Split Annotations

Optimizing Data-Intensive Computations in Existing Libraries

Shoumik Palkar and Matei Zaharia

STANFORD
DAWN

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

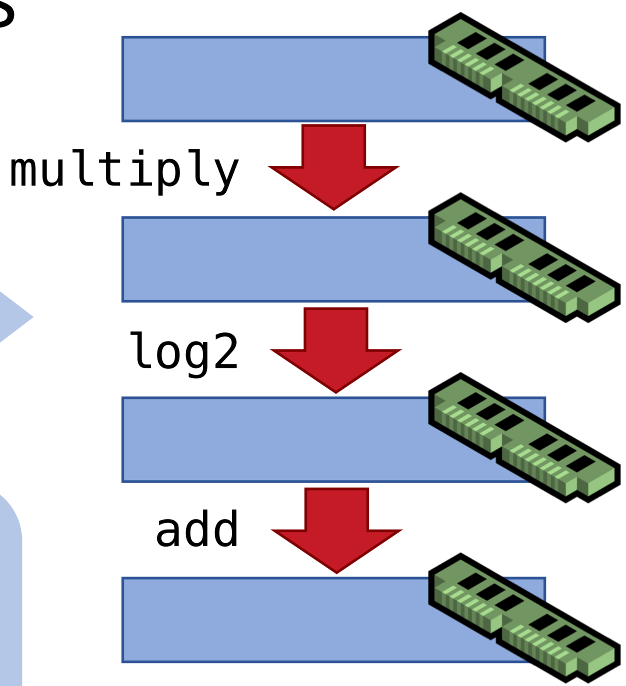
(prior work: up to **30x** performance left on table)

Why is calling existing APIs slow?

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

```
// 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

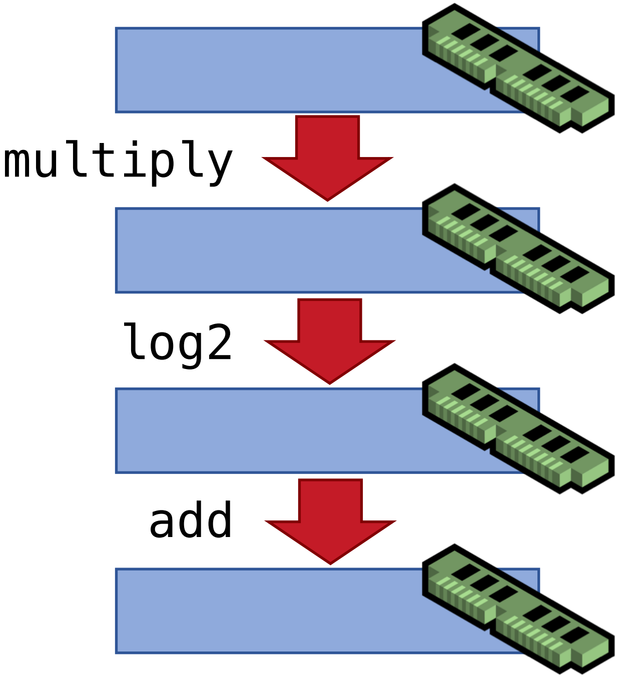
TensorFlow XLA, TorchScript, Weld, Numba, Bohrium

PyTorch Weld Numba Bohrium

JIT compilers improve E2E performance

Compilers fuse operators during compilation to **reduce data movement**.

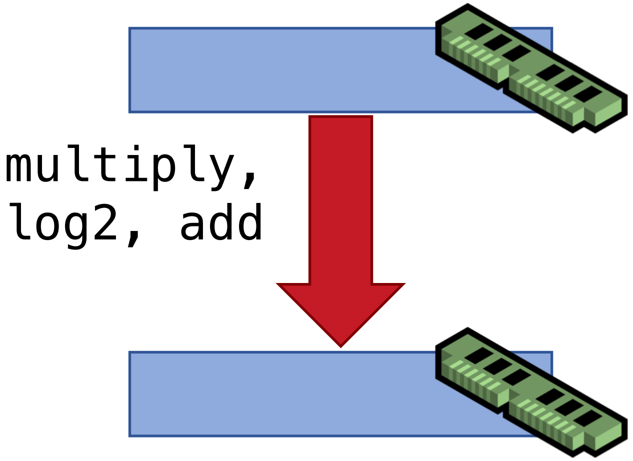
```
// From Black Scholes
// all inputs are vectors
d1 = price * strike
d1 = np.log2(d1) + strike
```



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]*

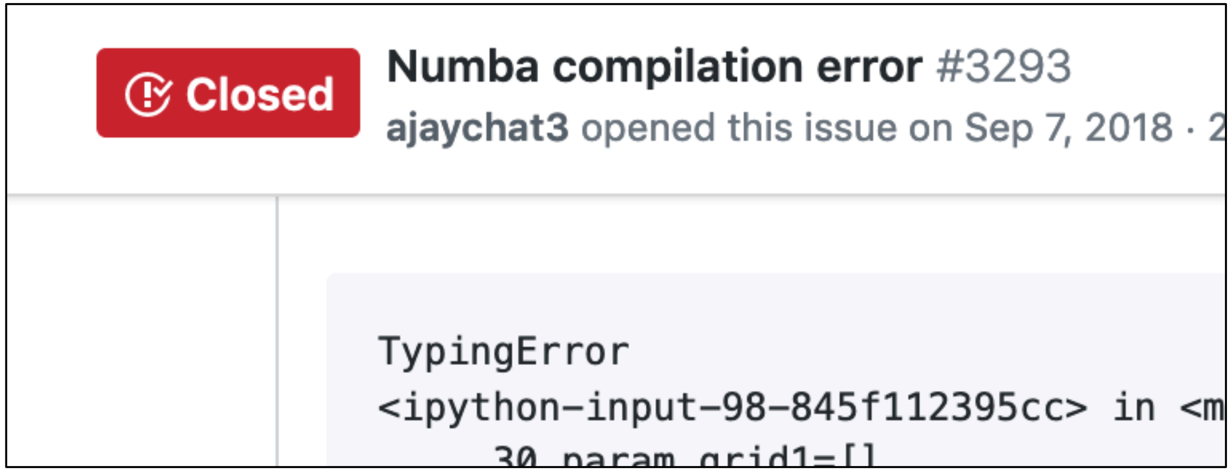
Problem: Huge Developer Effort

- 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!

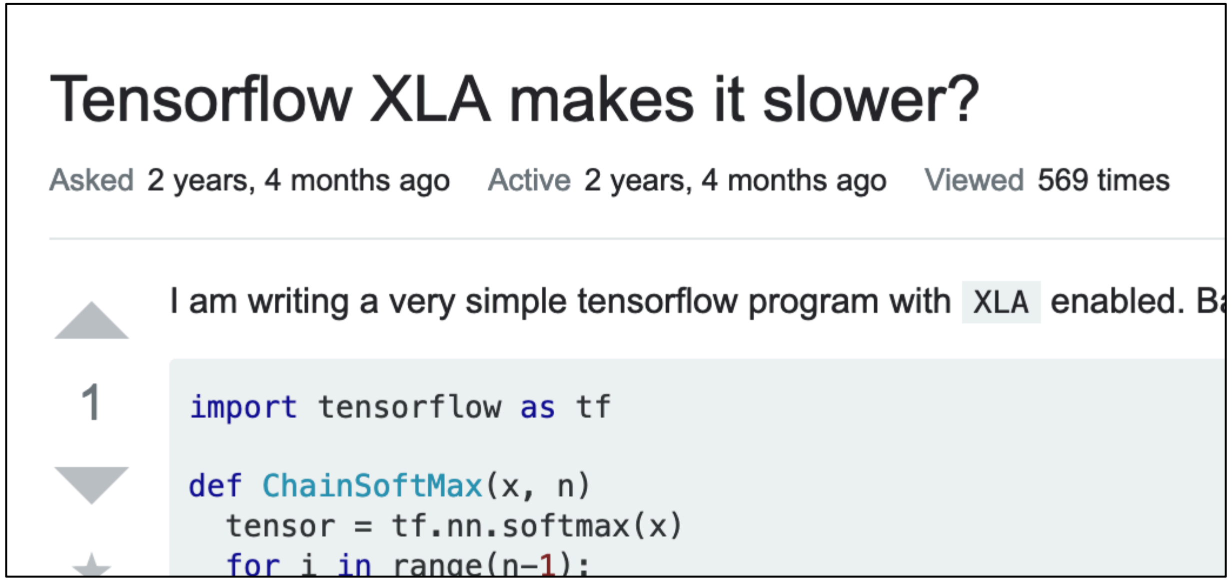


Numba compilation error #3293

ajaychat3 opened this issue on Sep 7, 2018 · 2

```
TypingError
<ipython-input-98-845f112395cc> in <m
    30 param_grid1=[]
```

“Sorry, our compiler doesn’t recognize this pattern yet”



Tensorflow XLA makes it slower?

Asked 2 years, 4 months ago Active 2 years, 4 months ago Viewed 569 times

I am writing a very simple tensorflow program with `XLA` enabled. B

```
import tensorflow as tf

def ChainSoftMax(x, n)
  tensor = tf.nn.softmax(x)
  for i in range(n-1):
```

“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:

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

SAs Enable Pipelining + Parallelism

Without SAs:

d1

price

strike

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

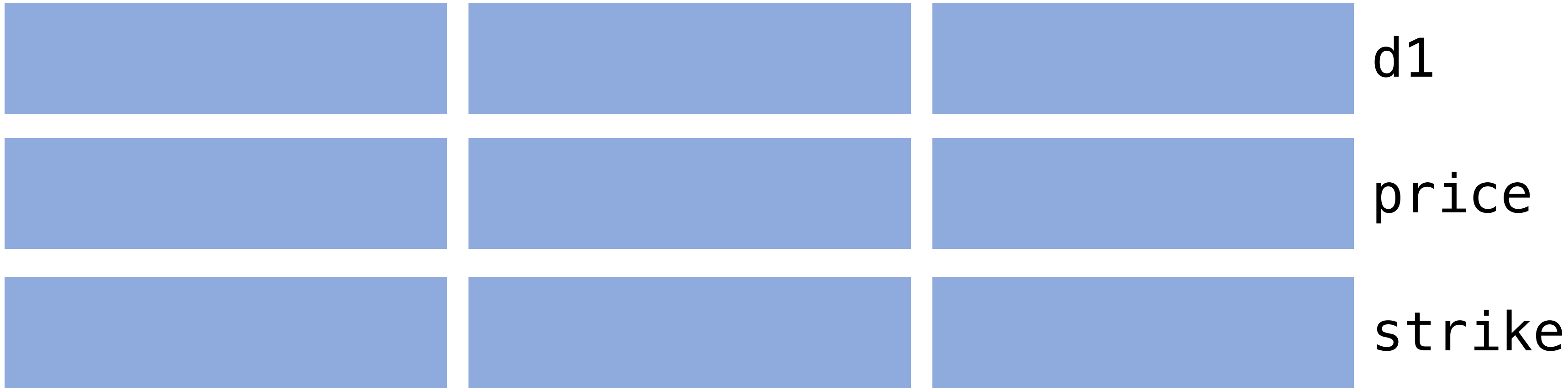
SAs Enable Pipelining + Parallelism

With SAs:

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

SAs Enable Pipelining + Parallelism

With SAs:

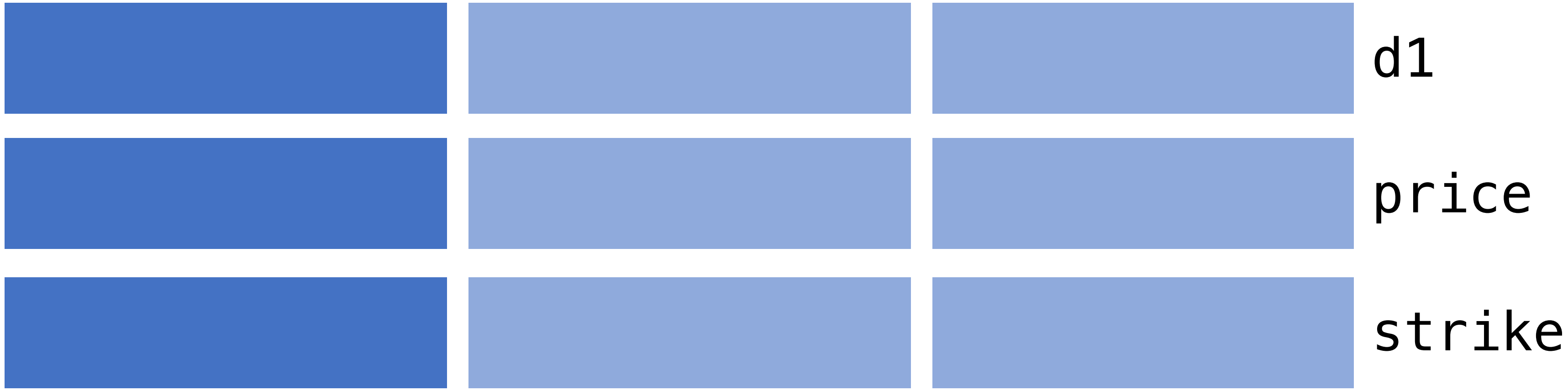


```
d1 = price * strike
d1 = np.log2(d1) + strike
```

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

SAs Enable Pipelining + Parallelism

With SAs:



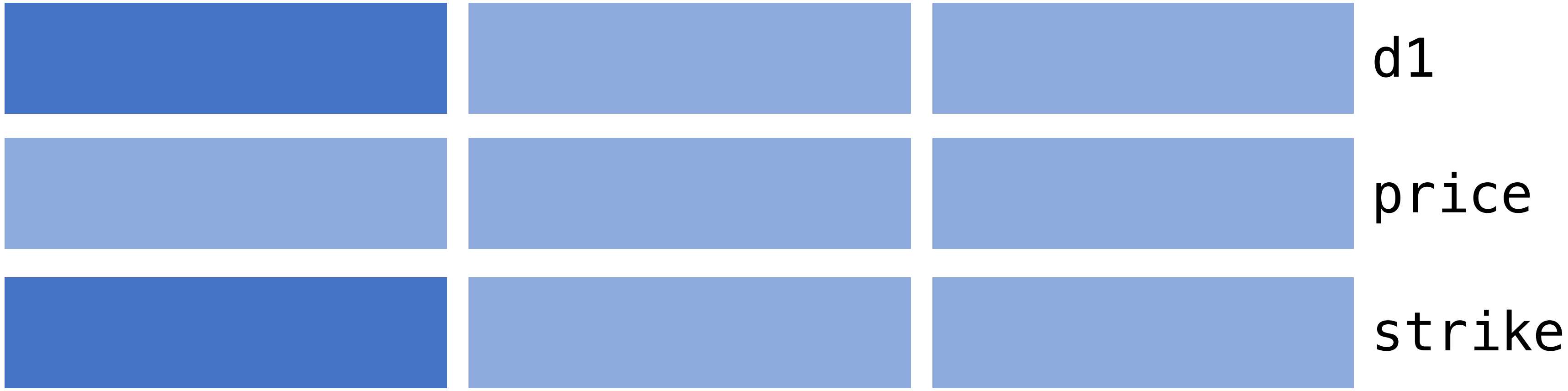
Collectively fit in cache

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

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

SAs Enable Pipelining + Parallelism

With SAs:



Collectively fit in cache

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

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

SAs Enable Pipelining + Parallelism

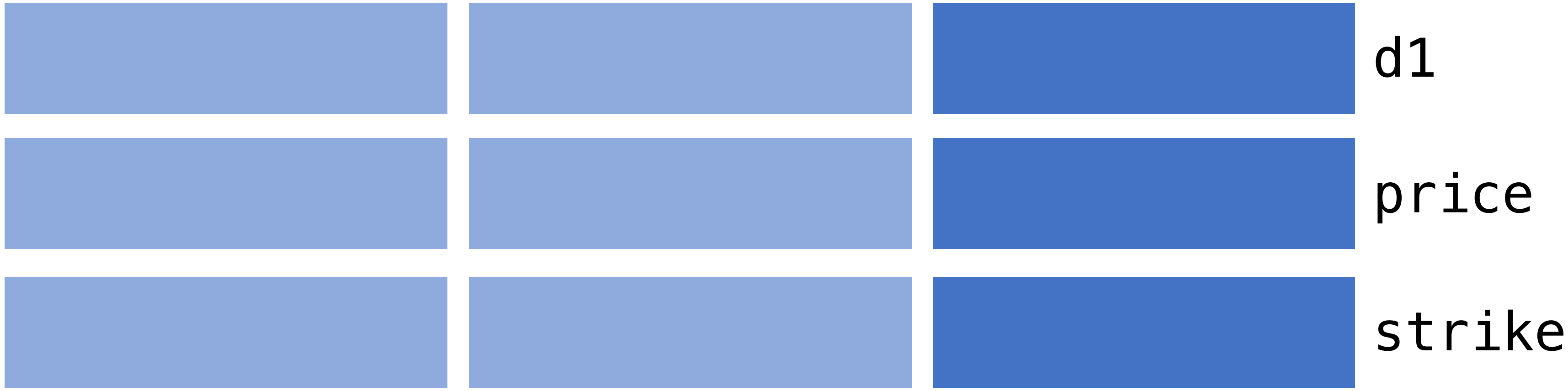
With SAs:

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

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

SAs Enable Pipelining + Parallelism

With SAs:

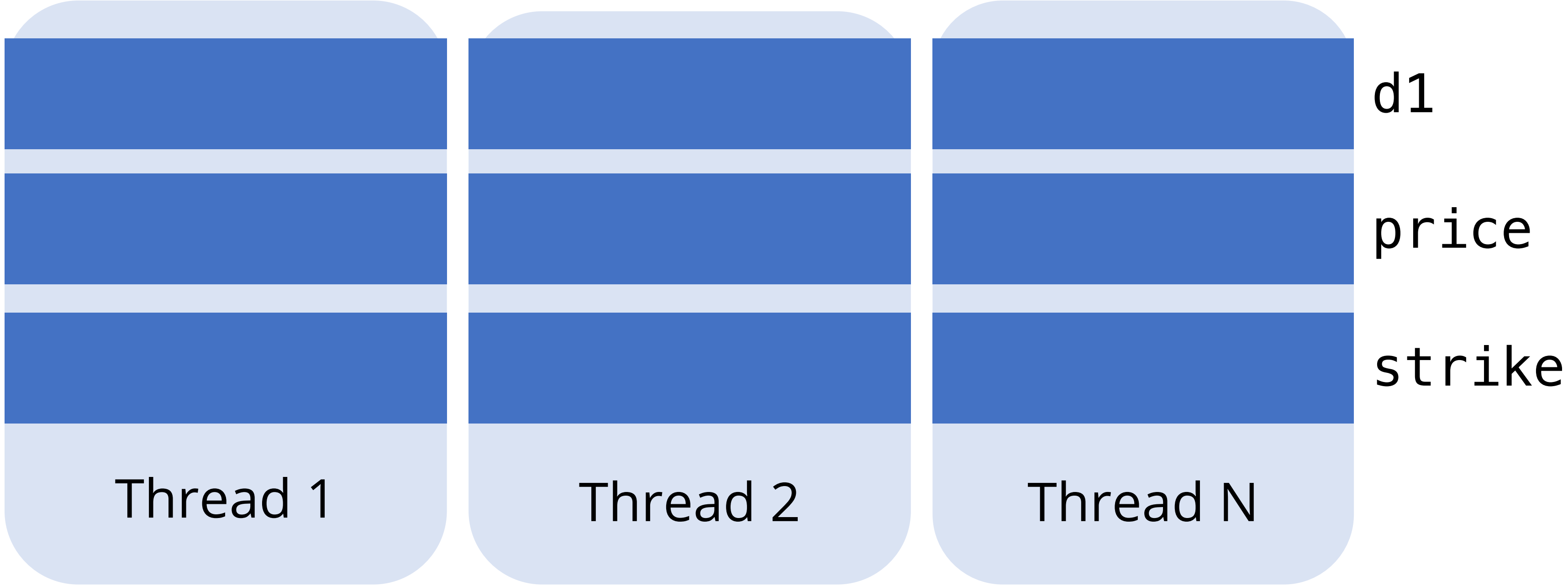


Build execution graph, **keep data in cache** by passing cache-sized splits to functions.

```
d1 = price * strike
d1 = np.log2(d1) + strike
```

SAs Enable Pipelining + Parallelism

With SAs:



Parallelize over split pieces

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

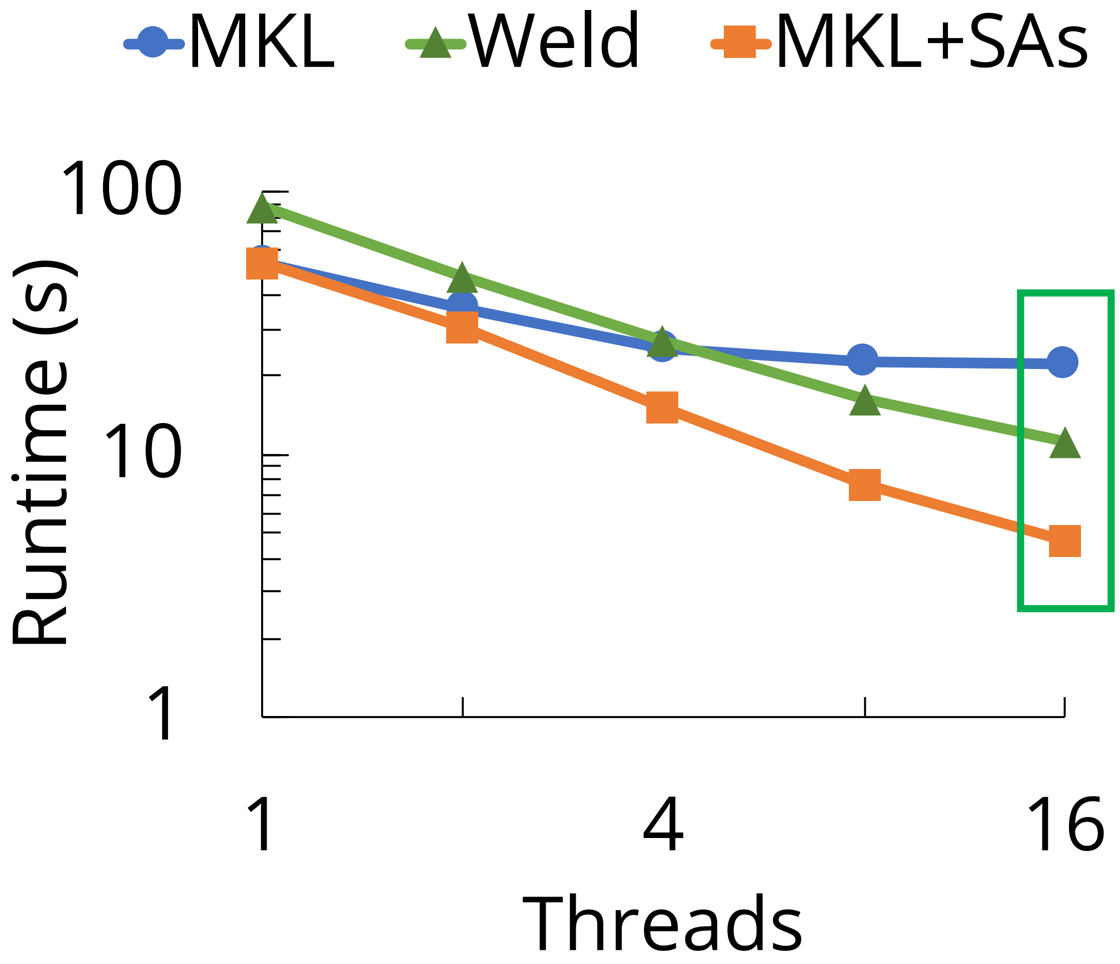
Example of a split annotation for MKL

```
@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

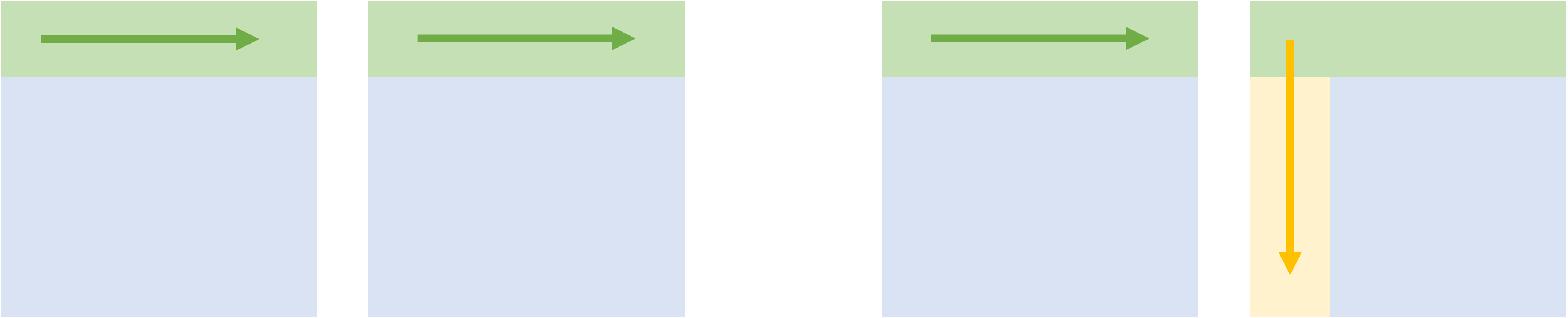
Challenges in designing SAs

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 <u>safe</u> 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.

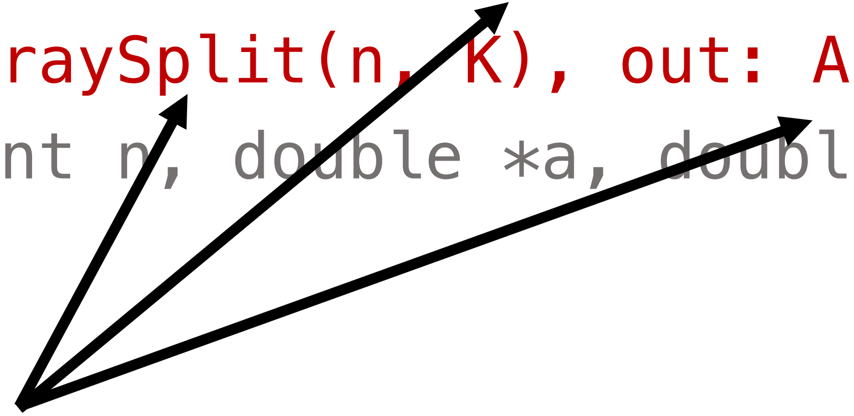
```
@sa(n: SizeSplit(n, K), a: ArraySplit(n, K),
        b: ArraySplit(n, K), out: ArraySplit(n, K))
void vdAdd(int n, double *a, double *b, double *out)
```

`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**.

```
@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)
```

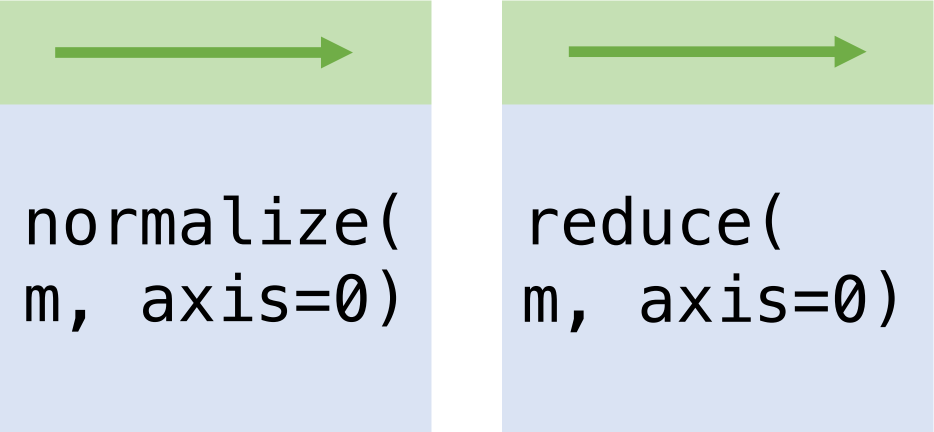


Same split types enforce values split in the same way: we **can pipeline** if data between functions has matching split types.

Example: Matrix Pipelining in NumPy

Split type for NumPy matrices encodes dimension + axis:

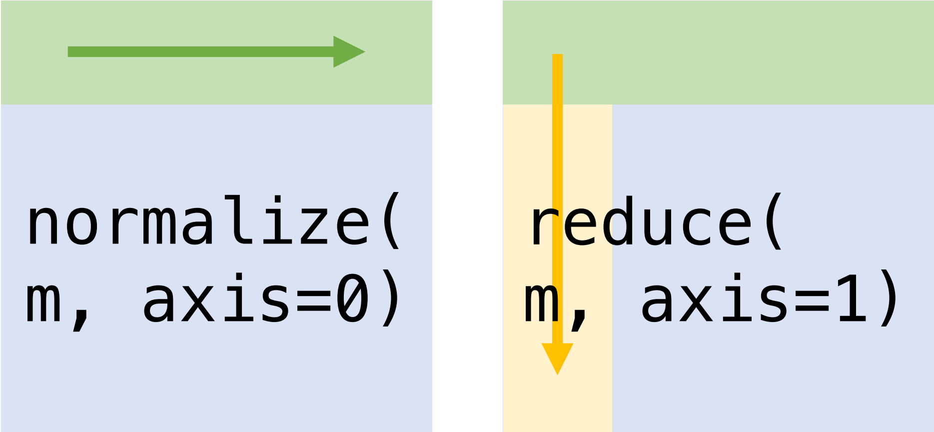
`MatrixSplit(Rows, Cols, Axis, K)`



```
normalize(
m, axis=0)
```

```
reduce(
m, axis=0)
```

Split types match: `axis=0` for both function calls



```
normalize(
m, axis=0)
```

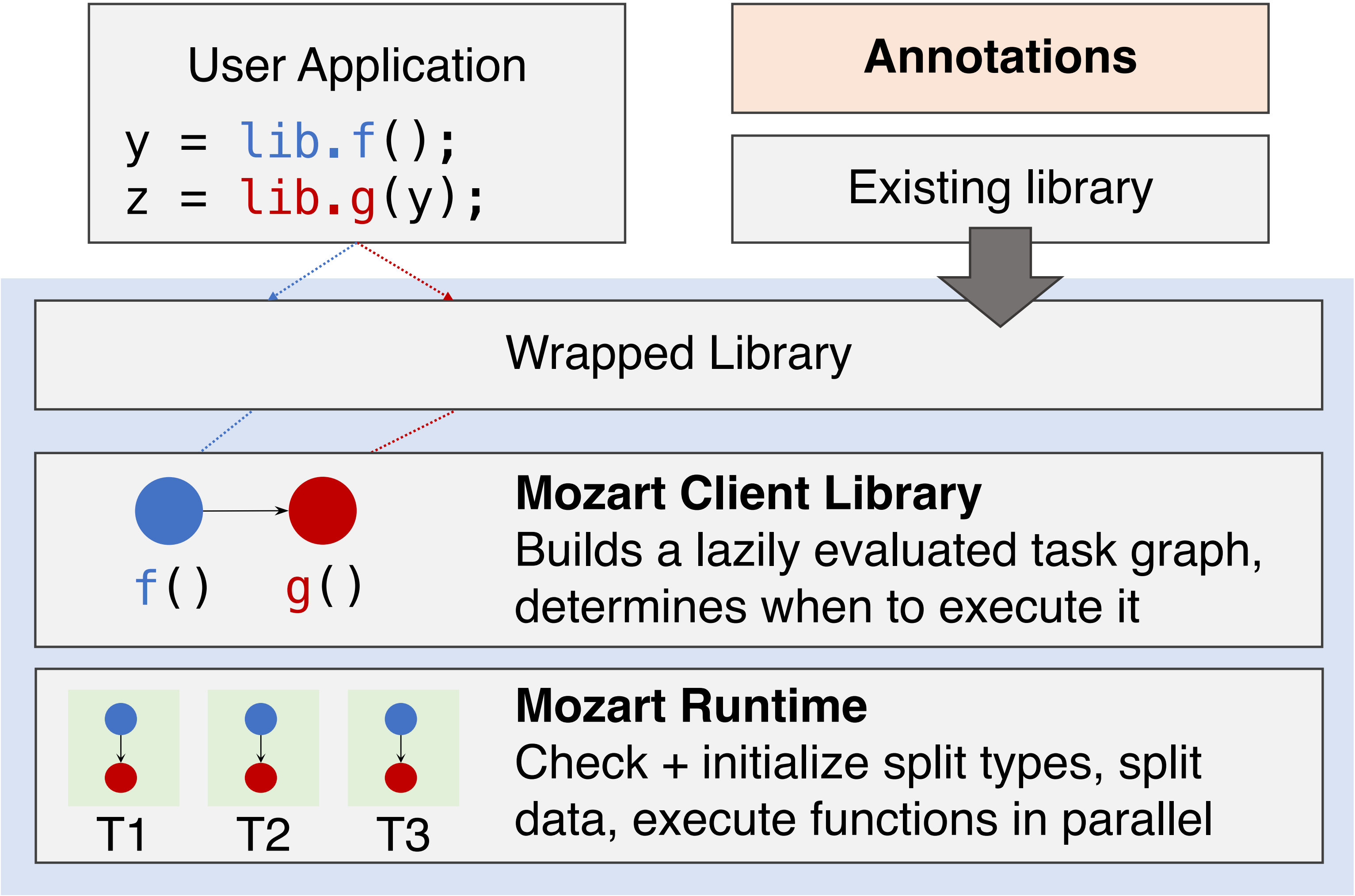
```
reduce(
m, axis=1)
```

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 = lib.f();
z = lib.g(y);
```

Annotations

Existing library

Wrapped Library

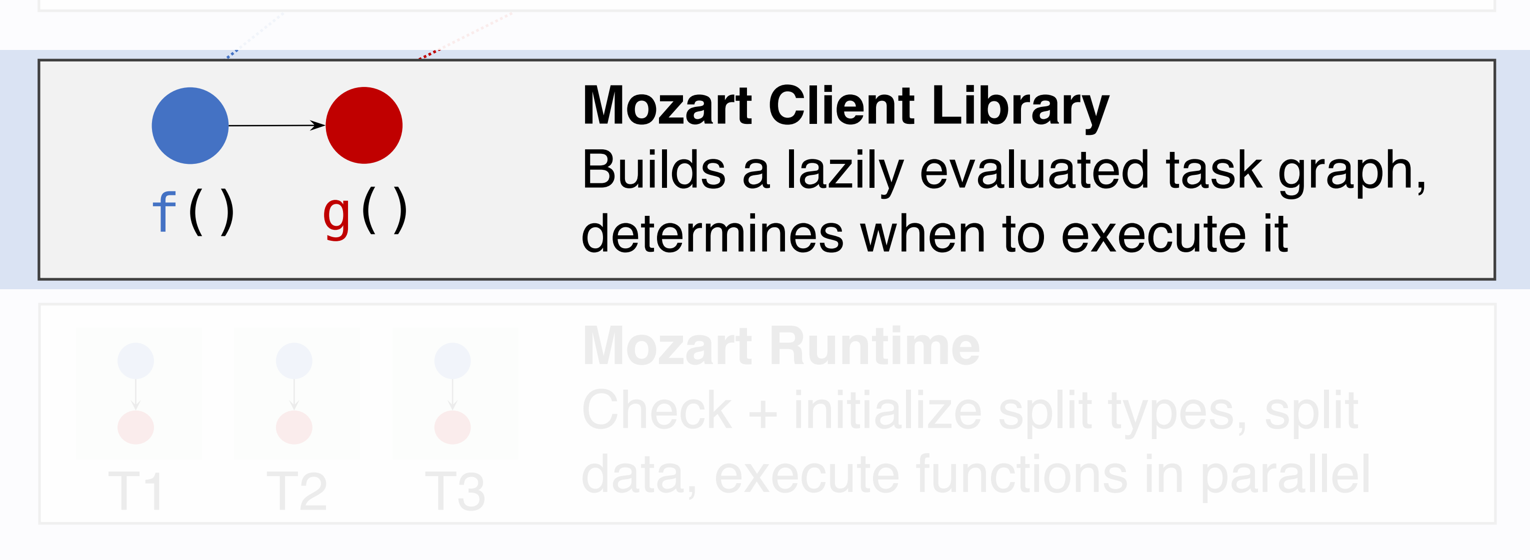
Mozart Client Library
Builds a lazily evaluated task graph, determines when to execute it

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

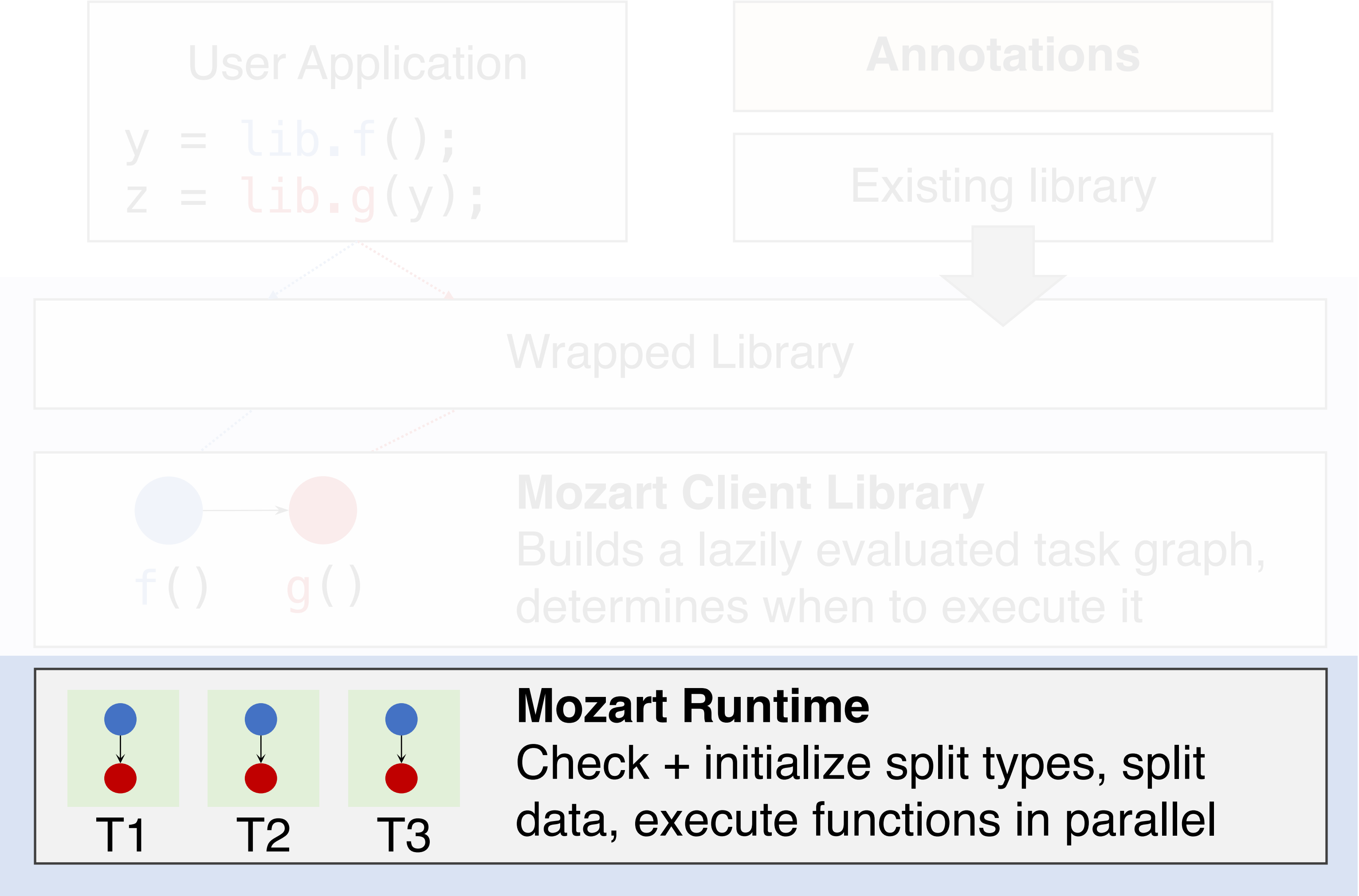
See paper for details!



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 = lib.f();
z = 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:

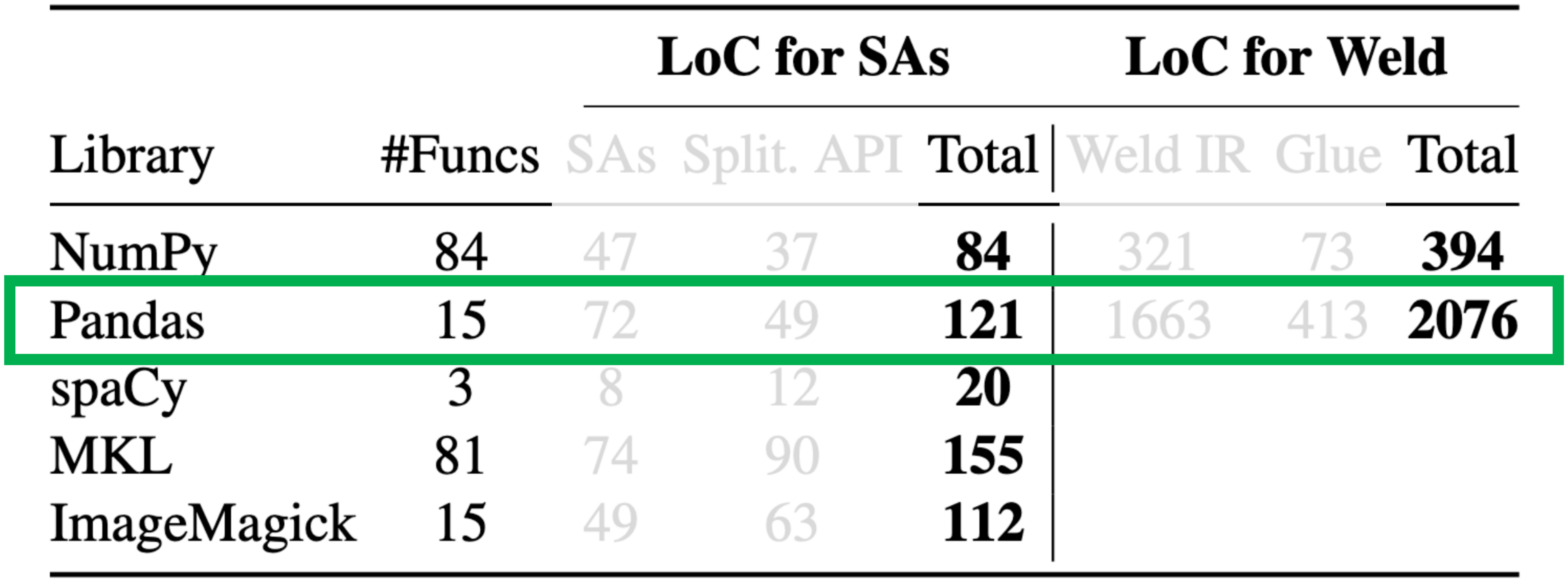
1. What kinds of workloads can SAs accelerate?
2. How much effort is required to use SAs vs. compilers?
3. How do SAs perform compared to JIT compilers?

Data Types and Libraries Demonstrated

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

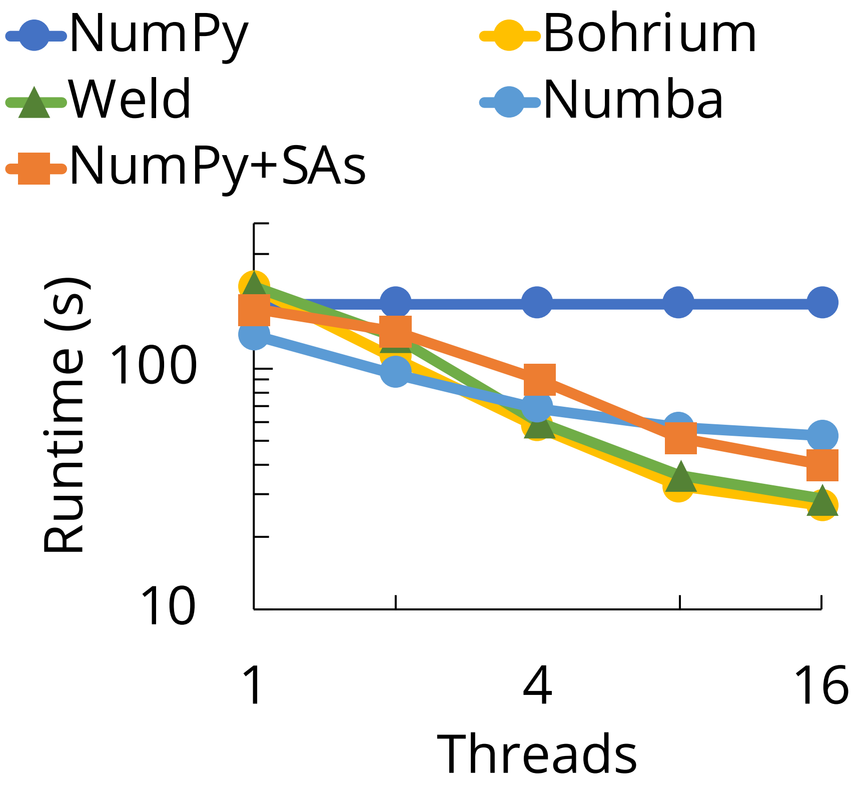
Data types and operators: Arrays, Tensors, Matrices, DataFrame joins, grouping aggregations, image processing algorithms, functional operators (map, reduce, etc.)

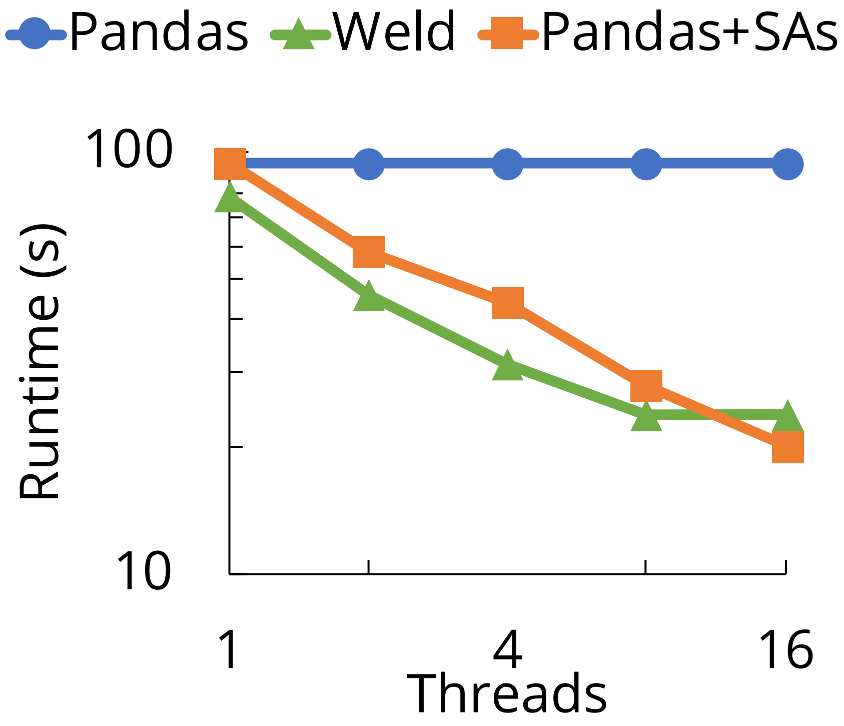
SAs require less integration effort than compilers



		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**			

SAs can match JIT compilers under existing APIs



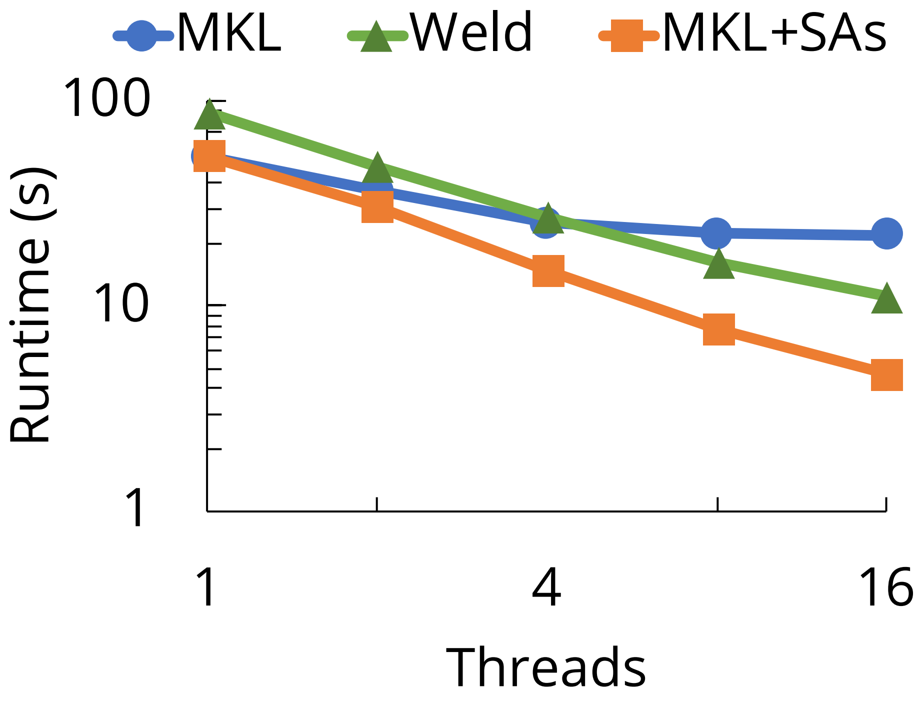


nBody simulation: **4.6x speedup** over NumPy

pandas

Birth Analysis: **4.7x speedup** over pandas

SAs can accelerate highly optimized libraries



Black Scholes: **5x speedup** over MKL

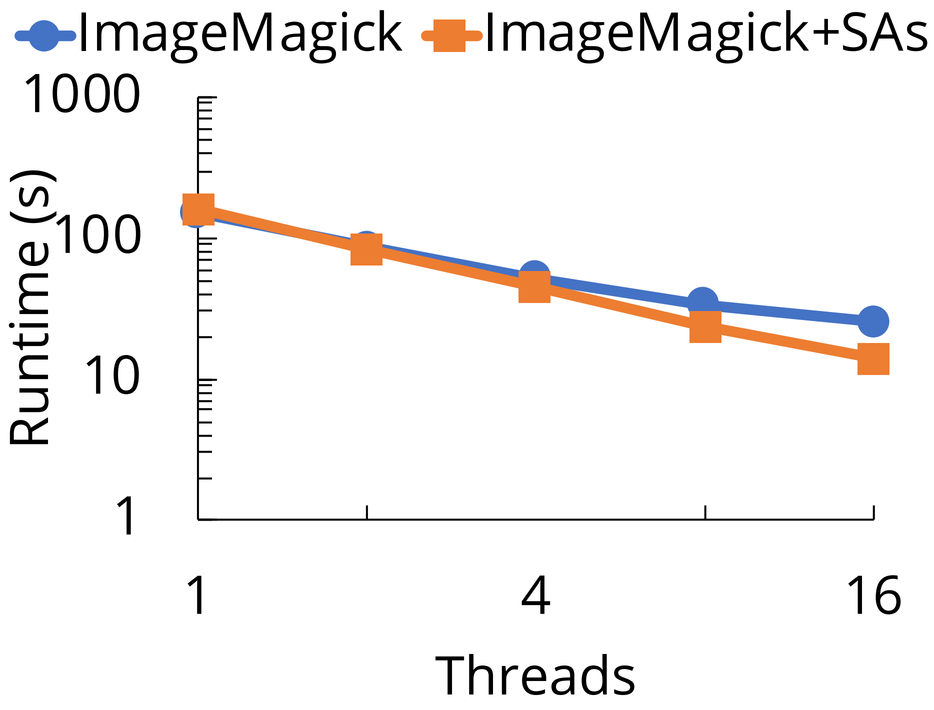


Image filter: **1.8x speedup** over ImageMagick

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

Artifacts Available acm

Artifacts Evaluated Functional acm