Weld: Accelerating Data Science by 100x

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

Stanford DAWN, *MIT CSAIL, **Imperial College London

STANFORD INFOLAB

www.weld.rs

Motivation

Modern data applications combine many disjoint processing libraries & functions

+ Great results leveraging work of 1000s of authors

Motivation

Modern data applications combine many disjoint processing libraries & functions

+ Great results leveraging work of 1000s of authors

– No optimization across functions

How Bad is This Problem?

Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```

How Bad is This Problem?

Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```

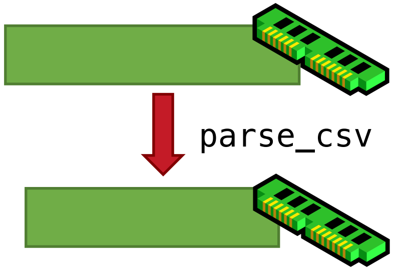
How Bad is This Problem?

Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```



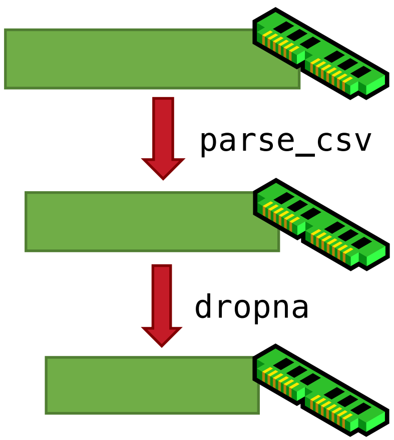
How Bad is This Problem?

Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```



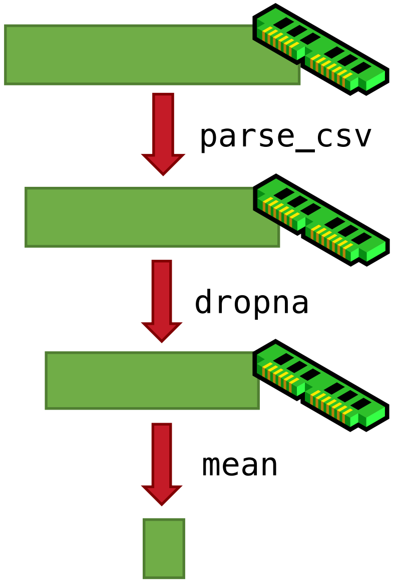
How Bad is This Problem?

Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```



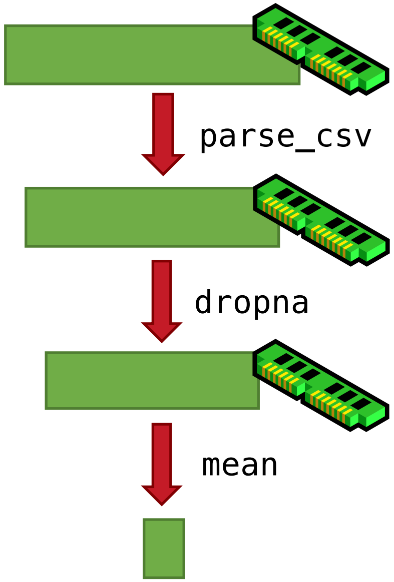
How Bad is This Problem?

Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```



Up to 30x slowdowns in NumPy, Pandas, TensorFlow, etc. compared to an optimized C implementation

Data Science Today

Data scientists "`pip install`" libraries needed for prototype/get the job done

Data Science Today

Data scientists "**`pip install`**" libraries needed for prototype/get the job done

Observe **performance issues** in pipelines composed of fast data science tools

Data Science Today

Data scientists "`pip install`" libraries needed for prototype/get the job done

↓

Observe **performance issues** in pipelines composed of fast data science tools

↓

Hire engineers to optimize your pipeline, leverage new hardware, etc.

Data Science Today

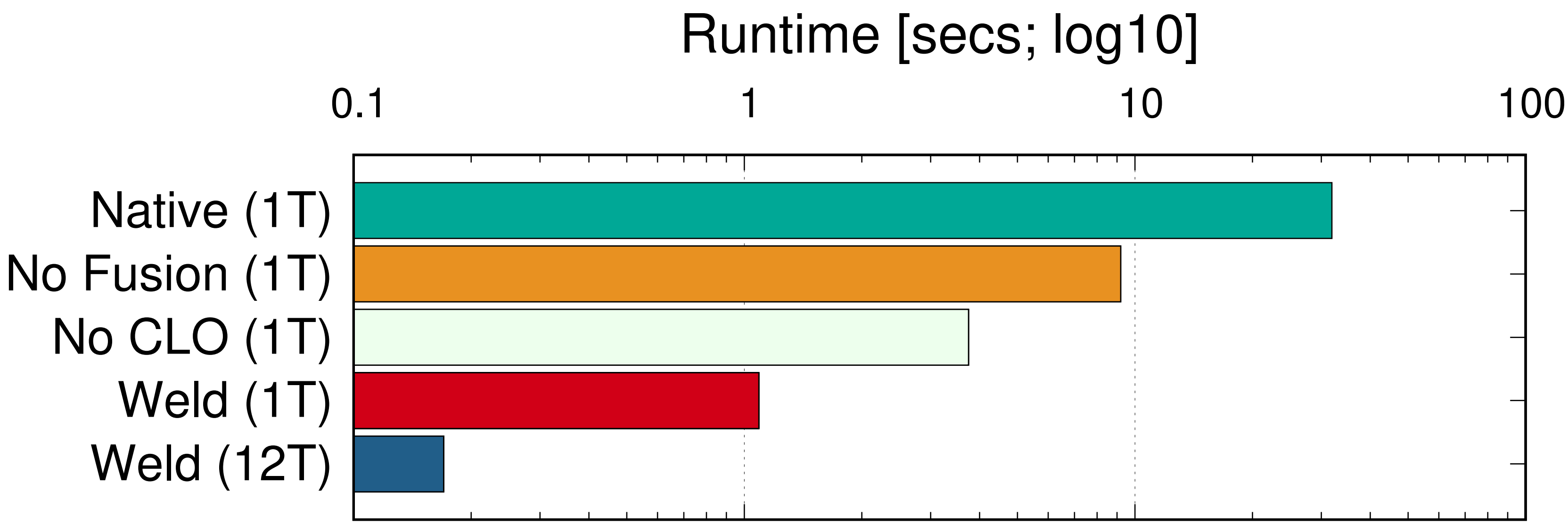
Data scientists “`pip install`” libraries needed for prototype/get the job done

Observe **performance issues** in pipelines composed of fast data science tools

Hire engineers to optimize your pipeline, leverage new hardware, etc.

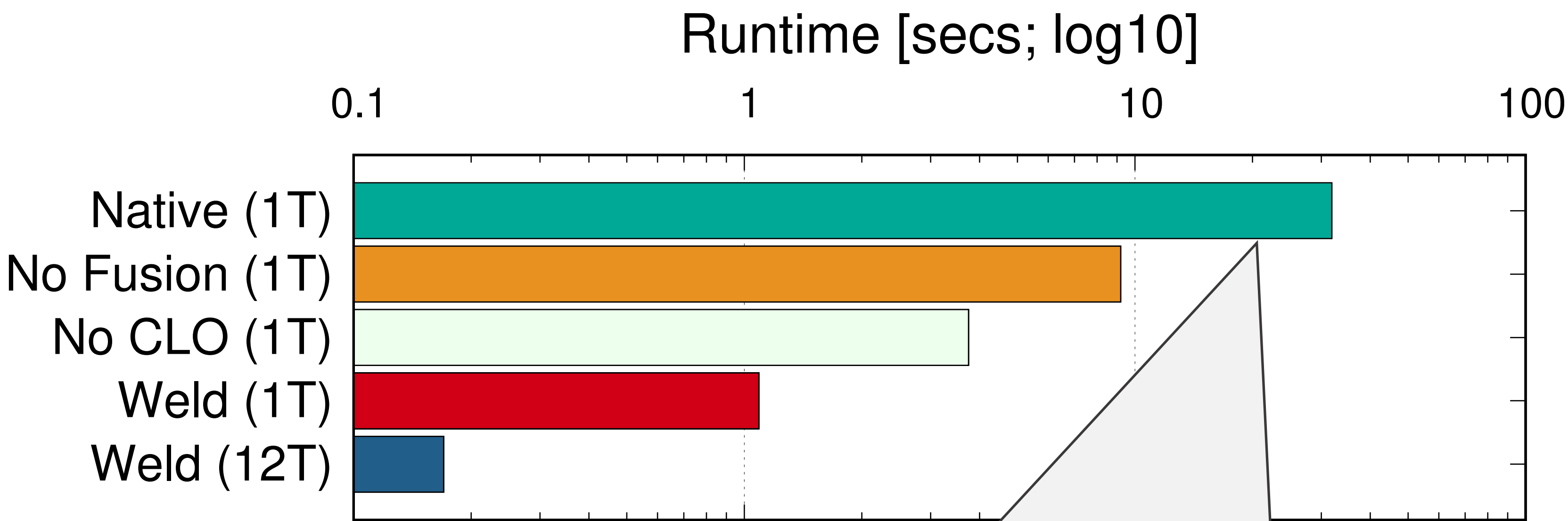
Weld’s vision: bare metal performance for data science out of the box!

Weld: An Optimizing Runtime



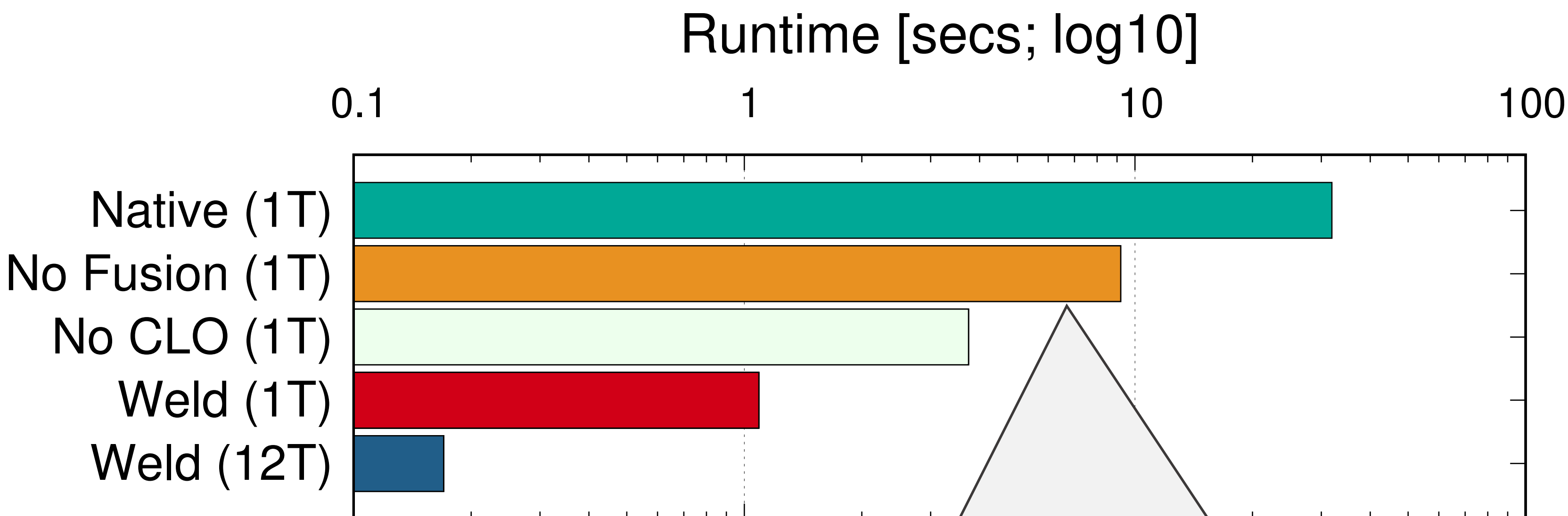
Filter Dataset → Compute a Linear Model → Aggregate Indices
Uses NumPy and Pandas (both backed by C)

Weld: An Optimizing Runtime



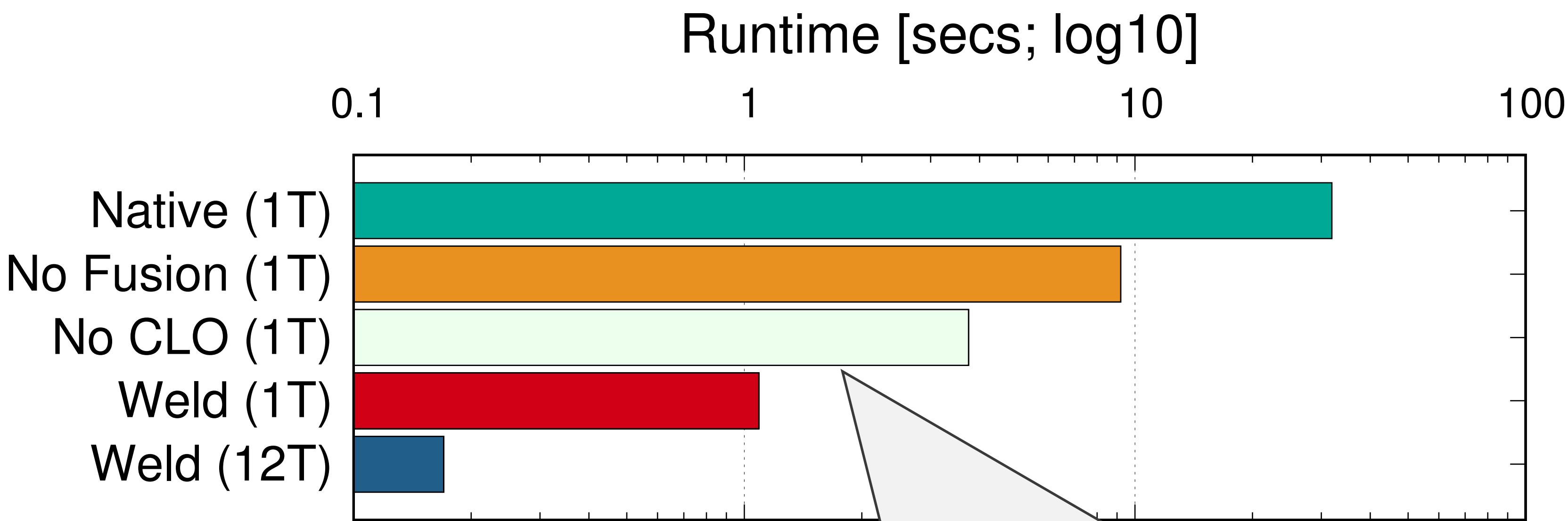
Native NumPy and Pandas

Weld: An Optimizing Runtime



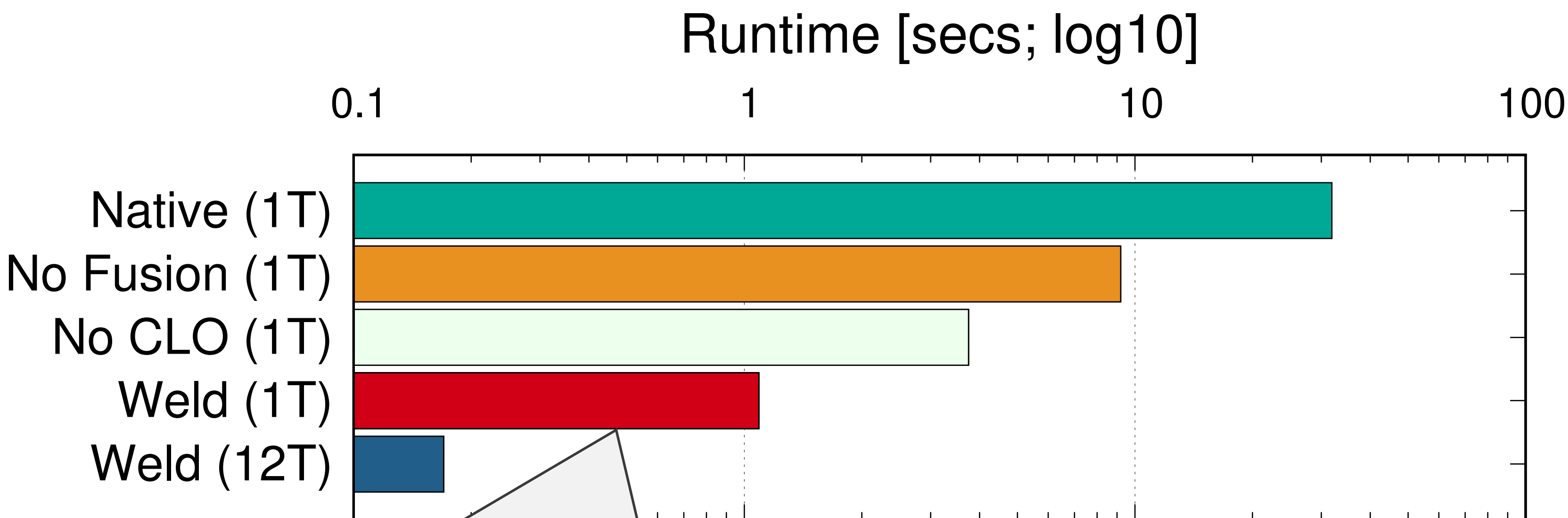
~**3x** Speedup from code generation
(SIMD instructions + other standard compiler optimizations)

Weld: An Optimizing Runtime



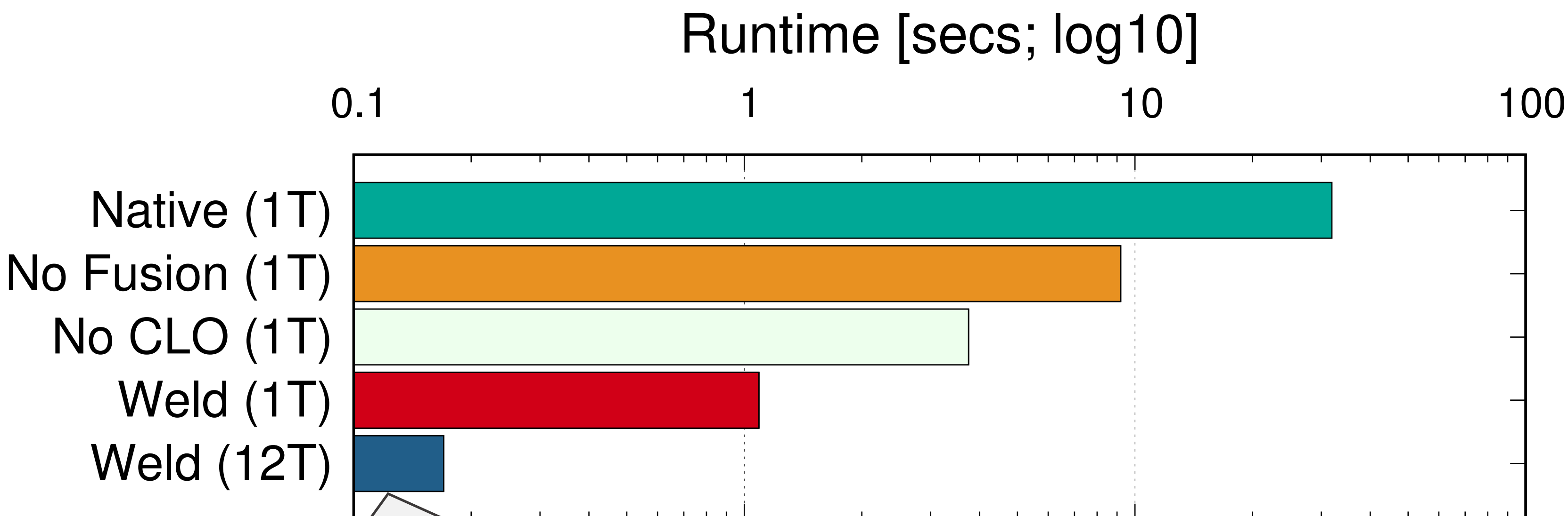
~**8x** Speedup from fusion within each library
(eliminates within-library memory movement)

Weld: An Optimizing Runtime



~29x Speedup from fusion across libraries library
(eliminates cross-library memory movement, co-optimizes library calls)

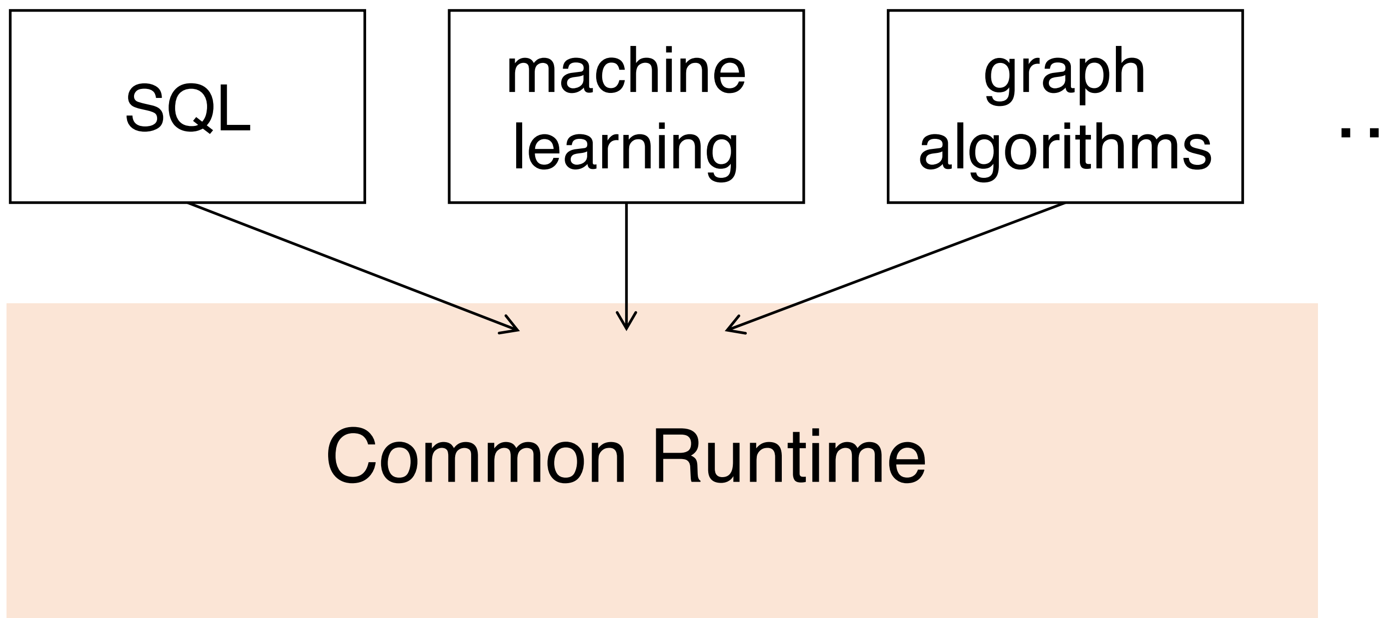
Weld: An Optimizing Runtime



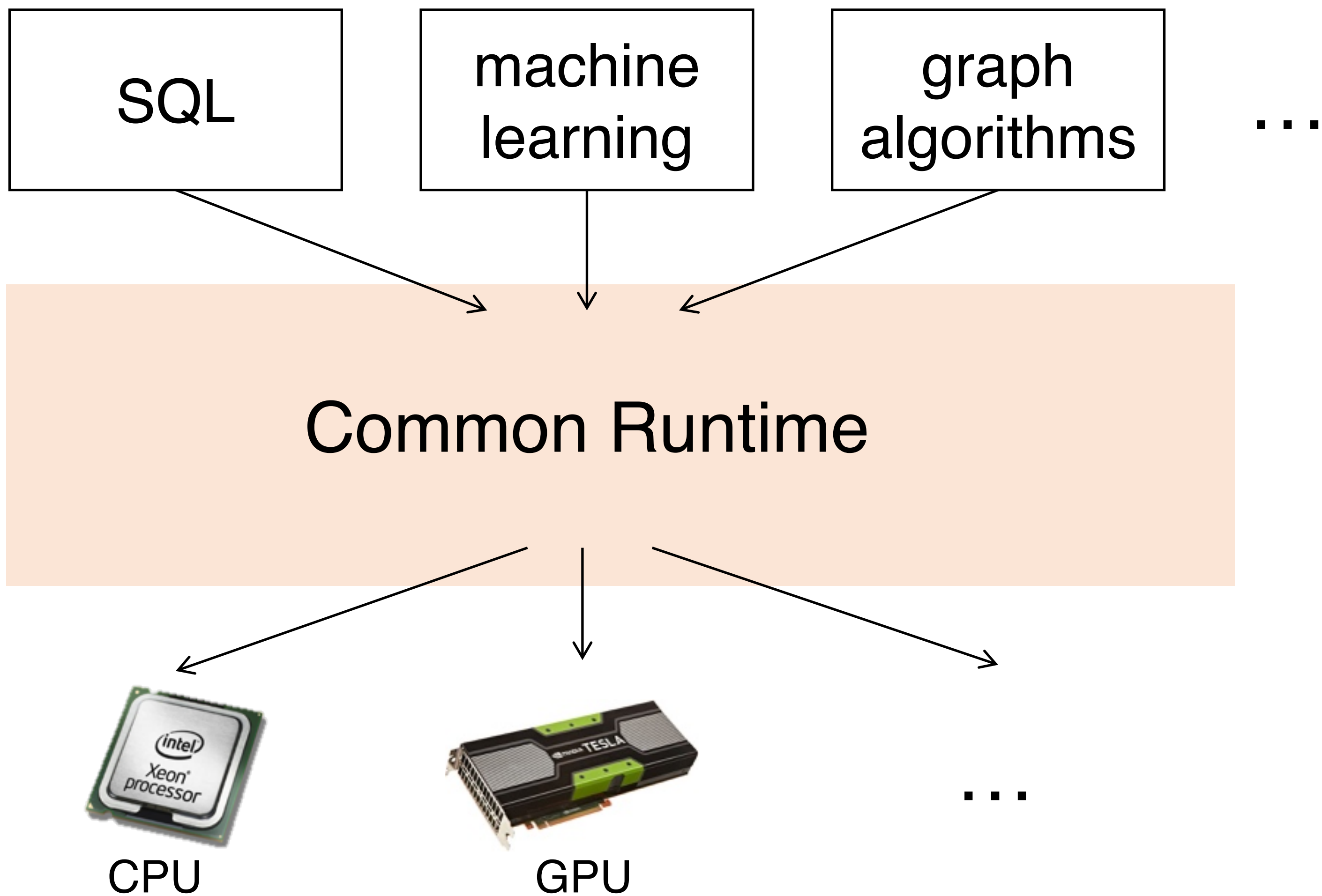
~180x Speedup with automatic parallelization
(eliminates cross-library memory movement, co-optimizes library calls)

Weld Architecture

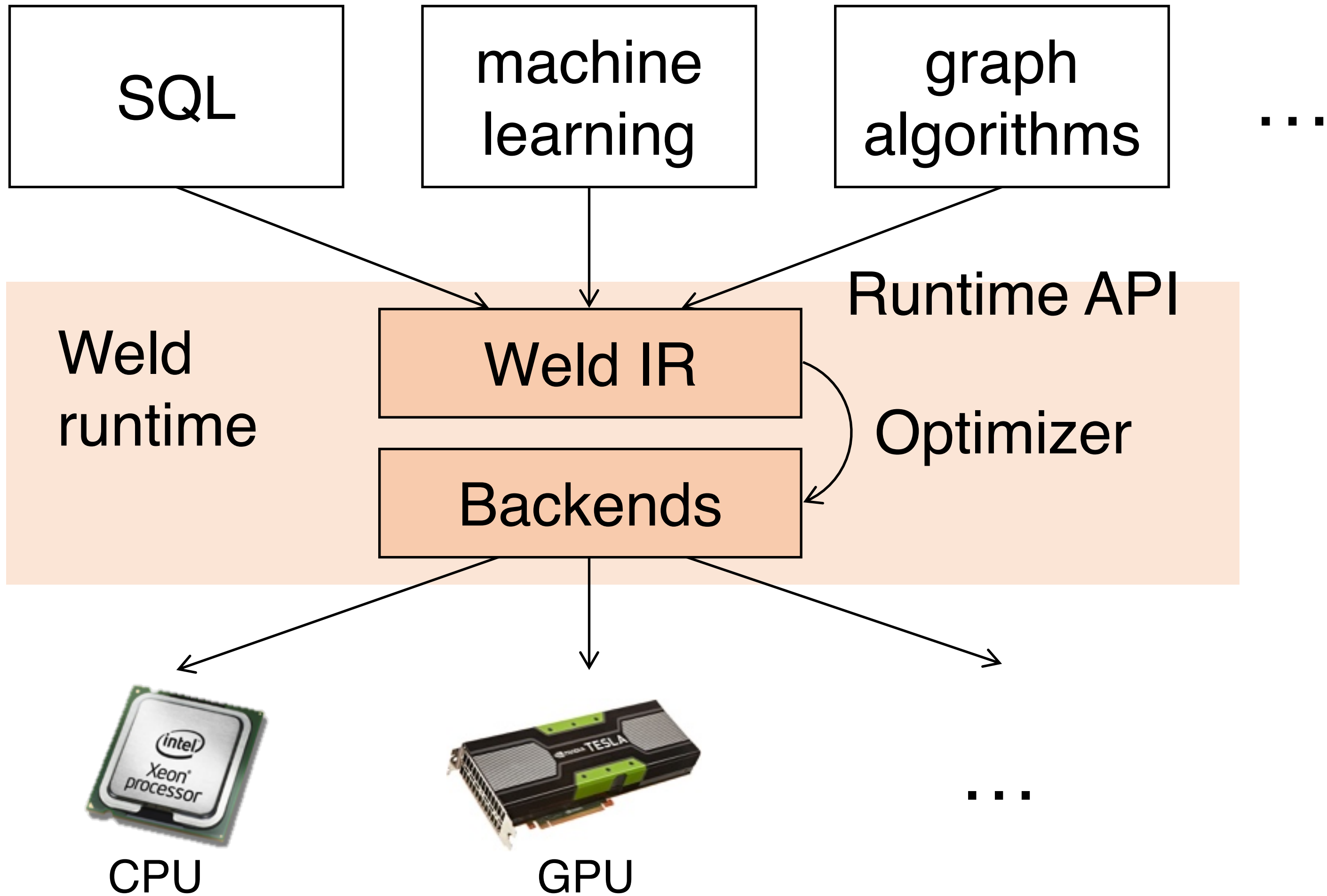
Weld Architecture



Weld Architecture



Weld Architecture



Rest of this Talk

Runtime API – How applications “speak” with Weld

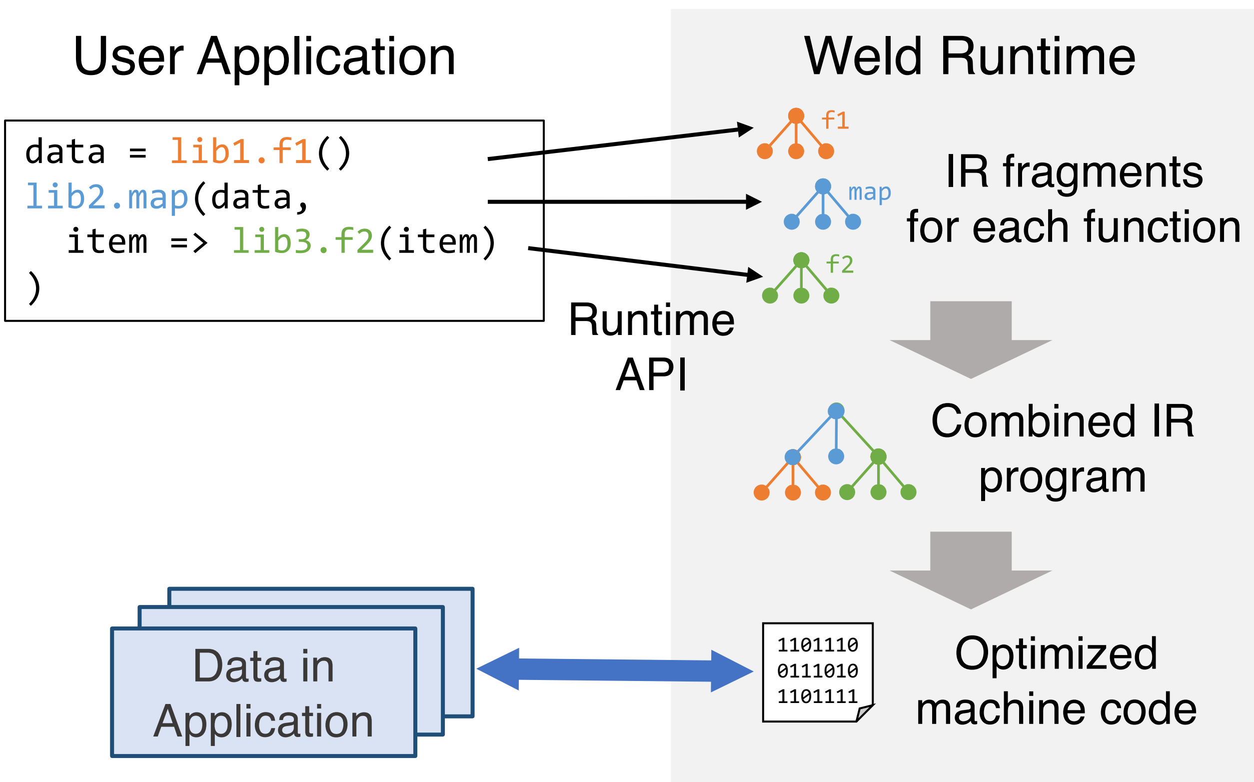
Weld IR – How applications express computation

Results

Demo

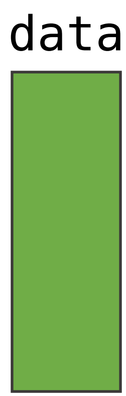
Runtime API

Uses lazy evaluation to collect work across libraries



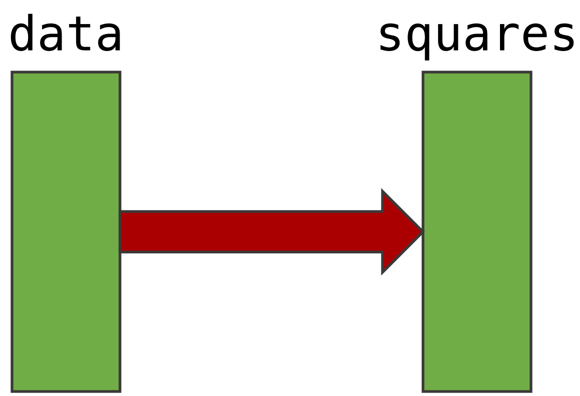
Without Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



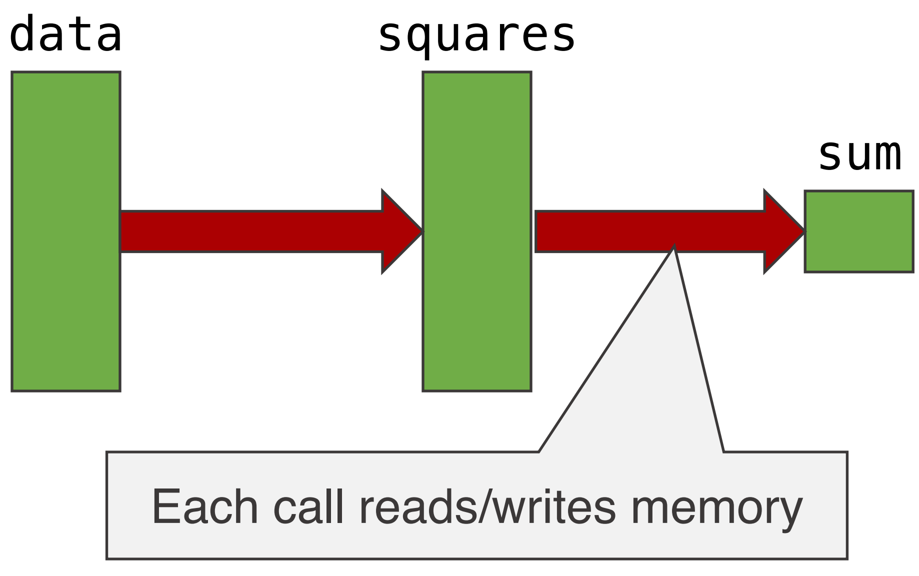
Without Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



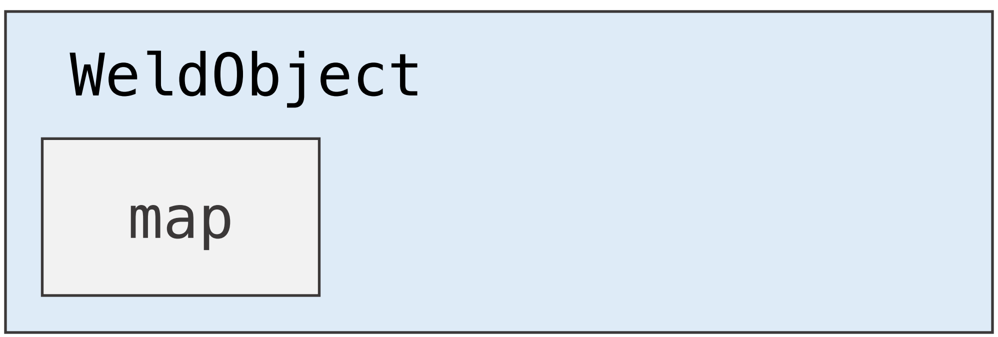
Without Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



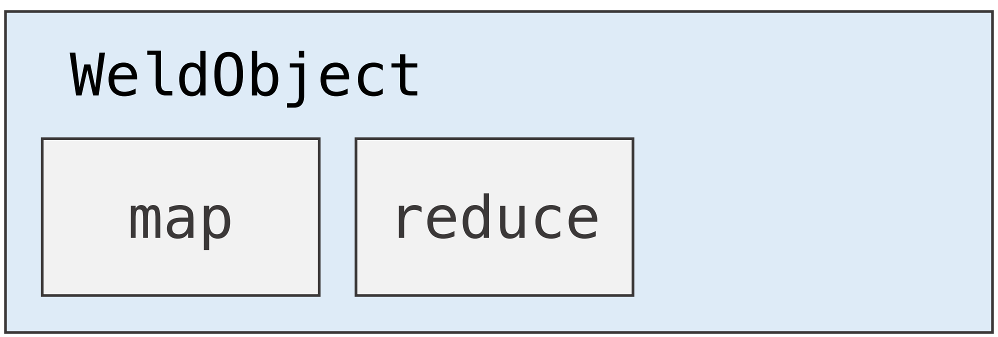
With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



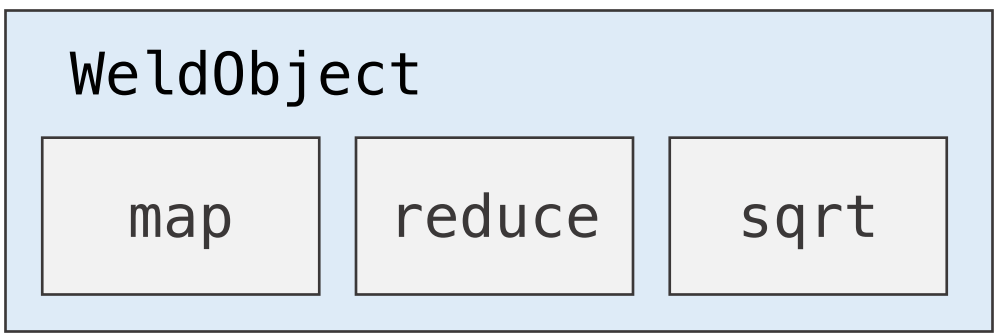
With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



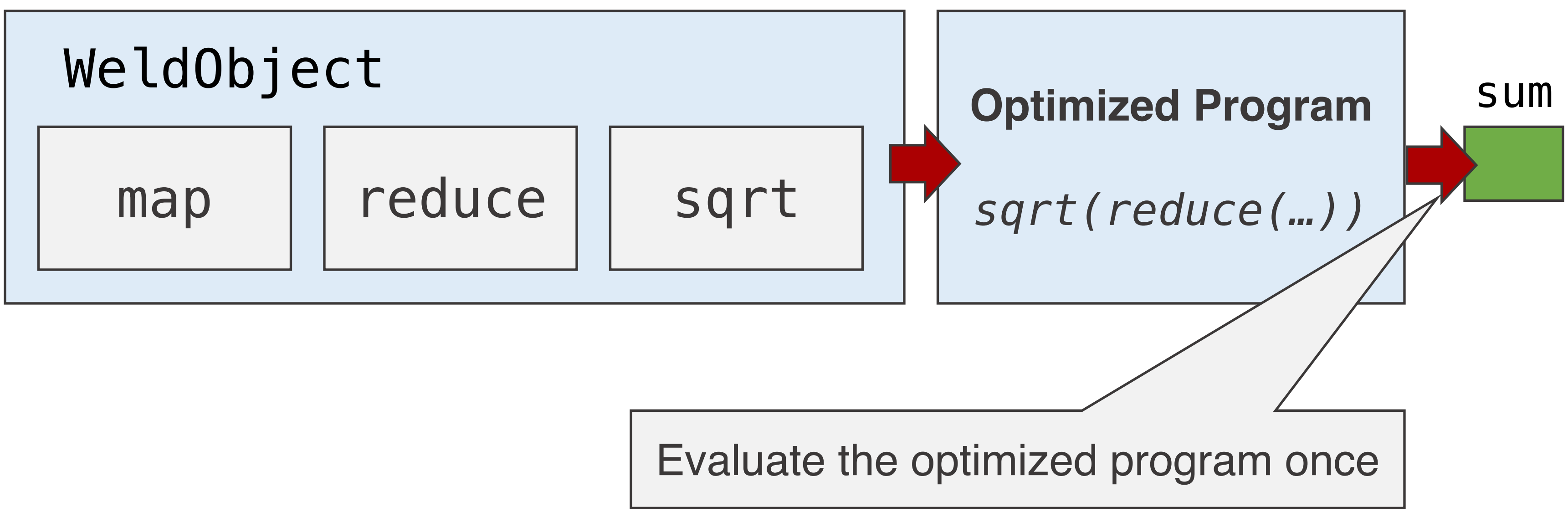
With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



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 and targeting parallel hardware**

Weld IR: Internals

Small IR* with only two main constructs.

Parallel loops: iterate over a dataset

Builders: declarative objects for producing results

- » E.g., append items to a list, compute a sum
- » Can be implemented differently on different hardware

Weld IR: Internals

Small IR* with only two main constructs.

Parallel loops: iterate over a dataset

Builders: declarative objects for producing results

» E.g., append items to a list, compute a sum

» Can be implemented differently on different hardware

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

Examples: Functional Ops

Examples: Functional Ops

Functional operators using builders

```
def map(data, f):
    builder = new appender[i32]
    for x in data:
        merge(builder, f(x))
    result(builder)
```

Examples: Functional Ops

Functional operators using builders

```
def map(data, f):
    builder = new appender[i32]
    for x in data:
        merge(builder, f(x))
    result(builder)


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

Example Optimizations

```
squares = map(data, |x| x * x)
sum = reduce(data, 0, +)
```

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

Loops can be merged into one pass over data and vectorized

Other Features

Interactive REPL for debugging Weld programs

Serialization/Deserialization operators for Weld data

Configurable memory limit and thread limit

Trace Mode for tracing execution at runtime to catch bugs

Rich logging for easy debugging

Utilities for generating C bindings to pass data into Weld

C UDF Support for calling arbitrary C functions

Ability to Dump Code for debugging

Syntax Highlighting support for Vim

Type Inference in Weld IR to simplify writing code manually for testing

Implementation

Implementation

APIs in C and Python (with Java coming soon)

- Full LLVM-based CPU backend SIMD support

Written in ~30K lines of Rust, LLVM, C++

- Fast, safe native language with no runtime

Implementation

APIs in C and Python (with Java coming soon)
- Full LLVM-based CPU backend SIMD support

Written in ~30K lines of Rust, LLVM, C++
- Fast, safe native language with no runtime

Partial Prototypes of **Pandas**, **NumPy**, TensorFlow and Apache Spark

Grizzly

A subset of Pandas integrated with Weld

- Operators include `unique`, `filter`, `mask`, `group_by`, `pivot_table`

Transparent single-core and multi-core speedups

Interoperates with Pandas with same API

Grizzly in Action

Adapted from http://pandas.pydata.org/pandas-docs/stable/tutorials.html (chapter 7)

Grizzly in Action

```
import pandas as pd


# Read dataframe from file
requests = pd.read_csv(‘filename.csv’)


# Fix requests with extra digits
requests['Incident Zip'] = requests['Incident Zip'].str.slice(0, 5)

# Fix requests with 00000 zipcodes
zero_zips = requests['Incident Zip'] == '00000’
requests['Incident Zip'][zero_zips] = np.nan

# Display unique incident zips
print requests['Incident Zip'].unique()
```

Adapted from http://pandas.pydata.org/pandas-docs/stable/tutorials.html (chapter 7)

Grizzly in Action

Pandas for I/O

```
import pandas as pd
import grizzly as gr

# Read dataframe from file
requests = gr.DataFrameWeld(pd.read_csv('filename.csv'))


# Fix requests with extra digits
requests['Incident Zip'] = requests['Incident Zip'].str.slice(0, 5)

# Fix requests with 00000 zipcodes
zero_zips = requests['Incident Zip'] == '00000'
requests['Incident Zip'][zero_zips] = np.nan

# Display unique incident zips
print requests['Incident Zip'].unique()
```

Adapted from http://pandas.pydata.org/pandas-docs/stable/tutorials.html (chapter 7)

Integration Effort

Integration Effort

Small up front cost to enable Weld integration

- 500 LoC for each library we prototyped

Integration Effort

Small up front cost to enable Weld integration

- 500 LoC for each library we prototyped

Easy to port over each operator

- 30 LoC each

Integration Effort

Small up front cost to enable Weld integration

- 500 LoC for each library we prototyped

Easy to port over each operator

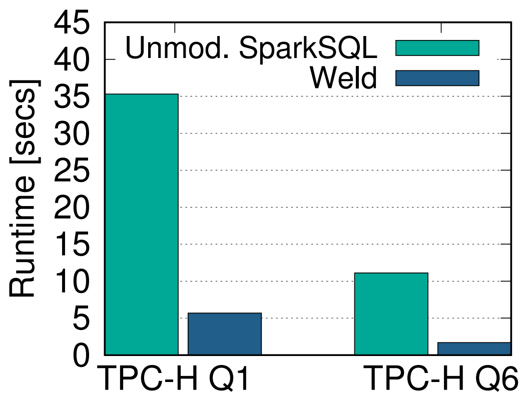
- 30 LoC each

Incrementally Deployable

- Weld-enabled ops work with native ops

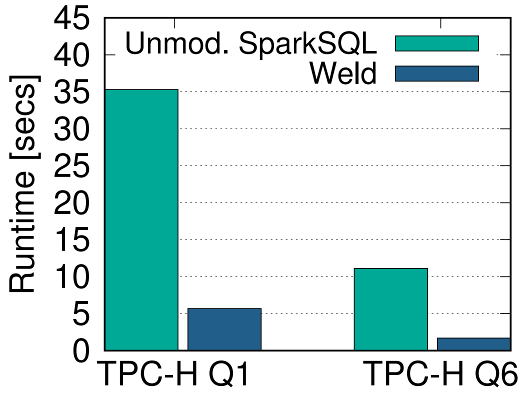
Weld Accelerates Existing Libraries

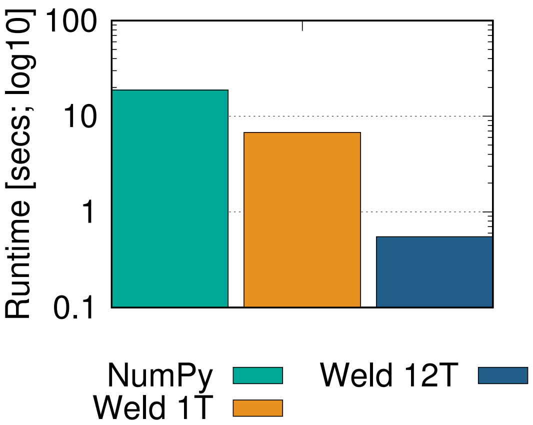
Weld Accelerates Existing Libraries



TPC-H:
3.5x speedup

Weld Accelerates Existing Libraries

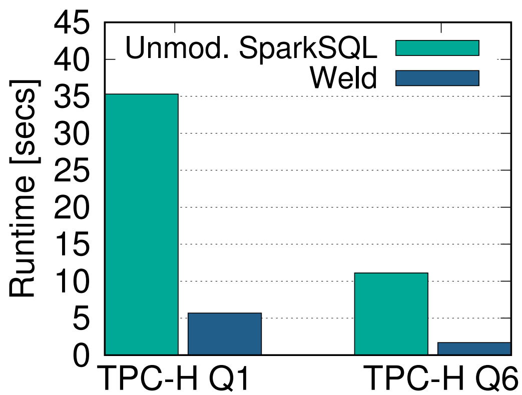


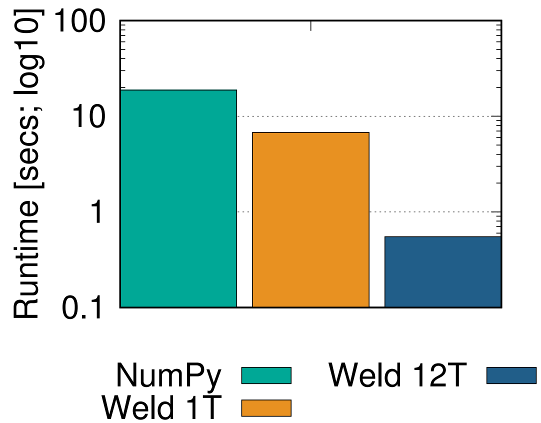


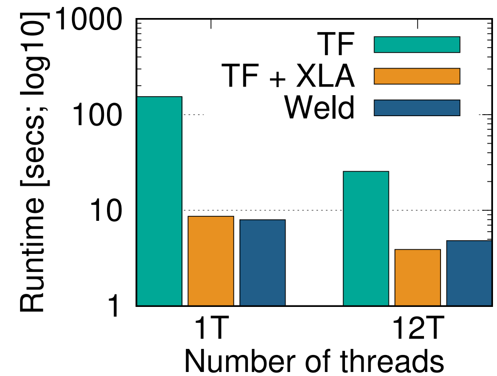
TPC-H:
3.5x speedup

Black Scholes:
4.5x speedup

Weld Accelerates Existing Libraries







Spark SQL

TPC-H:
3.5x speedup

NumPy

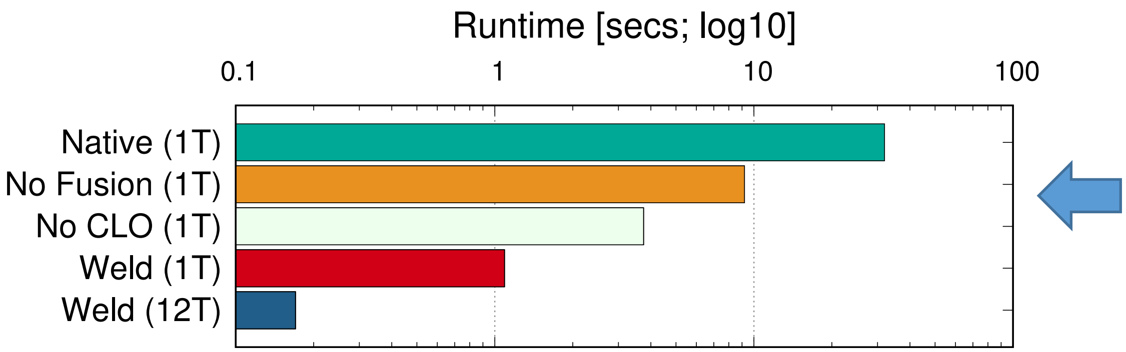
Black Scholes:
4.5x speedup

TensorFlow

Logistic Regression:
Competitive with XLA

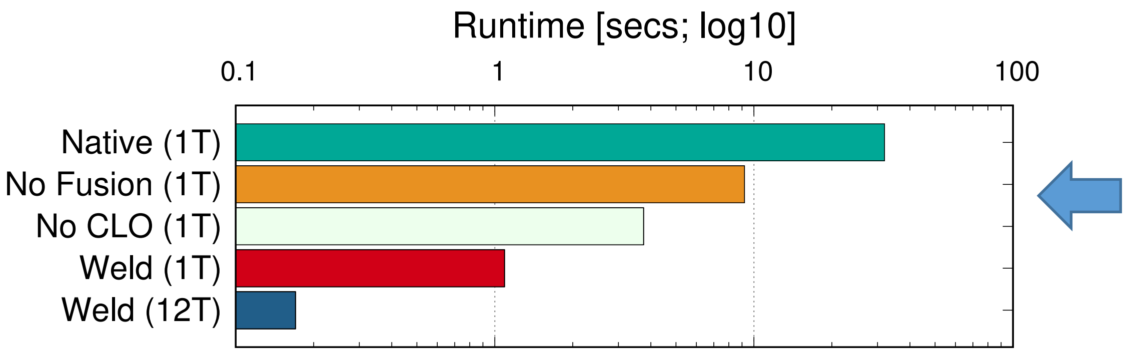
Weld Accelerates Multi-Library Workflows

Weld Accelerates Multi-Library Workflows



Data cleaning + lin. alg. with Pandas + NumPy: **180x** speedup

Weld Accelerates Multi-Library Workflows



Data cleaning + lin. alg. with Pandas + NumPy: **180x** speedup

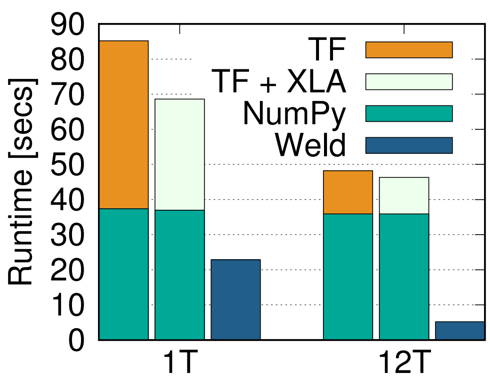
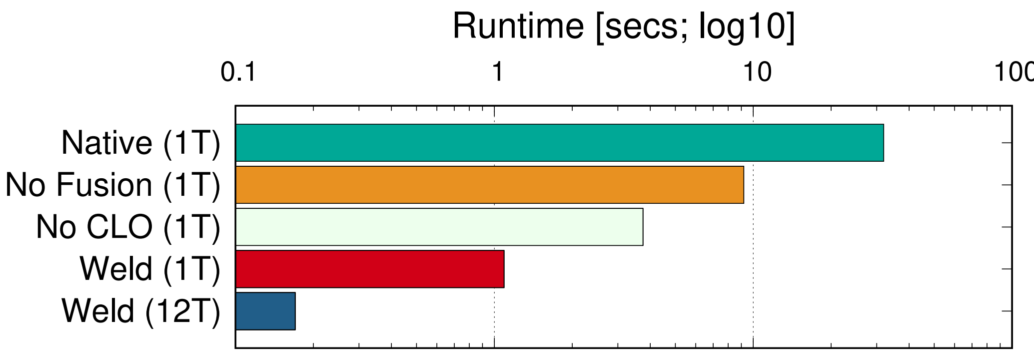


Image whitening + linear regression with TensorFlow + NumPy: **8.9x** speedup

Weld Accelerates Multi-Library Workflows



Data cleaning + lin. alg. with Pandas + NumPy: **180x** speedup

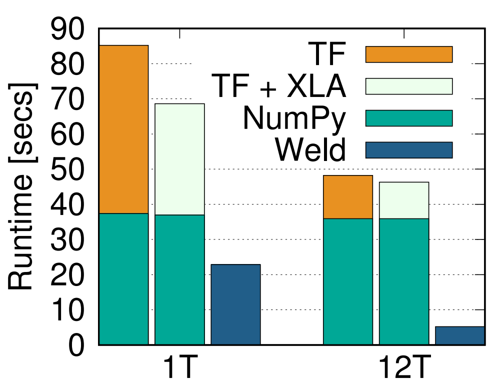
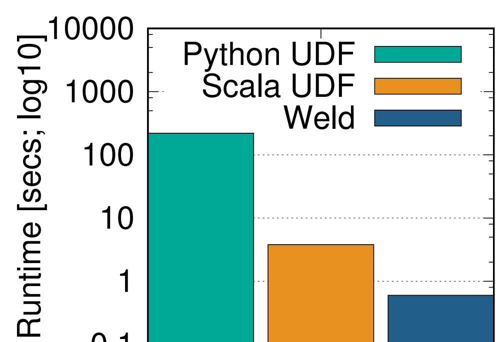
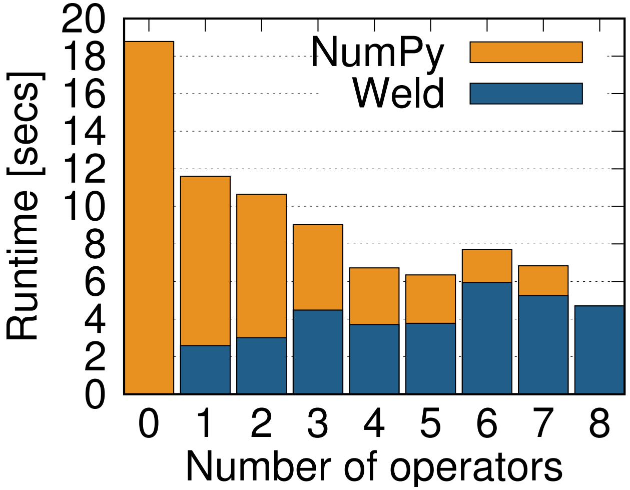


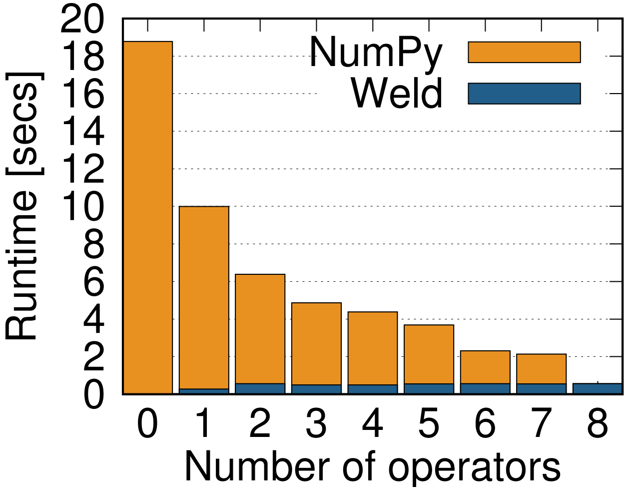
Image whitening + linear regression with TensorFlow + NumPy: **8.9x** speedup



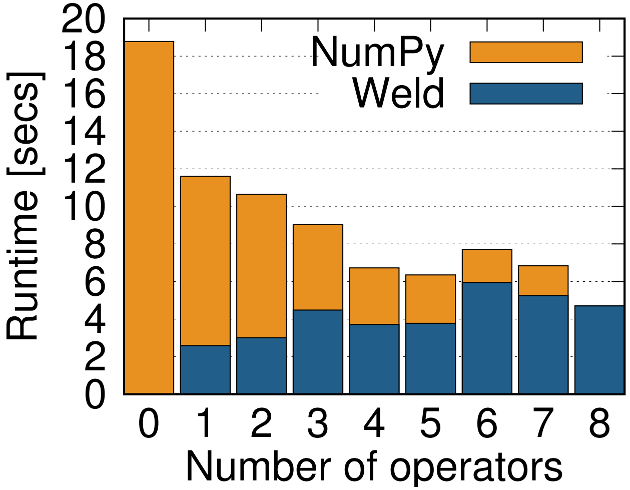
Linear model eval. with Spark SQL UDF: **6x** speedup

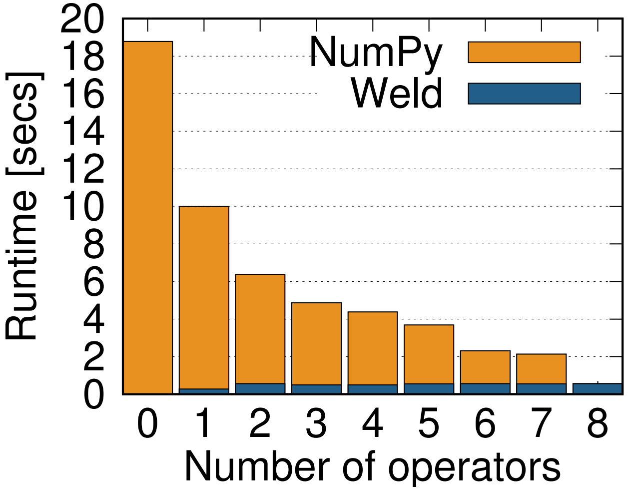
Incremental Integration

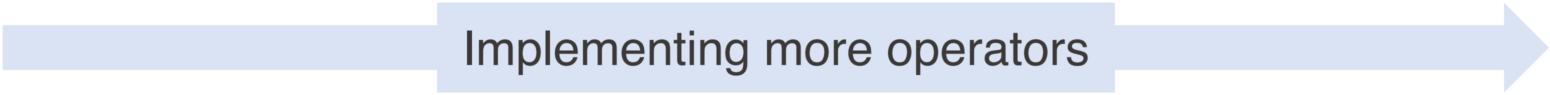




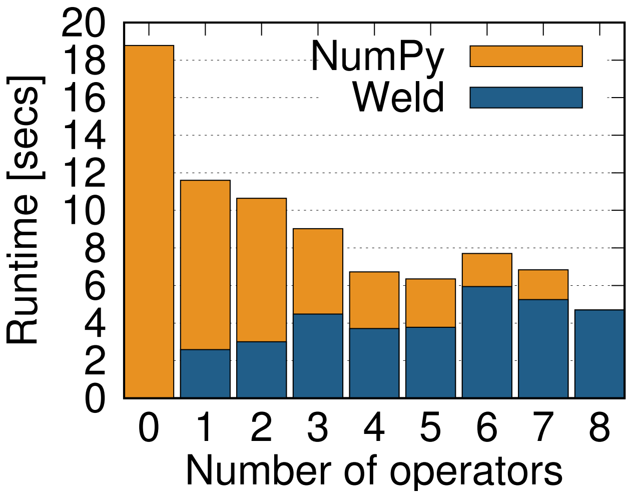
Incremental Integration

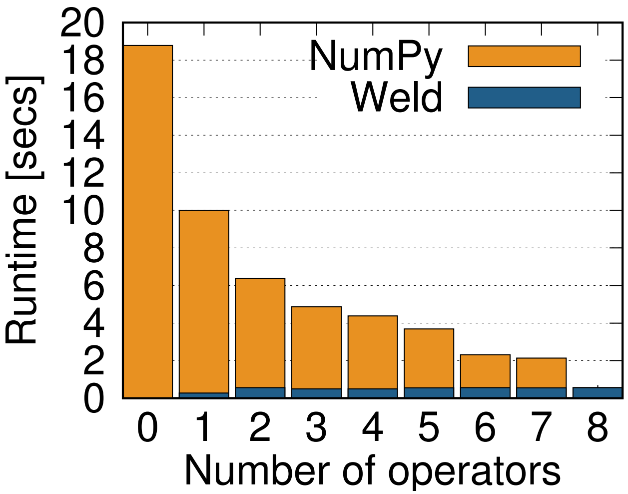






Incremental Integration





Implementing more operators

NumPy Black Scholes workload:

Incremental benefits with incremental integration.

Demo.

Conclusion

Changing the interface between libraries can speed up data analytics applications by 10-100x on modern hardware

Try out Weld for yourself, or contribute!

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

https://www.weld.rs

```
$ pip install pyweld
$ pip install pygrizzly
$ pip install weldnumpy
```

STANFORD INFOLAB