Split Annotations
Optimizing Data-Intensive Computations in Existing Libraries
Shoumik Palkar and Matei Zaharia
Motivation for split annotations

Modern data analytics applications combine many disjoint processing libraries & functions

+ Great results leveraging 1000s of functions
– No end-to-end optimization across function calls
  (prior work: up to 30x performance left on table)
Why is calling existing APIs slow?

One major reason: on modern hardware, processing speeds have outpaced memory speeds

```plaintext
// From Black Scholes
// all inputs are vectors
d1 = price * strike
d1 = np.log2(d1) + strike
```

Data movement is often dominant bottleneck in composing existing functions
Existing ideas for optimizing E2E applications under high-level APIs

Researchers have proposed **JIT compilers** and **runtimes** to optimize code on a per-app basis.

**Examples**
TensorFlow XLA, TorchScript, Weld, Numba, Bohrium
JIT compilers improve E2E performance

Compilers fuse operators during compilation to reduce data movement.

// From Black Scholes
// all inputs are vectors
d1 = price * strike
d1 = np.log2(d1) + strike
JIT compilers improve E2E performance

Compilers fuse operators during compilation to reduce data movement.

// From Black Scholes
// all inputs are vectors
d1 = price * strike
d1 = np.log2(d1) + strike

Up to 30x speedups from data movement optimizations such as loop fusion [Weld, XLA]
Problem: Huge Developer Effort

- Need to replace every function to use compiler intermediate representation (IR)
- IR may not even support all optimizations present in hand-optimized code

Example
Weld needs 1000s of LoC to support NumPy, Pandas

JIT compiler from our research group!
“Sorry, our compiler doesn’t recognize this pattern yet”

“Some ops are expected to be slower compared to hand-optimized kernels”
Can we obtain similar speedups to JIT compilers with only *existing functions*?
Split Annotations (SAs)

Data movement optimizations + parallelization of existing APIs without library code changes!
SAs Enable Pipelining + Parallelism

Key idea: split data to pipeline and parallelize it.
SAs Enable Pipelining + Parallelism

Without SAs:

\[ d_1 = \text{price} \times \text{strike} \]
\[ d_1 = \log_2(d_1) + \text{strike} \]
SAs Enable Pipelining + Parallelism

Without SAs:

\[
d1 = price \times strike
\]

\[
d1 = \log_2(d1) + strike
\]
SAs Enable Pipelining + Parallelism

With SAs:

\[
d1 = \text{price} \times \text{strike}
\]

\[
d1 = \log_2(d1) + \text{strike}
\]
SAs Enable Pipelining + Parallelism

With SAs:

\[ d1 = \text{price} \times \text{strike} \]
\[ d1 = \log_2(d1) + \text{strike} \]

Build execution graph, keep data in cache by passing cache-sized splits to functions.
SAs Enable Pipelining + Parallelism

With SAs:

\[ d_1 = \text{price} \times \text{strike} \]

\[ d_1 = \log_2(d_1) + \text{strike} \]

Build execution graph, keep data in cache by passing cache-sized splits to functions.

Collectively fit in cache

\[ d_1 = \text{price} \times \text{strike} \]

\[ d_1 = \log_2(d_1) + \text{strike} \]
SAs Enable Pipelining + Parallelism

With SAs:

$$d_1 = \text{price} \times \text{strike}$$

$$d_1 = \text{np.log2}(d_1) + \text{strike}$$

Build execution graph, **keep data in cache** by passing cache-sized splits to functions.

Collectively fit in cache

$$d_1 = \text{price} \times \text{strike}$$

$$d_1 = \text{np.log2}(d_1) + \text{strike}$$
SAs Enable Pipelining + Parallelism

With SAs:

\[ d1 = \text{price} \times \text{strike} \]
\[ d1 = \log_2(d1) + \text{strike} \]

Build execution graph, keep data in cache by passing cache-sized splits to functions.
SAs Enable Pipelining + Parallelism

With SAs:

\[
d1 = \text{price} \times \text{strike}
\]

\[
d1 = \text{np.log2}(d1) + \text{strike}
\]

Build execution graph, keep data in cache by passing cache-sized splits to functions.
SAs Enable Pipelining + Parallelism

With SAs:

Parallelize over split pieces

Build execution graph, **keep data in cache** by passing cache-sized splits to functions.
Example of a split annotation for MKL

@sa(n: SizeSplit(n, K), a: ArraySplit(n, K),
    b: ArraySplit(n, K), out: ArraySplit(n, K))

    // Computes out[i] = a[i] + b[i] element-wise
    void vdAdd(int n, double *a, double *b, double *out)

Benefits compared to JIT compilers:
+ No intrusive library code changes
+ Reuses optimized library function implementations
+ Does not require access to library code
SAs can sometimes outperform compilers

Black Scholes using Intel MKL

5x speedups by reducing data movement
Challenges in designing SAs

1. Defining how to split data and enforcing safe pipelining

2. Building a lazy task graph transparently

3. Designing a runtime to execute tasks in parallel
Challenges in designing SAs

1. Defining how to split data and enforcing **safe** pipelining

2. Building a lazy task graph **transparently**

3. Designing a **runtime** to execute tasks in parallel

See paper for implementation details!
How do SAs enforce **safe** pipelining?

E.g., preventing pipelining between matrix functions that iterate over row vs. over column:

Okay to pipeline – split matrix by row, pass rows to function.  

Cannot pipeline – second function reads incorrect values.
SAs use a type system to enforce safe pipelining

A split type uniquely defines how to split function arguments and return values.

@sa(n: SizeSplit(n, K), a: ArraySplit(n, K),
    b: ArraySplit(n, K), out: ArraySplit(n, K))
void vdAdd(int n, double *a, double *b, double *out)
SAs use a **type system** to enforce safe pipelining

A **split type** uniquely defines how to split function arguments and return values.

```plaintext
@sa(n: SizeSplit(n, K), a: ArraySplit(n, K),
    b: ArraySplit(n, K), out: ArraySplit(n, K))
void vdAdd(int n, double *a, double *b, double *out)
```

**ArraySplit** depends on function arg. **n**, the **runtime size** of an array, and **K**, the **number of pieces**.
Same split types = values can be pipelined

An SA defines a unique “splitting” for a value using a primitive called a split type.

```plaintext
@sa(n: SizeSplit(n, K), a: ArraySplit(n, K),
    b: ArraySplit(n, K), out: ArraySplit(n, K))
void vdAdd(int n, double *a, double *b, double *out)
```

Same split types enforce values split in the same way: we can pipeline if data between functions has matching split types.
Example: Matrix Pipelining in NumPy

Split type for NumPy matrices encodes dimension + axis: 
MatrixSplit(Rows, Cols, Axis, K)

Split types match: axis=0 for both function calls

Split types don’t match: axis=0 for first call, axis=1 for second call
How an annotator writes SAs

1. Define a split type (e.g., ArraySplit, MatrixSplit)

2. Write a **split function** and **merge function** for the type

3. Annotate functions using the defined split types
Mozart: Our system implementing SAs

User Application
\[ y = \text{lib.f}(); \]
\[ z = \text{lib.g}(y); \]

Annotations

Existing library

Wrapped Library

Mozart Client Library
Builds a lazily evaluated task graph, determines when to execute it

Mozart Runtime
Check + initialize split types, split data, execute functions in parallel
Mozart: Our system implementing SAs

**In C++:** Memory protection for lazy evaluation

**In Python:** Meta-programming for lazy evaluation

See paper for details!

Mozart Client Library
Builds a lazily evaluated task graph, determines when to execute it

Mozart Runtime
Check + initialize split types, split data, execute functions in parallel
Mozart: Our system implementing SAs

User Application
\[ y = \text{lib.f}(); \]
\[ z = \text{lib.g}(y); \]

Annotations

Existing library

Wrapped Library

Mozart Client Library
Builds a lazily evaluated task graph, determines when to execute it

Mozart Runtime
Check + initialize split types, split data, execute functions in parallel
Results
Results

Setup: EC2 m4.10xlarge (160GB memory, 40 vCPUs) running Linux.

Questions:
1. What kinds of workloads can SAs accelerate?
2. How much effort is required to use SAs vs. compilers?
3. How do SAs perform compared to JIT compilers?
Data Types and Libraries Demonstrated

Libraries: L1 + L2 BLAS (MKL), NumPy, Pandas, spaCy, ImageMagick

Data types and operators: Arrays, Tensors, Matrices, DataFrame joins, grouping aggregations, image processing algorithms, functional operators (map, reduce, etc.)
SAs require less integration effort than compilers

<table>
<thead>
<tr>
<th>Library</th>
<th>#Funcs</th>
<th>SAs</th>
<th>Split. API</th>
<th>Total</th>
<th>Weld IR</th>
<th>Glue</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>84</td>
<td>47</td>
<td>37</td>
<td>84</td>
<td>321</td>
<td>73</td>
<td>394</td>
</tr>
<tr>
<td>Pandas</td>
<td>15</td>
<td>72</td>
<td>49</td>
<td>121</td>
<td>1663</td>
<td>413</td>
<td>2076</td>
</tr>
<tr>
<td>spaCy</td>
<td>3</td>
<td>8</td>
<td>12</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MKL</td>
<td>81</td>
<td>74</td>
<td>90</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ImageMagick</td>
<td>15</td>
<td>49</td>
<td>63</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SAs can match JIT compilers under existing APIs

- NumPy
- Bohrium
- Weld
- Numba
- NumPy+SAs

nBody simulation: **4.6x speedup** over NumPy

Birth Analysis: **4.7x speedup** over pandas
SAs can accelerate highly optimized libraries

Black Scholes: 5x speedup over MKL

Image filter: 1.8x speedup over ImageMagick
Across the 15 workloads we benchmarked:

SAs **perform within 1.2x of all compilers** in **nine** workloads

SAs **outperform all compilers** in **four** workloads

**Compilers outperform SAs by >1.2x** in **two** workloads

- Up to **6x slower**: This happens when code generation (e.g., compiling interpreted Python) matters

See paper for more details!
Conclusion

Split Annotations:
- Enable order-of-magnitude speedups over existing APIs
- Require less than 10x LoC to use compared to compilers

https://www.github.com/weld-project/split-annotations

Contact:
shoumik@cs.stanford.edu
https://www.shoumik.xyz