bit floats.

2. Haversine Distance [18], a workload that uses
NumPy to compute the Haversine distance for a
dataset of GPS coordinates to a fixed coordinate. Our
dataset again had 2^30 elements with 64-bit floats.

3. Data Cleaning [13], an example Pandas workload
from the Pandas Cookbook [8] that cleans a DataFrame
of ZIP codes by truncating each one to five digits using Python’s string slicing functions, replacing broken and zero-valued codes with nan, and filtering duplicate zip codes. We used a 25GB dataset of zip codes.

4. Crime Index [28,29], which uses Pandas to filter a DataFrame of city population statistics, evaluate a softmax regression based on features in the dataset, and perform a per-city aggregate to compute a crime index. We used a 25GB dataset of statistics.

The other NumPy and Pandas workloads from [28,29] were bottlenecked by a single hash table operation where Weld only provided speedups through a better multi-threaded execution. As we increase the number of threads, each workload becomes increasingly memory-bound. In all cases, Mozart and Weld outperform the native single-threaded library by parallelizing the workload.

**Performance Results.** Figure 3 show the end-to-end result of the five workloads running on 1–32 threads. The unmodified Pandas and NumPy libraries do not support multi-threaded execution. As we increase the number of threads, each workload becomes increasingly memory-bound. In all cases, Mozart and Weld outperform the native single-threaded library by parallelizing the workload.

![Figure 3: Mozart’s performance of the Python workloads.](image)

In most cases, Mozart’s performance nearly matches the performance of Weld and Bohrium, which require replacing library functions. Mozart only requires an external annotation to each function.

In our benchmarks, we found that the "naive" implementations of the Black Scholes and Haversine workloads allocated extraneous amounts of memory by using NumPy operators that materialize a result for each function call: these allocations caused large performance penalties due to demand paging [15]. We thus present results for both the naive implementation and an implementation that uses NumPy’s in-place operators to avoid demand paging.

**Limitations of SAs.** On a single thread, we note that each workload is compute-bound and exhibits little benefit from Mozart’s data movement optimizations. Weld accelerates workloads on a single thread by generating efficient code: for example, in Data Cleaning, Weld generates code to slice strings instead of calling Python’s string slice function, and for Crime Index, Weld implements filtering using fast AVX2-based predication instructions, which Pandas does not use natively. Since Mozart operates over black-box functions, it cannot apply these optimizations.

**Memory Usage.** Figure 4 shows the peak resident memory usage for each workload. Like Weld, Mozart reduces memory usage by preventing full intermediate result materialization. Mozart does this by reusing buffers as described in §5.3 We note that Mozart needs to know which values are intermediate values in the task graph to use this optimization: users specify this by marking values as outputs. Weld exhibits a high memory footprint for Data Cleaning because (unlike Mozart) it needs to encode Python strings as byte-arrays before processing, effectively requiring a clone of the full dataset. Mozart, Weld, and the in-place NumPy implementation both uses less memory than the naive NumPy implementation because they do not allocate an array for every operator.

![Figure 4: Max memory usage in the Python workloads.](image)

However, perhaps surprisingly, Mozart’s data pipelining and parallelization produces performance improvements competitive with Weld on multiple threads, even though Mozart does not JIT compile optimized code. This is because data movement optimizations become increasingly impactful as the ratio of compute cycles to memory bandwidth increases: by pipelining loops, Mozart reduces the number of main memory accesses, similar to how Weld reduces the number of memory accesses by fusing loops.

**7.2 C Workloads with MKL**

We also evaluated Mozart by adding SAs for Intel MKL’s vector functions. MKL internally parallelizes each of its individual functions, so adding SAs to it shows the impact of data movement optimizations in a parallel library. Because MKL is closed-source, this integration also shows that our implementation is a truly a black-box approach that only requires access to C function headers. Finally, MKL is highly tuned for x86, with many of its operators likely written in assembly [24]. The library is
Figure 5: Mozart’s performance vs. Intel MKL and Weld on the Black Scholes and Haversine Distance workloads. Mozart’s data movement optimizations enable 10× speedups on 32 threads.

Figure 6: Mozart’s performance against unmodified Pandas on the timestamp workload. With Mozart, we can parallelize the workload only by adding an annotation and writing a splitter function in Python.

Figure 7: Speedup of calling `Func(len, arr, arr)` 10 times using MKL with and without SAs. The array had $2^{30}$ double values. The plot shows that SAs are most impactful on workloads with memory-bound operators.

7.3 Benchmarks Not Supported by Current IRs

A key advantage of SAs is that they reuse code that library developers have already written rather than requiring developers to rewrite existing functionality. To exhibit this, we ran a Pandas workload from the Pandas cookbook [38] that parses a CSV containing UNIX timestamps and events, casts the timestamp into a date, and then filters events based on the date. The workload filters the timestamp by comparing an integer representing the UNIX time and a string representing a date (e.g., 1525231405 => ‘01-01-1970’). Weld does not support parsing CSVs or date comparisons between integers and strings in its Pandas port, and also does not support splitting a text file by newline. Porting these computations to run in Weld is difficult and would likely rely on UDFs, since Weld treats strings as byte arrays. In contrast, to parallelize the workload with Mozart, we just added an annotation on each Pandas function used in the workload, and wrote a splitter function in Python that splits a string buffer by newline by offsetting into the string.

Figure 6 shows the result on a 4GB dataset, comparing single-threaded Pandas vs. Mozart parallelizing the workload with 2–32 threads. By only adding 11 lines of code, we were able to parallelize the existing Pandas function calls to improve performance by 4× (the computation is then bottlenecked from pickling Python objects for multiprocessing): existing runtime systems would require porting these library-specific features to an IR.

7.4 Computational Intensity vs. Speedup

To study the relation between compute-boundedness and Mozart’s data movement optimizations, we also ran an artificial micro-benchmark that runs an MKL 1D-vector math function 10 times on an 8GB array. We picked several functions with varying computational intensity (we measured the intensity of each function by calling it repeatedly in a tight loop on a small array that fits in the L1 cache). We benchmarked the following floating-point operations, in order of increasing intensity: add, multiply, square root, division, error function, and exponentiation.
Figure 7a shows the results, plotting the speedup of Mozart over MKL on 1–32 threads: the legend is sorted in order of increasing intensity. The results confirm that Mozart has the largest impact on memory-intensive workloads by exhibiting decreasing speedups as the function called becomes more expensive. Figure 7b plots the normalized relative intensities of the functions we benchmarked.

7.5 Integration Effort

Table 1 summarizes the total integration effort for using Mozart with Pandas and NumPy, and MKL. Both integrations add an SA for each function we used in our library workloads. For MKL, we annotated every 1D-array vector function. We also added splitter implementations for NumPy’s ndarray and Pandas’ DataFrame and Series. For ndarray, our splitters used the builtin slice operators to create views into an existing array. Since DataFrame and Series are implemented on top of ndarray, the splitter implementation is similar. For MKL, we wrote C splitter functions for double* arrays and array lengths.

Overall, annotations were 3–5 lines of code and the splitters functions required to cover every workload across each library was less than 200 lines of code (in comparison, Weld required an average of 22 lines of code per operator for Pandas [29], after adding over 600 lines of “glue” code for marshalling, etc.). We thus found the effort of using Mozart to be low compared to runtimes that require rewriting library functions, for much of the same benefits with multi-threaded execution.

8 Related Work

Splitability annotations are related to several prior works on accelerating existing software by using parallel runtimes or interfaces.

SAs are influenced by work on building new common runtimes or IRs for data analytics [22][29][37][42] and machine learning [1][36][39]. Weld [29] and Delite [54] are two specific examples of systems that use a common IR to detect parallel patterns and automatically generate parallel code. Although SAs do not generate code, we show in §7 that in a parallel setting that the most impactful optimizations are the data movement ones, so SAs can achieve competitive performance without requiring developers to replace code. Systems that provide API-compatible replacements for existing libraries such as Bohrium [20] also have completely re-engineered backends: in contrast, SAs serve to accelerate code that has not been ported to a new system without modifying existing functions.

Several existing works provide black-box optimizations and automatic parallelization of functions. Numba [27] JITs code using a single decorator, while Pydron [25], Dask [12] and Ray [26] automatically parallelize Python code for multi-cores and clusters. In C, work such as Cilk [4] and OpenMP [11] automatically parallelize loops. Unlike these systems, in addition to parallelization SAs enable data movement optimizations on compositions of functions and support split types.

The optimizations SAs enable have been studied before: Vectorwise [43] and other vectorized databases [5][9][21] apply the same pipelining and parallelization techniques as SAs for improved cache locality. Unlike these databases, Mozart applies these techniques on a diverse set of black-box libraries and also reason about the safety of pipelining different functions using split types. SAs are also influenced by prior work in the programming languages community on automatic loop tiling and pipelining [7][17].

Finally, the SA splitting abstraction is conceptually related to Spark’s partitioners [55] and Scala’s parallel collections API [30]. Scala’s parallel collections API in particular features a Splitter and Combiner that partition and aggregate a data type, respectively. Unlike parallel collections, SAs enable pipelining and also introduce split types, which reason about the safety of black-box functions: Scala’s collections API still requires understanding the internal implementation of each collection. Spark’s Partitioners similarly do not enable pipelining.

9 Conclusion

Data movement is a significant bottleneck for data-intensive applications that combine functions from existing libraries. Although researchers have developed languages and runtimes that apply data movement optimizations on existing workflows, they often require intrusive changes to the libraries themselves. We introduced a new black-box approach called splitability annotations (SAs), which specify how to safely split data and pipeline it through a parallel computation to reduce data movement. We showed that SAs require no changes to existing library functions, are easy to integrate, and provide performance benefits competitive with clean-slate approaches in many cases.

References

[1] Abadi, Martín and Barham, Paul and Chen, Jianmin and Chen, Zhifeng and Davis, Andy and Dean, Jeffrey and Devin, Matthieu and Ghemawat, Sanjay and Irving, Geoffrey and Isard, Michael and others. TensorFlow: A System for Large-Scale Machine Learning. In OSDI, volume 16, pages 265–283, 2016.
[34] sharedmem. https://github.com/rainwoodman/sharedmem

[35] Spark Partitioner. https://spark.apache.org/docs/2.2.0/api/java/org/apache/spark/Partitioner.html, 2014.

[36] Sujeeth, Arvind and Lee, HyoukJoong and Brown, Kevin and Rompf, Tiark and Chafi, Hassan and Wu, Michael and Atreya, Anand and Odersky, Martin and Olukotun, Kunle. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[37] Sujeeth, Arvind K and Brown, Kevin J and Lee, Hyoukjoong and Rompf, Tiark and Chafi, Hassan and Odersky, Martin and Olukotun, Kunle. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.

[38] Pandas Cookbook Chapter 8: Parsing Unix timestamps. http://nbviewer.jupyter.org/github/jvns/pandas-cookbook/blob/master/cookbook/

[39] TVM: An End to End IR Stack for Deploying Deep Learning Workloads on Hardware Platforms. http://tvmlang.org/2017/08/17/tvm-release-announcement.html, 2017.

[40] Wulf, Wm A and McKee, Sally A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[41] TensorFlow XLA. https://www.tensorflow.org/performance/xla, 2018.

[42] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[43] Zukowski, Marcin and van de Wiel, Mark and Boncz, Peter. Vectorwise: A vectorized analytical DBMS. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1349–1350. IEEE, 2012.