Weld: A Common Runtime for Data Analytics


Stanford InfoLab, *MIT CSAIL
Motivation

Modern data apps combine many disjoint processing libraries & functions
» Relational, statistics, machine learning, ...
» E.g. PyData stack

+ Great results leveraging work of 1000s of authors
– No optimization across these 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)

5-30x slowdowns in NumPy, Pandas, TensorFlow, etc
How We Solve This

SQL
machine learning
graph algorithms

Common Runtime

CPU
GPU
How We Solve This

- SQL
- machine learning
- graph algorithms

Weld runtime

Weld IR

Backends

Runtime API

Optimizer

CPU

GPU
Runtime API

Uses lazy evaluation to collect work across libraries

data = lib1.f1()
lib2.map(data,
    item => lib3.f2(item)
)
Weld IR

Designed to meet three goals:

1. **Library composition**: support complete workloads such as nested parallel calls

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

3. **Explicit parallelism and targeting parallel hardware**
Why Don’t Compilers Solve This?

Languages and intermediate representations (IRs) make it hard to optimize across libraries

» Main abstraction is shared memory
  (must worry about aliasing, order, etc)

Most compilers don’t model parallel operations

» Makes high performance code generation for heterogeneous parallel hardware even more difficult
Weld IR

Small, powerful design inspired by “monad comprehensions”

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
Builders

Hardware independent and explicitly parallel

Three operators:

**merge(builder, value):** Merge a value into the builder and return a new builder

**result(builder):** destroy the builder and return a value

**for(data, builders, func):** iterate over data, potentially merging values into one or more builders in parallel
Examples

Implement functional operators using builders

```python
def map(data, f):
    builder = new vecbuilder[int]
    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 Optimization: Fusion**

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

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

Loops can be merged into one pass over data
Optimizer

Cost Based Optimizer similar to an RDBMS

**Builder Implementations:** How to implement a particular builder (e.g., global vs. local hash tables)

**Transforms:** Should expressions be fused, vectorized, inlined, etc.

Quantifies choices among optimizations using data from the program
Cost Model

Inspired by cost models for in-memory databases

+ modeling nested parallelism and choices among implementations of data structures

Models cache contention, costs of atomic instructions, etc.
Implementation

Prototype with APIs in Scala and Python
  » LLVM and Voodoo for code gen

Integrations: TensorFlow, NumPy, Pandas, Spark
Results: Individual Workloads

SQL (TPC-H)

PageRank

Word2Vec

TF-Op = C++ operator
Results: Existing Frameworks

- **SparkSQL**
- **Weld**

### TPC-H Q1, Q6

<table>
<thead>
<tr>
<th>Workload</th>
<th>SparkSQL</th>
<th>Weld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [secs]</td>
<td>40</td>
<td>5</td>
</tr>
</tbody>
</table>

### NExpr, NP

<table>
<thead>
<tr>
<th>Workload</th>
<th>NP</th>
<th>NExpr</th>
<th>Weld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [secs]</td>
<td>0.2</td>
<td>0.04</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### TF, Hand-opt, Weld

<table>
<thead>
<tr>
<th>Workload</th>
<th>TF</th>
<th>Hand-opt</th>
<th>Weld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [secs; log10]</td>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Integration effort:** 500 lines glue, 30 lines/operator
Results: Modeling Costs

Takeaway: Cost curves *resemble* actual runtimes
Results: Cross-Library Optimization

**Pandas + NumPy**

- Current
- Weld, no CLO
- Weld, CLO
- Weld, 12 core

**Spark SQL UDF**

- Scala UDF
- Weld
Conclusion

The way we compose software will have to change to efficiently use modern hardware

Lots of open questions and design decisions!
  » Leveraging specialized hardware, domain info, …

Open source: soon!
Related Work

HyPer, LegoBase, Tupleware: target relational algebra and serial UDFs; no nested parallelism

LLVM, OpenCL: low-level shared-memory model

NESL, parallel FP: not closed under optimizations

DSLs: Weld focuses on integration with existing libraries and cross-library optimization
Observation

Many analytics algorithms can be written with a few “embarrassingly parallel” operators
  » See how many run on MapReduce / Spark

Focus on these instead of general programs
The Goal

- SQL
- machine learning
- graph algorithms
- CPUs
- GPUs
- ...
Results on GPUs

SQL (TPC-H)

```
<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Ocelot</th>
<th>Weld</th>
<th>SQL (TPC-H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Q3</td>
<td>0.2</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>Q6</td>
<td>0.3</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>Q12</td>
<td>0.4</td>
<td>0.4</td>
<td>0.8</td>
</tr>
</tbody>
</table>
```

Nearest Neighbors

```
<table>
<thead>
<tr>
<th>Runtime [secs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF</td>
</tr>
<tr>
<td>5.9</td>
</tr>
</tbody>
</table>
```

PageRank

```
<table>
<thead>
<tr>
<th>Runtime [secs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond (CPU)</td>
</tr>
<tr>
<td>LightSpMV</td>
</tr>
<tr>
<td>Bond (GPU)</td>
</tr>
<tr>
<td>0.7</td>
</tr>
</tbody>
</table>
```
Example Transformations

def query(products: vec[{dept:int, price:int}]):
    sum = 0
    for p in products:
        if p.dept == 20: sum += p.price

def query(dept: vec[int], price: vector[int]):
    sum = 0
    for i in 0..len(users):
        if dept[i] == 20: sum += price[i]

    for i in 0..len(products) by 4:
        sum += price[i..i+4] * (dept[i..i+4] == [20,20,20,20])
Weld Results: TPC-H Q6

- Python: 0.53 sec
- Java: 0.14 sec
- C: 0.08 sec
- HyPer Database: 0.11 sec
- Optimized: 0.03 sec
- Weld: 0.03 sec
Effect of Optimizations

Transformations usable on any Weld program
How Weld Fits Into Applications

User Application

```javascript
data = lib1.f1()
lib2.map(data, item => lib3.f2(item))
```

Runtime API

IR fragments for each function

Combined IR program

Optimized program

Machine code

Weld Runtime

Data in application
Example: Spark + NumPy

data = spark.sql("select user.features from users where age > 20")
scores = data.map(lambda vec: scoreMatrix * vec)
average = scores.mean()
Example: Spark + NumPy

```python
data = spark.sql("select user.features from users where age > 20")

scores = data.map(lambda vec: scoreMatrix * vec)

average = scores.mean()
```
Runtime API

Uses lazy evaluation to collect work across libraries

User Application

```javascript
data = lib1.f1()
lib2.map(data,
  item => lib3.f2(item)
)
```

Weld Runtime

IR fragments for each function

Combined IR program

Optimized machine code

Data in application

Uses lazy evaluation to collect work across libraries.
Supported Optimizations

- Loop Fusion
-_loop Tiling
- Row-to-Column
- Constant Folding
- Common Subexpressions
- Branch Predication
- Inlining
- Vectorization
- Insert free() Calls
- …