Weld: Rethinking the Interface Between Data-Intensive Libraries

Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag[†], Rahul Palamuttam, Holger Pirk[‡], Malte Schwarzkopf[†], Saman Amarasinghe[†], Samuel Madden[†], Matei Zaharia

Stanford University, MIT CSAIL[†], Imperial College London[‡]

Abstract

Data analytics applications combine multiple functions from different libraries and frameworks. Even when each function is optimized in isolation, the performance of the combined application can be an order of magnitude below hardware limits due to extensive data movement across these functions. To address this problem, we propose Weld, a new interface between data-intensive libraries that can optimize across disjoint libraries and functions. Weld exposes a lazily-evaluated API where diverse functions can submit their computations in a simple but general intermediate representation that captures their data-parallel structure. It then optimizes data movement across these functions and emits efficient code for diverse hardware. Weld can be integrated into existing frameworks such as Spark, TensorFlow, Pandas and NumPy without changing their user-facing APIs. We demonstrate that Weld can speed up applications using these frameworks by up to 29×.

1. Introduction

The main way users are productive writing software is by *combining* libraries and functions written by other developers. This is especially true in data analytics applications, which often need to compose many disparate algorithms into one workflow. For instance, a typical machine learning pipeline might select some data using Spark SQL [1], transform it using NumPy and Pandas [2, 3], and train a model with TensorFlow [4], taking advantage of Python's rich ecosystem of data science libraries.

Traditionally, the interface for composing these libraries has been function calls. Each library function takes pointers to in-memory data, performs a computation, and writes a result back to memory. Unfortunately, this interface is increasingly inefficient for *data-intensive* applications. The gap between memory bandwidth and processing speeds has grown steadily over time [5], so that, on modern hardware, many applications spend most of their time on *data movement* between functions. For example, even though NumPy and Pandas use optimized C functions (*e.g.*, BLAS [6]) for their operators, we find that programs that use multiple such operators can be 8× slower than handwritten code, because the function call interface requires materializing intermediate results in memory after each operation. This problem gets worse when some libraries use hardware accelerators, such as GPUs, because data movement into these accelerators is even slower [7]. In short, the core interface developers have used to compose software for the past 50 years—functions that exchange data through memory—misuses the most precious resources on modern hardware.

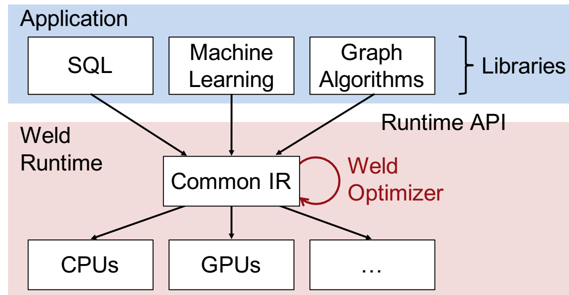


Figure 1: Weld captures diverse data-parallel workloads using a common intermediate representation (IR) to emit efficient code for the end-to-end application on diverse hardware.

This paper proposes Weld, a novel interface and runtime that can optimize *across* data-intensive libraries and functions while preserving their user-facing APIs. Weld consists of three key components (Figure 1). First, Weld asks libraries to represent their computations using a functional *intermediate representation (IR)*. This IR captures the data-parallel structure of each function to enable rich data movement optimizations such as loop fusion and tiling [8]. Libraries then submit their computations to Weld through a lazily-evaluated *runtime API* that can collect IR code from multiple functions before executing it. Finally, Weld's *optimizer* combines these IR fragments into efficient machine code for diverse parallel hardware. Weld's approach, which combines a unified IR with lazy evaluation, enables complex optimizations *across* independently written libraries for the first time.

Weld's first component is its IR. We sought to design an IR that is both highly general (to capture a wide range of data analytics computations) and amenable to complex optimizations (*e.g.*, loop fusion, loop tiling, vectorization, and execution on diverse hardware). To this end, Weld uses a small functional IR based on two concepts: nested parallel loops and an abstraction called *builders* for composing results in parallel. Builders are a hardware-independent abstraction that specify *what* result to compute (*e.g.*, a sum or a list) without giving a low-level implementation (*e.g.*, atomic instructions), allowing for different implementations on different hardware. While Weld's IR imposes some limitations, such as lack of support for asynchronous computation, we show that it is general enough to express relational, graph and machine learning workloads, and to produce code with state-of-the-art performance for these tasks.

Weld's second component is a runtime API that uses lazy evaluation to capture work across function call and library boundaries. Unlike interfaces such as OpenCL and

CUDA [9, 10], which execute work eagerly, Weld registers the work from multiple functions (even in different languages) and optimizes across them only when the program forces an evaluation (*e.g.*, before writing to disk). The API supports accessing data in the application's memory without changing its format, allowing Weld to work against the native in-memory formats of common libraries such as Spark SQL and NumPy.

Finally, Weld's optimizer performs a wide range of optimizations on its IR, including loop fusion, loop tiling, and vectorization, to combine the IR fragments from different libraries into efficient machine code. Although these optimizations are not novel, we show that they can be combined on IR fragments brought together by Weld's API to yield powerful optimizations across libraries that cannot be applied to the individual functions. For example, in an application that filters data using Spark SQL and applies a NumPy function to each row, Weld can vectorize the NumPy function across rows, or even apply loop tiling [8] across the two libraries.

We show that Weld can unlock order-of-magnitude speedups in data analytics applications, even when they use well optimized libraries. We implemented a prototype of Weld with APIs in C, Java and Python, a full backend for multicore x86 CPUs, and a partial backend for GPUs. We then integrated Weld into four common libraries: Spark SQL, NumPy, Pandas, and TensorFlow. In total, Weld can offer speedups of 3–29× in applications that use multiple libraries, and 2.5–6.5× even in applications that use multiple functions from the same library, by minimizing data movement and generating efficient machine code. Moreover, because Weld's IR is data-parallel, it can also parallelize the computations of single-threaded libraries such as NumPy and Pandas, yielding speedups of up to 180× when allowed to use more cores than the original computation with no additional programmer effort. Weld's compiler is also competitive with code generators for narrower domains, such as HyPer [11] for SQL and XLA [12] for linear algebra. Finally, porting each library to use Weld only required a few days of effort and could be done incrementally, with noticeable benefits even when just a few common operators were ported.

To summarize, the contributions of this paper are:

1. A novel interface to enable cross-library and cross-function optimizations in data-intensive workloads using (1) a general intermediate representation based on loops and builders and (2) a lazily evaluated runtime API.
2. An optimizer that combines Weld IR fragments from disjoint libraries into efficient code for multicores and GPUs.
3. An evaluation of Weld using integrations with Pandas, NumPy, TensorFlow and Spark that shows Weld can offer up to 29× speedups in existing applications.

2. System Overview

Figure 2 shows an overview of Weld. As described earlier, Weld has three components: a data-parallel IR for libraries to express work in, a lazy runtime API for submitting this work, and an optimizer that targets various hardware. These components can be integrated into existing user-facing libraries.

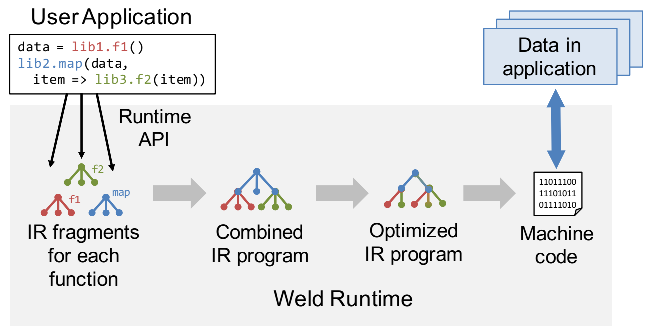


Figure 2: Overview of Weld. Weld collects fragments of IR code for each Weld-enabled library call and combines them into a single IR program. It then compiles this program to optimized code that runs on the application's in-memory data.

In practice, we expect libraries to integrate Weld in two main ways. First, many libraries, such as Pandas and NumPy, already implement their key functions in low-level languages such as OpenCL or C. Developers can port individual functions to emit Weld code instead, and then automatically benefit from Weld's cross-function optimizations. Libraries like NumPy and Pandas already have a compact in-memory representation for data (*e.g.*, NumPy arrays [2]), so Weld can work directly against their in-memory data at no extra cost. Weld's design allows functions to be ported incrementally and offers notable speedups even if only a few common operators are ported.

Second, some libraries, such as Spark SQL and TensorFlow, already implement code generation beneath a lazily evaluated API. For these libraries, Weld offers both the ability to interface efficiently with other libraries, and a simpler way to generate code. For example, much of the complexity in code generators for databases involves operator fusion logic: transforming a tree of operators into a single, imperative loop over the data [11, 13]. With Weld, each operator can emit a separate loop, independent of downstream operators; Weld will then fuse these loops into a single, efficient program.

We note that Weld focuses primarily on *data movement* optimizations for *data-parallel* operators from domains such as relational and linear algebra. These operators consume the bulk of time in many applications by causing memory traffic, and benefit from co-optimization. Domain-specific optimizations, such as reordering linear algebra expressions or ordering joins in SQL, still need to be implemented within each library (and outside of Weld). In addition, Weld supports calling existing C functions for complex non-data-parallel code that developers have already optimized. Nonetheless, we show that Weld's data movement optimizations have a significant impact.

2.1 A Motivating Example

We illustrate the benefit of Weld in a data science workflow adapted from a tutorial for Pandas and NumPy [14]. Pandas

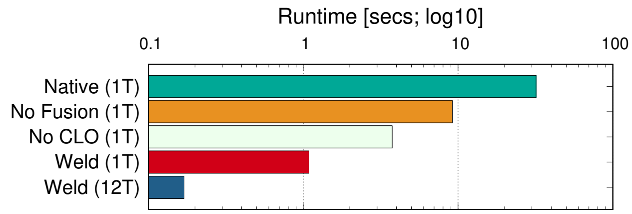


Figure 3: Performance of a data science workflow (log scale) in unmodified Pandas and NumPy (where only individual operators are written in C), Weld without loop fusion, Weld without cross-library optimization (CLO), Weld with all optimizations enabled, and Weld with 12 threads.

and NumPy are two popular Python data science libraries: Pandas provides a "dataframe" API for manipulating data in a tabular format, while NumPy provides fast linear algebra operators. Both Pandas and NumPy offer optimized operators, such as data filtering and vector addition, written in C or Cython. However, workloads that combine these operators still experience substantial overhead from materializing intermediate results.

Our workload consists of filtering large cities out of a population-by-cities dataset, evaluating a linear model using features in the dataframe to compute a crime index, and then aggregating these crime indices into a total crime index. It combines relational operators from Pandas with vector arithmetic operators from NumPy. Figure 3 shows its performance on a 6 GB dataset. Porting each operator to run on Weld yields a 3× speedup (shown in the No Fusion bar) due to Weld's more efficient, vectorizing code generator. Enabling Weld's loop fusion optimization then leads to a further 2.8× speedup *within* each library, and an additional 3.5× speedup when enabled *across* libraries. This gives Weld a total 29× speedup on a single thread, largely due to this data movement optimization (Weld 1T bar). Finally, Pandas and NumPy are single-threaded, but Weld can automatically parallelize its generated code without any change to the user application. Enabling multithreading gives a further 6.3× speedup on 12 cores, at which point Weld saturates the machine's memory bandwidth, for a total of 187× speedup versus single-core NumPy and Pandas.

2.2 Limitations

Weld's interface and implementation have several limitations. First, Weld currently only aims to accelerate single-machine code in a shared-memory environment (*e.g.*, multicore CPU or GPU). Its IR includes shared-memory operations such as random lookups into an array, which are difficult to implement efficiently in a distributed setting. Nonetheless, Weld can also be integrated into distributed systems such as Spark to accelerate each node's local computations. These distributed frameworks are often CPU- or memory-bound [1, 15, 16]. We show in § 7 that by optimizing computation on each node, Weld can accelerate Spark applications by 5-10×.

Second, Weld's IR cannot express asynchronous computations where threads race to update a result [17]; all Weld programs are race-free. It also lacks constructs to let programmers control locality (*e.g.*, pinning data to a CPU socket).

Third, Weld executes computations lazily, which makes programs harder to debug. Fortunately, lazy APIs are becoming very common in performance-sensitive systems such as LINQ [18], Spark and TensorFlow, and we believe that programmers can use similar debugging techniques (*e.g.*, printing an intermediate result).

Finally, Weld requires integration into libraries in order to speed up applications. As discussed at the beginning of §2, we believe that this is worthwhile for several reasons. First, many libraries already write their performance-sensitive operators in C or OpenCL; Weld offers a higher-level (functional) interface to write code that is more hardware-independent. Second, Weld enables significant speedups *even within a single library*, creating incentives for individual libraries to use it. Finally, Weld can be integrated incrementally and still offer substantial speedups, as we show in §7.3.

3. Weld's Intermediate Representation

Libraries communicate the computations they perform to Weld using a data-parallel intermediate representation (IR). This component of the Weld interface determines both which workloads can run on Weld and which optimizations can easily be performed. To support a wide range of data-intensive workloads, we designed Weld's IR to meet three goals:

1. Generality: we wanted an IR that could express diverse data analytics tasks (*e.g.*, relational and linear algebra), as well as *composition* of these tasks into larger programs.
2. Ability to support optimizations, especially for data movement optimizations such as loop fusion and loop tiling.
3. Parallelism: we wanted the IR to be explicitly parallel so that Weld can automatically generate code for modern parallel hardware, *e.g.*, multicores and GPUs.

To meet these goals, we designed a small IR inspired by monad comprehensions [19], similar to functional languages but operating at a lower level that makes it easier to express fusion optimizations.

3.1 Data Model

Weld's basic data types are scalars (*e.g.*, `int` and `float`), variable-length vectors (denoted `vec[T]` for a type `T`), structures (denoted `{T1,T2,...}`), and dictionaries (`dict[K,V]`). These types are nestable to support more complex data. We chose these types because they appear commonly in data-intensive applications and in low-level data processing code (*e.g.*, dictionaries are useful to implement database joins).

3.2 Computations

Weld's IR is a functional, expression-oriented language. It contains basic operators for arithmetic, assigning names to values, sequential `while` loops, and collection lookups. It also supports calling external functions in C.

Builder Types	
`vecbuilder[T]`	Builds a `vec[T]` by appending merged values of type `T`
`merger[T,func,id]`	Builds a value of type `T` by merging values using a commutative function `func` and an identity value `id`
`dictmerger[K,V,func]`	Builds a `dict[K,V]` by merging `{K,V}` pairs using a commutative function
`vecmerger[T,func]`	Builds a `vec[T]` by merging `{index,T}` elements into specific cells in the vector using a commutative function
`groupbuilder[K,V]`	Builds a `dict[K,vec[V]]` from values of type `{K,V}` by grouping them by key

Table 1: Builder types available in Weld.

In addition, the IR has two parallel constructs: a parallel `for` loop and a hardware-independent abstraction for constructing results in parallel called a *builder*. Parallel loops can be nested arbitrarily, which allows complex function composition. Each loop can merge values into multiple builders; for example, a single loop can merge values into one builder to produce a sum and another to produce a list.

Weld includes multiple types of builders, shown in Table 1. For example, a `vecbuilder[T]` takes values of type `T` and builds a vector of 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.

Builders support three basic operations. `merge(b, v)` adds a new value `v` into the builder `b` and returns a new builder to represent the result.[1] Merges into builders are associative, enabling them to run in parallel. `result(builder)` destroys the builder and returns its final result: no further operations are allowed on it after this. Finally, Weld's parallel `for` loop is also an operator that consumes and returns builders. `for(vectors, builders, func)` applies a function of type `(builders, index, elem)=> builders` to each element of one or more vectors in parallel, then returns the updated builders. Each call to `func` receives the index of the corresponding element and a struct with the values from each vector. The loop can also optionally take a start index, end index and stride for each input vector to express more complex strided access patterns over multidimensional arrays (*e.g.*, matrices).

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

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

// Loop over two vectors and merge results only
// on some iterations.
v0 := [1, 2, 3];
v1 := [4, 5, 6];
result(
  for({v0, v1},
     vecbuilder[int],
     (b,i,x) => if(x.0 > 1) merge(b, x.0+x.1) else b
)) // returns [7, 9]
```

Listing 1: Examples of using builders.

Weld's `for` loops can be nested arbitrarily, enabling Weld to express irregular parallelism (*e.g.*, graph algorithms) where different iterations of the inner loop do different amounts of work.

Finally, Weld places two restrictions on the use of builders for efficiency. 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. Formally, builders are a linear type [20]. Second, functions passed to `for` must return builders derived from their arguments. These restrictions let Weld's compiler safely implement builders using mutable state.

3.3 Higher-Level Operators

To aid developers, Weld also contains macros that implement common functional operators such as `map`, `filter` and `reduce` using builders, so that library developers familiar with functional APIs can concisely express their computations. These operators are all map into loops and builders. For example, the second snippet in Listing 1 implements a `map` over a vector.

3.4 Why Loops and Builders?

Given the broad use of functional APIs such as MapReduce and Spark, a strawman design for an IR might have used these functional operators as the core constructs rather than loops and builders. Unfortunately, this design prevents expressing many common optimizations in the IR. For example, consider Listing 2, which runs two operations on the same input vector:

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

Listing 2: A `map` and `reduce` over the same input vector.

Even though these `map` and `reduce` operations can be computed in a shared pass over the data, no operator akin to `mapAndReduce` exists that computes both values in one pass. Richer optimizations such as loop tiling are even harder to express in a high-level functional IR. By exposing a single loop construct that can update multiple builders, patterns like the above can easily be fused into programs such as Listing 3.

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

Listing 3: `for` operating over multiple builders to produce both a vector and an aggregate in one pass.

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

API Summary	
`NewWeldObject(data, type, encoder)`	Creates a `WeldObject` wrapping the given in-memory `data` and giving it Weld type `type`. The `encoder` object implements marshaling (§4.2).
`NewWeldObject(deps, expr, encoder)`	Creates a `WeldObject` with the given dependencies (other `WeldObjects`), a Weld IR expression, and an `encoder` for the resulting type.
`GetObjectType(o)`	Returns the Weld type of the `WeldObject` `o` (*e.g.*, `vec[int]`).
`Evaluate(o)`	Evaluates the Weld program and returns a result.
`FreeWeldObject(o)`	Frees memory for this `WeldObject`.
`FreeWeldResult(v)`	Free a `WeldResult`.

Table 2: A summary of the Weld API.

3.5 Generality of the IR

Weld's parallel loop and builders can express all of the functional operators in systems such as MapReduce, LINQ and Spark, as well as relational algebra, linear algebra and other data-parallel operations. Given the wide range of algorithms implemented over these APIs [21, 22], we believe Weld can benefit many important workloads. Our evaluation shows workloads from each of these domains. As discussed in § 2.2, the current IR does not capture asynchrony.

4. Runtime API

The second component of Weld is its runtime API. Unlike interfaces like OpenCL and CUDA, Weld uses a *lazy* API to construct a computation graph. Library functions use the API to compose fragments of IR code (perhaps across libraries) and provide access to data in the application. The API uses the IR fragments and data to build a DAG of computations. When libraries evaluate a computation, Weld fuses the graph into a single IR program, optimizes it and and executes it. We show examples of the API in Python here, it also supports Java and C.

Consider the program in Listing 4 as a motivating example, which uses the `itertools` library's `map` function to iterate over a set of vectors and apply `numpy.dot` to each one:

```
scores = itertools.map(vecs, lambda v: numpy.dot(v, x))
print scores
```

Listing 4: A Python program that combines library calls.

A standard call to `itertools.map` would treat `numpy.dot` as a black box and call it on each row. If both libraries use Weld, however, the result `scores` is instead an object encapsulating a Weld program capturing *both* functions. Weld evaluates this object only when the user calls `print`. Before evaluation, Weld will optimize the IR code for the entire workflow, enabling optimizations that would not make sense in either function on its own. For example, Weld can tile the loop to reuse blocks of the `x` vector across multiple rows of `v` for cache efficiency.

4.1 API Overview

Developers integrate Weld into their libraries using an interface called `WeldObject`, which represents either a lazily evaluated sub-computation or external data in the application. A `WeldObject` may depend on other `WeldObjects` (possibly from other libraries), forming a DAG for the whole program where leaves are external data. Table 2 summarizes Weld's API.

Developers create `WeldObjects` using the `NewWeldObject` call. This call has two variants: one to encapsulate external data dependencies in the application and one to encapsulate sub-computations and dependencies *among* `WeldObjects`. To encapsulate an external data dependency, developers pass as arguments a pointer to the data dependency, the Weld type of the dependency (*e.g.*, `vec[int]`), and an *encoder* for marshaling between native library formats and Weld-compatible objects. §4.2 discusses encoders in detail.

To encapsulate sub-computations and dependencies with other `WeldObjects`, developers pass a list of dependent `WeldObject` s (`deps`), a Weld IR expression representing the computation, and an encoder for the expected type of the `WeldObject`. If the library evaluates the `WeldObject`, this encoder is used to marshal data returned by Weld to a native-library format. The provided IR expression must depend only on the dependencies declared in `deps`. Listing 5 shows an example function for computing the square of a number using Weld's API.

```
def square(self, arg):
  # Programatically construct an IR expression.
  expr = weld.Multiply(arg, arg)
  return NewWeldObject([arg], expr)
```

Listing 5: A simple function that squares an argument, `arg`, passed in as a `WeldObject`.

The `Evaluate` API call evaluates a `WeldObject` instance and returns a result. Libraries can choose when to evaluate an object in several ways. In our integrations with Python libraries, we used methods that save or print the object (*e.g.*, the `__str__` method to convert it to a string) as evaluation points to introduce lazy evaluation behind the library's existing API. Systems like Spark and TensorFlow already have lazy APIs with well-defined evaluation points. `Evaluate` returns a special handle called `WeldResult` that can be checked for failure or queried for a pointer to the returned data.

Library developers manage the lifecycle of a `WeldObject` manually after instantiation. The `FreeWeldObject` call deletes an instance of `WeldObject` by freeing its internal state; this call *does not* free data dependencies or child `WeldObject` instances in other libraries. In languages with automatic memory management, such as Python, developers can add the `FreeWeldObject` call in their class's destructor.

4.2 Marshalling Data

Weld specifies a standard binary format for its basic data types that allows it to operate over existing in-memory data. Specifically, scalar types (`int`, `float`, *etc.*) and structs follow C packed structure layout, and vectors `vec[T]` are represented as

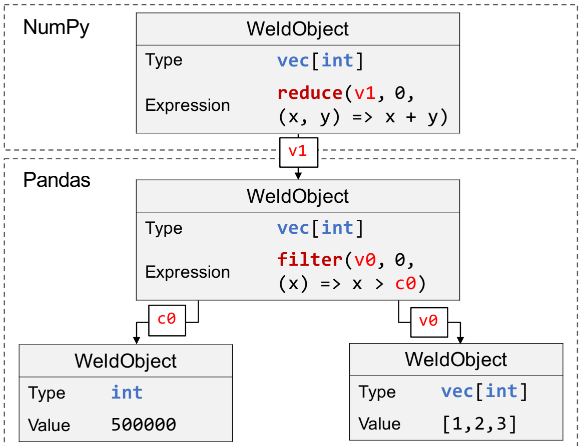


Figure 4: The example from Listing 7 as a computation graph.

an `{int64,T*}` structure. Dictionaries and builders cannot be passed into Weld from outside in our current implementation.

Library developers provide *encoders* to map types in their native programming languages (*e.g.*, Python) to Weld-usable data and vice-versa. The encoder interface is a single function `encode`. When used with `NewWeldObject` to declare a data dependency, this function maps an object in the library's native language and returns data understood by Weld's runtime. For example, an encoder for NumPy arrays would take in a NumPy object and extract a pointer to its internal data array, which already happens to be a packed array of primitive types in NumPy [2]. When used with `NewWeldObject` to declare a sub-computation, the function takes a pointer to data in Weld's format and returns an object in a library format.

4.3 Memory Management

Values in Weld are considered either *owned by library* or *owned by Weld*. Input data is always owned by libraries. These data are neither mutated nor freed by Weld. Values that Weld allocates during execution are owned by Weld, as are results of `Evaluate` (`WeldResults`). Decoders can either point to these values directly when wrapping them into library-specific objects (as long as the library does not free the corresponding `WeldResult`), or they can copy data.

Weld can also allocate memory during the execution of an IR program (*i.e.*, temporary values that are not part of the final result). The runtime frees this memory after each call to `Evaluate`, preserving only the objects needed for the resulting `WeldResult`. The `Evaluate` function takes a memory limit as an argument to prevent unbounded allocation, which is useful when integrating Weld into a larger engine such as Spark SQL.

4.4 User Defined Functions (UDFs)

Libraries such as Spark and `itertools` can take functions as arguments. To implement these, a `WeldObject`'s IR expression can also represent a function, with dependencies pointing to variables in its closure. In order to make it easy to pass functions into Weld, we implemented a basic UDF translator for Python that walks Python abstract syntax trees (ASTs) to convert them to Weld IR, based on techniques in existing systems [16, 23, 24]. Listing 6 shows an example; the `@weld` annotation provides a type signature for the function in Weld.

```
# Produces (x: int) => x + 1 in Weld IR
@weld("(int) => int")
def increment(x): return x + 1
```

Listing 6: Python UDF with a Weld type annotation.

4.5 Example: Combining NumPy and Pandas

Weld's API enables optimizations even across independent libraries. We illustrate this with the function in Listing 7, which uses the Python Pandas library and NumPy to compute the total population of all cities with over 500,000 residents. NumPy represents data in C arrays wrapped in Python objects. Pandas uses data frames [3], which are tables with named columns that are also encoded as NumPy arrays.

```
def large_cities_population(data):
  filtered = data[data["population"] > 500000]
  sum = numpy.sum(filtered)
  print sum
```

Listing 7: A sample Python program using Pandas and NumPy.

In the native Pandas and NumPy libraries, this code causes two data scans: one to filter out values greater than 500,000 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." The loop also benefits from optimizations such as vectorization.

To enable using Weld for this program, we must extend the `DataFrame` object in Pandas to return lazily evaluated `WeldObjects`. We must also provide a IR fragment for the `>` operator on `DataFrame` columns and the `numpy.sum` function.

Listing 8 shows the Weld expressions for implementations of each of these functions. Each implementation takes either a data dependency (*e.g.*, a NumPy array) or `WeldObject` as input, and incrementally builds a Weld computation graph by returning another composable `WeldObject` instance.

```
# DataFrame > filter
# Input: Vector v0, constant c0
filter(v0, (x) => x > c0)

# numpy.sum
# Input: Vector v0
reduce(v0, 0, (x, y) => x + y)
```

Listing 8: Pandas and NumPy functions using Weld.

After porting these operators to Weld, the libraries can use the API to lazily compose a computation graph for the full workload without any modifications to the user's code. Listing 9 shows the final fused Weld expression of the variable `sum` (now a `WeldObject`) before it is printed. Calling `print` on this instance invokes the `evaluate` method to compute a result. Figure 4 shows the computation graph for the workload.

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

Listing 9: The combined Weld program.

Optimizations Passes	
Loop Fusion	Fuses adjacent loops to avoid materializing intermediate results when the output of one loop is used as the input of another. Also fuses multiple passes over the same vector.
Size Analysis	Infers the size of output vectors statically.
Loop Tiling	Breaks nested loops into blocks to exploit caches by reusing values faster [8].
Vectorization & Predication	Transforms loops with simple inner bodies to use vector instructions. Branches inside the loop body are transformed into unconditional select instructions (predication).
Common Subexpression Elimination	Transforms the program to not run the same computation multiple times.

Table 3: Optimization passes implemented in Weld.

After optimization, this function becomes a single parallel loop using builders, as shown in Listing 10:

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

Listing 10: The Weld program after loop fusion.

The optimizer then applies optimizations such as predication and vectorization here; for clarity, we omit these above.

5. Optimizer and Hardware Backends

Weld's optimizer combines IR fragments composed by different libraries through the runtime API into efficient machine code for parallel hardware. The optimizer starts by applying some general optimization rules in the IR (*i.e.*, outputting new code in the same IR). It then passes the code to hardware-specific backends. Although Weld uses standard compiler optimizations, our contribution is to show that its compilation strategy produces efficient code for libraries combined using Weld's runtime API, even when these libraries implement their functions using Weld's IR in isolation.

IR-level Optimizations. We modeled our optimizer after LLVM [25], which applies a wide range of hardware-independent optimizations at the IR level and then passes a largely optimized program to a hardware backend. This approach is powerful because it allows composing passes at the IR level in arbitrary ways, and it fits well with Weld's loop-and-builder based IR (§3), which can express fused and tiled code within the same IR as the original program.

We implemented several different optimization passes at the IR level, shown in Table 3. These passes are implemented using pattern-matching rules on sub-trees of the abstract syntax tree (AST). Weld applies passes in a static order, with rules at each level applied repeatedly until the AST no longer changes. Specifically, we apply the loop fusion transformations first, then size analysis, then loop tiling, then vectorization and finally common subexpression elimination.

Weld's functional IR makes standard optimizations significantly easier to apply than to C or LLVM. For example, common optimizations such as vectorization are hard to apply in C or LLVM because of pointer aliasing (determining whether two pointers could refer to overlapping addresses). In contrast, Weld's immutable data values and "write-then-read" builders (which have a separate "write-only" phase followed by a read-only phase after computing the result) make it straightforward to transform sub-trees of the AST in isolation. As we show in §7, Weld's optimization passes produce efficient code across a variety of domains, even when starting with disparate IR fragments combined at runtime using Weld's API.

Multicore x86 Backend. Weld's CPU backend emits explicitly multithreaded and vectorized code for each program. We use LLVM [25] to generate code, and vectorize the code explicitly in our compiler to target AVX2 [26] instructions. The backend could in principle run on non-x86 architectures supported by LLVM, but we have only evaluated it on x86.

At runtime, the generated code links with a multicore work-stealing execution engine inspired by Cilk [27], which schedules work for each parallel `for` loop dynamically. This allows Weld to support irregular parallelism. In code with nested loops, new tasks are created by splitting up the outermost loop that still has more than one iteration remaining. This policy ensures that expensive outer loops will be split across cores, but smaller inner loops can often stay on the same core without incurring task creation overhead.

We implemented the data structures within the backend, including builders, using standard multithreaded programming techniques such as cache line padding to reduce false sharing. Multicore builders, including the `merger`, are largely implemented with per-core copies that are aggregated when `result` is called. Other builder implementations are possible and potentially faster in certain cases, as described in §7.7.

Partial GPU Backend. Our GPU backend is built on top of OpenCL and supports the `merger`, `vecmerger` and `vecbuilder` builders. Weld's optimization passes fuse computations into a single OpenCL kernel that we then submit to the GPU.

In order to obtain reasonable performance on the GPU, we had to implement one more IR transformation rule specific to the GPU backend. Since GPUs cannot easily support dynamic parallelism at runtime, we use un-nesting [28] to transform a nested parallel program into a regular one. In addition, our implementations of builders on the GPU are different. For example, we use a parallel aggregation tree to combine intermediate results for `merger` and `vecmerger` efficiently across thousands of GPU cores.

Currently, programs in Weld must either execute *completely* on the CPU or on the GPU. We do not yet support partial offloading of computation to a coprocessor.

6. Library Integrations

To evaluate Weld, we integrated it into four popular libraries: Spark SQL, TensorFlow, NumPy, and Pandas. Each integra-

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 4: Number of lines of code in our library integrations.

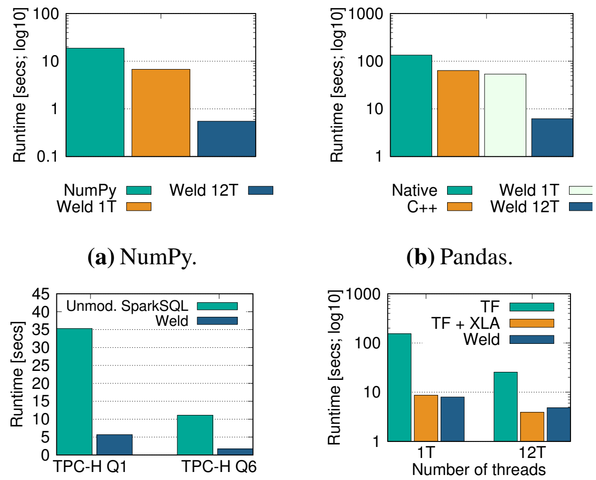
tion required some up front "glue code" for marshalling data and enabling lazy evaluation (if the library was eagerly evaluated), as well as code for each ported operator. Overall, we found the integration effort to be modest across the board, as shown in Table 4. Each library required 100–900 lines of glue code and an additional 5–85 lines of code per operator; operators can be added incrementally and. interoperate with native operators in each library.

Spark SQL. Weld's integration with Spark SQL [1] accelerates its local computations on each node. If performance is bounded by local resources [15], Weld accelerates queries even in a distributed setting. Spark SQL already has a lazy API to build an operator graph, and already performs Java code generation using a similar technique to HyPer [11], so porting this framework was straightforward: we only needed to replace the emitted Java bytecode with Weld IR via Weld's API. Spark SQL's existing Java code generator uses complex logic [13] to directly generate imperative loops for multiple chained operators because the Java compiler cannot perform these optimizations automatically. In contrast, our Weld port emits a separate IR fragment for each operator without considering its context, and Weld automatically fuses these loops.

TensorFlow. Like Spark SQL, TensorFlow [4] also has a lazily evaluated API that generates a data flow graph composed of modular operators. Our integration with TensorFlow required two components: *(i)* a user-defined `WeldOp` operator that runs an arbitrary Weld expression, and *(ii)* a graph transformer that replaces a subgraph of the TensorFlow data flow graph with an equivalent `WeldOp` node. Before execution, the transformer searches the original data flow graph for subgraphs containing only operators that are understood by our port, and replaces each such subgraph with a `WeldOp` node for their combined expression, relying on Weld to fuse these expressions. Our integration leverages TensorFlow's support for user-defined operators and graph rewriting and makes no changes to the core TensorFlow engine.

NumPy and Pandas. Our integrations with NumPy and Pandas required more effort because these libraries' APIs are *eagerly* evaluated. To enable lazy evaluation in NumPy, we built a wrapper for its array data type, `ndarray`, which contains a `WeldObject`. We also built a Weld encoder to wrap the pointer to the data buffer in the `ndarray` structure using the Weld vector type.[2] We overwrote the functions that print arrays or extract elements from them to force evaluation. Finally, all of our

[2] NumPy's internal data format is already a packed array of primitive types.



(a) NumPy. **(b)** Pandas.

(c) Spark SQL. **(d)** TensorFlow.

Figure 5: Workloads using individual libraries.

ported operators accept either a standard `ndarray` or our wrapper for their input arguments, and return a `WeldObject` wrapper with those arguments as dependencies (§4.1).

We ported Pandas in a similar way, by creating a wrapper object around dataframes. We ported Pandas' filtering, sorting, predicate masking, aggregation, per-element string slicing and `getUniqueElements` functions to Weld. Internally, Pandas represents dataframe columns as NumPy arrays, so we could use the same encoder as in NumPy and simply pass a pointer to these raw primitive arrays to Weld.

7. Evaluation

To evaluate Weld, we sought to answer the following questions: (1) How much does Weld speed up data analytics workloads end-to-end, both within and across libraries? (2) Does Weld exhibit performance benefits when deployed incrementally? (3) Can Weld's generated code match the performance of existing optimized systems?

Experimental Setup. Unless otherwise noted, we ran tests on a server with an Intel Xeon E5-2680v3 CPU with 12 cores (24 hyperthreads) based on the Haswell micro-architecture and 128 GB of memory. Our C++ baselines are compiled using LLVM 3.5 (with `O3` and `LTO`). Results average over five runs.

7.1 Accelerating Existing Frameworks

Weld accelerates programs using individual libraries by applying optimizations across calls to the library, fusing operators, and generating vectorized, multithreaded machine code. We benchmark Weld integrations with the four frameworks from §6: Spark 2.0, TensorFlow 0.12 (without the XLA compiler), TensorFlow 1.0 (with XLA), NumPy 1.8.2, and Pandas 0.19.1.

NumPy. We evaluate the NumPy Weld integration with a Black Scholes implementation on a set of 100 million records. Black Scholes is a computationally expensive data parallel workload with mathematical operators such as square roots, exponents, and logarithms. Figure 5a shows the results. Weld emits vectorized exponent and logarithm functions using AVX2, while NumPy does not vectorize these calls. This leads to a 3× speedup over the native NumPy implementation and a 33× improvement on 12 cores. §7.6 breaks down the speedups in this workload.

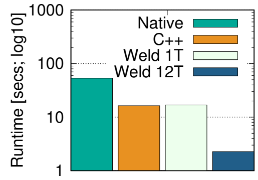
Pandas. We evaluate our Pandas integration on a data science workload from the Pandas Cookbook [14], using a 6GB dataset of zipcodes. The workload uses Pandas to clean the dataset by first "slicing" the zipcodes to represent each one with five digits, removing all nonexistent zipcodes, and filtering duplicate zipcodes. The Pandas library we compare against implements operators in either C or Cython already.

Figure 5b shows the results. Weld fuses each operator in Pandas; in this data-intensive workload, materialization and redundant traversals over the data prove expensive. This renders loop fusion useful, leading to a 2.5× speedup over native Pandas. Weld also facilitates multithreading and transparently parallelizes this workload (Pandas is a single-threaded framework) without any application changes, enabling a 21.6× improvement over native Pandas on 12 cores.

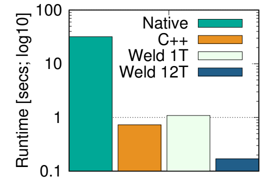
Spark SQL. To illustrate Weld's benefits in a distributed framework, we evaluate Spark SQL's Weld integration on TPC-H queries 1 and 6 with 20 Amazon EC2 r3.xlarge worker instances (2 cores, 30 GB memory) and 800GB of TPC-H data (scale factor 800). Data was read from the Spark SQL in-memory cache. As shown in Figure 5c, Weld provides a 6.2× speedup for TPC-H Q1 and 6.5× for Q6. Weld's performance improvement comes largely from its ability to generate low-level, vectorized machine code; Spark natively generates Java bytecode, which the Java JIT compiler cannot vectorize.

TensorFlow. We evaluate TensorFlow by training a binary logistic regression classifier using the MNIST dataset [29] to recognize a digit as zero or nonzero. We test the performance of the default TensorFlow implementation against a Weld-enabled TensorFlow. We use two different versions of TensorFlow: one with the XLA compiler [12] and one without. XLA is a JIT compiler which converts TensorFlow computations into machine code. The TensorFlow developers introduced XLA as an optional feature in its 1.0 release after observing that data movement across TensorFlow operators was a dominant cost [12]; XLA performs many of the optimizations Weld does within TensorFlow (*e.g.*, loop fusion).

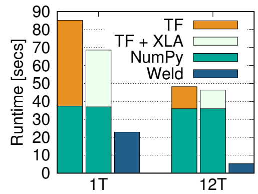
Figure 5d shows the results. On a single core, Weld's lazy evaluation allows IR fragments from each operator to be fused and optimized, demonstrating a 19.3× speedup over TensorFlow without XLA. This speedup comes from loop fusion across operators enabled by Weld's lazily evaluated runtime. Weld accelerates the computation by 1.09× over TensorFlow with XLA enabled. Despite Weld's greater generality, its performance on this workload matches XLA, which is specialized for TensorFlow's linear algebra routines. Unlike XLA, Weld's integration with TensorFlow additionally enables optimization across other libraries as well, as shown in §7.2.



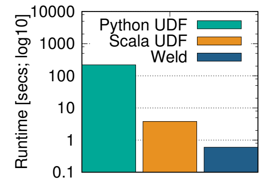
(a) Pandas + NumPy, App. 1.



(b) Pandas + NumPy, App. 2.



(c) NumPy + TensorFlow.



(d) Spark SQL UDF.

Figure 6: Workloads which *combine* libraries.

With 12 threads, the speedup over TensorFlow without XLA reduces to 5.3×, since the Weld version becomes bound on memory access; the overhead associated with TensorFlow's runtime also becomes sizable. Weld is within 20% of TensorFlow with XLA on 12 threads. The accuracy on a held-out validation set is unchanged for all of our implementations.

7.2 Optimizing Across Libraries

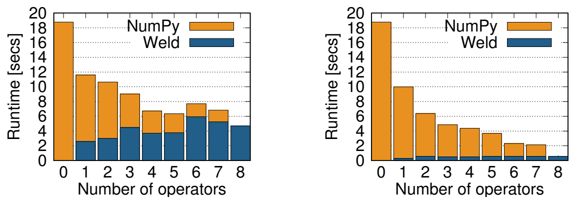
Weld also optimizes workflows which call into multiple libraries in a single application. We evaluate our system on four representatitive workloads that combine libraries.

NumPy + Pandas. To demonstrate the benefits of cross library optimization, we evaluate Weld on workloads that combine NumPy and Pandas, adapted from the Pandas Cookbook [14].

In the first workload [14], we filter small cities in a population-by-cities dataframe, evaluate a softmax regression model based on features in the dataframe, and aggregate the resulting crime index predictions on a per-state basis. In the second workload, we perform the same filtering operation then evaluate a linear model based on the same feature set, and finally compute an average crime index across all cities.

Figures 6a and 6b show the results. With Weld integration, the API collects IR fragments *across* library boundaries, enabling the NumPy calls to be co-optimized with calls into Pandas. We observe speedups of up to 3.1× and 29× on a single core, and speedups of 23× and 187× across 12 cores. These optimizations come from cross-library loop fusion, vectorization of operators across libraries (*e.g.*, using predication instructions from the Pandas call with arithmetic vector instructions generated from the NumPy library call), and other standard whole-program optimizations such as inlining. §7.6 breaks down the speedups seen in the second workload in more detail.

NumPy + TensorFlow. We applied Weld to a NumPy and TensorFlow image processing workflow. NumPy whitens images from MNIST [30] (a standard preprocessing task for



(a) Black Scholes 1T. **(b)** Black Scholes 12T.

Figure 7: Incremental Integration with one and 12 threads.

image scoring) and TensorFlow scores them using a standard logistic regression model trained to recognize digits.

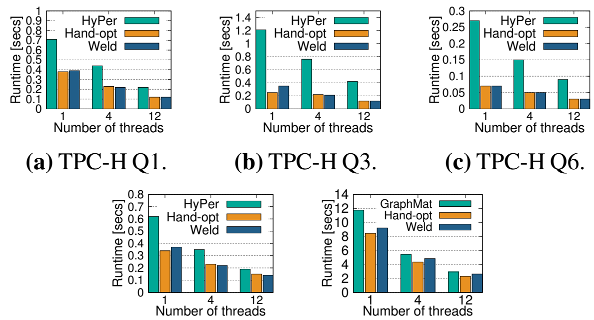
Figure 6c shows the results. With Weld integration we observe a 3× performance improvement over NumPy and TensorFlow with XLA, on a single thread. This performance improvement increases to 8.9× over the native library implementations with 12 threads. Weld provides a speedup despite TensorFlow's specialized XLA compiler optimizing the scoring computation because it co-optimizes the image whitening task with the model scoring task. With 12 cores, the speedup is due to parallelizing the whitening computation; in the native library implementation of the workload with 12 cores, TensorFlow parallelizes the model scoring but NumPy continues to run on a single thread. Performance improvements over NumPy and TensorFlow without XLA were slightly better.

Spark SQL UDF. We also demonstrate the benefits of operator composition by comparing a Weld-enabled UDF with Scala and Python UDFs in Spark SQL. The query is similar to one of our Pandas workloads: it evaluates a linear model implemented by the UDF on each row of a 2.4GB dataset and computes the average result. The Weld-enabled version of the UDF was written in Python and translated to Weld automatically using the library described in § 4.4.

Figure 6d shows the results. Calling the Scala and Python UDFs for each row is expensive due to data marshaling between Spark SQL and these languages. In contrast, the Weld UDF is co-optimized with the rest of the query (implemented using Weld-enabled Spark SQL operators) and the program is vectorized, producing a 6× speedup over the Scala UDF and a 360× speedup over the Python UDF. We note here that despite the performance cost of UDFs, in practice many analytics jobs use them [1, 31]. This indicates that developers seem unwilling to invest time in crafting fast parallel code in these use cases which increases the value of a runtime that transparently accelerates workflows.

7.3 Incremental Integration

To show that Weld is incrementally deployable, we ran the Black Scholes workload from § 7.2 on a single thread and incrementally implemented one operator at a time using Weld's interface. Operators which were not Weld-enabled used native NumPy. The Black Scholes workload uses eight unique operators; we measured which operators were the most expensive in terms of total number of CPU cycles and ported them from most to least computationally expensive.



(a) TPC-H Q1. **(b)** TPC-H Q3. **(c)** TPC-H Q6.

(d) TPC-H Q12. **(e)** PageRank.

Figure 8: CPU microbenchmarks on TPC-H and PageRank

Figure 7 shows the results. On a single core, implementing the first operator in Weld (the erf function, which evaluates an exponent and a closed integral) gives a 1.6× speedup, and implementing half the operators gives a 2.7× speedup over native NumPy, by providing vectorized implementations for functions which NumPy runs sequentially. Implementing more operators shows further (yet diminishing) speedups. With 12 threads, the single threaded NumPy operators prove to be a bottleneck, a consequence of Amdahl's Law. Implementing half the operators in Weld gives a modest 4× speedup, implementing all but one operator in Weld gives a 9× speedup, and implementing all the operators in Weld gives a 33× speedup since the entire workflow can be parallelized.

7.4 CPU Backend Performance

We now evaluate Weld's code generation on x86 CPUs by comparing the performance of a handful of data processing workloads against several state-of-the-art, domain-specific compilers and systems.

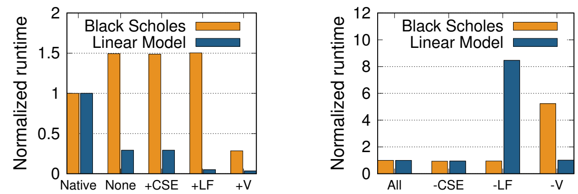
TPC-H queries. Figure 8 show the results for the TPC-H queries (scale factor 10), compared against the HyPer [11] database and a hand-optimized C baseline using Intel's AVX2 intrinsics for vectorization and OpenMP [32] for parallelization. HyPer generates LLVM IR for SQL queries, which is then compiled to machine code before execution. Weld uses the same query plan as HyPer; query plans were obtained from [33]. Execution time is competitive with HyPer across the board. Weld outperforms HyPer on Q6 and Q12 because it applies predication in the IR and generates explicitly vectorized LLVM IR. HyPer depends on LLVM for vectorization, which does not automatically perform predication.

Linear Algebra. Figure 5d shows a comparison of Weld against TensorFlow's specialized XLA compiler. As we previously reported, Weld's generated code matches the performance of the XLA's generated code, despite Weld optimizing a more general IR (XLA is specialized for TensorFlow's linear algebra operators).

PageRank. Figure 8e shows results for a PageRank implementation written in the Weld IR, compared against the GraphMat [34] graph processing framework. GraphMat had the

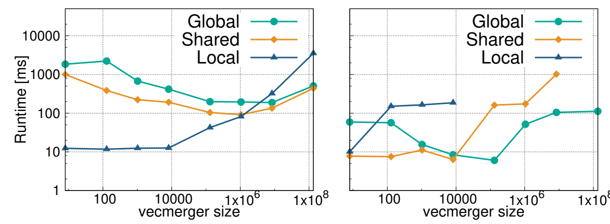
(a) TPC-H. **(b)** NN. **(c)** PageRank.

Figure 9: GPU microbenchmarks on TPC-H, PageRank, and nearest neighbor classification.



(a) Adding Optimizations **(b)** Removing Optimizations

Figure 10: Effects of individual optimizations on Black Scholes and the Pandas linear model workload. CSE, LF, and V correspond to Common Subexpression Elimination, Loop Fusion, and Vectorization respectively.



(a) Xeon E5-2680. **(b)** GeForce Titan X.

Figure 11: Builder implementations on heterogeneous hardware. Missing data points indicate that the strategy ran out of memory or did not finish.

fastest multicore PageRank implementation we found. Weld's per-iteration runtime for both serial and parallel code outperforms GraphMat, and is competitive with a hand-optimized C baseline which uses Cilk [27] for multi-threading.

In all cases, Weld's runtime memory usage also matched the memory usage of our C implementations. In summary, Weld produces machine code competitive with existing systems on a variety of workloads, demonstrating both the expressiveness of the IR and the effectiveness of our optimizer.

7.5 GPU Backend Performance

Weld's IR is designed to support diverse hardware platforms. We ran a set of Weld programs on a prototype GPU backend to illustrate that the abstractions in the IR are sufficient to enable fast code generation across platforms. We compare our GPU results to other existing systems which also target GPU execution using an Nvidia GeForce GTX Titan X with 12 GB of internal DDR5 memory running Nvidia's OpenCL implementation (version 367.57, CUDA 7.5).

TPC-H queries. Figure 9a shows the results of running the same four TPC-H queries from §7.4 against Ocelot [35], a database optimized for GPUs. We find that for most queries, Weld generates OpenCL code that generally outperforms the CPU backend and is on par with Ocelot, apart from Q1 where it is within a factor of 2 of Ocelot.

Nearest Neighbor Classification. Figure 9b shows the results of running the same nearest neighbor classifier as before, using TensorFlow (with its GPU backend enabled) with and without Weld integration. Weld's generated code outperforms TensorFlow's handwritten CUDA operator by $2.2\times$.

PageRank. The PageRank implementation is based on the un-nesting strategy described in § 5 without any dynamic load balancing. Nevertheless, the GPU-based implementation achieves a roughly $5\times$ speedup over the CPU-based implementation on account of the high degree of data parallelism that can be exploited by the GPU. For reference, Figure 9c includes the per-iteration performance of the LightSpMV implementation of PageRank [36] which is, to the best of our knowledge, the fastest implementation on the compressed sparse row representation of the graph. LightSpMV, however, implements dynamic load balancing instead of static partitioning which comes with a $4\times$ runtime overhead.

7.6 Effects of Individual Optimizations

We evaluate the effect individual optimizations have on two tasks introduced earlier in the evaluation section: the Black Scholes workload that uses NumPy and the linear model workload that uses Pandas and NumPy. For each experiment, we show the effect of incrementally adding optimizations in Weld and the effect of removing single optimizations from the entire optimization suite. Figure 10 shows the results obtained by running both workloads on a single thread.

The Black Scholes and Pandas workloads differ in that the Black Scholes workload is *compute intensive* and does not benefit from data movement optimizations. This is evident in the results; the only transformation which has a substantial impact is vectorization, which makes more effective use of the execution units on a single core. In contrast, the Pandas workload is *data intensive*, with only a few cycles of computation per byte; most time goes into performing scans over memory. Loop fusion provides the greatest gain here by transforming the entire computation to take a single traversal over the data and preventing intermediate results from being written back to memory.

7.7 Builder Abstraction

As described in §5, builders in Weld are implemented differently on CPUs and GPUs. In this section we further motivate why builders are a powerful abstraction by considering several potential implementation strategies for the `vecmerger` builder on the CPU and GPU. The `vecmerger` takes an input vector and allows merges of new values into arbitrary positions in the vector. We use the `vecmerger` to implement a Weld program that counts the number of occurrences of each key in a list.

We tested three different implementation strategies for the `vecmerger` on each platform – thread-local copies combined with a final aggregation step (local), thread block- or NUMA node-local copies updated with atomic instructions and combined with a smaller final aggregation (shared), and a single global copy updated with atomic instructions (global). Each strategy ran with all available threads on the platform. All runs merged 10^8 32-bit integer keys.

Figure 11 shows that the size at which each `vecmerger` strategy becomes optimal varies widely across the two platforms. Even if a developer wrote code in a generic hardware-independent language, such as OpenCL [9], to choose the strategy dynamically, this code would not be portable across platforms due to different transition points. In contrast, a runtime based on the builder abstraction, such as Weld, can generate completely different code for each platform to optimize the `vecmerger` on it. (Note that we have not yet implemented all the variants here in our prototype.)

7.8 Compilation Times

Weld's compile times (including optimization of the IR and code generation via LLVM) ranged from 62 ms to 257 ms (mean 126 ms and median 117 ms) across all experiments. Since we expect most analytics workloads on large datasets to run for several seconds, we believe that these times are acceptable for Weld's target applications.

8. Related Work

Weld builds on ideas in multiple fields, including compilers, parallel programming models and database engines. Unlike existing systems, however, Weld aims to design an interface that can be used to optimize across *diverse existing libraries* instead of creating a monolithic programming model in which all applications should be built. We show that such cross-library optimization is crucial for performance in workloads that combine multiple libraries. In addition, unlike systems that do runtime code generation for more specific workloads (*e.g.*, databases [11] or linear algebra [12]), Weld offers a small and general IR that can achieve state-of-the-art performance *across* these workloads.

Multiple prior systems aim to simplify programming parallel hardware. Hardware-independent intermediate languages such as OpenCL [9], LLVM [25], and SPIR [37] are based on low-level sequential instruction sets where threads communicate via shared memory, which makes it hard to perform complex optimizations for *parallel* code, such as loop fusion or loop tiling across parallel functions. Moreover, the APIs to these systems are evaluated eagerly, resulting in multiple independent invocations when applications combine multiple libraries. In .NET languages, LINQ [18] offers a lazily evaluated API that has been used to target clusters [22] and heterogeneous hardware [23] using relational algebra as an IR. However, LINQ is designed for the "closed" world of .NET programs and does not provide a means of interfacing with code outside the .NET VM. Moreover, although its relational IR can support various fusion optimizations [38], it is difficult to express other optimizations such as loop tiling. Spark [21], FlumeJava [39] and other systems [40, 41] also perform optimizations using a relational or functional IR, while Pydron [24] uses annotations to parallelize Python code.

Runtime code generation has been used in systems including databases [1, 11, 16] and TensorFlow's XLA compiler [12]. However, most of these systems are limited to a narrow workload domain (*e.g.*, linear algebra or SQL with sequential user-defined functions [16]). Our work shows that Weld's more general IR, together with its optimizer, enables code generation on par with these systems for multiple key workloads and simultaneously allows optimizing across them.

The compilers literature is also rich with parallel languages, IRs and domain-specific languages (DSLs) [27, 42–45]. However, most of this work focuses on static compilation of an entire program, whereas Weld allows dynamic compilation at runtime, even when programs load libraries dynamically or compose them in a dynamically typed language like Python. Weld's IR is closest to monad comprehensions [19] and to Delite's DMLL [46, 47], both of which support nested parallel loops that can update multiple results per iteration.[3] Builders are also similar to Cilk reducers [48] and to LVars [49], although these systems to not implement them in different ways on different hardware platforms. Finally, Weld's optimization techniques (*e.g.*, loop fusion) are standard, but our contribution is to demonstrate that they can be applied easily to Weld's IR and yield high-quality code even when libraries are combined using the narrow interface Weld provides.

9. Conclusion

We presented Weld, a novel interface and runtime for accelerating data-intensive workloads using disjoint functions and libraries. Today, these workloads often perform an order of magnitude below hardware limits due to extensive data movement. Weld addresses this problem through an intermediate representation (IR) that can capture data-parallel applications and a runtime API that allows IR code from different libraries to be combined. Together, these ideas allow Weld to bring powerful optimizations to multi-library workflows without changing their user-facing APIs. We showed that Weld is easy to integrate into Spark SQL, TensorFlow, Pandas and NumPy and provides speedups of up to 6.5× in workflows using a single library and 29× across libraries.

10. Acknowledgements

We thank Pratiksha Thaker, Chirstopher Aberger, Firas Abuzaid, Parimarjan Negi, Paroma Varma, Peter Bailis, and the many members of the Stanford InfoLab for their valuable feedback on this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN

[3] Weld has some differences from these IRs, however, most notably that it represents builders as a first-class type. For example, in Delite, the result of each loop is a collection, not a builder, meaning that nested loops cannot efficiently update the same builder unless the compiler treats this case specially.

project – Intel, Microsoft, Teradata, and VMware – as well as NSF CAREER grant CNS-1651570 and NSF Graduate Research Fellowship under grant DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proc. ACM SIGMOD*, 2015, pp. 1383–1394.

[2] NumPy. [Online]. Available: http://www.numpy.org/

[3] W. McKinney, " Data Structures for Statistical Computing in Python ," in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 51 – 56.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. USENIX OSDI*, 2016, pp. 265–283.

[5] A. Kagi, J. R. Goodman, and D. Burger, "Memory Bandwidth Limitations of Future Microprocessors," in *ISCA*, 1996, pp. 78–78.

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[7] N. AlSaber and M. Kulkarni, "Semcache: Semantics-aware caching for efficient gpu offloading," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465021

[8] Loop tiling. [Online]. Available: https://en.wikipedia.org/wiki/Loop_tiling

[9] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[10] CUDA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[11] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proc. VLDB*, vol. 4, no. 9, pp. 539–550, 2011.

[12] TensorFlow XLA. [Online]. Available: https://www.tensorflow.org/performance/xla/

[13] S. Agarwal, D. Liu, and R. Xin, "Apache Spark as a compiler: Joining a billion rows per second on a laptop." [Online]. Available: https://databricks.com/blog/2016/05/23/

[14] Pandas Cookbook example. [Online]. Available: http://nbviewer.jupyter.org/github/jvns/pandas-cookbook/blob/v0.1/cookbook/Chapter%207%20-%20Cleaning%20up%20messy%20data.ipynb

[15] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 293–307.

[16] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, "An Architecture for Compiling UDF-centric Workflows," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1466–1477, August 2015.

[17] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 693–701.

[18] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling Object, Relations and XML in the .NET Framework," in *SIGMOD*. ACM, 2006, pp. 706–706.

[19] T. Grust, *Monad Comprehensions: A Versatile Representation for Queries*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 288–311.

[20] D. Walker, "Substructural Type Systems," in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, 2005, ch. 1.

[21] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, October 2016.

[22] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[23] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A Compiler and Runtime for Heterogeneous Systems," in *Proc. ACM SOSP*. ACM, 2013, pp. 49–68.

[24] S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy, "Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud." in *OSDI*, 2014, pp. 645–659.

[25] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004, pp. 75–86.

[26] Intel AVX2. [Online]. Available: https://software.intel.com/en-us/node/523876

[27] R. D. Blumenofe, C. F. Joerg, B. C. Kurzmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[28] G. E. Blelloch and G. W. Sabot, "Compiling Collection-oriented Languages onto Massively Parallel Computers," in *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of.* IEEE, 1988, pp. 575–585.

[29] MNIST. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[31] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: SQL and Rich Analytics at Scale,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. ACM, 2013, pp. 13–24.

[32] OpenMP. [Online]. Available: http://openmp.org/wp/

[33] HyPer Web Interface. [Online]. Available: http://hyper-db.de/interface.html

[34] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “GraphMat: High Performance Graph Analytics Made Productive,” *Proc. VLDB*, vol. 8, no. 11, pp. 1214–1225, 2015.

[35] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, “Hardware-oblivious parallelism for in-memory column-stores,” *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 709–720, 2013.

[36] Y. Liu and B. Schmidt, “LightSpMV: faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs,” in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 82–89.

[37] J. Kessenich. (2015) An introduction to SPIR-V. [Online]. Available: https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf

[38] D. G. Murray, M. Isard, and Y. Yu, “Steno: Automatic Optimization of Declarative Queries,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 121–131.

[39] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, “FlumeJava: easy, efficient data-parallel pipelines,” in *ACM SIGPLAN Notices*, vol. 45, no. 6. ACM, 2010, pp. 363–375.

[40] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, “Musketeer: All for One, One for All in Data Processing Systems,” in *Proc. ACM EuroSys*, 2015, pp. 2:1–2:16.

[41] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl, “Implicit Parallelism Through Deep Language Embedding,” in *SIGMOD ’15*, 2015.

[42] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding Fork-Join Parallelism into LLVM’s Intermediate Representation,” in *PPoPP*. ACM, 2017, pp. 249–265.

[43] Data Parallel Haskell. [Online]. Available: https://wiki.haskell.org/GHC/Data_Parallel_Haskell

[44] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, “Implementation of a Portable Nested Data-parallel Language,” *SIGPLAN Not.*, vol. 28, no. 7, pp. 102–111, 1993.

[45] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, “A Domain-specific Approach to Heterogeneous Parallelism,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’11. New York, NY, USA: ACM, 2011, pp. 35–46.

[46] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky, “Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging,” in *POPL ’13*, 2013.

[47] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun, “Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO 2016. ACM, 2016, pp. 194–205.

[48] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, “Reducers and Other Cilk++ Hyperobjects,” in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’09. New York, NY, USA: ACM, 2009, pp. 79–90.

[49] L. Kuper and R. R. Newton, “LVars: Lattice-based Data Structures for Deterministic Parallelism,” in *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, ser. FHPC ’13. New York, NY, USA: ACM, 2013, pp. 71–84.