Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations

Shoumik Palkar and Matei Zaharia
Stanford University

Abstract

Data movement between main memory and the CPU is a major bottleneck in parallel data-intensive applications. In response, researchers have proposed using compilers and intermediate representations (IRs) that apply optimizations such as loop fusion under existing high-level APIs such as NumPy and TensorFlow. Even though these techniques generally do not require changes to user applications, they require intrusive changes to the library itself: often, library developers must rewrite each function using a new IR. In this paper, we propose a new technique called *split annotations (SAs)* that enables key data movement optimizations over unmodified library functions. SAs only require developers to *annotate* functions and implement an API that specifies how to partition data in the library. The annotation and API describe how to enable cross-function data pipelining and parallelization, while respecting each function's correctness constraints. We implement a parallel runtime for SAs in a system called Mozart. We show that Mozart can accelerate workloads in libraries such as Intel MKL and Pandas by up to 15×, with *no* library modifications. Mozart also provides performance gains competitive with solutions that require rewriting libraries, and can sometimes outperform these systems by up to 2× by leveraging existing hand-optimized code.

ACM Reference Format:
Shoumik Palkar and Matei Zaharia. 2019. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359652

1 Introduction

Developers build software by composing optimized libraries and functions written by other developers. For example, a typical scientific application may compose routines from hand-optimized libraries such as Intel MKL [5], while machine learning practitioners build workflows using a rich ecosystem of Python libraries such as Pandas [52] and PyTorch [22]. Unfortunately, on modern hardware, optimizing each library function in isolation is no longer enough to achieve the best performance. Hardware parallelism such as multicore, SIMD, and instruction pipelining has caused computational throughput to outpace memory bandwidth by an order of magnitude over several decades [44, 51, 66]. This gap has made *data movement* between memory and the CPU a fundamental bottleneck in data-intensive applications [55].

In recognition of this bottleneck, researchers have proposed redesigning software libraries to use optimizing compilers and runtimes [27, 46, 48, 56, 61, 63–65]. For example, Weld [56] and XLA [7] are two recent compilers that propose rewriting library functions using an intermediate representation (IR) to enable cross-function data movement optimization, parallelization, and JIT-compilation. In both, data movement optimizations such as loop fusion alone have shown improvements of two orders of magnitude in real data analytics pipelines [8, 55].

Although this compiler-based approach has shown promising results, it is highly complex to implement. This manifests in two major disadvantages. First, leveraging these compilers requires highly intrusive changes to the library itself. Many of these systems [7, 34, 36, 46, 56] require reimplementing each operator in entirety to obtain any benefits. In addition, the library must often be redesigned to "thread a compiler" through each function by using an API to construct a dataflow graph (e.g., TensorFlow Ops [10] or Weld's Runtime API [56]). These restrictions impose a large burden of effort on the library developer and hinder adoption. Second, the code generated by compilers might not match the performance of code written by human experts. For example, even state-of-the-art compilers tailored for linear algebra [7, 34, 61] generate convolutions and matrix multiplies that are up to 2× slower than hand-optimized implementations [3, 14, 18]. Developers thus face a tough choice among expanding these complex compilers, dropping their own optimizations, or forgoing optimization across functions.

In this paper, we propose a new technique called *split annotations*, which provides the data movement and parallelization optimizations of existing compilers and runtimes *without* requiring modifications to existing code. Unlike prior work that requires reimplementing libraries, our technique only requires an *annotator* (e.g., a library developer or third-party programmer) to annotate functions with a split annotation

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359652

(SA) and to implement a *splitting API* that specifies how to split and merge data types that appear in the library. Together, the SAs and splitting API define how data passed into an unmodified function can be partitioned into cache-sized chunks, pipelined with other functions to reduce data movement, and parallelized automatically by a runtime transparent to the library. We show that SAs yield similar performance gains with up to 17× less code compared to prior systems that require function rewrites, and can even outperform them by as much as 2× by leveraging existing hand-optimized functions.

There are several challenges in enabling data movement optimization and automatic parallelization across black-box library functions. First, a runtime cannot naïvely pipeline data among all the annotated functions: it must determine whether function calls operating over split data are compatible. For example, a function that operates on rows of pixels can be split and pipelined with other row-based functions, but not with an image processing algorithm that operates over patches (similar to determining valid operator fusion rules in compilers [7, 38, 46, 56]). This decision depends not on the data itself, but on the shape of the data (e.g., image dimensions) at runtime. To address this challenge, we designed the SAs around a type system with two goals: *(1)* determining how data is split and merged, and *(2)* determining which functions can be scheduled together for pipelining. Annotators use SAs to specify a *split type* for each argument and return value in a function. These types capture properties such as data dimensionality and enable the runtime to reason about function compatibility. For each split type, annotators implement a *splitting API* to define how to split and merge data types.

Second, in order to pipeline data across functions, the runtime requires a lazily evaluated dataflow graph of the annotated functions in an application. While existing systems [17, 32, 46, 55] require library developers to change their functions to explicitly construct and execute such a graph using an API, our goal is to enable these optimizations without library modifications. This is challenging since most applications will not only make calls to annotated library functions, but also execute arbitrary un-annotated code that cannot be pipelined. To address this challenge, we designed a client library called `libmozart` that captures a dataflow graph from an existing program at runtime and determines when to execute it with no library modification, and minor to no changes to the user application. We present designs for `libmozart` for both C++ and Python. Our C++ design generates transparent wrapper functions to capture function calls lazily, and uses memory protection to determine when to execute lazy values. Our Python design uses function decorators and value substitution to achieve the same result. In both designs, `libmozart` requires no library modifications.

Finally, once the client library captures a dataflow graph of annotated functions, a runtime must determine how to execute it efficiently. We designed a runtime called *Mozart* that uses the split types in the SAs and the dependency information in the dataflow graph to split, pipeline, and parallelize functions while respecting each function's correctness constraints.

We evaluate SAs by integrating them with several data processing libraries: NumPy [20], Pandas [52], spaCy [24], Intel MKL [5], and ImageMagick [15]. Our integrations require up to 17× less code than an equivalent integration with an optimizing compiler. We evaluate SAs' performance benefits on the data science benchmarks from the Weld evaluation [55], as well as additional image processing and numerical simulation benchmarks for MKL and ImageMagick. Our benchmarks demonstrate the generality of SAs and include options pricing using vector math, simulating differential equations with matrices and tensors, and aggregating and joining SQL tables using DataFrames. End-to-end, on multiple threads, we show that SAs can accelerate workloads by up to 15× over single-threaded libraries, up to 5× compared to already-parallelized libraries, and can *outperform* compiler-based approaches by up to 2× by leveraging existing hand-tuned code. Our source code is available at https://www.github.com/weld-project/split-annotations.

Overall, we make the following contributions:

1. We introduce split annotations, a new technique for enabling data movement optimization and automatic parallelization with no library modifications.
2. We describe `libmozart` and Mozart, a client library and runtime that use annotations to capture a dataflow graph in Python and C code, and schedule parallel pipelines of black-box functions safely.
3. We integrate SAs into five libraries, and show that they can accelerate applications by up to 15× over the single-threaded version of the library. We also show that SAs provide performance competitive with existing compilers.

2 Motivation and Overview

Split annotations (SAs) define how to split and pipeline data among a set of functions to enable data movement optimization and automatic parallelization. With SAs, an *annotator*—who could be the library developer, but also a third-party developer—annotates functions and implements a splitting API that defines how to partition data types in a library. Unlike prior approaches for enabling these optimizations under existing APIs, SAs require no modification to the code library developers have already optimized. SAs also allow developers building new libraries to focus on their algorithms and domain-specific optimizations rather than on implementing a compiler for enabling cross-function optimizations. Our annotation-based approach is inspired by the popularity of systems such as TypeScript [25], which have demonstrated that third-party developers can annotate existing libraries (in TypeScript's case, by adding type annotations [26]) and allow other developers to reap their advantages.

```
// inputs are `double` arrays with `len` elements
vdLog1p(len, d1, d1);          // d1 = log(d1)
vdAdd(len, d1, tmp, d1);       // d1 = d1 + tmp
vdDiv(len, d1, vol_sqrt, d1); // d1 = d1 / vol_sqrt
```

Listing 1. Snippet from the Black Scholes options pricing benchmark implemented using Intel MKL.

```
@splittable(
  size: SizeSplit(size), a: ArraySplit(size),
  mut out: ArraySplit(size))
void vdLog1p(long size, double *a, double *out);

@splittable(
  size: SizeSplit(size), a: ArraySplit(size),
  b: ArraySplit(size), mut out: ArraySplit(size))
void vdAdd(long size, double *a,
           double *b, double *out);
void vdDiv(long size, double *a,
           double *b, double *out);
```

Listing 2. SAs for three functions in Intel MKL.

2.1 Motivating Example: Black Scholes with MKL

To demonstrate the impact of SAs, consider the code snippet in Listing 1, taken from an Intel MKL implementation of the Black Scholes options pricing benchmark. This implementation uses MKL's parallel vector math functions to carry out the main computations. Each function parallelizes work using a framework such as Intel TBB [62] and is hand-optimized using SIMD instructions. Unfortunately, when combining several of these functions on a multicore CPU, the workload is bottlenecked on *data movement*.

The data movement bottleneck arises in this workload because each function completes a full scan of every input array, each of which contains `len` elements. For example, the call to `vdLog1p`, which computes the element-wise logarithm of the `double` array `d1` in-place, will scan `sizeof(double)*len` bytes: this will often be much larger than the CPU caches. The following call to `vdAdd` thus cannot exploit any locality of data access, and must *reload values for `d1` from main memory*. Since individual MKL functions only accept lengths and pointers as inputs, their internal implementation has no way to prevent these loads from main memory *across* different functions. On modern hardware, where computational throughput has outpaced memory bandwidth between main memory and the CPU by over an order of magnitude [44, 51, 66], this significantly impacts multicore scalability and performance, even in applications that already use optimized libraries.

SAs and their underlying runtime, Mozart, address this data movement bottleneck by splitting function arguments into smaller pieces and pipelining them across function calls. Listing 2 shows the SAs an annotator could provide for the three functions in Listing 1. We describe the SAs fully in §3, but at a high level, annotators use the SA and a new abstraction called split types to define how to split each function argument into small pieces. For example, the annotator can define a split type `ArraySplit` to indicate that the array arguments be split into smaller, regularly-sized arrays. The split type `SizeSplit` then indicates that the size argument be split to represent the lengths of these arrays. Annotators then bridge the abstraction of the split type with code that performs the splitting (e.g., by offsetting into the original array pointer) by implementing a *splitting API* for each split type.

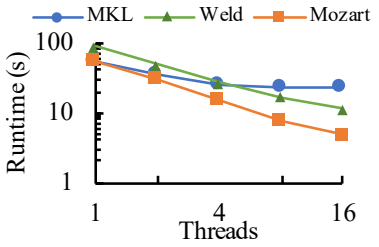


Figure 1. Performance of the Black Scholes benchmark on 1–16 threads with MKL, Weld, and MKL with Mozart.

Given these SAs, Mozart executes the code in Listing 1 in a markedly different way. Instead of calling each MKL function on the full input arrays at once, Mozart splits each array into small, equally-sized chunks that collectively fit in the CPU cache (e.g., chunks of 4096 elements per array). Mozart then assigns these chunks across threads, and has each thread call all the functions in the pipeline in sequence (`vdLog1p`, `vdAdd`, etc.) on one chunk at a time. Data for each chunk resides in each CPU's local cache across all functions. Although the total number of loads and stores remains unchanged (i.e., across all chunks, each function still loads and processes all elements of `d1`, `tmp`, and `vol_sqrt`), each array element is loaded from main memory *only once* and served from cache for all subsequent accesses. This is a stark reduction compared to the original execution, which loads each array from main memory for each function call.

Figure 1 shows the impact of SAs on the full Black Scholes benchmark, which contains 32 vector operations, on a modern Intel Xeon CPU. The benchmark runs on 11GB of data. While un-annotated MKL bottlenecks on memory at around four threads, Mozart scales to all the cores on the machine. In this benchmark, Mozart also *outperforms* the optimizing Weld compiler, which applies optimizations such as loop fusion to reduce data movement by keeping data in CPU registers. We found this was because Weld does not generate vectorized code for several operators that MKL does vectorize. Although it is feasible to extend Weld to include SIMD versions of these operators, this benchmark is one example of the advantages of leveraging code that developers have already hand-optimized.

To enable these performance improvements, Mozart first captures a dataflow graph of annotated functions called in

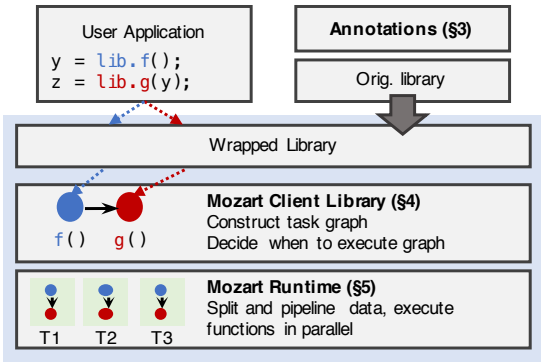


Figure 2. Overview of Mozart.

the application (Figure 2). Since our goal is to leave libraries unmodified, we wish to capture such a graph without an explicit API that libraries implement. The *`libmozart` client library* (§4) transparently handles constructing such a graph and determining when to execute it, using a combination of auto-generated wrapper functions and memory protection in C/C++ and function decorators in Python. Mozart then runs a planning step to determine which functions to pipeline, and then executes tasks using the SAs and dataflow graph (§5).

2.2 Limitations and Non-Goals

SAs have a number of restrictions and non-goals. First, since SAs make repeated calls to black-box functions to achieve pipelining and parallelism, they are limited to functions that do not cause side effects or hold locks. Second, SAs do not apply the compute-based optimizations (e.g., common subexpression elimination or SIMD vectorization) that systems like Weld and TensorFlow XLA can provide by re-implementing functions in an IR. Nevertheless, because data movement has been shown to be one of the major bottlenecks in modern applications, we show that SAs can provide speedups competitive with these compilers without rewriting functions.

3 Split Annotation Interface

In this section, we present the split annotation (SA) interface. We first discuss the challenges in pipelining arbitrary functions to motivate our design. We then introduce *split types* and *split annotations*, the core abstractions used to address these challenges. We conclude this section by describing the *splitting API* that annotators implement to bridge these abstractions with code to split and merge data.

3.1 Why Split Types?

Split types are necessary because some data types in the user program can be split in multiple ways. As an example, matrix data can be split into collections of either rows or columns, or can even be partitioned into blocks. However, some functions require that data be split in a specific way to ensure correctness. As an example of this, consider the program below, which uses a function to first normalize the rows of a matrix (indicated via `axis=0`), followed by the columns of a matrix.

```
normalizeMatrixAxis(matrix, axis=0);
normalizeMatrixAxis(matrix, axis=1);
```

In the first call, the matrix must be split by rows, since the function requires access to all the data in a row when it is called. Similarly, the second call requires that the matrix be split into columns. Split types allow distinguishing these cases, even when they depend on runtime values (e.g., `axis`). Beyond ensuring correctness, split types also enable pipelining data split in the same way across functions.

3.2 Split Types and Split Annotations

A *split type* is a parameterized type $N\langle V_0 \dots V_n \rangle$ defined by its name N and a set of parameter values $V_0 \dots V_n$. A split type defines how a function argument is split. Two split types are equal if their names and parameters are equal. If the split types for two arguments are equal, it means that they are split in the same way, and their corresponding pieces can be passed into a function together. [1] Each split type is associated with a single concrete data type (e.g., an integer, array, etc.).

The library annotator decides what a "split" means for the data types in her library. As an example, an annotator for MKL's vector math library, which operates over C arrays, can choose to split arrays into multiple regularly-sized pieces. The annotator can then define a split type $\mathit{ArraySplit}\langle \mathit{int}, \mathit{int} \rangle$ that uniquely specifies how an array is split with two integer parameters: the length of the full array and the number of pieces the array is split into. An array with 10 elements can be split into two length-5 pieces with a split type $\mathit{ArraySplit}\langle 10, 2 \rangle$, or into five length-2 pieces with a split type $\mathit{ArraySplit}\langle 10, 5 \rangle$: these splits have *different* split types since their parameters are different, even if the underlying data refers to the same array. In the remainder of this section, we omit the parameter representing the number of pieces, because *(1)* every split type depends on it, and *(2)* Mozart sets it automatically (§5) and guarantees its equality for split types it compares.

A *split annotation (SA)* is then an annotation over a side-effect-free function that assigns a name and a split type to each of the function's arguments and its return value. Listing 3 shows the full syntax of a single SA. For arguments that should not be split (and thus copied to each pipeline), annotators can give them a "missing" split type, denoted with "`_`" (e.g., "`arg: _`"). In addition to providing split types, the SA specifies which of the function arguments are *mutable* using the `mut` tag, which Mozart uses to detect data dependencies between functions when building a dataflow graph (§4). Listing 2 shows SAs for MKL array functions. Taking the `vdAdd` function as an example, the SA assigns the names `size`, `a`, `b`, and `out` to the arguments and assigns each a split

[1] Since type equality depends not only on the type name but also on the parameter values, split types are formally *dependent types* [12].

```
@splittable(
  [mut] <arg1-name>: [<arg1-split-type>|_], ...
) [-> <ret-split-type>]
/* one or more functions */
```

Listing 3. Full syntax of a split annotation.

type. The SA marks `out` as **`mut`** to indicate that the function mutates this argument.

Split Type Constructors. One subtlety in writing SAs arises due to the split types' parameters. Even though a split type's name is known when the annotator writes an SA, its parameters will generally not be known until *runtime*. For example, in the *ArraySplit* split type defined above, the length of an array will not be known until the program executes. This creates a challenge because Mozart needs to know the full split type including the concrete values of its parameters.

To address this, a split type can use function arguments to compute its parameters at runtime. Specifically, for an SA over a function F, each split type in the SA uses a *constructor* $A_0 \ldots A_n \Rightarrow V_0 \ldots V_n$ to construct its parameters, where $A_0 \ldots A_n$ are zero or more arguments of F. Within an SA, we use the syntax **`Name`**`(A0...An)` to refer to a constructor for a split type with a name **`Name`** that constructs its parameters with function arguments `A0...An`, where `A0...An` are names assigned to arguments in the SA. Note that the split type constructor is a part of the *splitting API* that annotators implement for each split type—we discuss this API further in §3.3. Unless otherwise noted, we assume in this paper that split types use the identity function $A_0 \ldots A_n \Rightarrow A_0 \ldots A_n$ as their constructor.

As an example, consider Ex. 1 in Listing 4, which takes a matrix argument and an axis that determines whether the function operates over rows or columns (similar to the function in §3.1). We can represent splitting this matrix by either rows or columns by using a split type with three integer parameters called $MatrixSplit\langle int, int, int\rangle$. The parameters represent the matrix dimensions and the axis to iterate over. Within an SA, an annotator can write **`MatrixSplit`**`(m, axis)` to represent this split type: Listing 4 shows the constructor definition for this split type, which maps the matrix and axis into the split type's three parameters. The split type for matrices does not depend on the matrix data itself, since the underlying data does not affect how the matrix is split. The SAs in Listing 2 similarly use the `size` argument (but not the array itself) to construct the *ArraySplit* split type.

With split types, Mozart can determine whether two functions can be pipelined safely. For each annotated function that Mozart captures in a dataflow graph, it initializes the parameters of the split types using the function's arguments. If all the data passed between two functions have matching corresponding split types, they can be pipelined. Otherwise, already-split data must be *merged* and re-split before passing it to the next function to prevent pipelining errors. Returning to the example from §3.1, we can use the split type from Ex. 1 in Listing 4 to assign the matrix arguments the split types $MatrixSplit\langle rows, cols, 0\rangle$ and $MatrixSplit\langle rows, cols, 1\rangle$: since these split types do not match, Mozart will not pipeline them.

```
// Parameters are (rows, cols, axis)
splittype MatrixSplit(int, int, int)
// Constructor for MatrixSplit
MatrixSplit(m, axis) => (m.rows, m.cols, axis)

// Ex. 1: Normalize along an axis in a matrix.
@splittable(mut m: MatrixSplit(m, axis), axis: _)
void normalizeMatrixAxis(matrix m, int axis);

// Ex. 2: Add two matrices element-wise.
@splittable(left: S, right: S) -> S
matrix add(matrix left, matrix right);

// Ex. 3: Scale a matrix element-wise.
@splittable(mut m: S, val: _)
void scaleMatrix(matrix m, double val);

// Ex. 4: Remove zero-valued rows from a matrix.
@splittable(m: S) -> unknown
matrix filterZeroedRows(matrix m);

// Ex. 5: Reduce a matrix to a vector by summing.
@splittable(m: MatrixSplit(m, axis), axis: _)
  -> ReduceSplit(axis)
vector sumReduceToVector(matrix m, int axis);
```

Listing 4. Examples of SAs over matrices. Ex. 1 shows concrete split types, Ex. 2-3 show generics, Ex. 4 shows unknown split types, and Ex. 5 shows a reduction function.

Generics. SAs also support assigning generics to an argument. Generics in an SA are similar to generic types in languages such as Java or Rust: if two generics within an SA have the same name (e.g., `S`), the runtime ensures that the split types they are assigned are equal. Names for generics are local to an SA. Generics across SAs are propagated via *type inference* (§5), another common feature of existing type systems [28, 58].

Ex. 2 and 3 in Listing 4 show generics. Ex. 2 shows a function that adds two matrices element-wise: if `left` and `right` are split in the same way (indicated by their matching generic `S`), the function can process them together.

Unknown Split Type. Some functions will *change* the split type of a value in an unknown way upon execution. Ex. 4 in Listing 4 shows an example. The `filterZeroedRows` function changes the dimensions of its input, so its output split type is unknown after the call. We represent this in an SA using a special split type **`unknown`**, which represents a *unique* split

type. Uniqueness prevents pipelining `unknown` values with any other split value: for example, if we tried to pass two `unknown` values to `add`, the types would not match, thus preventing pipelining. However, generic functions such as `scaleMatrix` (Ex. 3), which take a *single* argument split in any way, can still accept `unknown` values. Generics and `unknown` together enable SAs to support operators such as filters, which are common in data-processing libraries like Pandas.

3.3 Splitting and Merging with the Splitting API

Annotators bridge the split type abstraction with an implementation using the splitting API. This API has several roles: it provides the constructor for constructing parameters, defines how to split data, and defines how to merge split pieces back into a full result. Table 1 summarizes these functions.

Constructor. The constructor maps values that will appear as function arguments to the split type's parameters. In our *ArraySplit* example, the constructor takes a `long` value representing an array's size and returns that size as its parameter. The constructor should not modify its arguments.

***Split* Function.** The split function performs the splitting operation using the original function argument and the split type parameters. It returns a split value representing the range $[start, end)$ in the function argument. Returning to the MKL *ArraySplit* example, the split function would return pointers offset from the base array pointer. Mozart dynamically selects the number of elements in $[start, end)$ and ensures that end does not surpass the total number of items in the argument. The `Info` function relays information to Mozart for this (§5). In our implementation, the split function also takes additional parameters such as a thread ID and the number of threads. This allows splits that are not based on integer ranges.

***Merge* Function.** The associative merge function coalesces split pieces into a single value once Mozart finishes processing them. In our MKL SAs, updates occur in-place, so no merge operation is needed, but if each function returned a new array instead, the merge function could concatenate the split arrays into a final result. This function can also be used to perform operations such as reductions in parallel using SAs.

Ex. 5 in Listing 4 shows an example of a reduction operator that collapses a matrix into a vector by summing either its rows or columns. The input matrix is split using *MatrixSplit* defined earlier, but the result is a new split type $ReduceSplit\langle axis\rangle$ that represents partial results. In particular, *ReduceSplit* represents either reduced row values or column values, depending on `axis`: its merge function uses `axis` to reconstruct either a row-vector or a column-vector.

3.4 Conditions to Use SAs

To summarize, a side-effect-free function $F(a, b, \ldots) \rightarrow c$ can be annotated with an SA with split types $(A, B, \ldots) \rightarrow C$ if:

Splitting API Summary (§3.3)
`NameConstructor(A0,...An)=> Parameters`
`Split(D arg, int start, int end, Parameters)=> D`
`Merge(Vector<D>, Parameters)=> D`
`Info(D arg, Parameters)=> RuntimeInfo`

Table 1. The API annotators implement for a split type $Name\langle Parameters\rangle$. The argument has a data type `D`.

$$F(a, b, \ldots) = Merge_C(F(a_1, b_1, \ldots), F(a_2, b_2, \ldots), \ldots)$$

where $Split_A(a) \rightarrow [a_1, a_2, \ldots]$ is the split function for split type A and $Merge_C(c_1, c_2, \ldots)$ is the associative merge function for split type C. There are no constraints on the number of splits each split function produces, as long as all split functions produce the same number of splits for a given function.

3.5 Summary: How Annotators Use SAs

To annotate a library, an annotator first decides how to split the library's core data types. She then defines split types and implements their splitting APIs. The annotator then writes SAs for side-effect-free functions using the defined split types. For functions that perform reductions or require custom merge operations, the annotator implements per-function split types that only implement the merge function. In our integrations, we required up to three split types for the core data types per library (e.g., DataFrames in Pandas), and one split type per reduction function. We generated most SAs using a script, since functions with matching function signatures can share the same SA. We discuss effort of integration in detail in §8.

3.6 Generality of SAs

A split type can capture a variety of data formats by splitting inputs and enabling pipelining and parallelization because annotators define their splitting behavior. We show in our library integrations in §7 and in our evaluation in §8 that SAs can split and optimize numerical and scientific workloads that use arrays, matrices, and tensors, SQL-like workloads that use DataFrames for operations ranging from projections and selections to groupBys and joins, image processing workloads, and natural language processing workloads. We note that, because SAs primarily aim to accelerate data parallel workloads that can be pipelined (i.e., cases where data movement optimizations in IRs are most applicable), they will be most impactful over collection-like data.

4 The Mozart Client Libraries

Mozart relies on a lazily evaluated dataflow graph to enable cross-function data movement optimizations. Nodes in the dataflow graph represent calls to annotated functions and their arguments, and edges represent data passed between functions. Constructing such a graph without library modifications

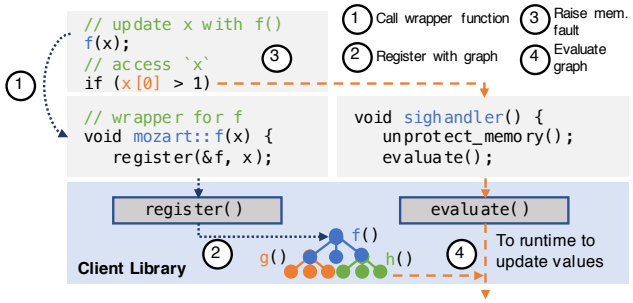


Figure 3. Overview of the C++ client library.

(e.g., by using an API as in prior work [10, 57]) is challenging because applications will contain a mix of annotated function calls and arbitrary code that may access lazy values. The `libmozart` client library is responsible for capturing a graph and determining when to evaluate it. The library has a small interface: *register(function, args)* registers a function and its arguments with the dataflow graph, and *evaluate()* evaluates the dataflow graph (§5 describes the runtime) when arbitrary code accesses lazy values. Since the `libmozart` design is coupled with the annotated library's language, we discuss its design in two languages: C++ and Python.

4.1 C++ Client Library

Our C++ client library uses code generation and OS-level memory protection to build a dataflow graph and to determine when to evaluate it. Figure 3 outlines its design.

Writing Annotations. An annotator registers split types, the splitting API, and SAs over C++ functions by using a command line tool we have built called `annotate`. This tool takes these definitions as input and generates namespaced C++ *wrapper functions* around each annotated library function. These wrapper functions are packaged with the splitting API and a lookup table that maps functions to their SAs in a shared library. The application writer links this wrapped library and calls the wrapper functions instead of the original library functions as always. This generally requires a namespace import and no other code changes–we note one exception below.

Capturing a Graph. The wrapper functions are responsible for registering tasks in the dataflow graph. When an application calls a wrapper, its function arguments are copied into a buffer, and the `libmozart` *register* API adds the function and its argument buffer as a node in the dataflow graph. The wrapper knows which function arguments are mutable, based on which arguments in the SA were marked **`mut`** (the SA is retrieved using the lookup table). This allows `libmozart` to add the correct data-dependency edges between calls.

Determining Evaluation Points. We evaluate the dataflow graph upon access to lazy values. There are two cases: *(1)* the accessed value was returned by an annotated function, and *(2)* the accessed value was allocated outside of the dataflow graph but mutated by an annotated function.

To handle the first case, if the library function returns a value, its wrapper instead returns a type called `Future<T>`. For types where `T` is a pointer, `Future<T>` is a pointer with an overloaded dereference operator that first calls *evaluate* to evaluate the dataflow graph. For non-pointer values, the value can be accessed explicitly with a `get()` method, which forces the execution. We also override the copy constructor of this type to track aliases, so copies of a lazy value can be updated upon evaluation. Wrapper functions can accept both `Future<T>` values and `T` values, so `Future` values may be pipelined. Usage of `Future<T>` is the main code change applications must make when using SAs in C++.

We handle the second case (shown in Figure 3 in the access to `x[0]`) by using memory protection to intercept reads of lazy values. Applications must use our drop-in `malloc` and `free` functions for memory accessed in the dataflow graph (e.g., via `LD_PRELOAD`). Our version of `malloc` uses the `mmap` call to allocate memory with `PROT_NONE` permissions, which raises a protection violation when read or written. `libmozart` registers a signal handler to catch the violation, unprotects memory, and evaluates the dataflow graph registered so far. When calling a wrapper function for the first time after evaluation, `libmozart` re-protects all allocated memory to re-enable capturing memory accesses. This technique has been used in other systems successfully [45, 46] to inject laziness.

4.2 Python Client Library

Writing Annotations. Developers provide SAs by using Python function decorators. In Python, split types for positional arguments are required, and split types for keyword arguments default to "_" (but can be overridden).

Capturing a Graph. `libmozart` constructs the dataflow graph using the same function decorator used to provide the SA. The decorator wraps the original Python function into one that records the function with the graph using *register()*. The wrapper function then returns a placeholder `Future` object.

Determining Evaluation Points. Upon accessing a `Future` object, `libmozart` evaluates the task graph. In Python, we can detect when an object is accessed by overriding its builtin methods (e.g., `__getattribute__` to get object attributes). After executing the task graph, the `Future` object forwards calls to these methods to the evaluated cached value. To intercept accesses to variables that are mutated by annotated functions (based on **`mut`**), when `libmozart` registers a mutable object, it overrides the object's `__getattribute__` to again force evaluation when the object's fields are accessed. The original `__getattribute__` is reinstated upon evaluation.

5 The Mozart Runtime

Mozart is a parallel runtime that executes functions annotated with SAs. Mozart takes the dataflow graph generated by the client library and converts it into an *execution plan*. Specifically, Mozart converts the dataflow graph into a series

of *stages*, where each stage splits its inputs, pipelines the split inputs through library functions, and then merges the outputs. The SAs dictate the stage boundaries. Mozart then executes each stage over *batches* in parallel, where each batch represent one set of split inputs.

5.1 Converting a Dataflow Graph to Stages

Recall that each node in the dataflow graph is an annotated function call, and each edge from function f_1 to f_2 represents a data dependency, i.e., a value mutated or returned by f_1 and read by f_2. Mozart converts this graph into stages. The functions f_1 and f_2 are in the same stage if, for every edge between them, the source value and destination value have the same split type. If *any* split types between f_1 and f_2 do not match, split data returned by f_1 must be merged, and a new stage starts with f_2. Mozart traverses the graph and checks types to construct stages.

To check split types between a source and destination argument, Mozart first checks that the split types have the same name. If the names are equal, Mozart uses the function arguments captured as part of the dataflow graph to initialize the split types' parameters. If the parameters also match, the source and destination have the same split type. If either split type is a generic, Mozart uses type inference [28, 58] to determine its type by pushing known types along the edges of the graph to set generics. If a split type cannot be inferred (e.g., because all functions use generics), Mozart falls back to a default for the data type: in our implementation, annotators provide a default split type constructor per data type.

This step produces an execution plan, where each stage contains an ordered list of functions to pipeline. The inputs to each stage are split based on the inputs' split types, and the outputs are merged before being passed to the next stage.

5.2 Execution Engine

After constructing stages, Mozart executes each stage in sequence by *(1)* choosing a batch size, *(2)* splitting and executing each function, and *(3)* merging partial results.

Step 1: Discovering Runtime Parameters. Mozart sets the number of elements in each batch and the number of elements processed per thread as runtime parameters. Since the goal of pipelining is to reduce data movement, we use a simple heuristic for the batch size: each batch should contain roughly *sizeof(L2 cache)* bytes. To determine the batch size, Mozart calls each input's `Info` function (§3), which fills a struct called `RuntimeInfo`. This struct specifies the number of total elements that will be produced for the input (e.g., number of elements in an array or number of rows in a matrix), and the size of each element in bytes. The batch size is then set to $\frac{C \times L2CacheSize}{\sum sizeof(element)}$ (where C is a fixed constant). We found that this value works well empirically (see §8) when pipelines allocate intermediate split values too, since these values are still small enough to fit in the larger shared last-level-cache.

Workers partition elements equally among themselves. The user configures the number of workers. Mozart checks to ensure that each split produces the same total number of elements. We opted for static parallelism rather than dynamic parallelism (e.g., via work-stealing) because it is simpler to schedule and we found that it leads to similar results for most workloads: however, dynamic work-stealing schedulers such as Cilk [30] are also compatible with Mozart.

Step 2: Executing Functions. After setting runtime parameters, Mozart spawns worker threads and communicates to each thread the range of elements it processes. Each worker allocates thread-local temporary buffers for the split values and enters the main driver loop. The worker's driver loop calls the *Split* function for each input and writes the result into the temporary buffers. If *Split* returns `NULL`, the driver loop exits. For arguments with the missing "_" split type, the original input value is copied (usually, this is just a pointer-copy) rather than split. The *start* and *end* arguments of the *Split* function are set based on the batch size and the thread's range.

To execute the function pipeline per thread, Mozart tracks which temporary buffers should be fed to each function as arguments by using a mapping from unique argument IDs to buffers. The execution plan represents function calls using these argument IDs (e.g., a call $f_1(a0, a1, a2) \rightarrow a3$ will pass the buffers for $a0$, $a1$, and $a2$ as arguments and store the result in the buffer for $a3$). After each batch, these buffers are moved to a list of partial results, and Mozart starts the next batch.

Step 3: Merging Values. Once the driving loop exits, each worker merges each list of temporary buffers via the split type merge function (the stage tracks the split type of each result), and then returns the merged result. Once all workers return their partial results, Mozart calls the merge function again on the main thread to compute the final merged results.

6 Implementation

Our C++ version of `libmozart` and Mozart is implemented in roughly 3000 lines of Rust. This includes the parser for the SAs, the `annotate` tool for generating header files containing the wrapper functions, the client library (including memory protection), planner, and parallel runtime. We use Rust's threading library for parallelism. We make heavy use of the `unsafe` features of Rust to call into C/C++ functions. Memory allocated for splits is freed by the corresponding mergers. Mozart manages and frees temporary memory.

The Python implementation of these components is in around 1500 lines of Python. The SAs themselves use Python's function decorators, and split types are implemented as abstract classes with splitter and merger methods. We use process-based parallelism to circumvent Python's global interpreter lock. For native Python data, we only need to serialize data when communicating results back to the main thread. We leverage copy-on-write *fork()* semantics when starting workers, which also means that "_" values need not be cloned.

7 Library Integrations

We evaluate SAs by integrating them with five popular data processing libraries: NumPy [20], Pandas [52] spaCy [24], Intel MKL [5] and ImageMagick [15]. Table 3 in §8 summarizes effort, and we discuss integration details below.

NumPy. NumPy is a popular Python numerical processing library, with core operators implemented in C. The core data type in the library is the `ndarray`, which represents an N-dimensional tensor. We implemented a single split type for `ndarray`, whose splitting behavior depends on its shape and the axis a function iterates over (the split type's constructor maps `ndarray` arguments to its shape). We added SAs over all tensor unary, binary, and associative reduction operators. We implemented split types for each reduction operator to merge the partial results: these only required merge functions.

Pandas. Our Pandas integration implements split types over DataFrames and Series by splitting by row. We also added a *GroupSplit* split type for `GroupedDataFrame`, which is used for groupBy operations. Aggregation functions that accept this split type group chunks of a DataFrame, create partial aggregations, and then re-group and re-aggregate the partial aggregations in the merger. We only support commutative aggregation functions. We support most unary and binary Series operators, filters, predicate masks, and joins: joins split one table and broadcast the other. Filters and joins return the `unknown` split type, and most functions accept generics.

spaCy. SpaCy is a Python natural language processing library with operators in Cython. We added a split type that uses spaCy's builtin minibatch tokenizer to split a corpus of text. This allows any function (including user-defined ones) that accepts text and internally uses spaCy functions to be parallelized and pipelined via a Python function decorator.

Intel MKL. Intel MKL [5] is an optimized closed-source numerical computation library used as the basis for other computational frameworks [6, 20, 22, 27, 42, 46]. To integrate SAs, we defined three split types: one for matrices (with rows, columns, and order as parameters), one for arrays (with length as a parameter), and one for the size argument. Since MKL operates over inputs in place, we did not need to implement merger functions. We annotated all functions in the vector math header, the saxpy (L1 BLAS) header, and the matrix-vector (L2 BLAS) headers.

ImageMagick. ImageMagick [15] is a C image processing library that contains an API where images are loaded and processed using an opaque handle called `MagickWand`. We implemented a split type for the `MagickWand` type, where the split function uses a crop function to clone and return a subset of the original image. ImageMagick also contains an API for appending several images together by stacking them—our split function thus returns entire rows of an image, and this API is used in the merger to reconstruct the final result.

7.1 Experiences with Integration

Unsupported Functions. We found that there were a handful of functions in each library that we could not annotate. For example, in the ImageMagick library, the `Blur` function contains a boundary condition where the edges of an image are processed differently from the rest of the image. SAs' split/merge paradigm would produce incorrect results here, because this special handling would occur on each split rather than on just the edges of the full image. Currently, annotators must manually determine whether each function is safe to annotate (similar to other annotation-based systems such as OpenMP). We found that this was straightforward in most cases by reading the function documentation, but tools that could formally prove an SA's compatibility with a function would be helpful. We leave this to future work. We did not find any functions that internally held locks or were not callable from multiple threads in the libraries we annotated.

Debugging and Testing. Since annotators must manually enforce the soundness of an SA, we built some mechanisms to aid in debugging and testing them. The `annotate` tool, for example, will ensure that a split type is always associated with the same concrete type. The runtimes for both C and Python include a "pedantic mode" for debugging that can be configured to panic if a function receives splits with differing numbers of elements, receives no elements, or receives `NULL` data. The runtime can also be configured to log each function call on each split piece, and standard tools such as Valgrind or GDB are still available. Anecdotally, the logs and pedantic mode made debugging invalid split/merge functions and errors in SAs unchallenging. We also fuzz tested our annotated functions to stress their correctness.

8 Evaluation

In evaluating split annotations, we seek to answer several questions: *(1)* Can SAs accelerate end-to-end workloads that use existing unmodified libraries? *(2)* Can SAs match or outperform compiler-based techniques that optimize and JIT machine code? *(3)* Where do performance benefits come from, and which classes of workloads do SAs help the most?

Experimental Setup. We evaluate workloads that use the five libraries in §7: NumPy v1.16.2, Pandas v0.22.0, spaCy v2.1.3, Intel MKL 2018 and ImageMagick v7.0.8. We ran all experiments on an Amazon EC2 `m4.10xlarge` instance with Intel Xeon E5-2676 v3 CPUs (40 logical cores) and 160GB of RAM, running Ubuntu 18.04 (Linux 4.4.0). Unless otherwise noted, results average over five runs.

8.1 Workloads

We evaluate the end-to-end performance benefits of SAs using Mozart on a suite of 15 data analytics workloads (of which four are repeated in NumPy and Intel MKL). Eight of the benchmarks are taken from the Weld evaluation [55], which

Workload	Libraries	Description (# Operators)
Black Scholes	NumPy, MKL	Computes the Black Scholes [1] formula over a set of vectors. (32)
Haversine	NumPy, MKL	Computes Haversine Dist. [11] from a set of GPS coordinates to a fixed point. (18)
nBody	NumPy, MKL	Uses Newtonian force equations to determine the position/velocity of stars over time. (38)
Shallow Water	NumPy, MKL	Estimates the partial differential equations used to model the flow of a disturbed fluid [23]. (32)
Data Cleaning [21]	Pandas	Cleans a DataFrame of 311 requests [9] by replacing `NULL`, broken, or missing values with `NaN`. (8)
Crime Index	Pandas, NumPy	Computes an average "crime index" score, given per-city population and crime information. (16)
Birth Analysis [49]	Pandas, NumPy	Given a dataset of number of births by name/year, computes fraction of names starting with "Lesl" grouped by gender and year-of-birth. (12)
MovieLens	Pandas, NumPy	Joins tables from the MovieLens dataset [43] to find movies that are most divisive by gender. (18)
Speech Tag	spaCy	Tags parts of speech and extracts features from a corpus of text. (8)
Nashville	ImageMagick	Image pipeline [19] that applies color masks, gamma correction, and HSV modulation. (31)
Gotham	ImageMagick	Image pipeline [13] that applies color masks, saturation/contrast adjustment, and modulation. (15)

Table 2. Workloads used in our evaluation. Descriptions for workloads from Weld are taken from [55]. The number in parentheses shows the number of library API calls.

obtained them from popular GitHub repositories, Kaggle competitions, and online tutorials. We also evaluate an additional numerical analysis workload (Shallow Water) over matrix operations, taken from the Bohrium paper [46] (Bohrium is an optimizing NumPy compiler that we compare against here). Finally, we evaluate two open-source image processing workloads [16] that use ImageMagick, and a part-of-speech tagging workload [24] that uses spaCy.

8.2 End-to-end Performance Results

Figure 4 shows Mozart's end-to-end performance on our 15 benchmarks on 1–16 threads. Each benchmark compares Mozart vs. a base system without SAs (e.g., NumPy or MKL without SAs). We also compare against optimizing compiler-based approaches that enable parallelism and data movement optimization without changing the library's API, but require *re-implementing functions* under-the-hood.

Summary of all Results. On 16 threads, Mozart provides speedups of up to 14.9× over libraries that are single-threaded, and speedups of up to 4.7× over libraries that are already parallelized due to its data movement optimizations. Across the 15 workloads, Mozart is within 1.2× of all compilers we tested in six workloads, and outperforms all the compilers we tested in four workloads. Compilers outperform Mozart by over 1.5× in two workloads. Compilers have the greatest edge over Mozart in workloads that contain many operators implemented in interpreted Python, since they naturally benefit more from compilation to native code than they do from data movement optimizations. We discuss workloads below.

NumPy Numerical Analysis (Figures 4a-d). We evaluate the NumPy workloads against un-annotated NumPy and three Python JIT compilers: Bohrium [46], Numba [6], and Weld. Each compiler requires function rewrites under-the-hood. Figures 4a-b show the performance of Black Scholes and Haversine, which apply vector math operators on NumPy arrays. All systems enable near-linear scalability since all operators can be pipelined and parallelized. Overall, Mozart enables speedups of up to 13.6×.

The nBody and Shallow Water workloads operate over tensors and matrices, and contain operators that cannot be pipelined. For example, Shallow Water performs several row-wise matrix operations and then aggregates along columns to compute partial derivatives. Mozart captures these boundaries using split types and still pipelines the other operators in these workloads. Figures 4c-d show that Mozart enables up to 4.6× speedups on 16 threads. Bohrium outperforms other systems in the Shallow Water benchmark because it captures indexing operations that Mozart cannot split and that the other compilers could not parallelize (Bohrium converts the indexing operation into its IR, whereas Mozart treats it as a function call over a single element that cannot be split).

Data Science with Pandas and NumPy (Figures 4e-h). We compare the Pandas workloads against Weld: the other compilers did not accelerate these workloads. The Data Cleaning and Crime Index workloads use Pandas and NumPy to filter and aggregate values in a Pandas DataFrame. Figures 4e-f show the results. Mozart parallelizes and pipelines both of these workloads and achieves an up to 14.9× speedup over the native libraries. However, Weld outperforms Mozart by up to 5.85× even on 16 threads, because both contain operators that use interpreted Python code which Weld compiles.

Figures 4g-h shows the results for the Birth Analysis and MovieLens workloads. These workloads are primarily bottlenecked on grouping and joining operations implemented in C. In Birth Analysis, Mozart accelerates groupBy aggregations by splitting grouped DataFrames and parallelizing

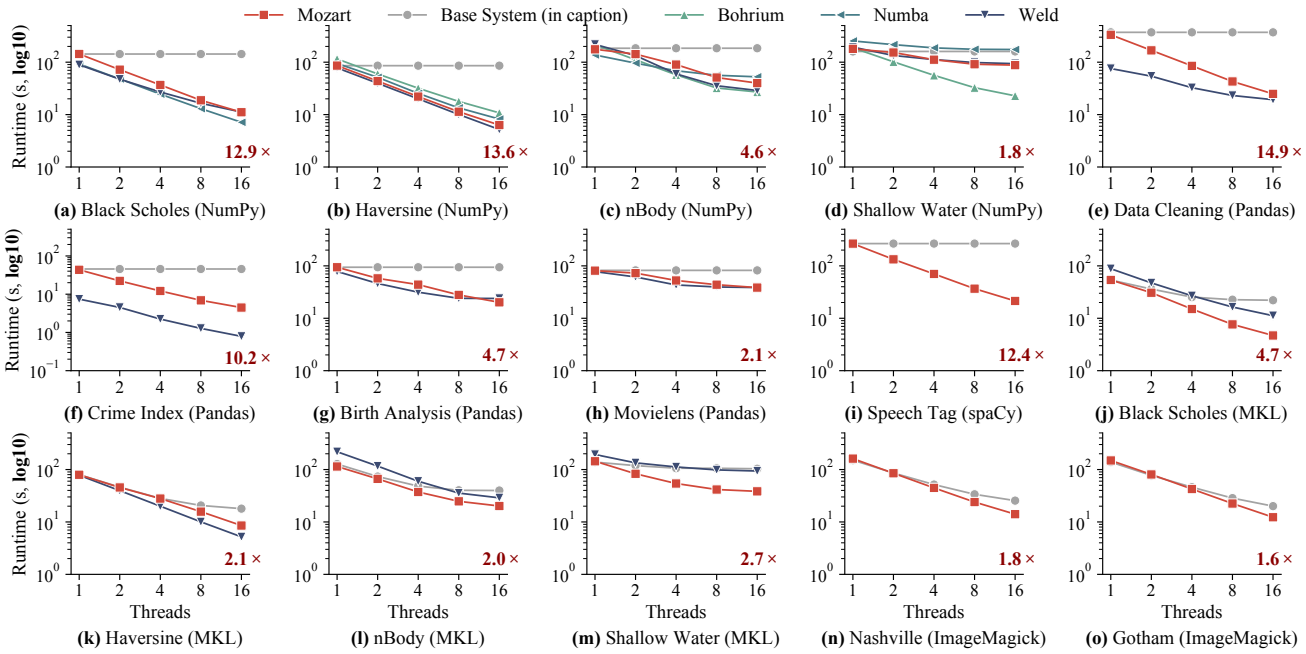


Figure 4. End-to-end performance on 15 benchmarks compared against a base system (in caption, e.g., NumPy) and several optimizing compilers that require rewriting libraries. We show results on 1–16 threads. Each plot displays the speedup (in red) that Mozart enables on 16 threads over the base system.

(there are no pipelined operators), leading to a 4.7× speedup. In MovieLens, we pipeline and parallelize two joins and parallelize a grouping aggregation, leading to a 2.1× speedup. In both, Mozart outperforms Weld. Weld's parallel grouping implementation bottlenecked on memory allocations around 8 threads in Birth Analysis. In MovieLens, speedups were hindered due to serialization overhead (Weld marshals strings before processing them, and Mozart sends large join results via IPC), but the Weld serialization could not be parallelized.

Speech Tagging with spaCy (Figures 4i). Figure 4i shows the performance of the speech tagging workload with and without Mozart. This workload operates over a corpus of text from the IMDb sentiment dataset [50]. It tags each word with a part of speech and normalizes sentences using a preloaded model. Mozart enables 12× speedups via parallelization. Unfortunately, no compilers supported spaCy.

Numerical Workloads with MKL (Figures 4j-m). We evaluate Mozart with MKL using the same numerical workloads from NumPy. Unlike NumPy, MKL already parallelizes its operators, so the speedups over it come from optimizing data movement. Figures 4j-m show that Mozart improves performance by up to 4.7× on 16 threads, even though MKL also parallelizes functions. Mozart outperforms Weld on three workloads here, because MKL vectorizes and loop-blocks matrix operators in cases where Weld's compiler does not.

Image workloads in ImageMagick (Figures 4n-o). Figure 4 shows the results on our two ImageMagick workloads. Like MKL, ImageMagick also already parallelizes functions, but Mozart accelerates them by pipelining across operators. Mozart outperforms base ImageMagick by up to 1.8×. End-to-end speedups were limited despite pipelining because splits and merges allocate and copy memory excessively. Mozart sped up just the computation by up to 3.4× on 16 threads.

8.3 Effort of Integration

In contrast to compilers, Mozart only requires annotators to add SAs and implement the splitting API to achieve performance gains. To quantify this effort, we compared the lines of code required to support our benchmarks with Mozart vs. Weld. Table 3 shows the results. Our Weld results only count code for operators that we also support, and only counts integration code. We do not count code for the Weld compiler itself (which itself is over 25K LoC and implements a full compilation backend based on LLVM). Similarly, we only count code that an annotator adds for Mozart, and do not count the runtime code.

Overall, for our benchmarks, SAs required up to 17× less code to enable similar performance to Weld in many cases. Weld required at least tens of lines of IR code per operator, a C++ extension to marshal Python data, and usage of its runtime API to build a dataflow graph. Anecdotally, through communication with the Pandas-on-Weld authors, we found

		LoC for SAs			LoC for Weld		
Library	#Funcs	SAs	Split. API	Total	Weld IR	Glue	Total
NumPy	84	47	37	**84**	321	73	**394**
Pandas	15	72	49	**121**	1663	413	**2076**
spaCy	3	8	12	**20**			
MKL	81	74	90	**155**			
ImageMagick	15	49	63	**112**			

Table 3. Integration effort for using Mozart. Numbers show the total lines of code per library. Mozart requires up to 15× fewer LoC to support the same operators as Weld.

	Black Scholes			Haversine		
	MKL	Mozart (-pipe)	**Mozart**	MKL	Mozart (-pipe)	**Mozart**
Normalized Runtime	1.00	1.01	**0.21**	1.00	0.97	**0.48**
LLC Miss (avg/stddev)	39.86% (2.26%)	42.96% (1.56%)	**22.12% (1.99%)**	50.44% (7.99%)	57.47% (0.40%)	**41.18% (3.67%)**
Inst/Cycle (avg/stddev)	0.536 (0.06)	0.511 (0.04)	**1.221 (0.10)**	0.892 (0.08)	1.362 (0.22)	**1.65 (0.33)**

Table 4. Hardware counters show that pipelining reduces cache misses, which translates to higher performance.

that just supporting the Birth Analysis workload, even after having implementing the core integration, was a multi-week effort that required several extensions to the Weld compiler itself and also required extensive low-level performance tuning. In contrast, we integrated SAs with the same Pandas functions in roughly half a day, and the splitting API was implemented using existing Pandas functions with fewer than 20 LoC each.

8.4 Importance of Pipelining

The main optimization Mozart applies beyond parallelizing split data is pipelining it across functions to reduce data movement. To show its importance, Table 4 compares three versions of the Black Scholes and Haversine workloads on 16 threads: un-annotated MKL, Mozart with pipelining, and Mozart without pipelining (i.e. Mozart parallelizes functions on behalf of MKL). We also used the Linux `perf` utility to sample the hardware performance counters for the three variations of the workload. Mozart without pipelining does not result in a speedup over parallel MKL. In addition, the most notable difference with pipelining is in the last level cache (LLC) miss rate: the miss rate decreases by a factor of 2×, confirming that pipelining does indeed generate less traffic to main memory and reduce data movement. This in turn leads to better overall performance on multiple threads. We found no other notable differences in the other reported counters. We saw similar results for the other MKL workloads as well.

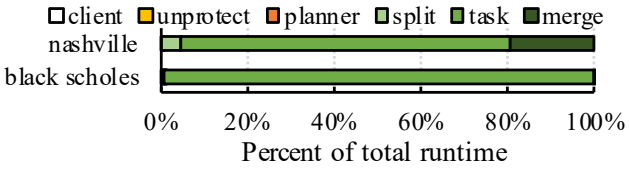
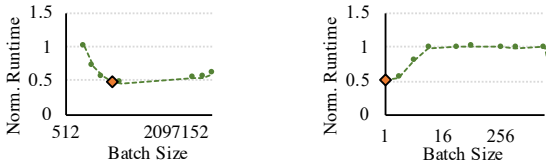


Figure 5. Breakdown of total running time in the Nashville and Black Scholes workloads. Across all workloads, we observed 0.5% overhead from other components.



(a) Black Scholes. **(b)** nBody.

Figure 6. Effect of batch size on two workloads. Mozart selects a batch size near the optimal using L2 cache size.

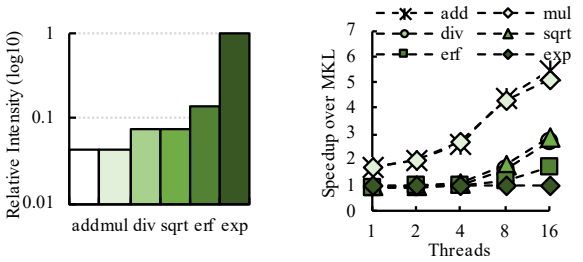
8.5 System Overheads

To measure the system overheads that Mozart imposes, Figure 5 shows the breakdown in running time for the Black Scholes and Nashville workloads on 16 threads. We report the client library time for registering tasks, the memory protection time for unprotecting pages during execution, planning, splitting, task execution, and merging. The Nashville workload had the highest relative split and merge times, since both the splitters and mergers allocate memory and copy data. Across all workloads, the execution time dominates the total running time, and the client library, memory protection, and planner account for less than 0.5% of the running time.

Most of the overhead is attributed to handling memory protection. In our setup, unprotecting each gigabyte of data took roughly 3.5ms, indicating that this overhead could be significant on task graphs that perform little computation. One mechanism for reducing the overhead of memory protection is the recent `pkeys` [59] set of system calls, which allows for O(1) memory permission changes by associating pages with a registered protection key. After tagging memory pages with a key, the cost of changing their permissions is a register write, so the time to unprotect or protect *all* memory allocated with Mozart becomes negligible (tens of microseconds to unprotect 1GB in a microbenchmark we ran).

8.6 Effect of Batch Size

We evaluate the effect of batch size by varying it on the Black Scholes and nBody workloads and measuring end-to-end performance for each parameter. We benchmarked these two workloads because they contain "elements" of different sizes: Black Scholes treats each `double` as a single element, while the matrices' split types in nBody treat rows of a matrix (256KB in size each) as a single element. Figure 6 shows the results. The marked point shows the batch size selected by



(a) Relative Intensity **(b)** Speedup over no SAs

Figure 7. Impact of compute-intensiveness in Mozart.

Mozart using the strategy described in §5. The plots show that batch size can have a significant impact on overall running time (too low imposes too much overhead, and too high obviates the benefits of pipelining), and that Mozart's heuristic scheme selects a reasonable batch size. Across all the workloads we benchmarked, Mozart chooses a batch size within 10% of the best batch size we observed in a parameter sweep.

8.7 Compute- vs. Memory-Boundedness

To study when Mozart's data movement optimizations are most impactful, we measured the intensity (defined as *cycles spent per byte of data*) of several MKL vector math functions by calling them in a tight loop on an array that fits entirely in the L2 cache. We benchmarked the following operations, in order of increasing intensity: add, mul, sqrt, div, erf, and exp. Figure 7a shows the relative intensities of each function (i.e., `vdExp` spends roughly 7× more cycles per byte of data than `vdErf`). We then ran each math function 10 times on a large 8GB array, with and without Mozart. Figure 7b shows the *speedup* of Mozart over un-annotated MKL on 1–16 threads. Mozart has the largest impact on memory-intensive workloads that spend few cycles per byte, and shows increasing speedups as increasing amounts of parallelism starve the available memory bandwidth.

9 Related Work

SAs are influenced by work on building new common runtimes or IRs for data analytics [48, 56, 65, 67] and machine learning [27, 34, 64]. Weld [56] and Delite [65] are two specific examples of systems that use a common IR to detect *parallel patterns* and automatically generate parallel code. Although Mozart does not generate code, we show in §8 that in a parallel setting, the most impactful optimizations are the data movement ones, so SAs can achieve competitive performance without requiring developers to replace code. API-compatible replacements for existing libraries such as Bohrium [46] also have completely re-engineered backends.

Several existing works provide black-box optimizations and automatic parallelization of functions. Numba [6] JITs code using a single decorator, while Pydron [53], Dask [4] and Ray [54] automatically parallelize Python code for multicores and clusters. In C, frameworks such as Cilk [30] and OpenMP [37] parallelize loops using an annotation-style interface. Unlike these systems, in addition to parallelization, SAs enable data movement optimizations across functions and reason about pipelining safety.

The optimizations that SAs enable have been studied before: Vectorwise [68] and other vectorized databases [31, 35, 47] 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 [41], pipelining [33, 40], and link-time optimization [29, 39], though we found these optimizations most effective over nested C loops in user code, and not over compositions of complex arbitrary functions.

Finally, split types are conceptually related to Spark's partitioners [2] and Scala's parallel collections API [60]. Scala's parallel collections API in particular features a `Splitter` and `Combiner` that partition and aggregate a data type, respectively. Unlike this API, SAs enable pipelining and also reason about the safety of pipelining black-box functions: Scala's collections API still requires introspecting collection implementations. Spark's Partitioners similarly do not enable pipelining.

10 Conclusion

Data movement is a significant bottleneck for data-intensive applications that compose functions from existing libraries. Although researchers have developed compilers 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 split 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 functions, are easy to integrate, and provide performance competitive with clean-slate approaches in many cases.

11 Acknowledgements

We thank Paroma Varma, Deepak Narayanan, Saachi Jain, Aurojit Panda, and the members of the DAWN project for their input on earlier drafts of this work. We also thank our shepherd, Raluca Ada Popa, and the anonymous SOSP reviewers for their invaluable feedback. This research was supported in part by supporters of the Stanford DAWN project (Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, SAP, Teradata, and VMware), by Amazon Web Services, Cisco, and by NSF CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] 2013. Black Scholes Formula. http://gosmej1977.blogspot.com/2013/02/black-and-scholes-formula.html.

[2] 2014. Spark Partitioner. https://spark.apache.org/docs/2.2.0/api/java/org/apache/spark/Partitioner.html.

[3] 2017. XLA: TensorFlow, Compiled! (TensorFlow Dev Summit 2017). https://www.youtube.com/watch?v=kAOanJczHA0&feature=youtu.be&t=41m46s.

[4] 2018. Dask. https://dask.pydata.org.

[5] 2018. Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

[6] 2018. Numba. https://numba.pydata.org.

[7] 2018. TensorFlow XLA. https://www.tensorflow.org/performance/xla/.

[8] 2018. TensorFlow XLA JIT. https://www.tensorflow.org/performance/xla/jit.

[9] 2019. 311 Service Requests Dataset. https://github.com/jvns/pandas-cookbook/blob/master/data/311-service-requests.csv.

[10] 2019. Adding a New Op. https://www.tensorflow.org/guide/extend/op.

[11] 2019. A Beginner's Guide to Optimizing Pandas Code for Speed. goo.gl/dqwmrG.

[12] 2019. Dependent Type. https://en.wikipedia.org/wiki/Dependent_type.

[13] 2019. Gotham. https://github.com/acoomans/instagram-filters/tree/master/instagram_filters/filters/gotham.py.

[14] 2019. How to optimize GEMM on CPU. https://docs.tvm.ai/tutorials/optimize/opt_gemm.html.

[15] 2019. ImageMagick. https://imagemagick.org/.

[16] 2019. instagram-filters. https://github.com/acoomans/instagram-filters/tree/master/instagram_filters/filters.

[17] 2019. Intel MKL-DNN. https://intel.github.io/mkl-dnn/.

[18] 2019. Matrix Multplication is 3X Slower than OpenBLAS. https://github.com/halide/Halide/issues/3499.

[19] 2019. Nashville. https://github.com/acoomans/instagram-filters/tree/master/instagram_filters/filters/nashville.py.

[20] 2019. NumPy. http://www.numpy.org/.

[21] 2019. Pandas Cookbook chapter 7: cleaning up messy data. https://github.com/jvns/pandas-cookbook/.

[22] 2019. PyTorch. http://pytorch.org.

[23] 2019. ShallowWater. https://github.com/mrocklin/ShallowWater/.

[24] 2019. spaCy. https://spacy.io/.

[25] 2019. Typescript. https://www.typescriptlang.org/.

[26] 2019. Typescript Decorators. https://www.typescriptlang.org/docs/handbook/decorators.html/.

[27] 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. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, Vol. 16. 265–283.

[28] Alexander Aiken and Edward L Wimmers. 1993. Type inclusion constraints and type inference. In *FPCA*, Vol. 93. Citeseer, 31–41.

[29] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2011. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices* 46, 4 (2011), 41–52.

[30] Robert D. Blumenofe, Christopher F. Joerg, Bradley C. Kurzmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55–69. https://doi.org/10.1006/jpdc.1996.0107

[31] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[32] 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 (PPoPP '11)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/1941553.1941561

[33] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*. ACM, New York, NY, USA, 10–18. https://doi.org/10.1145/1248648.1248652

[34] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 578–594.

[35] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. 2015. Tupleware: Big Data, Big Analytics, Small Clusters. In *CIDR*.

[36] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, and Omar Kanawi. 2018. Intel nGraph. https://ai.intel.com/intel-ngraph/.

[37] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[38] dphaskell 2018. Data Parallel Haskell. https://wiki.haskell.org/GHC/Data_Parallel_Haskell.

[39] Mary F Fernandez. 1995. *Simple and effective link-time optimization of Modula-3 programs*. Vol. 30. ACM.

[40] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. 2005. With-Loop fusion for data locality and parallelism. In *Symposium on Implementation and Application of Functional Languages*. Springer, 178–195.

[41] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

[42] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[43] F Maxwell Harper and Joseph A Konstan. 2016. The Movielens Datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4 (2016), 19.

[44] Kagi, A and Goodman, James R and Burger, Doug. 1996. Memory bandwidth limitations of future microprocessors. In *Computer Architecture, 1996 23rd Annual International Symposium on*. IEEE, 78–78.

[45] Helena Kotthaus, Ingo Korb, Michael Engel, and Peter Marwedel. 2015. Dynamic page sharing optimization for the R language. *ACM SIGPLAN Notices* 50, 2 (2015), 79–90.

[46] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. 2014. Bohrium: a virtual machine approach to portable parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 312–321.

[47] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

[48] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro* 31, 5 (2011), 42–53.

[49] Wayne Liu. 2014. Python and Pandas Part 4: More Baby Names. http://beyondvalence.blogspot.com/2014/09/python-and-pandas-part-4-more-baby-names.html.

[50] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the*

Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015

[51] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[52] Wes McKinney. 2010. Data Structures for Statistical Computing in Python . In *Proceedings of the 9th Python in Science Conference*. 51 – 56.

[53] Stefan C Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. 2014. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud.. In *OSDI*. 645–659.

[54] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Michael I. Jordan, and Ion Stoica. 2017. Real-Time Machine Learning: The Missing Pieces. *CoRR* abs/1703.03924 (2017). arXiv:1703.03924 http://arxiv.org/abs/1703.03924

[55] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. 2018. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1002–1015.

[56] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[57] Shoumik Palkar, James J. 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

[58] Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

[59] pkeys 2019. pkeys. http://man7.org/linux/man-pages/man7/pkeys.7.html.

[60] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A generic parallel collection framework. In *European Conference on Parallel Processing*. Springer, 136–147.

[61] 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. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[62] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc.

[63] Rocklin, Matthew. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*. Citeseer.

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

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

[66] Wm. A. Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.

[67] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.. In *OSDI*, Vol. 8. 1–14.

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