Weld: A Common Runtime for High Performance Data Analytics

Shoumik Palkar, James J. Thomas, Anil Shanbhag†, Deepak Narayanan, Holger Pirk†, Malte Schwarzkopf†, Saman Amarasinghe†, Matei Zaharia

Stanford InfoLab †MIT CSAIL

Abstract

Modern analytics applications combine multiple functions from different libraries and frameworks to build increasingly complex workflows. Even though each function may achieve high performance in isolation, the performance of the combined workflow is often an order of magnitude below hardware limits due to extensive data movement across the functions. To address this problem, we propose Weld, a runtime for data-intensive applications that optimizes across disjoint libraries and functions. Weld uses a common intermediate representation to capture the structure of diverse data-parallel workloads, including SQL, machine learning and graph analytics. It then performs key data movement optimizations and generates efficient parallel code for the whole workflow. Weld can be integrated incrementally into existing frameworks like TensorFlow, Apache Spark, NumPy and Pandas without changing their user-facing APIs. We show that Weld can speed up these frameworks, as well as applications that combine them, by up to 30×.

1. INTRODUCTION

Modern data applications use an increasingly diverse mix of algorithms, libraries and systems. In most domains, applications now incorporate statistical techniques, machine learning or graph analytics in addition to relational processing. This shift has given rise to a wide range of libraries and frameworks for advanced analytics, such as TensorFlow [1], Apache Spark [37], SciDB [7], and graph frameworks [21, 31]. Given the demand for advanced analytics, future applications are likely to further embrace these frameworks.

Unfortunately, this increased diversity has also made it harder to achieve high performance. In the past, an application could push all of its data processing work to an RDBMS, which would optimize the entire application. Today, in contrast, no single system understands the whole application. Instead, researchers and developers have optimized individual libraries and functions, such as machine learning or graph algorithms [21, 26, 31]. Because real applications combine many analytics functions and frameworks, however, this approach leaves a great deal of performance on the table.

In particular, most data analytics algorithms are *data-intensive*, meaning they perform little computation per byte of input data. With separately-optimized processing functions, each individual function is fast, but data movement across the functions can dominate the execution time. For example, even though TensorFlow uses optimized BLAS kernels in its individual operators, algorithms using multiple operators can be 10–30× slower than hand-tuned code. We find similar slowdowns in workflows using Python or Spark.

To address this problem, we propose Weld, a common runtime for data analytics libraries that optimizes across them. Although defining a single runtime that *fully* understands all workloads (*e.g.*, machine learning, graphs, or SQL) is unrealistic, our key insight is that we can still find simple abstractions that capture the *structure* of these workloads and enable powerful cross-library optimizations. We show that this approach enables order-of-magnitude speedups both within and across current data processing libraries.

Weld is based on two key ideas. First, to optimize across diverse libraries, Weld asks them to express their work using a functional-like intermediate representation (IR) that is highly amenable to cross-library optimizations such as loop fusion, vectorization, data layout changes, and loop tiling [35]. This IR is sufficiently expressive to capture SQL, machine learning, graph computations and other workloads. Second, Weld offers a runtime API based on lazy evaluation that lets applications build up a Weld computation by calling different libraries, and then optimizes across them. For example, if a user runs a NumPy [28] function on each row returned by a Spark SQL [3] query, Weld can vectorize calls to the function across rows, or push predicates from the function into Spark.

We evaluate Weld using both individual algorithms and applications that combine them. On SQL, machine learning and graph benchmarks, Weld is competitive with hand-tuned code and optimized systems such as HyPer [24] and GraphMat [31]. In addition, we have integrated Weld into Spark SQL, NumPy, Pandas, and TensorFlow. Without changing the user APIs of these frameworks, Weld provides speedups of up to 6× for Spark SQL, 32× for TensorFlow, and 30× for Pandas. Moreover, in applications that combine these libraries, cross-library optimization enables speedups of up to 31×.

Finally, Weld's approach goes somewhat against the "one size does not fit all" mantra in high-performance database systems [30]. We argue that as applications start to use more and more disjoint libraries, *one size will have to fit all* to achieve bare-metal performance across them. The cost of data movement through memory is already too high to ignore in current workloads, and will likely get worse with the growing gap between processor and memory speeds. The traditional method of composing libraries, through functions that pass pointers to in-memory data, will be unacceptably slow. To solve this problem, the research community must search for new interfaces that enable efficient composition. Weld's common runtime approach is one such design point to bring end-to-end optimization to advanced analytics.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

2. DESIGN PHILOSOPHY

The main question in designing a runtime like Weld is the level of abstraction at which it should operate. Clearly, there is a tradeoff between how much the runtime "knows" about workloads (*e.g.*, does it have machine learning operators as first-class primitives?), and how difficult it is to implement and maintain. In theory, for example, one could imagine extending an RDBMS with primitives for machine learning, graph analytics, and other workloads, and incorporating them into its optimizer and cost model. While past research has shown good results pushing various analytics algorithms into an RDBMS [7, 9, 14, 15], this approach is an uphill battle. For example, in-database machine learning libraries like MADlib [14] do not support new methods such as deep learning, and natural expressions of these methods in SQL do not perform well. Thus, most machine learning development still happens outside databases. Moreover, extending an RDBMS with a new data type or operator is complex, which limits the set of people who can do it, and further limits the rate at which new methods will be made available to users.

In Weld, we take a fundamentally different approach. We believe that the industry will continue to develop disparate libraries for data analysis, and we wish to design a *simple, minimal* runtime that will perform key optimizations across them while giving users the freedom to pick and choose libraries from any developer. Concretely, our design is based on the following three principles:

1. **Work with independently written libraries.** We believe that future applications will continue to mix multiple libraries, and developers will not wait for a single über-system to incorporate the best algorithms. The runtime must extract enough information from *existing, specialized* libraries to optimize across them, and offer enough performance benefits that library developers adopt it, but otherwise get out of their way.

2. **Enable the most impactful cross-library optimizations.** Although many complex domain-specific optimizations (*e.g.*, optimizing mathematical expressions) could be applied across libraries, we believe that the most bang-for-the-buck will be in two simple areas that affect all libraries: data movement and parallelism. We need to make sure that our runtime supports data movement optimizations such as pipelining across libraries and loop tiling, and that it understands the parallel structure of each computation and can therefore optimize parallel code. We show that just these optimizations give some computations in existing libraries a 31× performance boost on a single core and further speedups on multiple cores.

3. **Integrate incrementally into existing systems.** Because the runtime requires integration with each library, we design it for incremental adoption. In particular, Weld's runtime API allows libraries to move some of their functions to Weld (*e.g.*, specific TensorFlow operators) while keeping the rest outside. Moreover, Weld's programming interface is familiar to developers: it exposes functional operations such as `map` and `reduce`, which are already widely used in data analytics, and it can call existing C functions. Finally, Weld offers key benefits even *within* a single library by optimizing across separate functions—something that would require complex code generation systems otherwise [24]. Many library developers already write their most performance-intensive operators in languages such as C or OpenCL [1, 28], so we believe there are compelling reasons to write them in Weld.

In practice, data processing systems can integrate Weld in several ways. Many systems, such as RDBMS engines and TensorFlow, break down computations into a small set of physical operators. Writing these operators in Weld will automatically allow both high performance for each operator and optimization across them. Some systems also perform runtime code generation to LLVM, C or Java [3, 24]; these can be modified to emit Weld IR instead.

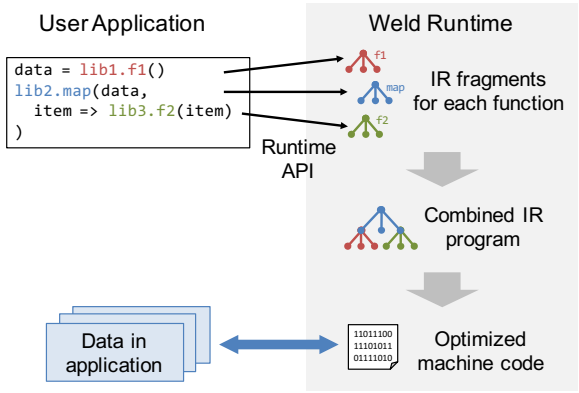


Figure 1: As applications call Weld-enabled libraries, the runtime collects fragments of IR code, which it then combines into a single Weld program. This complete program is optimized jointly and then compiled to parallel machine code, which executes against the application's in-memory data.

3. SYSTEM OVERVIEW

At a high level, Weld is based on three key ideas:

1. An **intermediate representation (IR)** that captures the structure of common data-parallel algorithms, and supports rich optimizations across them. For this purpose, we designed a minimal IR that is lower-level than both functional and relational operators, but is still able to capture their parallel structure and express complex optimizations. The IR is sufficiently general to support functional APIs such as Spark, relational operators, linear algebra and graph algorithms.

2. A **runtime API** that lets libraries expose parts of their computation as Weld IR fragments. As applications call Weld-enabled functions, the system builds a DAG of such fragments, but only executes them lazily when libraries force an evaluation. This lets the system optimize across different fragments.

3. A **compiler backend** that maps the final, combined Weld IR program to efficient multithreaded code. We implemented our current backend using LLVM [18]. Because the Weld IR is explicitly parallel, our backend automatically implements multithreading and vectorization using Intel AVX2.

Figure 1 shows how these components combine to optimize an application using multiple libraries. In the rest of this paper, we sketch the current prototype of Weld, focusing especially on the IR. We then present results that illustrate the impact of Weld.

4. WELD IR

The most important architectural element of Weld is its intermediate representation (IR) for code, which is somewhat akin to the logical operator algebra in a database. The IR determines both what workloads can run on Weld and what optimizations can easily be applied. Because our goal is to both support as wide a range of workloads as possible and to perform data movement and parallelism-aware optimizations, we needed an IR with the following properties:

- **Library composition.** Applications can combine libraries in many ways: for example, they may call one library function on the result of another, or even nest parallel functions inside each other (*e.g.*, launching a `map` operation in Spark and calling a parallelizable function such as a NumPy vector sum inside it). Because Weld will combine IR fragments from multiple libraries, the IR must support composition and nesting.

- **Explicit parallelism.** The operators in the IR must be explicitly parallel, letting the runtime know what can be done in parallel without having to infer it from sequential code.

- **Ability to express optimizations.** We wanted the IR to be able to express both the initial computation and optimizations we make over it, such as loop fusion, so that our backend can implement a wide variety of transformations without leaving the representation. This is akin to how logical plans in a database are both the input and output of optimization steps, and also follows the design of compiler IRs such as LLVM.

We investigated several existing IRs for Weld, but found that they did not meet these goals. Relational algebra expressions, as used in databases, contain explicitly parallel operators but do not support complex composition such as nesting. For example, it would be hard to express the concept of a "table" that contains vectors, each of which can be acted on in parallel.[1] In contrast, compiler IRs such as LLVM are not explicitly parallel, and would require us to infer parallel structure from low-level operations such as loads and stores.

Instead, we developed a simple parallel IR similar to monad comprehensions [11]. Our IR is based on parallel loops and a construct for merging results called "builders." Parallel loops can be nested arbitrarily, which allows complex composition of functions. Within these loops, the program can update various types of builders, which are declarative data types for constructing results in parallel (*e.g.*, computing a sum or adding items to a list). Multiple builders can be updated in the same loop, making it easy to express optimizations such as loop fusion or tiling, which change the order of execution but produce the same result.

4.1 Data Model

Weld's IR supports several common data types, shown in Table 1. Apart from scalars, it offers structures, variable-length vectors, and dictionaries. We chose these types because they appear commonly in data-intensive applications as well as low-level data processing code (*e.g.*, dictionaries are useful for hash joins). These types can be nested to represent more complex data.

4.2 Operators

Weld's IR contains basic operators for arithmetic, assigning names to values, sequential looping, and read operations on collections (*e.g.*, lookups into a hash table). It also contains a foreign function interface for calling external C functions. Our IR uses Static Single Assignment form, meaning that variables are immutable once defined, simplifying its analysis.

The IR has two parallel operators: a *parallel loop* to iterate over data, and *builders* for constructing results. A builder is a declarative data type that computes a result in parallel. Builders are *write-only, build-once*; expressions such as parallel loops can merge values into a builder, but a final result can only be materialized once, after these merges are done. Weld includes multiple types of builders, as shown in Table 2. For example, a `vecbuilder[T]` takes values of type `T` and builds a vector from the merged values. A `merger[T,func,id]` takes a commutative function and an identity value and combines values of type `T` into a single result.

[1] Of course, in relational algebra, foreign keys and joins can be used to express arbitrary relationships, but analyzing and optimizing applications expressed this way becomes challenging.

Primitive Data Types	
Scalars	`char`, `int`, `long`, `float`, ...
Structures	`{T1, T2, ...}` for field types `T1`, `T2`, ...
Vectors	`vec[T]`
Dictionaries	`dict[K, V]`

Table 1: Primitive data types in Weld.

Builder Types	
`vecbuilder[T]`	Builds a `vec[T]` from merged values of type `T`.
`merger[T,func,id]`	Builds a value of type `T` by merging values with a commutative function `func` and an identity value `id`.
`dictmerger[K,V,func]`	Builds a `dict[K,V]` by merging values with a commutative function.
`vecmerger[T,func]`	Builds a `vec[T]` from an initial vector by merging values `{index,T}` using a commutative function `func`.
`groupbuilder[K,V]`	Builds a `dict[K,vec[V]]` from merged values of type `{K,V}`.

Table 2: Builder types in Weld.

Builders support three basic operations. `merge(b, v)` adds a new value `v` into the builder `b` and returns a new builder[2] to represent the result. Merges into builders are associative, allowing them to be reordered. `result(builder)` destroys the builder and returns its final result: no further operations are allowed on it after this call. Finally, the `for(vector, builders, func)` operator applies a function of type `(builders, T) => builders` to each element of a vector in parallel, updating one more builders for each one, and returns the final set of builders. The `for` operator is the only way to launch parallel work in Weld: the iterations of the loop will run in parallel and merge their results into the provided builders.

```
// Merging two values into a builder
b1 := vecbuilder[int];
b2 := merge(b1, 5);
b3 := merge(b2, 6);
result(b3)   // returns [5,6]

// Using a for loop to merge multiple values
b1 := vecbuilder[int];
b2 := for([1,2,3], b1, (b, x) => merge(b, x+1));
result(b2)   // returns [2,3,4]

// Merging results only for some iterations
result(
  for([1,2,3],
     vecbuilder[int],
     (b, x) => if (x>1) merge(b, x) else b
)) // returns [2,3]
```

Listing 1: Some simple examples of using builders.

Note that because `for` itself returns a set builders, it can be nested and composed (see Listing 2). This lets Weld express nested parallel programs, including irregular parallelism (where instances of the

[2] In practice, some mutable state will be updated with the merged value, but Weld's IR treats all values as immutable, and so we represent the result as a new builder object in the IR.

inner loop do different amounts of work). It also makes Weld amenable to a wide range of loop transformations.

```
lists := [[1,2], [3,4,5], [6]];
result(
  for(lists, vecbuilder[int], (b, list) =>
    for(list, b, (b1, el) => merge(b1, el))
  )
)  // returns [1,2,3,4,5,6]
```

Listing 2: Using nested loops to compute a result over nested data. Here we flatten some nested lists.

Weld places two restrictions on the use of builders for efficiency and correctness. First, each builder must be consumed (passed to an operator) exactly once per control path to prevent having multiple values derive from the same builder, which would require copying its state. Therefore, formally builders are a *linear type* [34]. Second, functions passed to `for` must return builders derived from their arguments. These restrictions let the backend safely implement builders using mutable state.

4.3 Generality of the IR

Even though loops and builders are the only parallel operators in Weld, they can be used to implement a wide range of programming abstractions. Specifically, Weld supports all of the functional operators in systems like Spark, as well as all of the physical operators needed for relational algebra. For example, the last two snippets in Listing 1 implement the `map` and `filter` functional operators. Simply switching these loops to use a `merger` can implement a `reduce`. Thus, the Weld IR can express a wide variety of parallel algorithms that have been implemented in functional or relational systems such as MapReduce, Spark, MADlib, and others [9, 14, 37].

One limitation of the current version of the IR is that it is fully deterministic, so it cannot express asynchronous algorithms where threads race to update a result, such as Hogwild! [26]. We plan to investigate adding such primitives in a future version of the IR.

4.4 Why Loops and Builders?

Weld contains only loops and builders as its core operators. A strawman design for an IR might have used higher-level operators than a for loop. Unfortunately, this design prevents many optimizations from being expressed easily. Consider the example in Listing 3, where two operations produce a result over the same input vector:

```
data := [1,2,3];
r1 := map(data, x => x+1);
r2 := reduce(data, 0, (x, y) => x+y);
```

Listing 3: A `map` and `reduce` over the same input vector.

Even though both the `map` and `reduce` operations could be computed in a single pass over the data, no operator akin to `mapAndReduce` exists to compute both values in one pass. More complex optimizations, such as loop tiling, are even more difficult to express in a functional IR. By exposing all parallelism through a single loop construct over builders, patterns like the above can easily be fused into programs such as Listing 4.

```
data := [1,2,3];
result(
  for(data, {vecbuilder[int], merger[+]},
     (bs, x) =>
       {merge(bs.0, x+1), merge(bs.1, x)}
)) // returns {[2,3,4], 6}
```

Listing 4: `for` loop operating over multiple builders to produce both a vector and an aggregate in one pass.

5. RUNTIME API

Weld uses its IR in conjunction with a *runtime API* to enable optimizations even across independent libraries in an application. The runtime API, currently available in Python and Scala, lets applications build up a Weld IR expression using lazy evaluation, by combining expressions from different libraries (even across language boundaries). Libraries can then call an `evaluate()` API in methods that need to output results to run the whole expression.

To illustrate, we consider the function in Listing 5, which uses the Python Pandas [22] and NumPy libraries to compute the total population of all cities with over 500,000 residents. NumPy stores data in C-style arrays. Pandas provides a table-like "data frame" API, where each column is stored as a NumPy array.

```
def large_cities_population(data):
  # data is a Pandas DataFrame object.
  filtered = data[data["population"] > 500000]
  sum = numpy.sum(filtered)
  print sum
```

Listing 5: A sample Python program using Pandas and NumPy.

In the standard eagerly-evaluated Pandas and NumPy libraries, this code causes two data scans: one to filter out values greater than 500,000 (`filtered`), and one to sum the values. Using Weld, these scans can be fused into a single loop and the sum can be computed "on the fly"—all without changes to the user's code. Additionally, this fused loop can benefit from optimizations such as vectorization and predication for further performance gains.

To use Weld for this program, methods in the `DataFrame` class in Pandas must be extended to return a lazily evaluated Weld object. We must also provide a Weld implementation for the `>` operator on `DataFrame` columns and for the `numpy.sum` function.

Listing 6 shows the Weld IR expressions for implementations of each of these functions. Both implementations use functional "sugar" operators (`filter` and `reduce`) that compile into the IR as described in Section 4.3. The runtime API provides support for building Weld expressions from existing objects.

```
# DataFrame column > filter
# Input Weld expressions: v0: vec[int], c0: int
filter(v0, x => x > c0)

# numpy.sum
# Input Weld expressions: v0: vec[int]
reduce(v0, 0, (x, y) => x+y)
```

Listing 6: Implementations of Pandas and NumPy functions using Weld. In the above example, `v0` and `c0` can be other Weld expressions.

With these implementations, we can now build a Weld program by combining them. Listing 7 shows the final fused Weld expression for the variable `sum`.

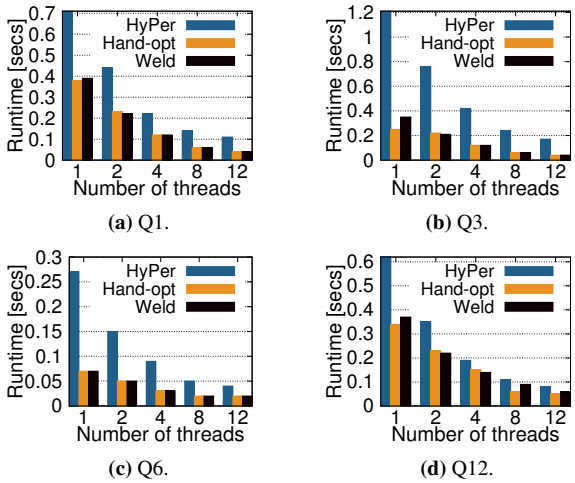
```
reduce(filter(v0, x => x > 500000),
    0, (x, y) => x+y)
```

Listing 7: The combined Weld program.

Finally, the `print` statement calls Python's `__str__` operation on this variable to convert it to a string. At this point, our modified NumPy library forces the runtime to optimize and evaluate the Weld expression. After optimization, this function becomes a single parallel loop using a sum builder, as shown in Listing 8:

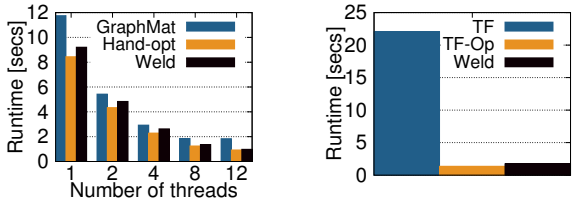
```
result(for(v0, merger[+,0],
    (b, x) => if (x > 500000) merge(b, x) else b))
```

Listing 8: The optimized Weld program after loop fusion.



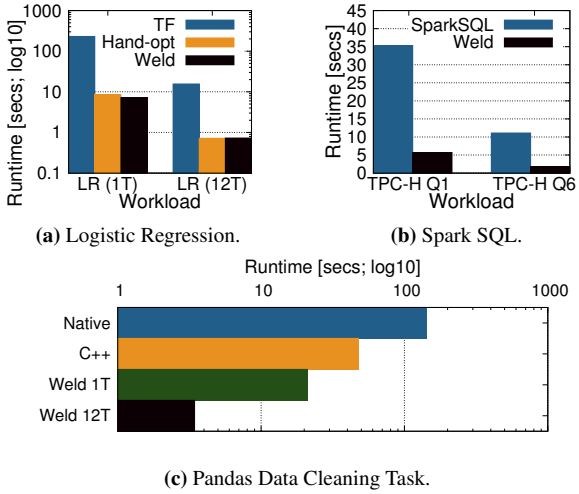
(a) Q1. **(b)** Q3. **(c)** Q6. **(d)** Q12.

Figure 2: Results for TPC-H queries. Weld generates code that is competitive with HyPer and with hand-optimized C++.



(a) PageRank. **(b)** Word2Vec.

Figure 3: Results for PageRank and Word2Vec. Weld's generated code outperforms GraphMat's PageRank implementation and is competitive with TensorFlow's hand-optimized Word2Vec operator (TF-Op).



(a) Logistic Regression. **(b)** Spark SQL.

(c) Pandas Data Cleaning Task.

Figure 4: *Library Integration Results.* **4a**. Integrating Weld into TensorFlow yields a 32× speedup on one core and 21× on 12 cores. **4b**. Replacing Java codegen with Weld in Spark SQL speeds up TPC-H Q1 and Q6 by 6×. **4c**. Weld speeds up Pandas by 4× on a single core, and by 27× on 12 cores.

6. PROTOTYPE EVALUATION

We have implemented a prototype of Weld with APIs in Scala and Python that compiles expressions to multithreaded code using LLVM. The prototype also supports capturing user-defined functions using AST introspection, similar to LINQ [36] and TupleWare [6], for integration into functional APIs such as Spark. To interact with data in the host program, Weld takes pointers directly to existing data, and is compatible with the layout of C arrays and `structs`. The prototype implements several optimizations, including loop fusion, loop tiling, common subexpression elimination, and vectorization.

In this section, we substantiate the core claims of Weld, namely (1) that Weld can generate highly efficient code for diverse data applications and (2) that there is substantial benefit to optimizing *across* data-intensive libraries and functions. We run our benchmarks on a machine with an Intel Xeon E5-2680 v3 CPU with 12 cores (24 hyperthreads). Our handwritten baselines are compiled using Clang 3.5 (`-O3`, `-march=native`).

6.1 Performance vs. State of the Art

To evaluate Weld's raw performance on diverse workloads, we implemented three benchmarks: a set of TPC-H [33] queries for SQL, PageRank for graph analytics, and a simple neural network called Word2Vec for machine learning. For each, we compare Weld to a hand-optimized C++ implementation and an existing high-performance framework. Our C++ implementations are vectorized manually using Intel AVX2 intrinsics and techniques such as predication; they represent our best effort to extract peak performance from our hardware.

Figure 2 shows Weld's execution time for four TPC-H queries running at scale factor of 10, compared to the HyPer v0.5 database [24] and handwritten implementations. Weld is competitive or outperforms HyPer for all queries. Weld's speedups for Q6 and Q12 come from implementing predication and vectorization in the backend; HyPer depends on LLVM to vectorize code, which it fails to do for these queries. Weld's speedups on other queries come from lightweight hash table implementations and dynamic load balancing of work across cores.

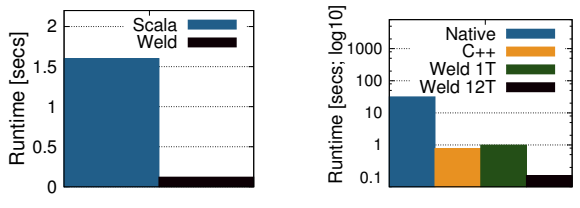
Figure 3a shows Weld's PageRank performance on the `twitter_rv` graph, comparing against a C++ implementation and against GraphMat [31], the fastest multicore framework we found for PageRank. Weld generates code which is competitive with C++ and outperforms GraphMat. GraphMat's worse performance is likely due to the extra work required to track the activeness of vertices (necessary for other graph algorithms, but not for this version of PageRank).

Finally, Figure 3b shows our results for a common neural network algorithm called Word2Vec [32], which maps words in a vocabulary to a smaller D-dimensional space. We compare Weld's implementation to TensorFlow. Because this algorithm is data-intensive, the TensorFlow developers have also developed a custom C++ kernel that combines the core operators in Word2Vec to eliminate data movement cost [32]. We also compare with this version, labeled TF-Op. Weld outperforms the standard TensorFlow version by 12× by eliminating data movement, and is competitive with the hand-written TF-Op. Weld can thus generate parallel code competitive with state-of-the-art systems across at least three domains.

6.2 Accelerating Existing Frameworks

We also prototyped integrations of Weld into four common data processing frameworks: Spark SQL for relational queries, TensorFlow for machine learning, NumPy for linear algebra, and Pandas for data science. In all cases, we added sufficient support to run the operators required for a small test workload.

TensorFlow. We evaluate TensorFlow on a binary logistic regression classifier trained on the MNIST dataset [20]. The classifier identifies each digit in the dataset as either zero or nonzero. We evaluate the default TensorFlow implementation against a Weld-enabled TensorFlow implementation and an optimized implementation using the Eigen [12] library. Figure 4a shows the results. On a single core,



(a) Spark SQL UDF. **(b)** Pandas + NumPy Workflow.

Figure 5: 5a. Optimizing across a Spark SQL query and a UDF with Weld gives a 14× speedup over unmodified Spark SQL. **5b**. Optimizing across NumPy and Pandas gives a 31× speedup on a single core and a further 8× speedup on 12 cores.

Weld's lazy evaluation allows IR fragments from each operator to be fused and co-optimized, and achieves a 32× speedup over the baseline. This speedup comes from whole-program optimizations such as dead code elimination, as well as loop fusion to prevent intermediate result materialization. With multiple cores, this speedup reduces to 21×, since the Weld version becomes memory-bound. Weld's performance is competitive with the C++/Eigen code.

Spark SQL. We also integrated Weld into Spark SQL to demonstrate how Weld can improve performance in distributed systems by accelerating single-node performance. We ran these experiments on a 21-node Amazon EC2 r3.xlarge cluster, with one instance running the Spark driver and the others running executors. We evaluated TPC-H queries 1 and 6 on data with a scale factor of 800. The data was read from Spark's in-memory cache. Figure 4b shows the result; Weld integration provides a 6.1× speedup for Query 1 and a 6.5× speedup for Query 6; these speedups come largely from generating native machine code instead of Java code as in Spark SQL.

Pandas. We ran our Pandas integration on a data science tutorial we found online [8]. The workload uses Pandas to clean a dataset of zipcodes, using operators such as string slicing and filtering to strip zipcodes to five digits, remove nonexistent zipcodes, and uniquify the list of zipcodes after these transformations. Figure 4c shows the results, compared with native Pandas and a handwritten C++ implementation. Weld produces fast code by fusing operators in Pandas; even though Pandas' underlying implementation is in C++, materializing intermediate DataFrames after each operation is costly. Weld also enables automatic multi-threading in the otherwise single-threaded Pandas library; in all, Weld provides a 4.2× improvement over the standard library implementation, and a further 6.5× improvement when transparently parallelizing to 12 cores.

6.3 Cross-Library Optimization

Weld's runtime API also enables substantial optimization *across* libraries. We illustrate this using a Spark SQL query that calls a User-Defined Function (UDF) written in Scala, as well as a Python data science workload that combines Pandas and NumPy.

Figure 5a shows the result for Spark SQL. We compare a Spark SQL query using an opaque Scala UDF to one using a Weld-backed UDF. The query calls the UDF on each row of a table, then sums the results. Without Weld, Spark SQL generates Java code for the entire query, but the call to the Scala UDF is costly and needs data conversions. Weld combines the whole task into a single IR program, which is optimized to vectorize the UDF calls *across* SQL table rows before summing them. This leads to a 14× speedup.

The Python workload starts with the Pandas data cleaning task in the previous section [8], then evaluates a simple linear model in NumPy to compute a crime index for each city. It then uses NumPy to aggregate these indices into a total crime index. Figure 5b shows the results. Because Weld collects IR fragments from disjoint libraries and optimizes them together, the system can prevent data movement even across library boundaries. Despite the native NumPy library running over BLAS kernels [19], we observe speedups of 31× over the baseline even on a single core. Apart from preventing materialization through loop fusion, we see further speedups by enabling vectorization of operators across libraries (*e.g.*, the filter in the Pandas library is vectorized and acts as a predicate for the computation carried out by the NumPy function). With multiple cores, we see a *further* 8× speedup over single-threaded Weld execution, for an overall 250× speedup over the Pandas baseline.

Library	Glue Code LoC	Per-Operator LoC
NumPy	Py: 84, C++: 24	avg: 16, max: 50
Pandas	Py: 416, C++: 153	avg: 22, max: 64
Spark SQL	Py: 5, Scala: 300	avg: 23, max: 63
TensorFlow	Py: 175, C++: 652	avg: 22, max: 85

Table 3: Number of lines of code in our library integrations.

6.4 Integration Effort

Our integrations into TensorFlow, Spark SQL, Pandas, and NumPy required fairly low effort, taking only a few graduate student days per framework. Each integration required about 500 lines of one-time "glue" code per framework, which mainly involves marshalling input data, calling Weld's API to build expressions, and subsequent un-marshalling of the output produced by Weld. In addition, each integration required 50–100 lines per operator ported after the glue code was written. This shows that Weld can be integrated into existing systems incrementally at relatively low cost.

TensorFlow and Spark SQL both have lazily evaluated APIs, so porting them to use Weld was straightforward. Glue code for TensorFlow also included rewriting the parts of the data-flow graph to use Weld operators instead, plus some work in wrapping raw data pointers into types that Weld's runtime understands. Spark SQL already performs Java code generation on a per-operator basis, so porting this framework simply entailed generating Weld code instead. In fact, Weld could automatically apply optimizations that required significant effort on the part of the Spark SQL developers, such as multi-operator fusion in the style of HyPer [24]. Pandas and NumPy are both eagerly evaluated frameworks, so some glue code was required to produce results *lazily* until certain "output" operators such as `print` are called. Table 3 summarizes the integration effort.

7. RELATED WORK

Weld builds on ideas in multiple fields, including compilers, parallel programming models, database engines, and domain-specific languages (DSLs). Unlike most existing systems, however, it aims to provide a runtime that can be used across *diverse existing libraries* instead of creating a new, standalone programming model in which data-intensive applications should be built.

Runtime code generation is used in RDBMS engines like HyPer [24], LegoBase [17] and Voodoo [25]. These systems are restricted to the relational model, however, and are often complex to write because they need to generate imperative code directly from multiple operators. Tupleware [6] also integrates LLVM-based UDFs into the generated code, but does not aim to integrate general, independently written parallel libraries.

NESL [4], Data-Parallel Haskell [13], and data flow engines like DryadLINQ, Musketeer and Spark [10, 23, 36, 37] use functional or relational operators as a parallel IR. While they support some types of fusion transformations, this choice makes it hard to express other transformations that can be captured in Weld, such as loops

that produce multiple results (§4.4) and loop tiling.

OpenCL [29] and SPIR [16] are low-level interfaces to diverse parallel hardware. They let users launch multiple copies of a kernel function in parallel, but do not aim to optimize across different kernel invocations in the same program. In addition, their intermediate representation is sequential (C- or LLVM-like) code.

Weld's IR is closest to monad comprehensions [11] and to Delite's multiloop construct [5, 27], both of which support nested parallel loops that emit multiple results, and perform sophisticated loop optimizations. However, unlike these systems, which apply the IR in a "closed" environment (relational algebra or DSLs written over a common framework), Weld focuses on optimizing across separately developed, existing libraries written in current languages.

8. CONCLUSION AND FUTURE WORK

With the advent of advanced analytics, data applications have become significantly more complex, combining multiple independently written libraries for functions such as feature transformation, graph analytics and machine learning. Unfortunately, the traditional method of combining libraries via function calls runs into a fundamental barrier on modern hardware: the cost of data movement starts to dominate, creating order-of-magnitude slowdowns. This cost is likely to get worse with the increasing gap between computing capacity and memory speeds.

To address this problem, libraries will need to adopt richer interfaces for composition, which facilitate cross-library optimizations while still giving their developers enough flexibility to implement sophisticated algorithms. We have explored one such interface in Weld, through an API and IR that capture enough structure from data-parallel libraries to perform key data movement and parallelism optimizations, yet support a wide range of workloads. We showed that Weld enables speedups of up to 32× in current data science frameworks such as Spark, TensorFlow, NumPy and Pandas, and can be integrated incrementally into each framework.

Weld represents only the first step towards a more efficient interface for advanced analytics applications, and one specific design point, but we hope that it helps frame important research questions. In particular, some questions we are exploring next include:

- **Optimization.** How should we design a program optimizer for Weld? Existing techniques from databases, such as cost-based optimization, are likely to be very helpful for the Weld IR, but they need to be adapted to the much broader set of workloads that Weld supports (e.g., nested loops and nested data structures). In addition, Weld's setting has fewer data statistics available by default. Alternatively, one can imagine sampling or adaptive re-optimization to respond to statistics measured at runtime.
- **Data placement.** Our current IR does not explicitly represent data placement and locality optimizations, though the loop fusion optimizations try to minimize data movement. We would like to extend the IR to support richer placement controls and optimization for targeting NUMA hardware devices.
- **Data access methods.** To integrate Weld into libraries with complex internal data formats, it may be necessary to read and write data to specialized formats such as Parquet, Protocol Buffers, or in-memory pointer-based formats. This process can be costly, making integration of query processing and (lazy) loading attractive [2]. Alternatively, it may be possible to transform Weld IR code to operate directly against other formats.
- **Domain-specific extensions.** While the current IR focuses solely on parallel loops and builders in order to fuse operations and minimize data movement, it is clear that further performance could be gained from adding more domain-specific knowledge. Where is the sweet spot between adding domain knowledge and adding complexity to the IR? For example, would adding linear algebra types and operations enable significant optimizations across math libraries? We plan to investigate extending Weld with User-Defined Types in the tradition of extensible optimizers and database engines.
- **Multi-query execution and optimization.** A classic database technique that is applicable more broadly is sharing work across multiple queries. By waiting to capture multiple Weld expressions within a data science program, we may be able to perform data sharing in complex machine learning or graph workloads.
- **Heterogeneous hardware.** Weld's explicitly parallel representation and support for complex transformations also makes it a good candidate to target additional hardware platforms, such as GPUs and FPGAs. In many domains, these platforms have become essential for performance, so expanding the set of backends we support could really help diversify Weld's use cases.

9. ACKNOWLEDGEMENTS

Sam Madden contributed significantly to the development of this project. We also thank the CIDR reviewers, Peter Bailis, Chris Ré, and our colleagues at the Stanford InfoLab and MIT for their thoughtful feedback.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proc. USENIX OSDI*. Savannah, GA, USA, 2016, pp. 265–283.

[2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. "NoDB: efficient query execution on raw data files". In: *Proc. ACM SIGMOD*. Scottsdale, AZ, USA, 2012, pp. 241–252.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, et al. "Spark SQL: Relational Data Processing in Spark". In: *Proc. ACM SIGMOD*. Melbourne, Victoria, Australia, 2015, pp. 1383–1394.

[4] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. "Implementation of a Portable Nested Data-parallel Language". In: *SIGPLAN Not.* 28.7 (1993), pp. 102–111.

[5] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. "Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns". In: *Proc. CGO*. Barcelona, Spain, 2016, pp. 194–205.

[6] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-centric Workflows". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1466–1477.

[7] Philippe Cudre-Mauroux, Hideaki Kimura, K.-T. Lim, Jennie Rogers, R. Simakov, Emad Soroush, et al. "A Demonstration of SciDB: A Science-oriented DBMS". In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1534–1537.

[8] Julia Evans. *Pandas Cookbook example*. URL: `http://nbviewer.jupyter.org/github/jvns/pandas-cookbook/blob/v0.1/cookbook/Chapter%207%20-%20Cleaning%20up%20messy%20data.ipynb`.

[9] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. "The Case Against Specialized Graph Analytics Engines". In: *Proc. CIDR*. Asilomar, CA, USA, 2015.

[10] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. "Musketeer: All for One, One for All in Data Processing Systems". In: *Proc. ACM EuroSys*. Bordeaux, France, 2015, 2:1–2:16.

[11] Torsten Grust. "Monad Comprehensions: A Versatile Representation for Queries". In: *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 288–311.

[12] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. URL: `http://eigen.tuxfamily.org`.

[13] HaskellWiki. *Data Parallel Haskell*. URL: `https://wiki.haskell.org/GHC/Data_Parallel_Haskell`.

[14] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, et al. "The MADlib Analytics Library: Or MAD Skills, the SQL". In: *Proc. VLDB* 5.12 (2012), pp. 1700–1711.

[15] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. "Vertexica: Your Relational Friend for Graph Analytics!" In: *Proc. VLDB* 7.13 (2014), pp. 1669–1672.

[16] John Kessenich. *An Introduction to SPIR-V*. 2015. URL: `https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf`.

[17] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-level Language". In: *Proc. VLDB* 7.10 (2014), pp. 853–864.

[18] Christian Lattner and Vikram Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: *Proc. CGO*. Palo Alto, CA, USA, 2004, pp. 75–86.

[19] Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (1979), pp. 308–323.

[20] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. *The MNIST database of handwritten digits*. URL: `http://yann.lecun.com/exdb/mnist/`.

[21] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB* 5.8 (2012), pp. 716–727.

[22] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proc. SciPy*. Austin, TX, USA, 2010, pp. 51–56.

[23] Derek G. Murray, Michael Isard, and Yuan Yu. "Steno: Automatic Optimization of Declarative Queries". In: *Proc. SIGPLAN PLDI*. San Jose, CA, USA, 2011, pp. 121–131.

[24] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *Proc. VLDB* 4.9 (2011), pp. 539–550.

[25] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. "Voodoo-a vector algebra for portable database performance on modern hardware". In: *Proc. VLDB* 9.14 (2016), pp. 1707–1718.

[26] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger. Curran Associates, Inc., 2011, pp. 693–701.

[27] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, et al. "Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging". In: *Proc. ACM POPL*. Rome, Italy, 2013.

[28] SciPy.org. *NumPy library*. URL: `http://www.numpy.org/`.

[29] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73.

[30] Michael Stonebraker and Ugur Cetintemel. ""One Size Fits All": An Idea Whose Time Has Come and Gone". In: *Proc. ICDE*. Tokyo, Japan, 2005, pp. 2–11.

[31] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, et al. "GraphMat: High Performance Graph Analytics Made Productive". In: *Proc. VLDB* 8.11 (2015), pp. 1214–1225.

[32] TensorFlow documentation. *Vector Representations of Words, `word2vec` example*. URL: `https://www.tensorflow.org/versions/r0.11/tutorials/word2vec/index.html`.

[33] Transaction Processing Performance Council. *TPC-H ad-hoc, decision support benchmark*. URL: `http://www.tpc.org/tpch/`.

[34] David Walker. "Substructural Type Systems". In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. MIT Press, 2005. Chap. 1.

[35] Michael E. Wolfe. "More Iteration Space Tiling". In: *Proc. ACM/IEEE Supercomputing*. Reno, NV, USA, 1989, pp. 655–664.

[36] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language". In: *Proc. USENIX OSDI*. San Diego, CA, USA, 2008, pp. 1–14.

[37] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (2016), pp. 56–65.