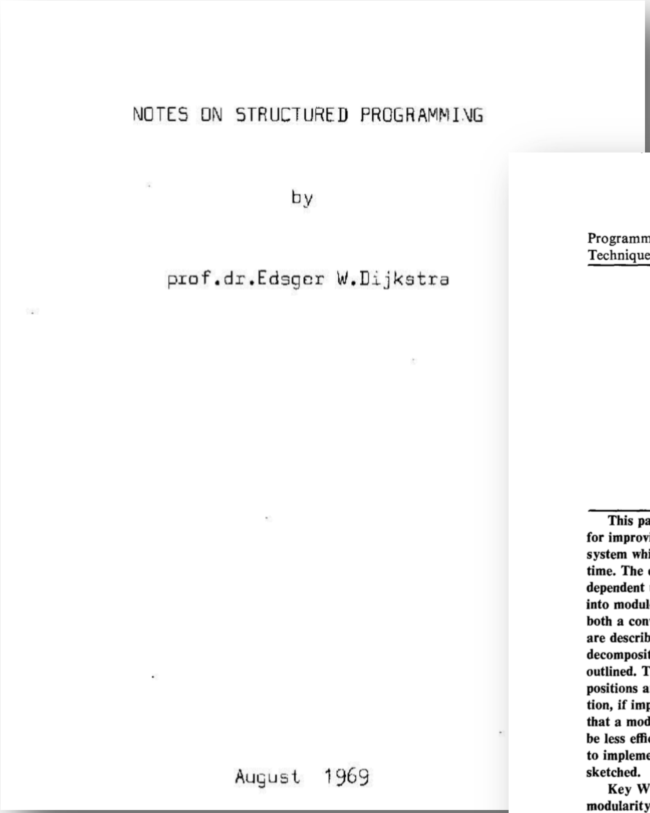
Interfaces for Efficient Software Composition on Modern Hardware

Shoumik Palkar

Dissertation Defense

April 2, 2020

Software composition: A mainstay for decades!

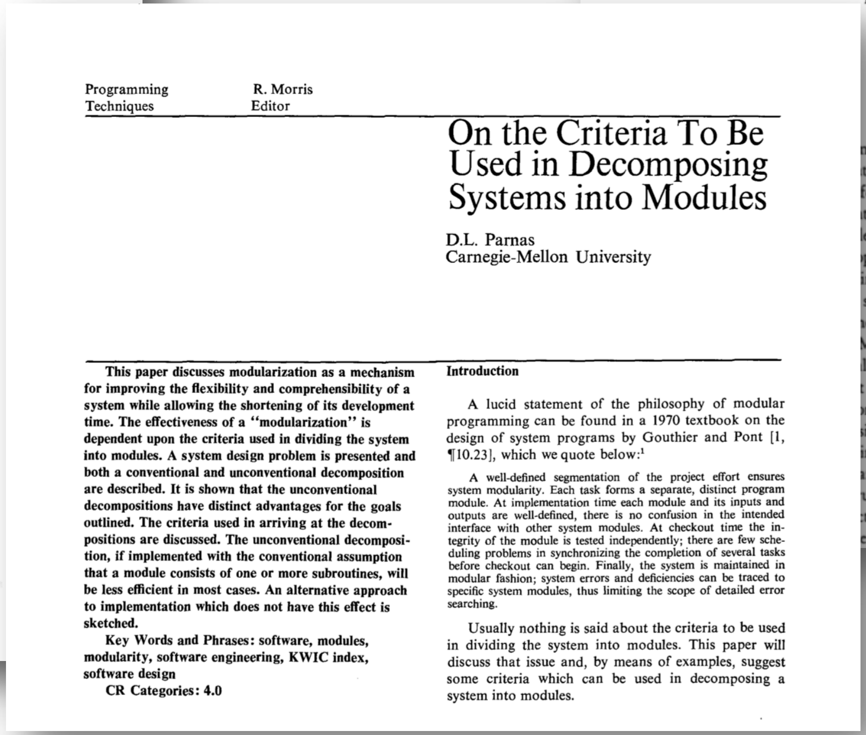


NOTES ON STRUCTURED PROGRAMMING

by

prof.dr.Edsger W.Dijkstra

August 1969



Programming Techniques — R. Morris, Editor

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design

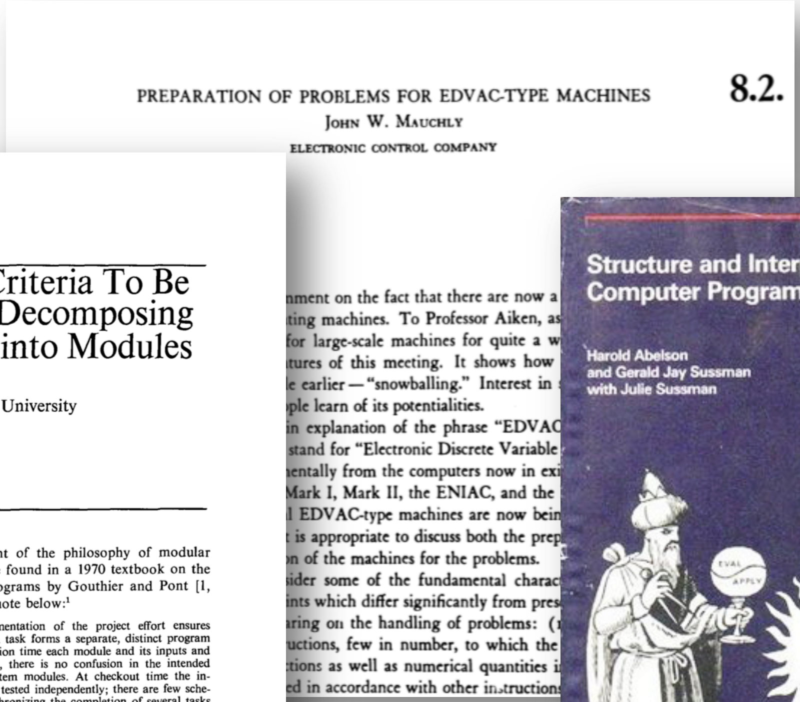
CR Categories: 4.0

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

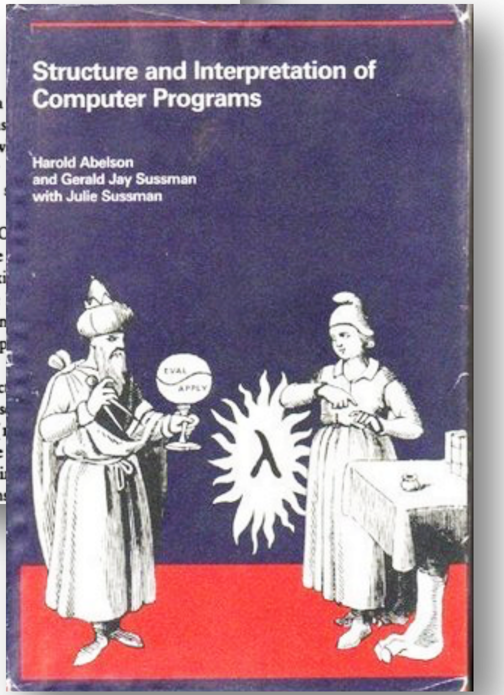


PREPARATION OF PROBLEMS FOR EDVAC-TYPE MACHINES — 8.2.

JOHN W. MAUCHLY

ELECTRONIC CONTROL COMPANY

nment on the fact that there are now a
ting machines. To Professor Aiken, as
for large-scale machines for quite a w
tures of this meeting. It shows how
e earlier—"snowballing." Interest in
ple learn of its potentialities.
in explanation of the phrase "EDVAC
stand for "Electronic Discrete Variable
entally from the computers now in exi
Mark I, Mark II, the ENIAC, and the
l EDVAC-type machines are now bein
t is appropriate to discuss both the prep
n of the machines for the problems.
sider some of the fundamental charac
nts which differ significantly from pres
aring on the handling of problems: (
uctions, few in number, to which the
tions as well as numerical quantities i
ed in accordance with other instructions



The result? An ecosystem of libraries + users

Intel Math Kernel Library

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

ImageMagick

scikit learn

NumPy

statsmodels

SciPy

K Keras

OpenCV

PYTORCH

SQLite

XGBoost

xtensor

Example: ML pipeline in Python

Example: ML pipeline in Python

\+ **Users can leverage** 1000s of expertly-developed libraries across many different domains

\- On modern hardware, composition is **no longer a "zero-cost" abstraction**

Example: the function call interface

Used to pass data between functionality via pointers to in-memory values.

```
void vdLog(float* a, float* out, size_t n) {
  for (size_t i = 0; i + 8 < n; i += 8) {
    __m256 v = _mm256_loadu_ps(a + i);
    ...
    _mm256_log2_ps(v, ...);
    ...
```

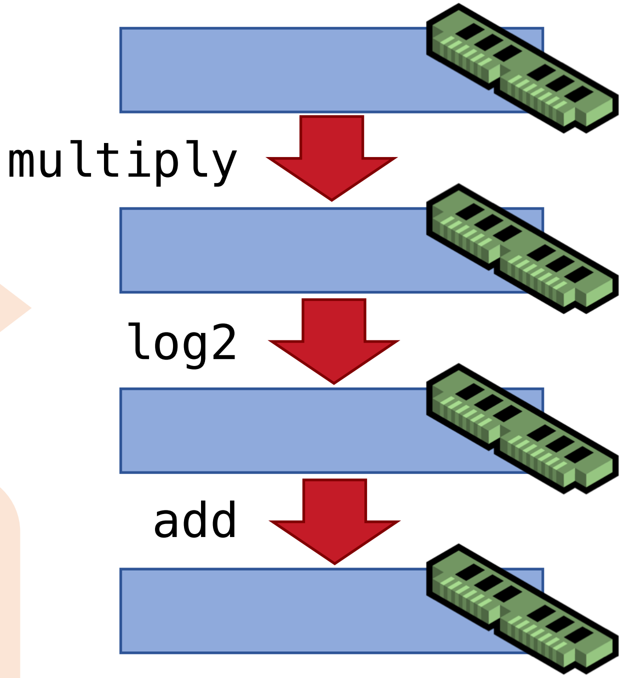
(1) Pass args through stack

Performance gap between these is growing!

Example: composition with function calls

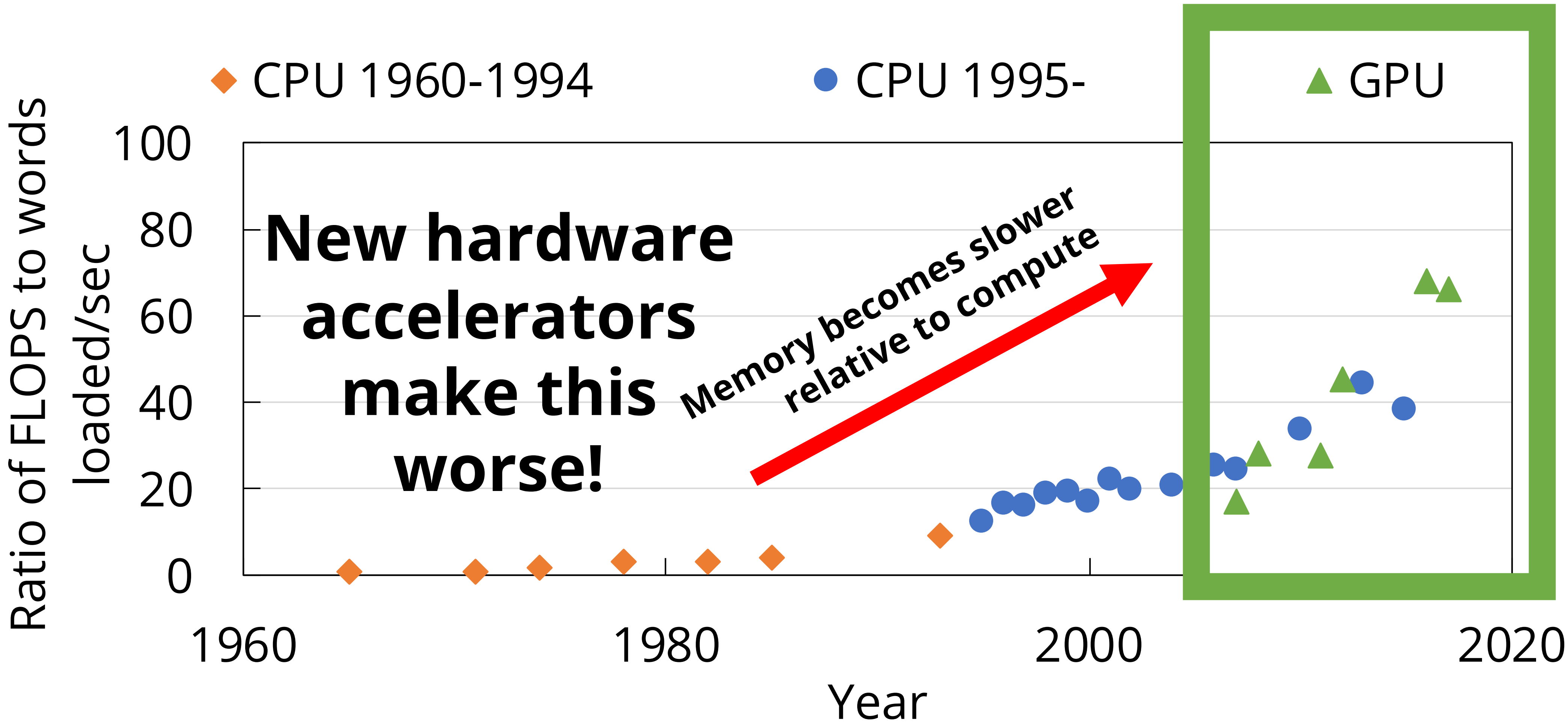
Growing gap between memory/processing speed makes function call interface worse!

```
// 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

Hardware Trends are Shifting Bottlenecks



1. Kagi et al. 1996. Memory Bandwidth Limitations of Future Microprocessors. ISCA 1996
2. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. TCCA 1995.

Do we need a new way to combine software?

- **Strawman: use a monolithic system**
  - “Legacy" applications: thousands of users of existing APIs
  - **Example:** Community of data scientists who use optimized Python libraries

- **Strawman: always use low-level languages (e.g., C++) or optimize manually**
  - Optimizations [still] require lots of manual work
  - **Example:** Manual optimizations in MKL-DNN

Challenges for software composition today

Research vision: make software composition a zero-cost abstraction again!

My Research: new interfaces to compose software on modern hardware

Key idea: Use ***algebraic properties*** of software APIs in ***new interfaces*** to enable new optimizations

Examples of algebraic properties:

- **`F()`**'s loops can be fused with **`G()`**'s loops
- **`F()`**'s args can be split + pipelined with **`G()`**
- **`F()`** is parallelizable after externally splitting its args

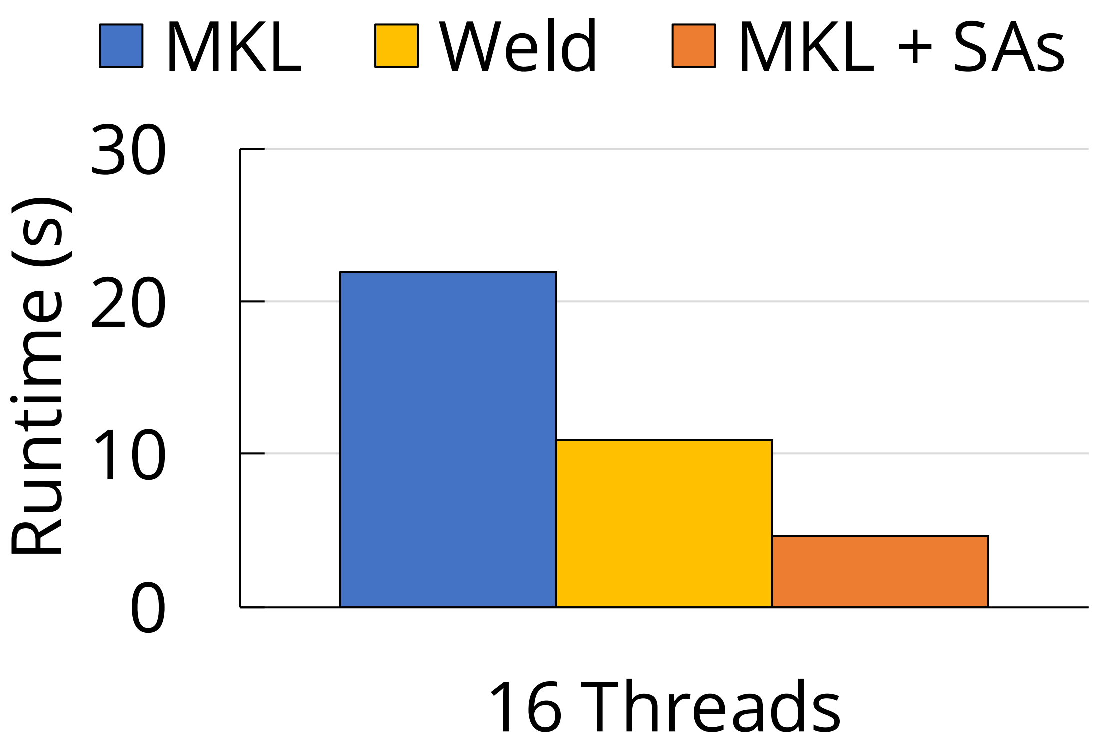
My Approach: Three interfaces with new systems to leverage their properties

Name	Interface/Properties	System
Weld		
Split annotations		
Raw filtering		

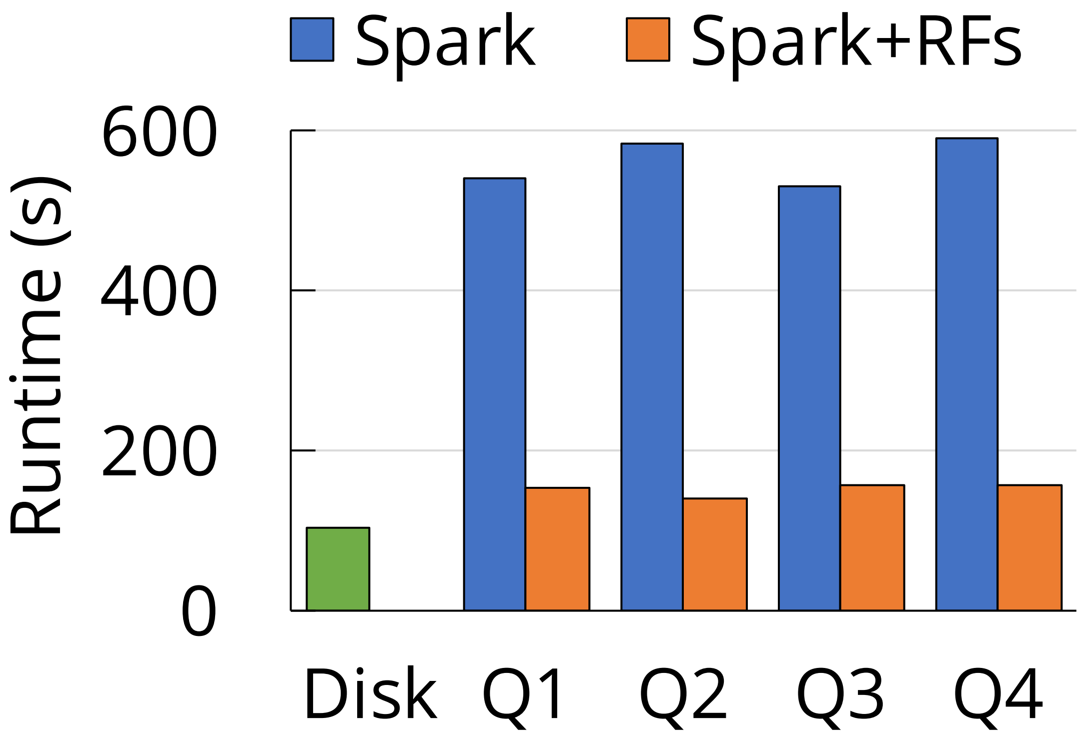
Focus: Data movement optimization and automatic parallelization over existing library APIs

Focus: I/O optimization via data loading

Preview: What a new interface can achieve



Black Scholes model with Intel MKL: **3-5x** speedup with Weld and SAs



Querying 650GB of Censys JSON data in Spark: **4x** speedup with raw filtering

Rest of this Talk

- Weld
- Split annotations
- Raw filtering
- Impact, open source, and concluding remarks

Weld: A Common Runtime for Data Analytics

CIDR '17

PVLDB '18

Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, Matei Zaharia

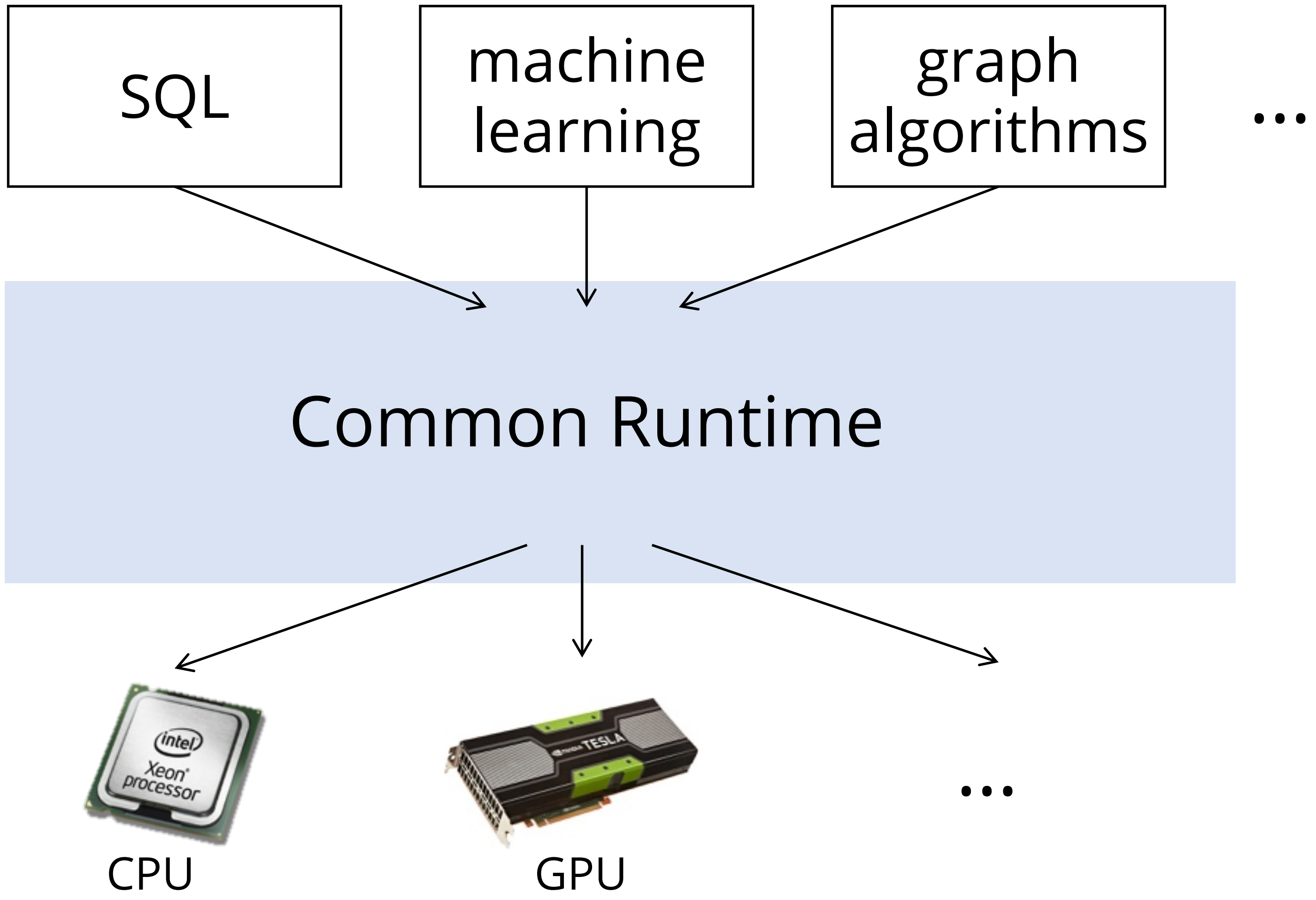
Motivation for Weld

+ Ecosystem of 100s of existing libraries and APIs
– Combining these libraries is no longer efficient!

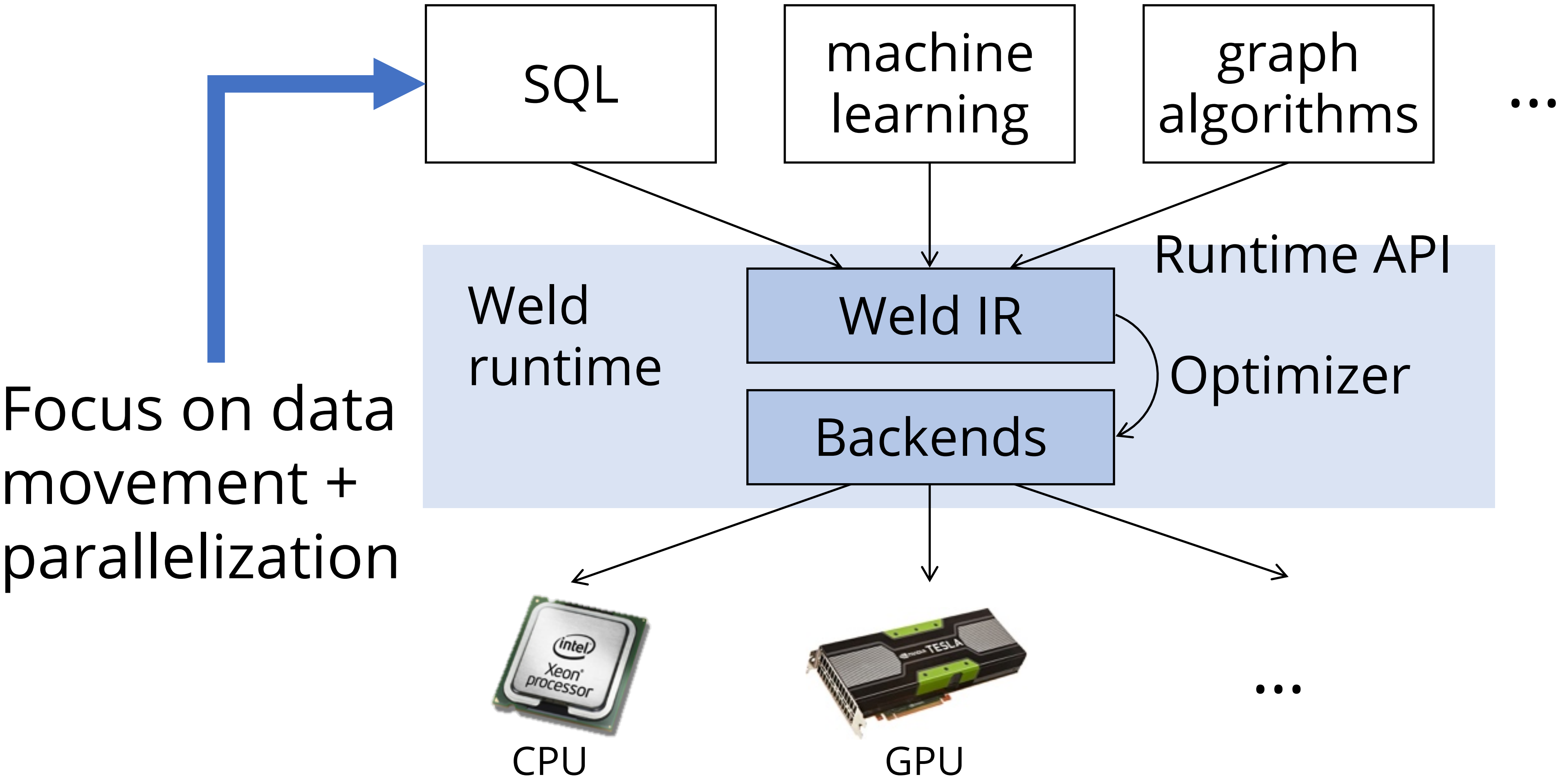
Example: Normalizing images in NumPy + classifying them in with log. reg. in TensorFlow: **13x difference** compared to an end-to-end optimized implementation

Can we enable existing APIs to compose efficiently on modern hardware?

Weld: A Common Runtime for Data Analytics

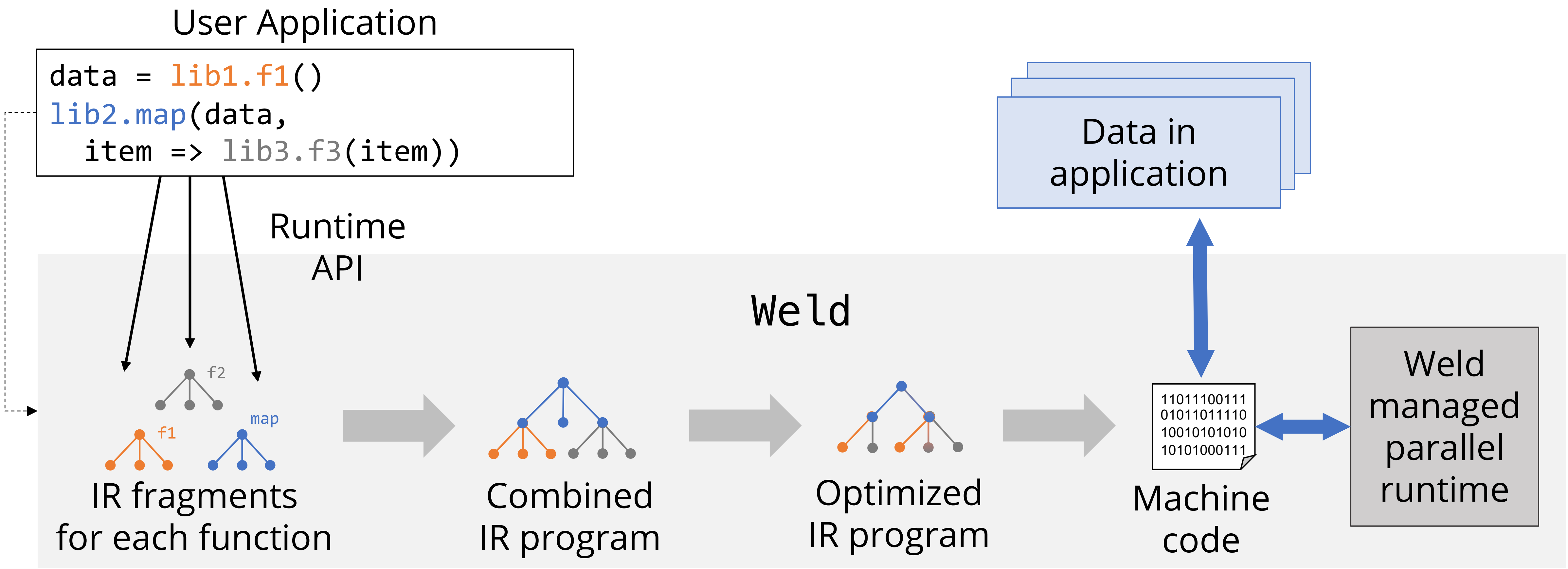


Weld: A Common Runtime for Data Analytics



Weld’s Runtime API

Runtime API uses lazy evaluation



Weld’s IR

Weld IR: Expressing Computations

Designed to meet three goals:

1. **Generality**
   support diverse workloads and nested calls

2. **Ability to express optimizations**
   e.g., loop fusion, vectorization, and loop tiling

3. **Explicit parallelism**

Weld IR: Internals

Small "functional" IR with two main constructs.

Parallel loops: iterate over a dataset

Builders: declarative objects to produce results

- *E.g.*, append items to a list, compute a sum
- Different implementations on different hardware
- Read after writes: enables mutable state

Captures relational algebra, functional APIs like Spark, linear algebra, and composition thereof

Weld's Loops and Builders

Example: Functional Operators

```
def map(data, f):
    builder = new appender[T]
    for x in data:
        merge(builder, f(x))
    result(builder)
```

Builder that appends items to a list.

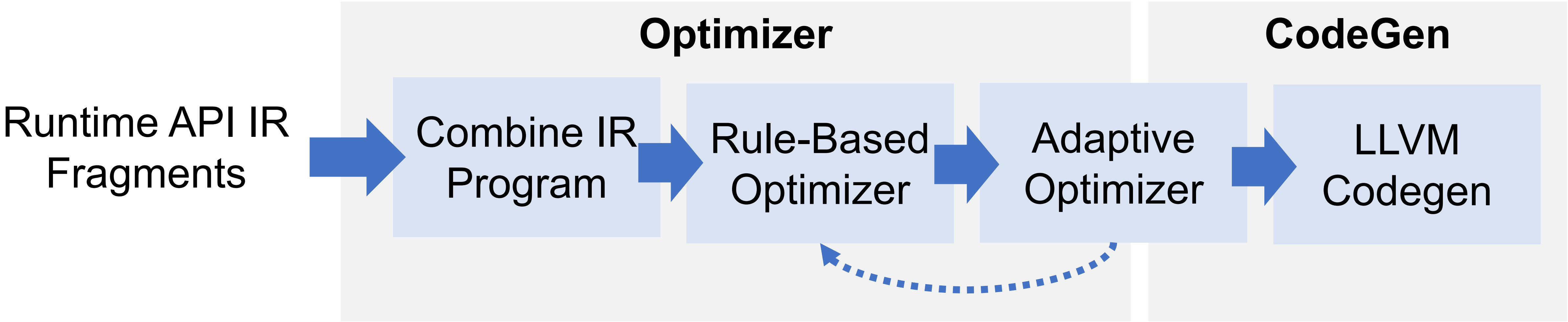
```
def reduce(data, zero, func):
    builder = new merger[zero, func]
    for x in data:
        merge(builder, x)
    result(builder)
```

Builder that aggregates a value.

Weld’s Optimizer

Optimizer Goal

Remove **redundancy** caused by composing independent libraries and functions.



Removing Redundancy

Rule-based optimizations for removing redundancy in generated Weld code.

Before:

```
tmp = map(data, |x| x * x)
res1 = reduce(tmp, 0, +)     // res1 = data.square().sum()
res2 = map(data, |x| sqrt(x))// res2 = np.sqrt(data)
```

Each line generated by separate function.

- Unnecessary materialization of `tmp`
- Two traversals of `data`
- Vectorization? Output size inference?

Removing Redundancy

Rule-based optimizations for removing redundancy in generated Weld code.

Before:

```
tmp = map(data, |x| x * x)
res1 = reduce(tmp, 0, +)
res2 = map(data, |x| sqrt(x))
```

After:

```
bld1 = new merger[0, +]
bld2 = new appender[i32]
          (len(data))
for x: simd[i32] in data:
    merge(bld1, x * x)
    merge(bld2, sqrt(x))
```

Removing Redundancy

Rule-based optimizations for removing redundancy in generated Weld code.

Before:

```
tmp = map(data, |x| x * x)
res1 = reduce(tmp, 0, +)
res2 = map(data, |x| sqrt(x))
```

After:

```
bld1 = new merger[0, +]
bld2 = new appender[i32]
          (len(data))
for x: simd[i32] in data:
    merge(bld1, x * x)
    merge(bld2, sqrt(x))
```

Example: Loop Fusion Rule to Pipeline Loops

Removing Redundancy

Rule-based optimizations for removing redundancy in generated Weld code.

Before:

```
tmp = map(data, |x| x * x)
res1 = reduce(tmp, 0, +)
res2 = map(data, |x| sqrt(x))
```

After:

```
bld1 = new merger[0, +]
bld2 = new appender[i32]
           (len(data))
for x: simd[i32] in data:
    merge(bld1, x * x)
    merge(bld2, sqrt(x))
```

Example: Vectorization to leverage SIMD in CPUs

Results

Partial Integrations with Several Libraries

Libraries: NumPy, Pandas, TensorFlow, Spark SQL

Evaluated on 10 data science workloads

+ microbenchmarks vs. specialized systems

Weld Enables Cross-Library Optimization

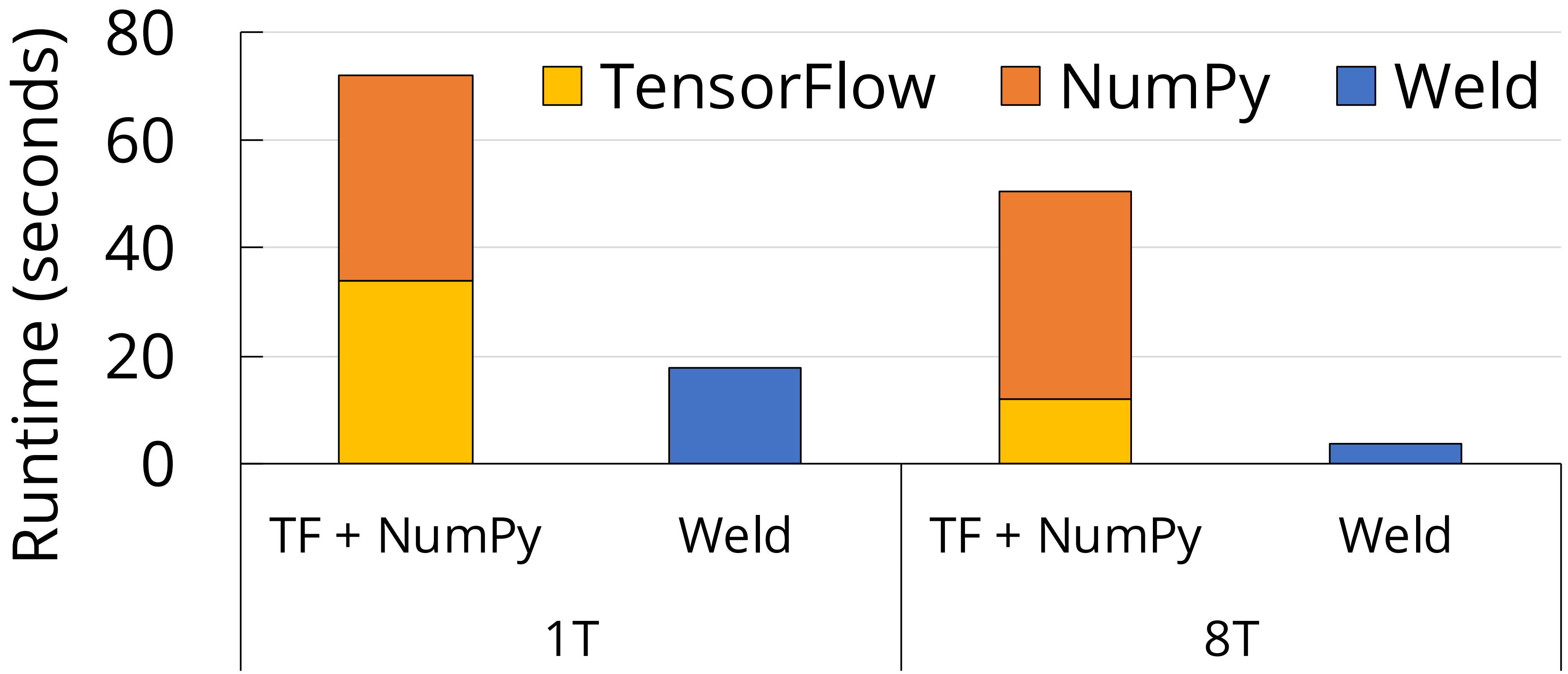
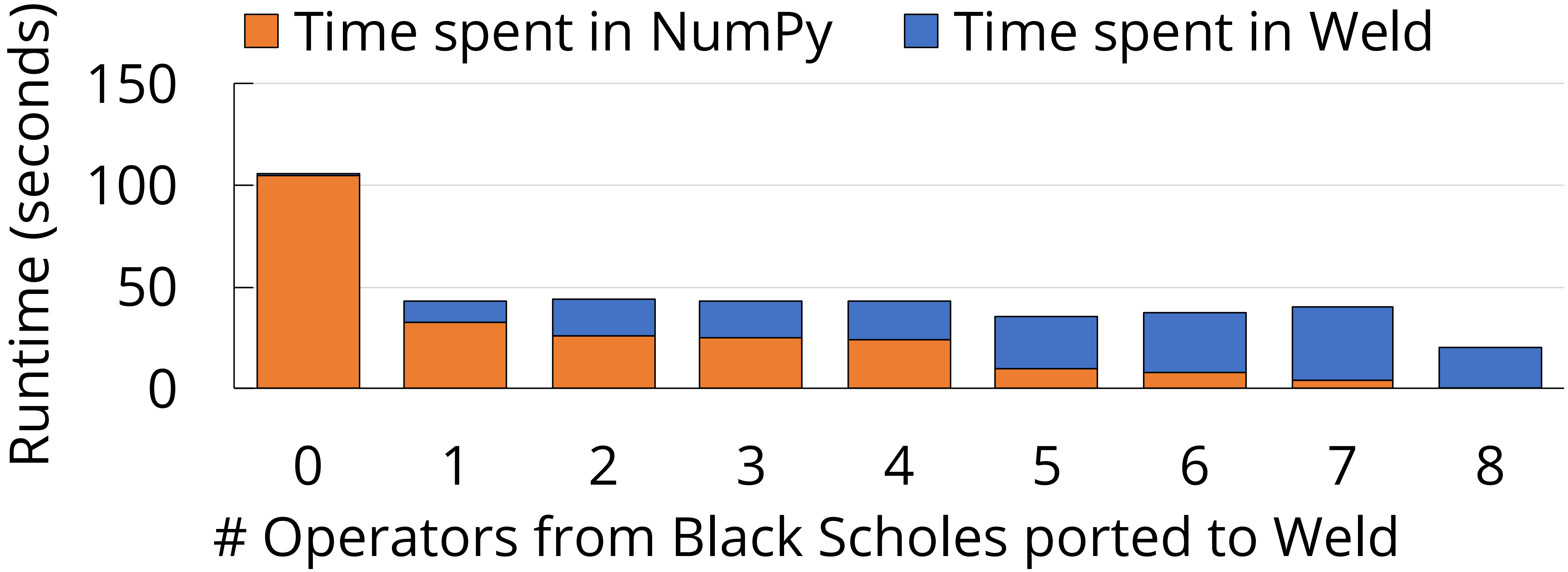


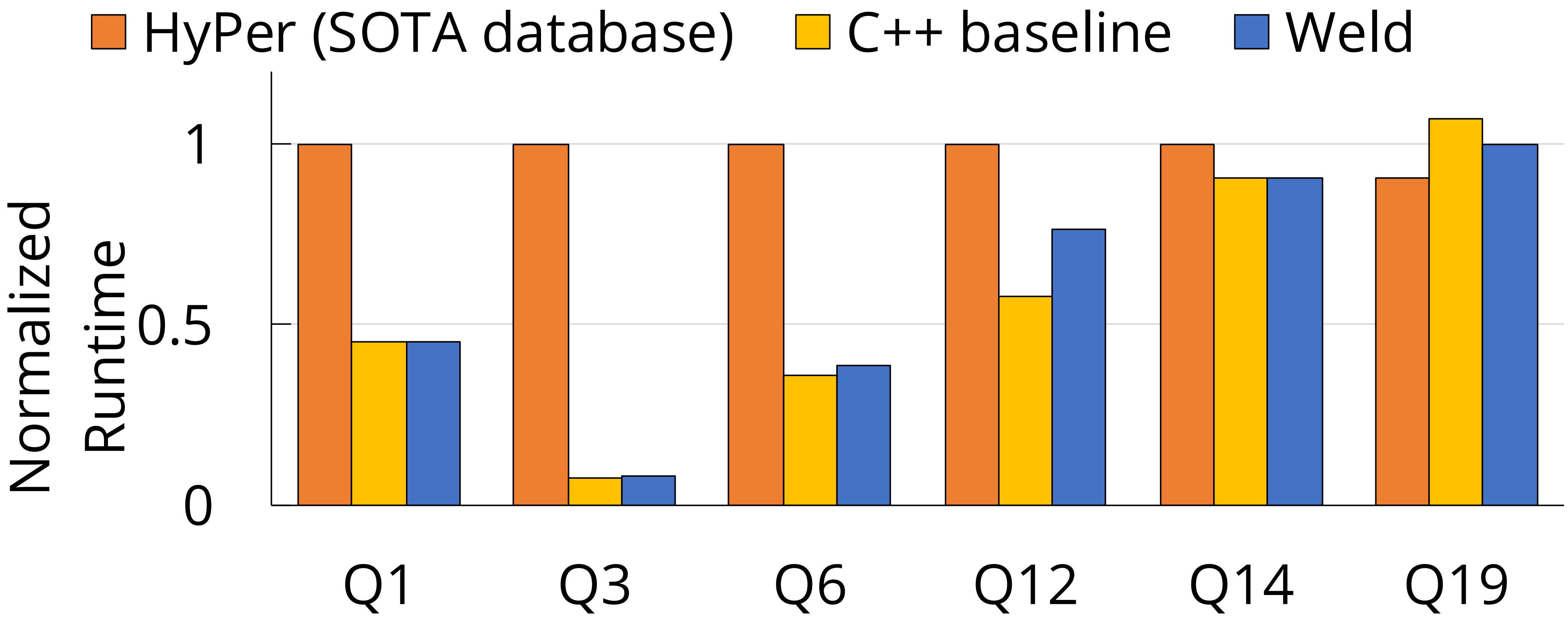
Image whitening + logistic regression classification with NumPy + TensorFlow: **13x** speedup

Weld can be integrated incrementally



Benefits with incremental integration.

Weld enables high quality code generation



SQL: Competitive with state-of-the-art and handwritten baseline (other benchmarks open source!)

Impact of Optimizations: 8 Threads

Experiment	All	-Fuse	-Unrl	-Pre	-Vec	-Pred	-Grp	-ADS	-CLO
DataClean	1.00	2.44	0.97	0.99	0.98	0.95			
CrimeIndex	1.00	195	2.04	1.00	1.02	0.96			3.23
BlackSch	1.00	6.68		1.44	1.95		1.64		
Haversine	1.00	3.97		1.20	1.02				
Nbody	1.00	1.78		2.22	1.01				
BirthAn	1.00	1.02		0.97	0.98				1.00
MovieLens	1.00	1.07		1.02	0.98				1.09
LogReg	1.00	20.18		1.00					2.20
NYCFilter	1.00	9.99		1.20	1.23	2.79			
FlightDel	1.00	1.27		1.01	0.96	0.96	5.50		1.47

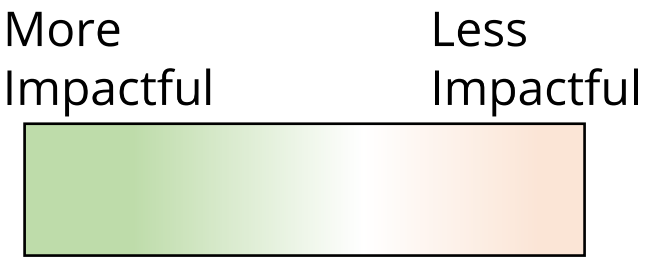
More Impactful

Less Impactful

All optimizations enabled.

Impact of Optimizations: 8 Threads

Experiment	All	-Fuse	-Unrl	-Pre	-Vec	-Pred	-Grp	-ADS	-CLO
DataClean	1.00	2.44	0.97	0.99	0.98	0.95			
CrimeIndex	1.00	195	2.04	1.00	1.02	0.96			3.23
BlackSch	1.00	6.68		1.44	1.95		1.64		
Haversine	1.00	3.97		1.20	1.02				
Nbody	1.00	1.78		2.22	1.01				
BirthAn	1.00	1.02		0.97	0.98				1.00
MovieLens	1.00	1.07		1.02	0.98				1.09
LogReg	1.00	20.18		1.00					2.20
NYCFilter	1.00	9.99		1.20	1.23	2.79			
FlightDel	1.00	1.27		1.01	0.96	0.96	5.50		1.47



Loop fusion: Pipeline loops to reduce data movement.
Up to **195x** difference

Weld Prior Work

- Runtime code generation in databases
  - HyPer, LegoBase, DBLAB, Voodoo, Tupleware
  - Only target SQL or don't explicitly support parallelism
- Languages for parallel hardware
  - OpenCL, CUDA, SPIR, DryadLINQ, Spark, etc.
  - No effective cross-function optimization (even with LTO etc.)
- Monad comprehensions, Delite multiloops
  - Weld supports incremental integration, cross-library API, adaptive optimizations

My Approach: Building three systems to leverage new interface properties

Name	Interface/Properties	System
Weld	IR to **extract parallel "structure"** of library functions	**Compiler** to enable data movement optimization + parallelization
Split annotations		
Raw filtering		

Split annotations: Optimizing Data-Intensive Computations in Existing Libraries

SOSP '19

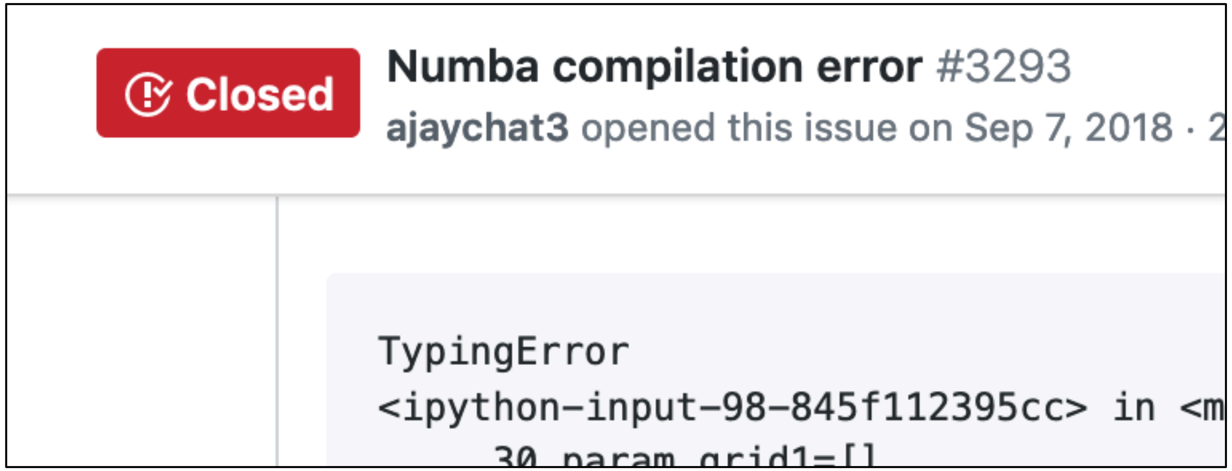
Shoumik Palkar and Matei Zaharia

Problem with Compilers: Developer Effort

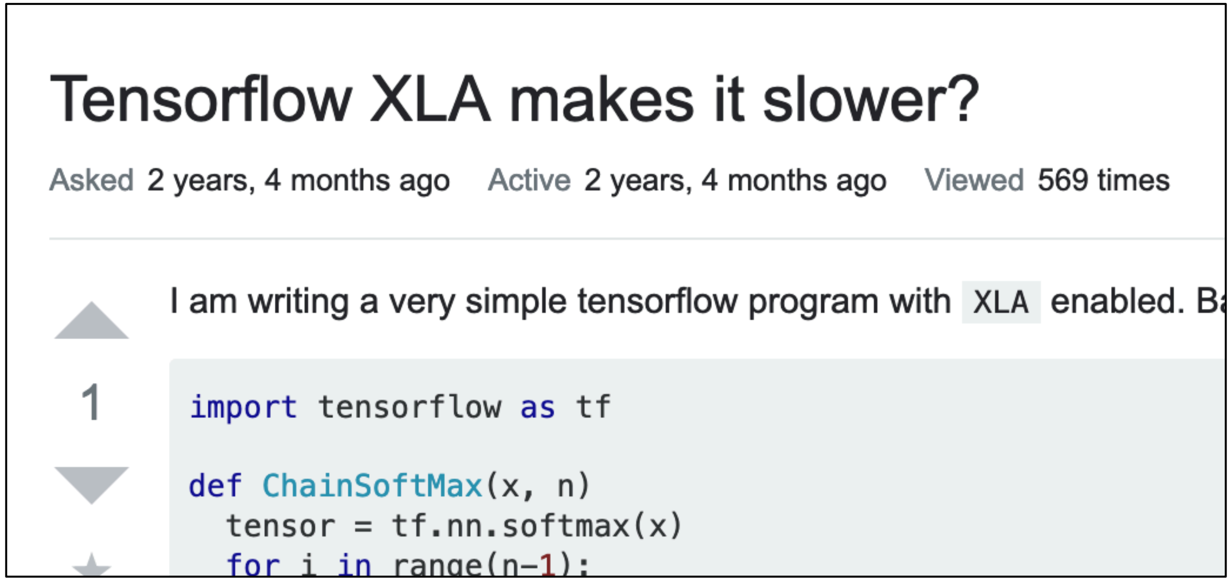
- Need to replace **every function** to use compiler intermediate representation (IR)
- IR **may not even support all optimizations** present in hand-optimized code

Examples

Weld needs 100s of LoC to support NumPy, Pandas



"Sorry, our compiler doesn't recognize this pattern yet"



"Some ops are expected to be slower compared to hand-optimized kernels"

Split Annotations (SAs)

Data movement optimizations and **automatic parallelization** on **unmodified library functions**

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

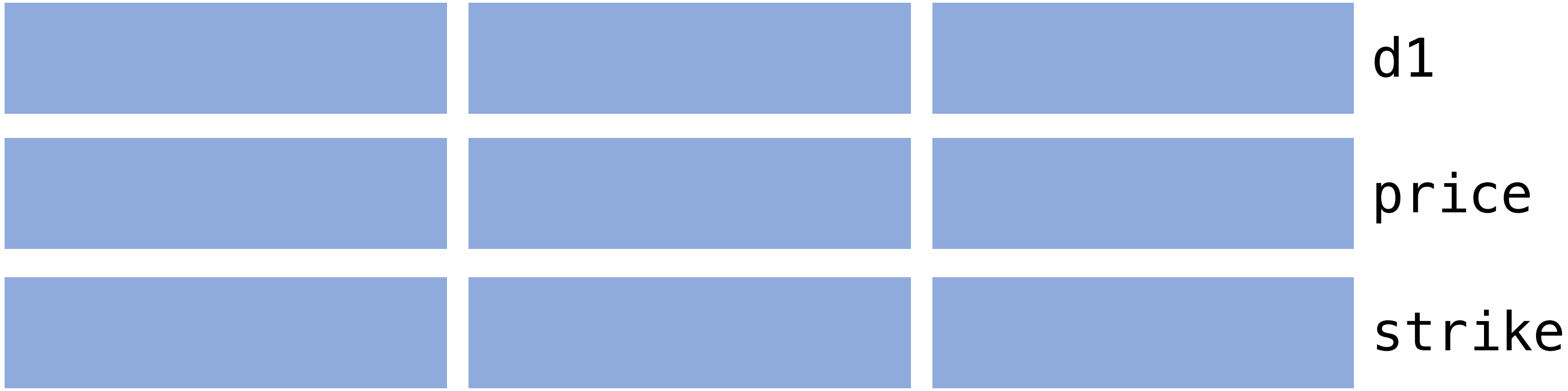
SAs Enable Pipelining + Parallelism

With SAs:


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

SAs Enable Pipelining + Parallelism

With SAs:

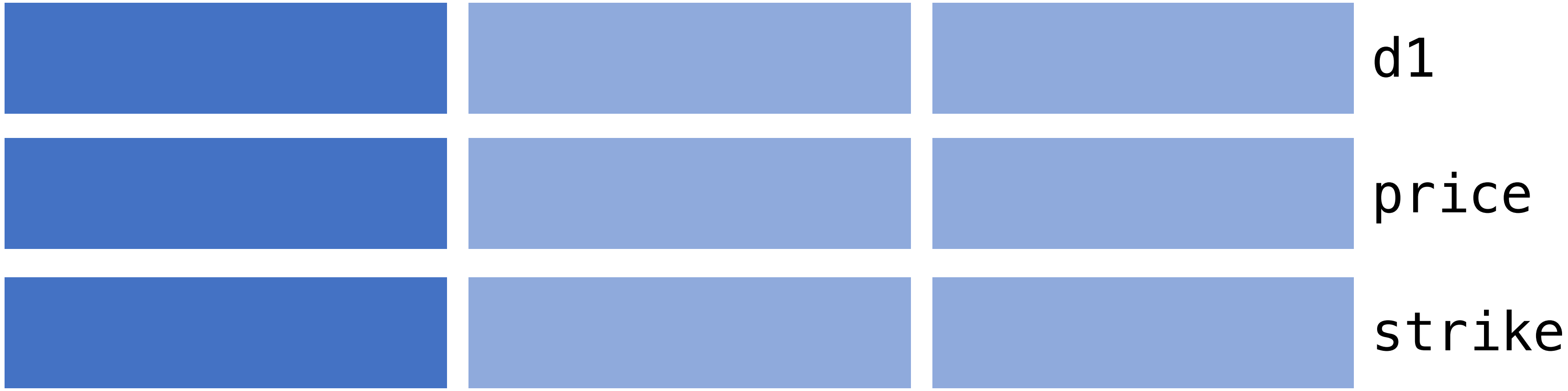



```
d1 = price * strike
d1 = np.log2(d1) + strike
```

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

SAs Enable Pipelining + Parallelism

With SAs:



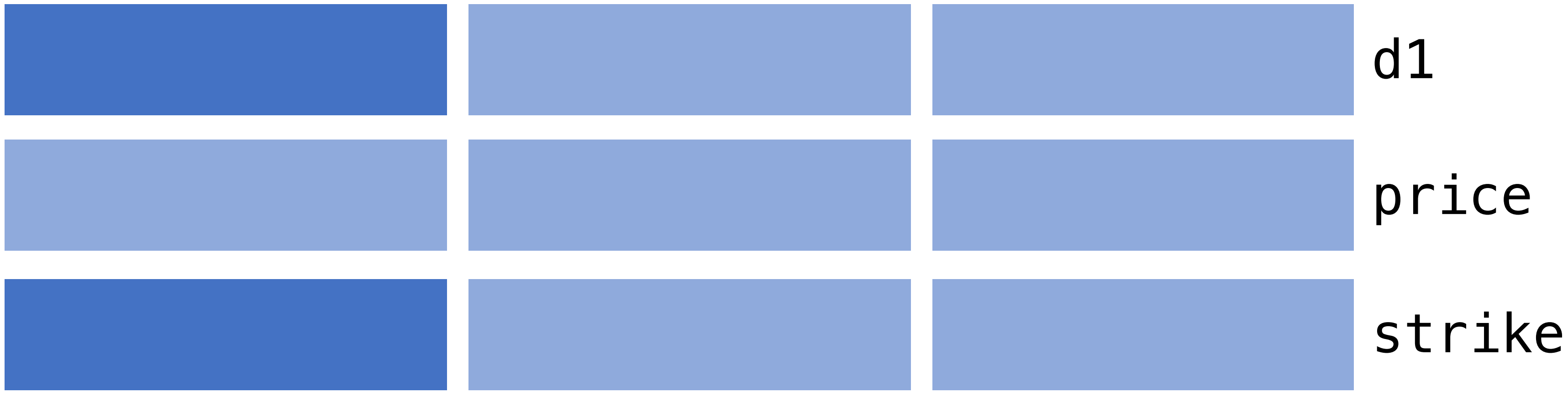
Collectively fit in cache

```
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

```
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:

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

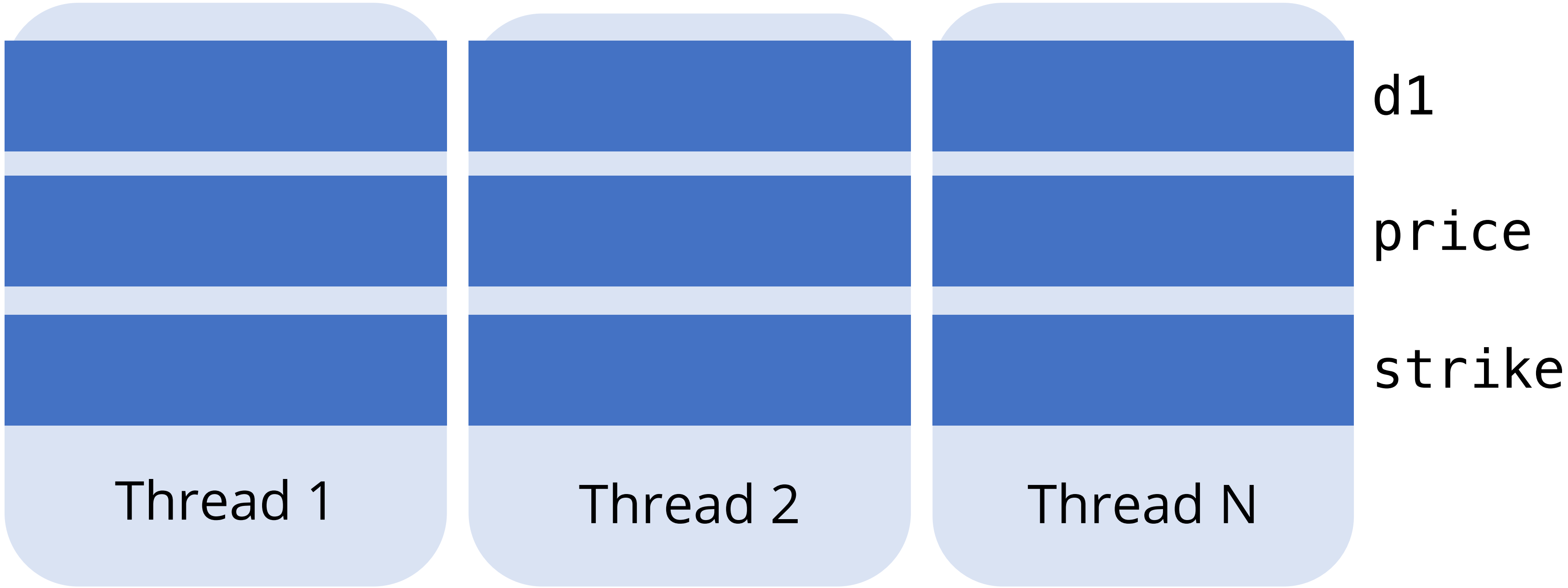
With SAs:


```
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:



Parallelize over split pieces

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

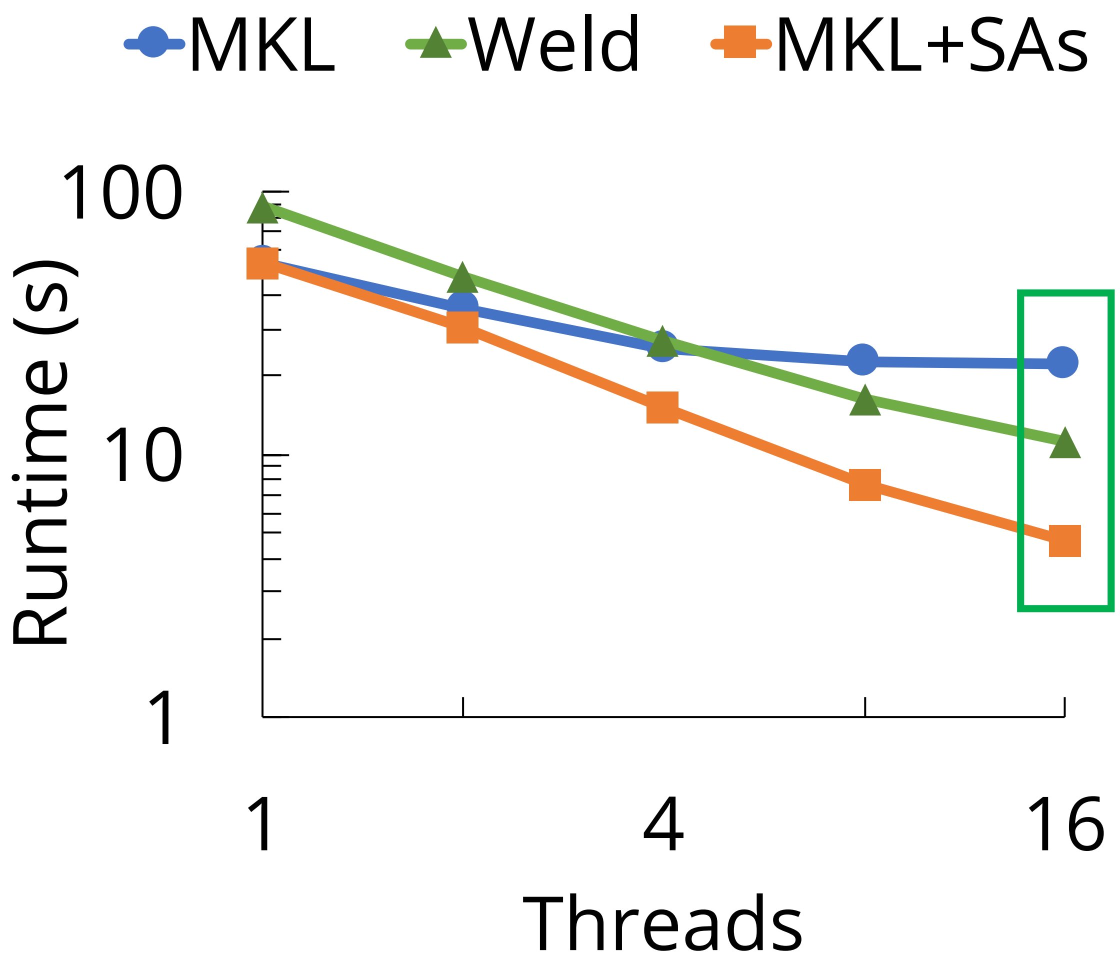
Example of a split annotation for MKL

```
@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

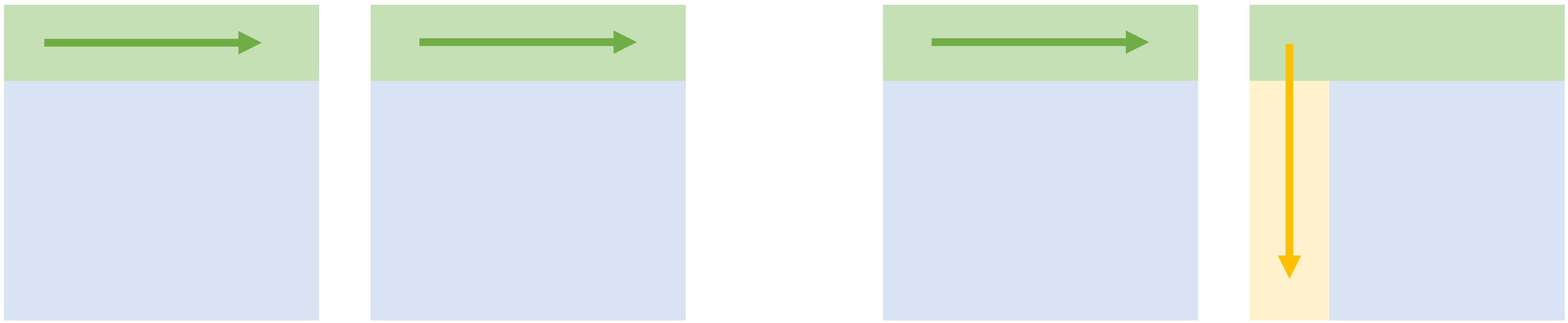
2. Building a lazy task graph **transparently**

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.

```
@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**.

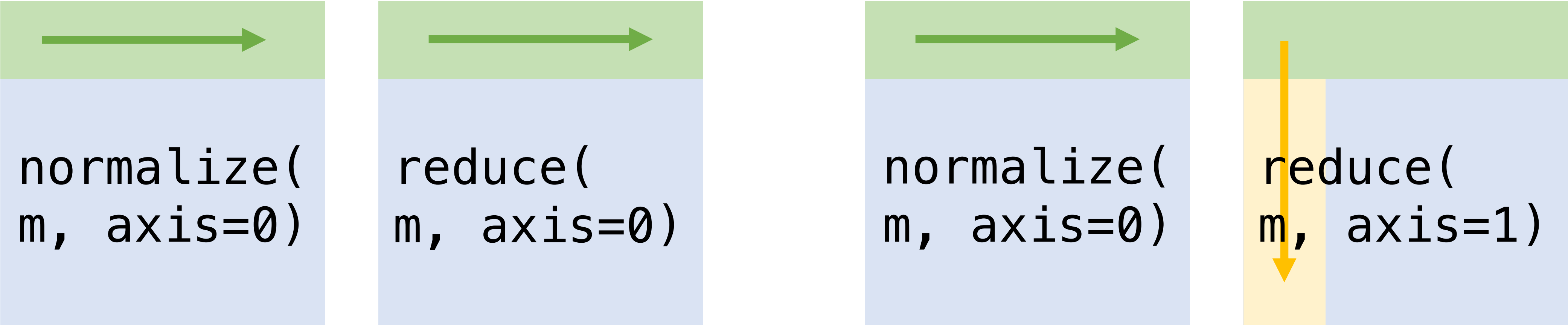
```
@sa(n: SizeSplit(n, K), a: ArraySplit(n, K),
       b: ArraySplit(n, K), out: ArraySplit(n, K))
void vdAdd(int n, double *a, double *b, double *out)
```

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)`**



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

Split types don't match: `axis=0` for first call, `axis=1` for second call

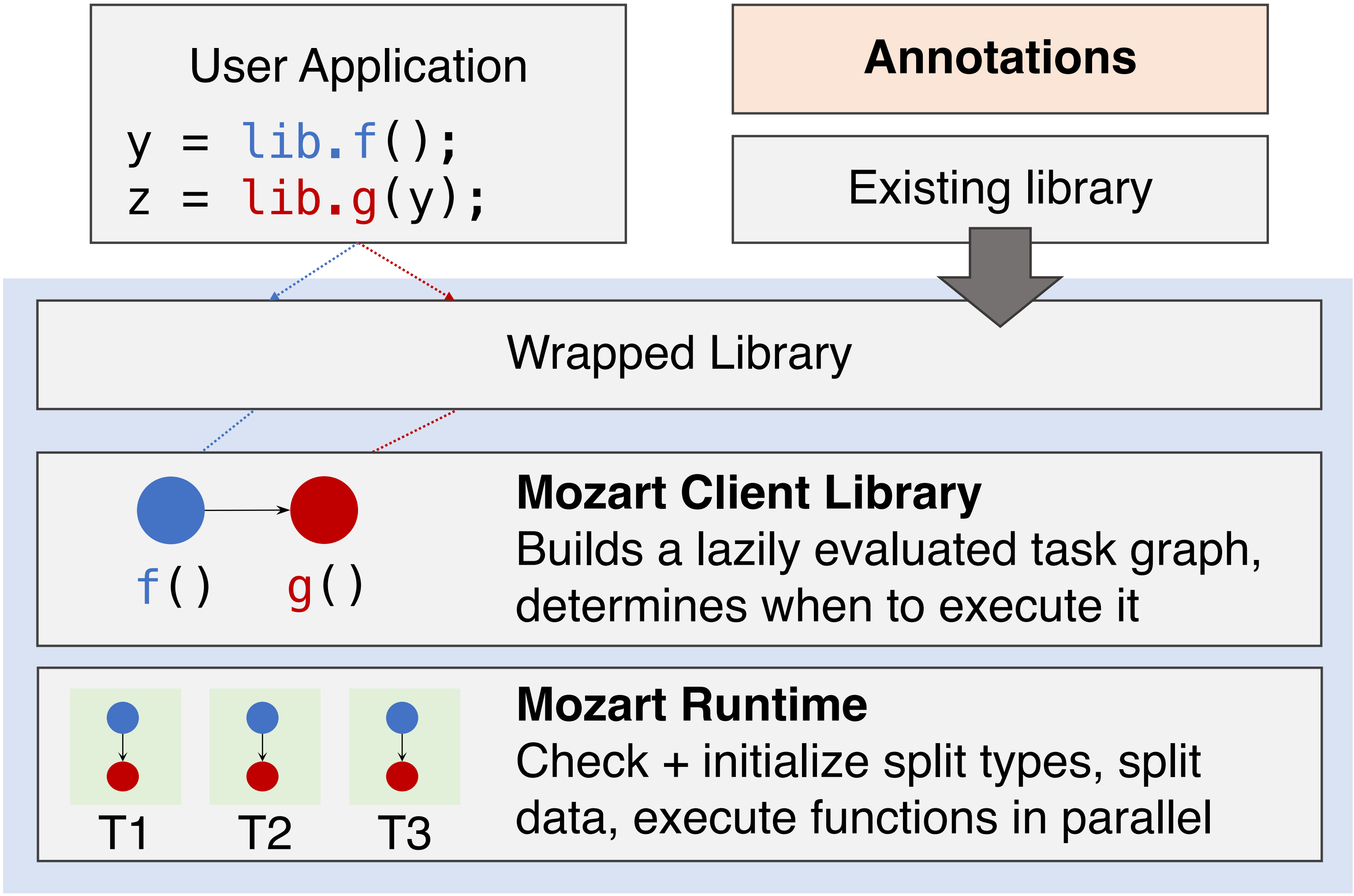
How an annotator writes SAs

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

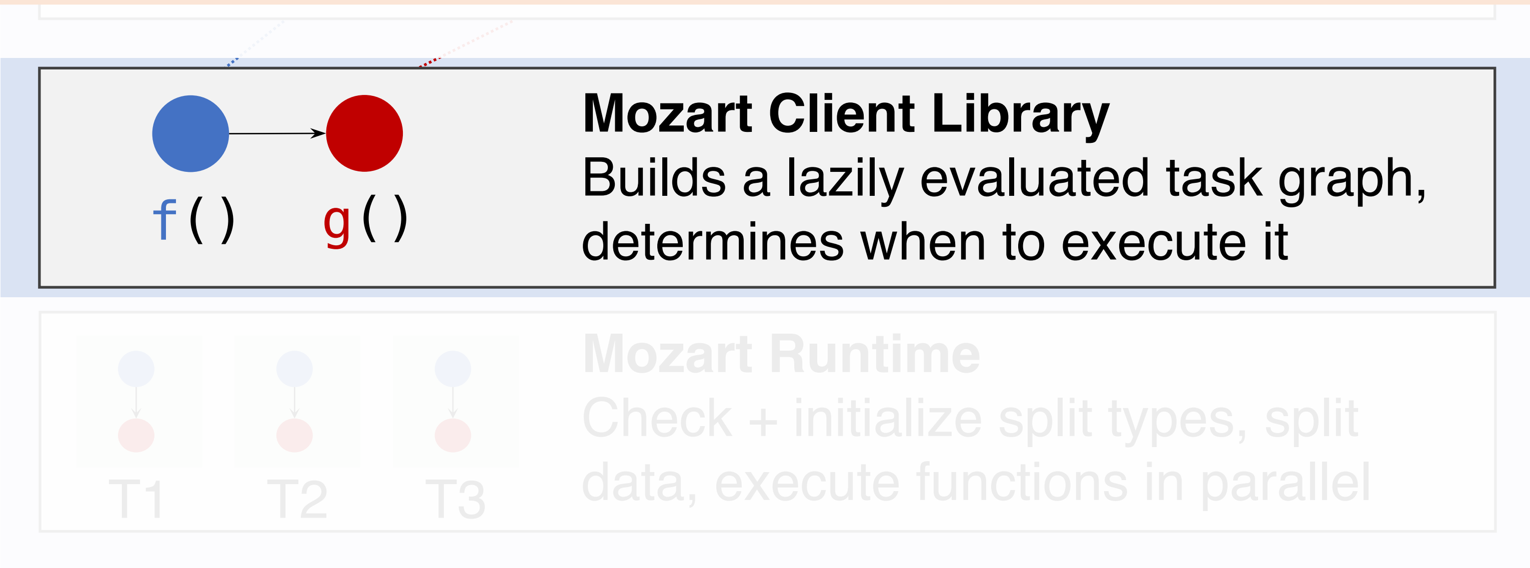
Mozart: Our system implementing SAs



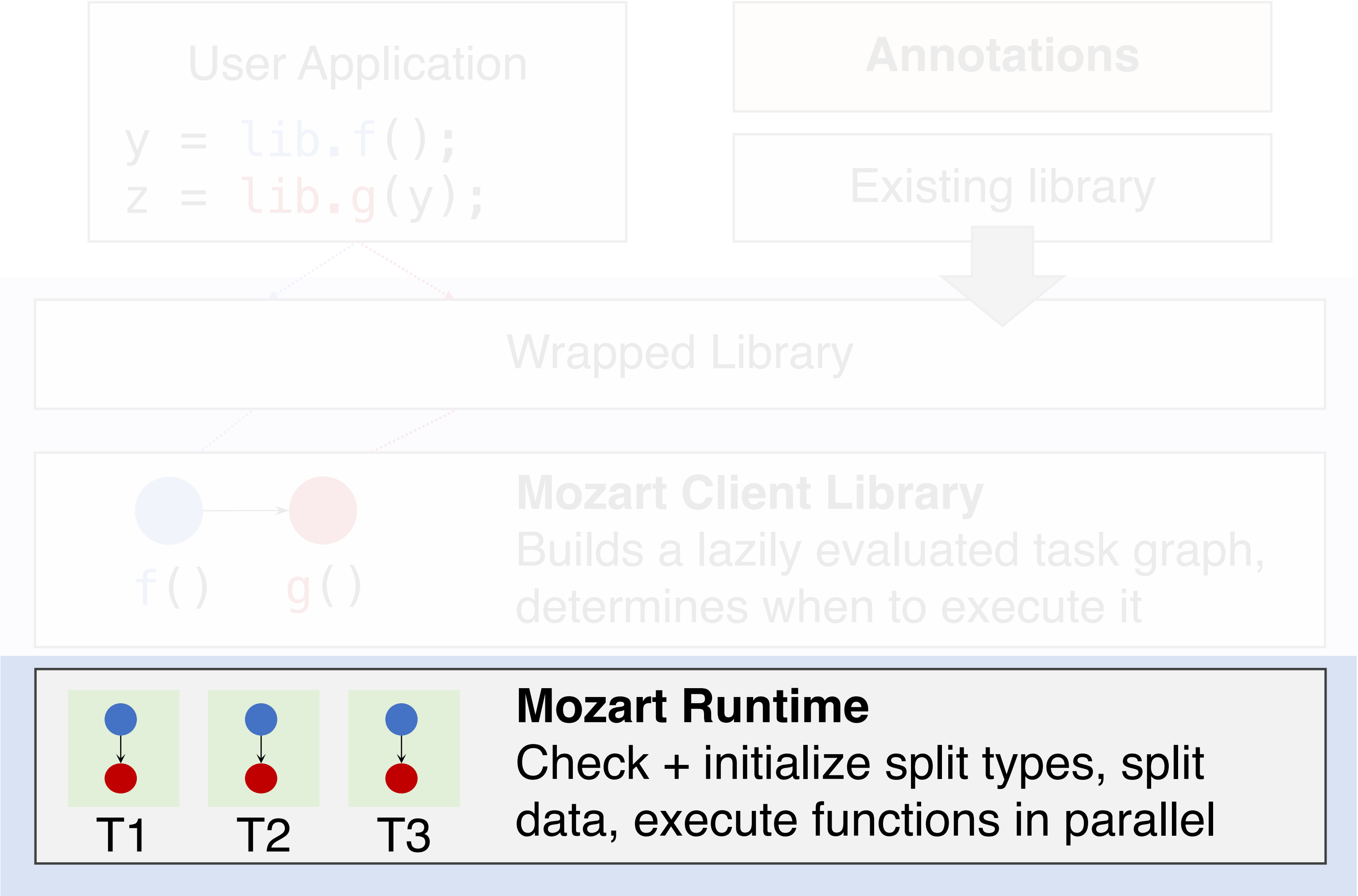
Mozart: Our system implementing SAs

In C++: Memory protection for lazy evaluation
In Python: Meta-programming for lazy evaluation

See paper for details!



Mozart: Our system implementing SAs



Results

Data Types and Libraries Demonstrated

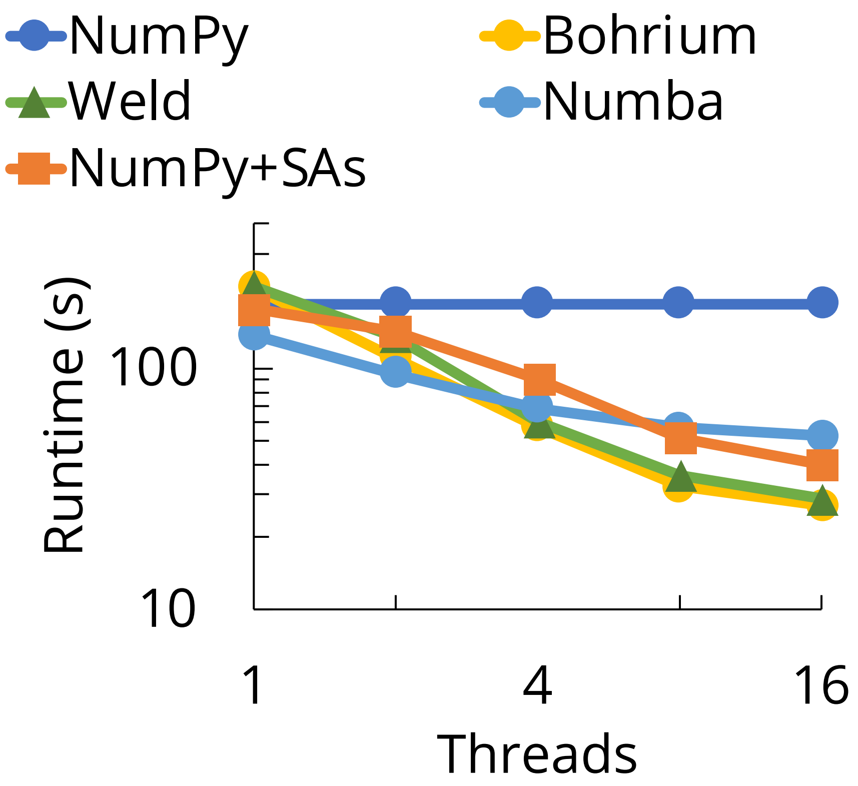
Libraries: L1 + L2 BLAS (MKL), NumPy, Pandas, spaCy, ImageMagick

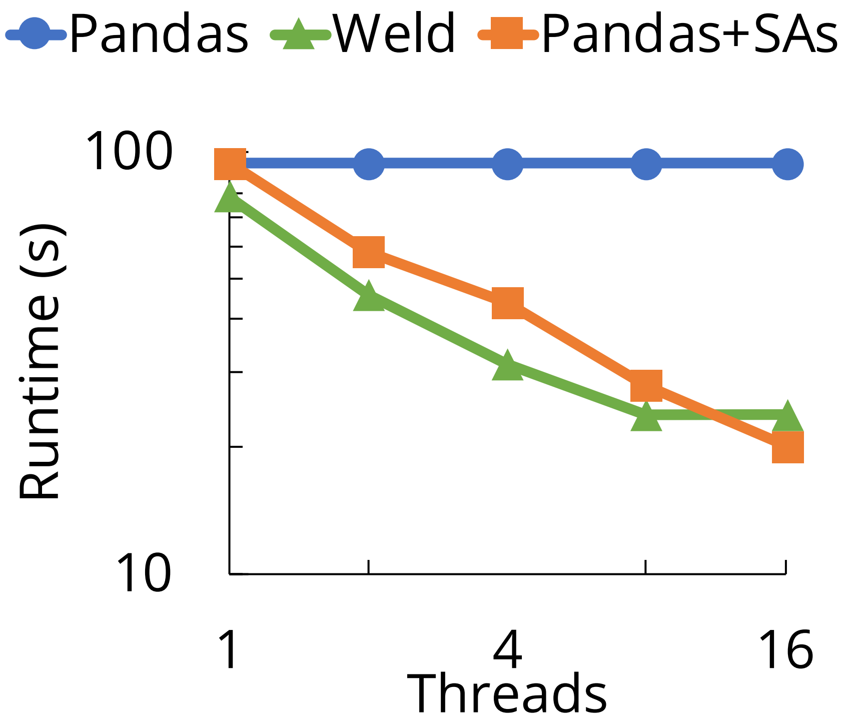
Data types and operators: Arrays, Tensors, Matrices, DataFrame joins, grouping aggregations, image processing algorithms, functional operators (map, reduce, etc.)

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





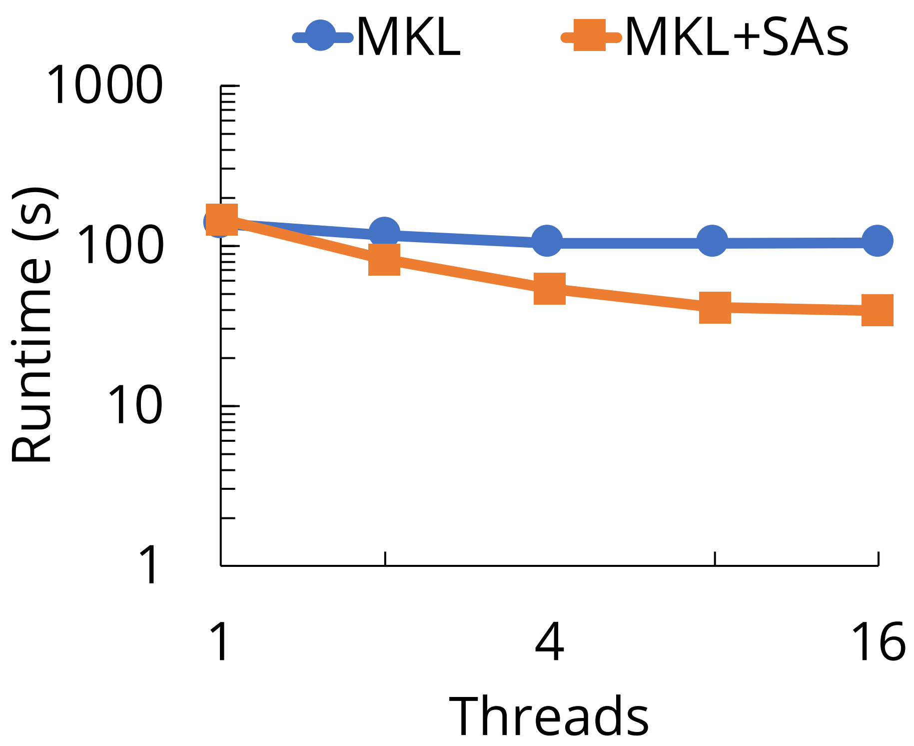
nBody simulation: **4.6x speedup** over NumPy

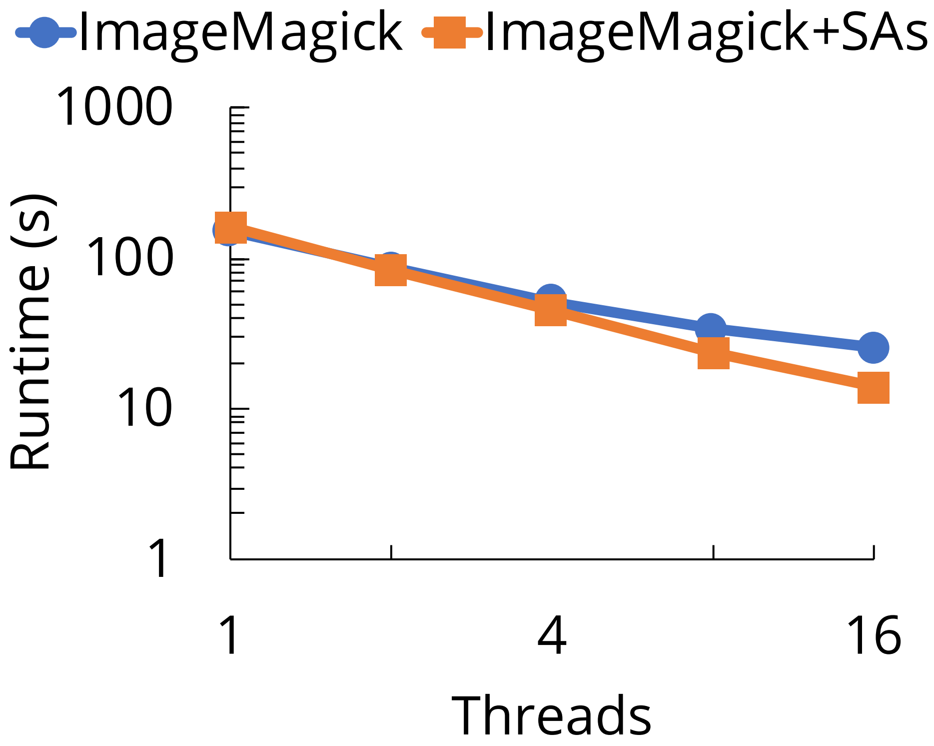
pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

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

SAs can accelerate highly optimized libraries





Shallow Water eqn: **3x 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** of our workloads

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

SAs Prior Work

- Black box code generation interface + parallelization
  - Numba, Pydron, Dask, Ray, Cilk, OpenMP
  - No pipelining/cross-function optimizations, which is focus of SAs
- Vectorization and Batch Processing
  - X100, MonetDB, Spark SQL
  - SAs enable these for arbitrary black-box libraries rather than SQL
- Automatic loop tiling and loop optimizations
  - Scala Collections, Polyhedral model in LLVM, etc.
  - Found to be ineffective over black-box functions, no pipelining

My Approach: Building three systems to leverage new interface properties

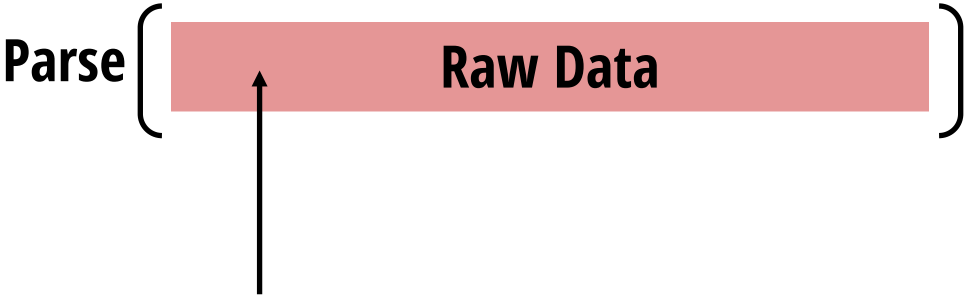
Name	Interface/Properties	System
Weld	IR to **extract parallel “structure”** of library functions	**Compiler** to enable data movement optimization + parallelization
Split annotations	Annotations to define **how to partition function inputs**	**Runtime** to pipeline data among **unmodified library functions**

Raw filtering: Optimizing I/O pipelines by restructuring data loading

PVLDB '18

Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia

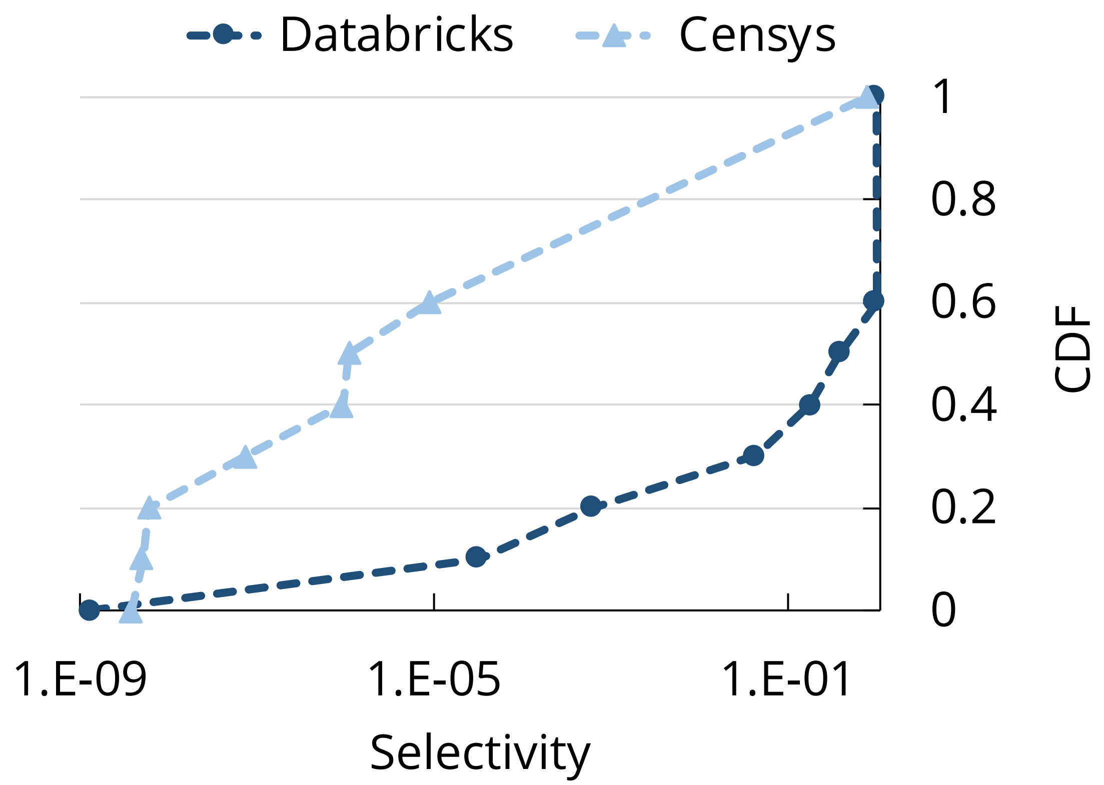
Parsing: A Computational Bottleneck



Today:
parse full input → slow!

Key Opportunity: High Selectivity

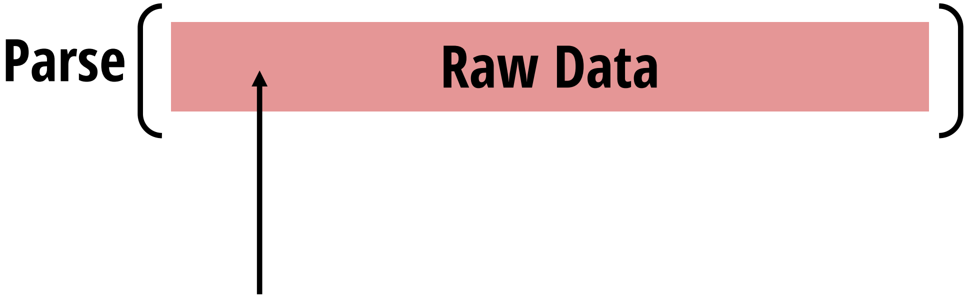
High selectivity especially true for **exploratory analytics.**



40% of customer Spark queries at Databricks **select < 20%** of data
99% of queries in Censys **select < 0.001%** of data

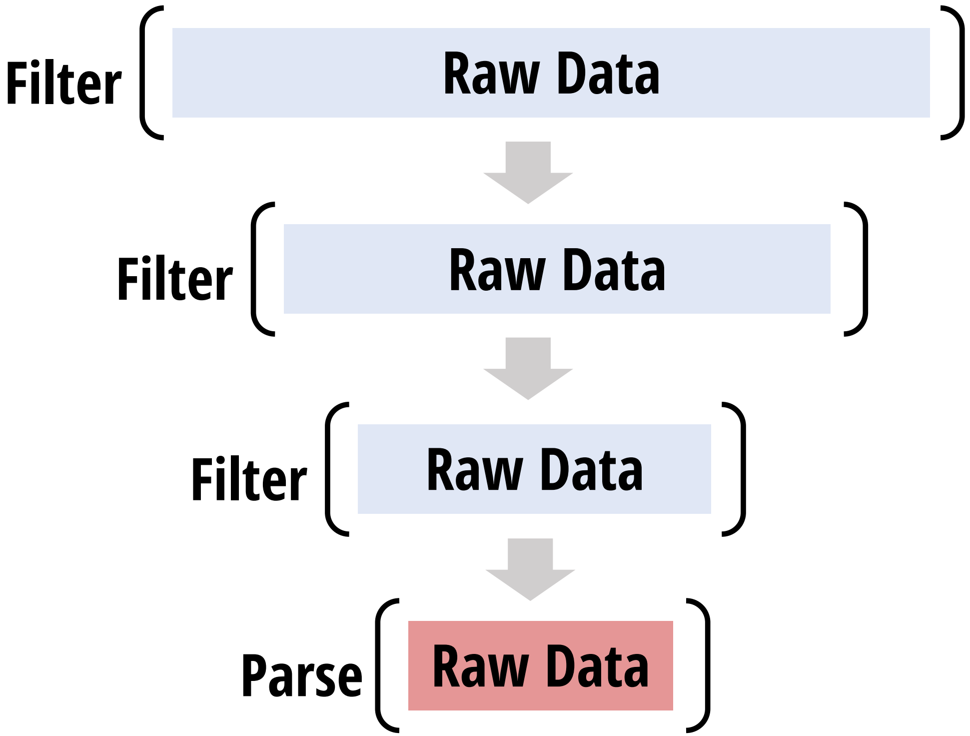
How can we exploit high selectivity to accelerate parsing?

Sparser: Filter Before You Parse

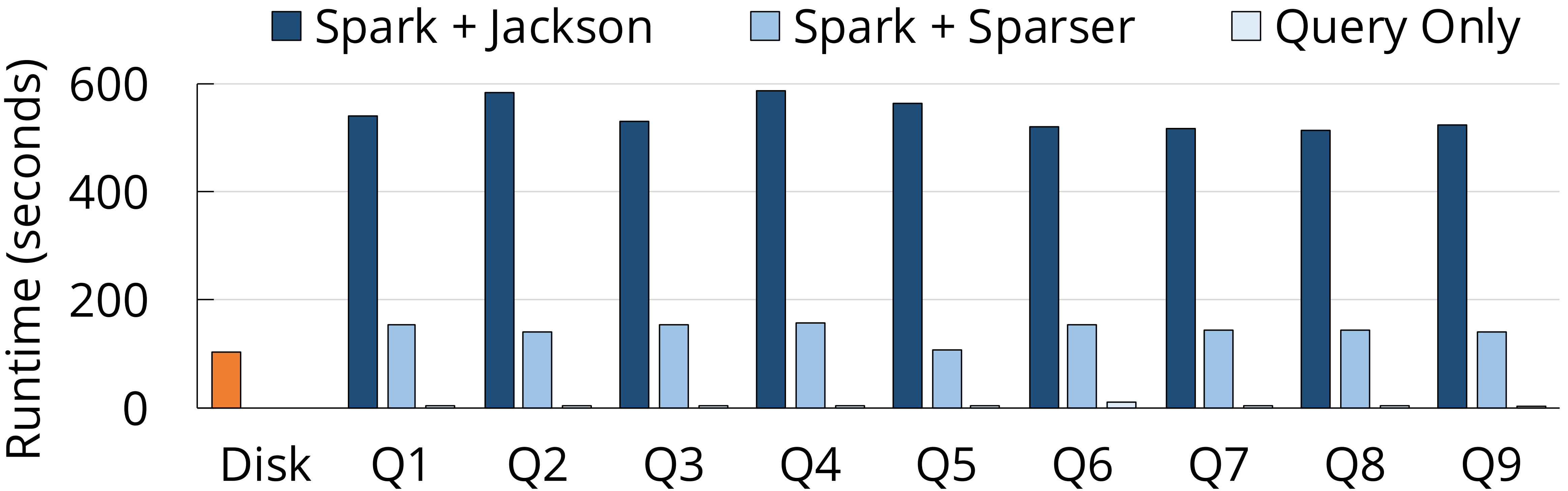


Today:
parse full input → slow!

Sparser: **Filter before parsing** first using fast filtering functions with **false positives, but no false negatives**



Results: Accelerating End-to-End Spark Jobs



Censys queries on 652GB of JSON data: up to **4x speedup** by using Sparser.

My Approach: Building three systems to leverage new interface properties

Name	Interface/Properties	System
Weld	IR to **extract parallel "structure"** of library functions	**Compiler** to enable data movement optimization + parallelization
Split annotations	Annotations to define **how to partition function inputs**	**Runtime** to pipeline data among **unmodified library functions**
Raw filtering	Composable filters with false positives	**Library** for accelerating I/O of serialized data

New composition interfaces can improve performance on modern hardware

- **Weld** used at NEC to support new vector accelerator, prototyped at Databricks, used in several labs

NEC CWI databricks KTH VETENSKAP OCH KONST

- Ongoing work at Stanford for extending **SAs** to bridge GPU and CPU libraries
- Teradata, Google have prototyped **raw filtering** internally

Acknowledgements

Acknowledgements

Thank you to my committee members!

Keith
Winstein

Christos
Kozyrakis

Mendel
Rosenblum

John
Duchi

Acknowledgements

Thank you Matei for an inspiring graduate career!

Acknowledgements

To FutureData, for great discussions, gossip, and friendships that I hope will last forever

Cody, Daniel, Deepti, Edward, Fiodar, Kaisheng, Keshav, Kexin, Peter Bailis, Peter Kraft, Pratiksha, Sahaana

To my office mates, for teaching me about sports, goofing off with me, and tolerating four years of terrible jokes

Deepak, Firas, James

To other friends who supported me outside of lab

Akshay, Aubhro, Jeff, Neil, Rohit, Stephanie, Sagar, Sahil, Yuval

And of course, to my wife Paroma, whose unwavering support made grad school one of the fondest times of my life, and the rest of my family: my parents Anjali and Prasad, my sister Ishani, my aunt and uncle Trupti and Sourja, and my two little cousins Shreya and Tvisha, all of who were collectively responsible for keeping me smiling for the last 26 years ☺

Conclusion

Thesis: We can use ***algebraic properties*** of software APIs in ***new interfaces*** to enable new optimizations

Demonstrated with three interfaces/systems:

- **Weld**
- **Split Annotations**
- **Raw filtering**

shoumik@cs.stanford.edu

STANFORD DAWN