Splitability Annotations
Optimizing Black-Box Function Composition in Existing Libraries

Shoumik Palkar and Matei Zaharia
Motivation

Users *compose* functions from software libraries when building applications.

Example: Adding vectors with MKL.

```c
// inp1 + inp2 + .. + inp10
vdAdd(inp1, inp2, res);
vDAdd(res, inp3, res);
...
vDAdd(res, inp10, res);
```

**Up to 8x slowdowns** compared to code that pipelines loops due to *data movement.*
Existing Solutions to Reduce Data Movement

Researchers have proposed **IRs** and **VMs** to reduce data movement and parallelize code.

**Examples**

Weld, Delite, XLA, TVM, Bohrium, Spark, etc.
Don’t These Systems use JIT Optimized Code?

Yes! But much of the impact comes from data movement.

Example: Impact of optimizations in Weld.

Loop fusion has the greatest impact by reducing repeated passes over data.
Problem: Huge Developer Effort

Need to replace every function to use IR
IR may not even support all optimizations present in hand-optimized code

Examples
Weld needs 100s of LoC to support NumPy, Pandas
Bohrium (port of NumPy) implements a new VM

Composing library functions using these systems is very Strauss-ful
Developers Have A Choice

1. Dropping their own optimizations and using existing IRs
2. Forgoing optimization under composition
Splitability Annotations (SAs)

Data movement optimizations and automatic parallelization on black-box functions for the first time: *without changing library functions!*
SAs Enable Pipelining + Parallelism

Split data to pipeline and parallelize it.
SAs Enable Pipelining + Parallelism

Without SAs:

```
vdAdd(inp1, inp2, res);
vdAdd(res, inp3, res);
...```

---
SAs Enable Pipelining + Parallelism

Without SAs:

vdAdd(inp1, inp2, res);
vdAdd(res, inp3, res);
...
SAs Enable Pipelining + Parallelism

With SAs:

```
vAdd(inp1, inp2, res);
vAdd(res, inp3, res);
...```

```cpp
int res, inp1, inp2, inp3;
```
SAs Enable Pipelining + Parallelism

With SAs:

\begin{align*}
\text{vdAdd}(& \text{inp1}, \text{inp2}, \text{res}) ; \\
\text{vdAdd}(& \text{res}, \text{inp3}, \text{res}) ; \\
\end{align*}

Keep data in fast memory by passing smaller splits to black-box functions.
SAs Enable Pipelining + Parallelism

With SAs:

vdAdd(inp1, inp2, res);
vdAdd(res, inp3, res);
...
SAs Enable Pipelining + Parallelism

With SAs:

```
vdsAdd(inp1, inp2, res);
vdsAdd(res, inp3, res);
...```

```
SAs Enable Pipelining + Parallelism

With SAs:

res

vdAdd(inp1, inp2, res);
vdAdd(res, inp3, res);

...
SAs Enable Pipelining + Parallelism

With SAs:

```
vDAdd(inp1, inp2, res);
vDAdd(res, inp3, res);
...```
SAs Enable Pipelining + Parallelism

With SAs:

Thread 1

Thread 2

Thread N

Parallelize over pieces
Splitability Annotations: Example

Splitability annotations (SAs) are small annotations on existing black-box functions.

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

S: “split data in the same way”

10 calls to the add function: **8x speedups** by reducing data movement
Rest of this Talk

1. System Overview
2. Splitability Annotation Abstractions
3. Examples
4. Splitter and Merger Functions
5. Runtime and Implementation
6. Results
System Overview

Provide annotations and splitter functions for the functions they want to optimize.

Provide a mechanism to construct a computation DAG to build a computation.

Pass computation DAG to a runtime to apply optimizations.
Splitability Annotations
Splitability Annotation Abstraction

An SA defines how to “split” (i.e., partition) values into multiple pieces using split types.

// @splittable
// (a: S, b: S, res: mut S)
void vdAdd(vec_t *a, vec_t *b, vec_t res);

Assign the same split type $S \rightarrow$ split these values in the same “way”: we can pipeline if data has matching split types.
Splitting Values

A value is split by “partitioning” it into multiple disjoint pieces using a user-defined **splitter function**. A split is thus data- and library-dependent.

Example: Splitting Arrays into regularly-sized chunks
Split Types: Capturing How Values are Split

A split type captures a **unique** partitioning for a **data type**.

Example:

Same underlying array, but **different** split types!
Example: A Vector Library

// (a: S, b: S, c: mut S)
void vdAdd(vec_t *a, vec_t *b, vec_t *res);

// (a: mut S) -> T
vec_t* vdFilter(vec_t *a, double max);
Example: A Vector Library

// (a: S, b: S, c: mut S)
void vdAdd(vec_t *a, vec_t *b, vec_t *res);

// (a: mut S) -> T
vec_t* vdFilter(vec_t *a, double max);
vdAdd Example: Same Split Types

We can add these: **same split type**, so pass corresponding pieces to function...

...but not these: different split types, so values don’t “line up.”

We need to be Chopin’ data up in a consistent way!
Example: A Vector Library

// (a: S, b: S, c: mut S)
void vdAdd(vec_t *a, vec_t *b, vec_t *res);

// (a: mut S) → T
vec_t* vdFilter(vec_t *a, double max);

Returns a different split type!
Split Types Enable **Safe Pipelining**

E.g., filtering data:

```c
res = vdFilter(inp1, 10.0);
vdAdd(res, inp2, res);
...
```

<table>
<thead>
<tr>
<th>inp1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>inp2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Split Types Enable **Safe Pipelining**

E.g., filtering data:

```cpp
res = vdFilter(inp1, 10.0);
vdAdd(res, inp2, res);
...```

```c
vdFilter
```
Split Types Enable **Safe Pipelining**

E.g., filtering data:

```plaintext
res = vdFilter(inp1, 10.0);
vdAdd(res, inp3, res);
...
```
Split Types Enable **Safe Pipelining**

E.g., filtering data:

```c
res = vdFilter(inp1, 10.0);
vdAdd(res, inp3, res);
...
```

vdFilter returns values with a different split type:

```
(a: S) \rightarrow T
vec_t* vdFilter(vec_t *a, double val);
```
Split Types Enable Safe Pipelining

E.g., preventing pipelining between matrices 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.
Split Type Semantics

Our implementation uses dependent types to represent split types in a program. Each split type has a name and a set of parameter values.

Type representing an array split into regular-sized pieces: `RegularSplit<Length, Pieces>`

`RegularSplit<1000, 10>` and `RegularSplit<1000, 100>` have different types (since they have different param. values)
SAs use Split Types to Define Constraints

(\text{arg1: } [\text{mut}] \ T1, \ \text{arg2: } [\text{mut}] \ T2, \ ...) \ [\rightarrow Tn]

- Argument is mutated
- Split types for each argument that is split
Examples of Using SAs
Examples: Matrices

//@splitttable
//((mut matrix: RowMajorMatrixSplit<R,C,P>,
//  rows: 1,
//  cols: cols)
void incrementByRow(double *matrix, int rows, int cols);

//@splitttable
//((mut matrix: ColMajorMatrixSplit<R,C,P>,
//  rows: rows,
//  cols: 1)
void incrementByCol(double *matrix, int rows, int cols);
Examples: Splitting Strings

// Parses a multi-line CSV string into an array of objects.
// @splittable
// (s: LineSplit<L,P>) -> S
tuple_t *read_csv(string *s);

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

Got a Handel on it?
Splitter and Merger Functions
Splitting Data with User-Defined Functions

Users provide **splitter functions** to split data in their library, and **merger functions** to merge split values.

Parameters in a split type become parameters in splitter function.

Example: to produce `RegularSplit<Length, Pieces>`, we use a splitter function that takes `length` and `pieces` as arguments.
Splitter and Merger Function APIs

For a split type with parameters Params:

```// Split data into a collection of pieces
fn split(data: D, Params) -> Collection<D>;

// merge a collection of pieces into a single value.
fn merge(Collection<D>) -> D;```
Splitter Function Example

Splitter functions “create” splits types.

To create `RegularSplit<Length, Pieces>` for a C Array:

```c
collection<vec_t> split(vec_t *data, 
  int length, int pieces) {
  ...
}
```
Runtime and Implementation
Using SAs to Optimize Computations

In computation DAG, if output of one function has same split type as the input of another, we can pipeline values.

Computes $\text{filter}(\text{inp1} + \text{inp2} + \text{inp3}) + \text{inp4}$

Pipeline functions with same split types

Split type changes: call merger function to un-split values

Call splitter function
Mozart: Our System Implementing SAs

C Implementation
- Written as a Rust library that can link with C program
- Rust threading for runtime
- Virtual memory protection for lazy evaluation

Python Implementation
- Function decorators for lazy evaluation
- Multiprocessing-based parallelism

No changes required for existing library functions in both implementations!
Capturing a DAG in C

Capture a DAG by using preprocessor macros/compile-time substitutions.

```c
void vdAdd(args) { <code> } becomes
    void vdAdd_impl(args) { <code> }
```

Wrapper functions to enable DAG creation:

```c
void vdAdd(args) { register_with_runtime(args, vdAdd); }

// Called by runtime.
void vdAdd_callback(void *args) {
    args = unpack(args);
    return vdAdd_impl(args);
}
```
Capturing a DAG in C

Replace allocation with `mozart_malloc`

```c
double *data = (double *) mozart_malloc(...);
```

• Internally: `mmap(PROT_NONE)`
• Catch SIGBUS/SIGSEGV, unprotect memory, execute graph

Limitations
• Requires updates to happen on heap allocated memory
• DAG creation cannot cross stack boundaries (dangling pointers)
Results
Results

Python - Integrated Mozart with NumPy and Pandas and compared performance against Weld

C – Integrated Mozart with MKL

Questions:
How does performance compare to code-generating systems such as Weld?
How much effort is required to add annotations, write splitter functions, etc.?
Results: Performance vs. Weld

Haversine Distance in NumPy: **within 10% of Weld** on 1—32 threads.

Crime Index in Pandas: **within 10% of Weld** on 32 threads. Code-gen has huge impact on one thread!

Data Cleaning in Pandas: **within 600% of Weld** on 32 threads: example of where compute-optimization is important.
Results: Performance vs. MKL

Black Scholes: **1.8x faster than Weld** on one thread, **13x faster than MKL** on 32 threads

Reasons for Speedup

- Reuse buffers to reduce page faults (speedup over MKL)
- Use MKL’s vectorized operators that Weld does not implement (speedup over Weld)
- Pipelined execution on multiple threads (scalability)
Micro-benchmark: Compute vs. Memory

Chain 10 of the same operator and measure speedup with Mozart. Try different operators with different intensities.

Takeaway
Mozart and data movement optimizations help most when workloads are memory bound
## Integration Effort

Overall, very low effort compared to other systems!

<table>
<thead>
<tr>
<th></th>
<th>Funcs. Annotated</th>
<th>Annotations LoC Total</th>
<th>Splitters LoC Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>13</td>
<td>47</td>
<td>37 (Python)</td>
</tr>
<tr>
<td>Pandas</td>
<td>15</td>
<td>72</td>
<td>49 (Python)</td>
</tr>
<tr>
<td>MKL</td>
<td>81</td>
<td>75 (Python)</td>
<td>90 (C)</td>
</tr>
</tbody>
</table>

Compare to Weld (~800 LoC for Pandas and NumPy each) and Bohrium (1000s of LoC for NumPy)
Conclusion

Splitability Annotations:
- Enable data movement optimizations and automatic parallelization over black-box library functions
- Use split types and a small type system to enforce safe composition
- Can improve performance of applications by up to 8x

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