Evaluating End-to-End Optimization for Data Analytics Applications in Weld

Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker,
Rahul Palamuttam, Parimajan Negi, Anil Shanbhag‡, Malte Schwarzkopf‡,
Holger Pirk†, Saman Amarasinghe‡, Samuel Madden‡, Matei Zaharia
Stanford University, †Imperial College London, ‡MIT CSAIL
{shoumik, jjthomas, deepakn, prthaker, rpalamut, pnegi, mzaharia}@stanford.edu,
pirk@imperial.ac.uk, {anil,malte,madden,saman}@csail.mit.edu

ABSTRACT

Modern analytics applications use a diverse mix of libraries and functions. Unfortunately, there is no optimization across these libraries, resulting in performance penalties as high as an order of magnitude in many applications. To address this problem, we proposed Weld, a *common runtime* for existing data analytics libraries that performs key physical optimizations such as pipelining under existing, imperative library APIs. In this work, we further develop the Weld vision by designing an automatic adaptive optimizer for Weld applications, and evaluating its impact on realistic data science workloads. Our optimizer eliminates multiple forms of overhead that arise when composing imperative libraries like Pandas and NumPy, and uses lightweight measurements to make data-dependent decisions at runtime in ad-hoc workloads where no statistics are available, with sub-second overhead. We also evaluate which optimizations have the largest impact in practice and whether Weld can be integrated into libraries incrementally. Our results are promising: using our optimizer, Weld accelerates data science workloads by up to 23× on one thread and 80× on eight threads, and its adaptive optimizations provide up to a 3.75× speedup over rule-based optimization. Moreover, Weld provides benefits if even just 4–5 operators in a library are ported to use it. Our results show that common runtime designs like Weld may be a viable approach to accelerate analytics.

PVLDB Reference Format:
S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, M. Zaharia. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB*, 11(9): 1002-1015, 2018.
DOI: https://doi.org/10.14778/3213880.3213890

1. INTRODUCTION

Modern data analytics workloads combine a broad mix of libraries, functions and processing systems. Although a workload might begin by selecting and transforming data through SQL, advanced analytics and machine learning pipelines typically involve external libraries such as Pandas [39], NumPy [43], TensorFlow [1], Apache Spark [61] and others in a programming language such as Python or R. These languages already offer thousands of third-party software libraries for data analytics [11, 50], and new ones are constantly being written to support new workloads [20, 51].

Proceedings of the VLDB Endowment, Vol. 11, No. 9

DOI: https://doi.org/10.14778/3213880.3213890

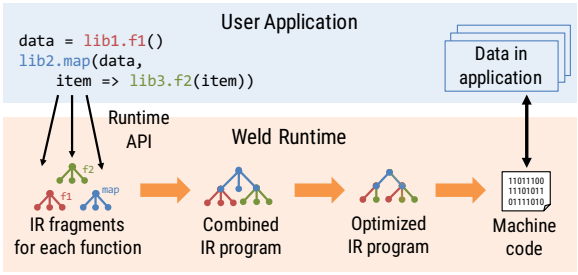


Figure 1: Weld's runtime lets diverse libraries submit computation in a functional intermediate representation (IR). After collecting fragments from multiple libraries via a lazy API, Weld generates optimized code that runs on the application's in-memory data.

Unfortunately, the analytics performed with these independent software libraries lack one of the main benefits of a DBMS: *end-to-end optimization*. A key reason for this is that users submit workloads to a DBMS using a unified query language—SQL—over which optimizations occur. In contrast, the interface to traditional software libraries is a rigid set of *library-specific functions*. Each library manages its own function execution, precluding optimization across different libraries, and sometimes even across functions in the same library. As a result, in applications using multiple libraries and functions, even simple optimizations like pipelining do not occur. This lack of end-to-end optimization has a serious impact on performance: we found that typical data science workloads using NumPy, Pandas and TensorFlow can run as much as 23× slower one thread compared to hand-optimized code, even when the individual functions they call (e.g., BLAS kernels) are heavily optimized.

To address this problem, we have proposed Weld [48], a *common runtime* for data analytics libraries that can perform key physical optimizations under diverse existing libraries. Weld provides a novel programming interface for analytics libraries that enables efficient composition. Weld lets libraries express the data-parallel structure of their computations (e.g., a map operation or an aggregation) using a functional *intermediate representation (IR)*, and uses a lazily evaluated *runtime API* to collect IR code from different libraries (Figure 1). As applications call Weld-enabled libraries and functions, Weld builds up a combined IR program, and then optimizes across the whole program before generating native code and running it. We showed that Weld IR can express a variety of workloads (e.g., relational and linear algebra) and that the IR supports a variety of

transformations to improve performance. However, this early work was limited in two ways: all the IR optimizations were invoked manually, and evaluation was limited to a small set of microbenchmarks.

In this paper, we extend our work in [48] to study what is needed to build a *fully automatic* Weld runtime system, including an optimizer for Weld's IR, and evaluate the "common runtime" approach using a variety of realistic applications. In particular, we answer the following four questions in designing a common analytics runtime:

1. **How should Weld optimize workloads?** Although the problem of optimizing applications in Weld is similar to database and compiler optimization at a high level, we address two challenges specific to the common runtime setting:
   (a) Unlike many human-authored SQL queries or programs, the Weld IR code generated by calling multiple libraries is often *highly redundant*, because imperative libraries like NumPy and Pandas materialize results after most operations. It is thus crucial to eliminate redundancies that arise from library composition, e.g., intermediate results that can be pipelined.
   (b) Weld needs to optimize ad-hoc analytics workloads with *no precomputed data statistics* (e.g., interactive data analysis in Python), without adding significant runtime overhead.

   To address these problems, we designed a fast adaptive optimizer for Weld based on (*i*) a rich set of pipelining, common subexpression elimination and loop fusion [30] rules to eliminate redundancy and (*ii*) adaptive optimizations for predicating branches [29] and choosing efficient data structures based on data sampled at runtime. We show that our optimization, compilation and sampling steps take sub-second time on realistic workloads and produce high performance code.
2. **How does Weld perform on complex real workloads?** Our prior work only evaluates microbenchmarks using minimal integrations of Weld with NumPy, Pandas and other libraries. In this paper, we study a full set of realistic data science workloads, including data cleaning and ad-hoc analytics in Pandas, image classification in TensorFlow, and physics simulation in NumPy, using expanded integrations with these frameworks.
3. **Can Weld be integrated into libraries incrementally?** Porting whole libraries to use Weld is time-consuming, so we evaluate the impact of porting just a few widely used operators.
4. **Which cross-library optimizations matter in practice?** We study the impact of each of our optimizations (e.g., pipelining, vectorization and adaptive predication) on our suite of workloads to identify which are critical to support in a common runtime.

More generally, this work substantiates the "analytics common runtime" vision set forth in [48] by designing and evaluating the first runtime that can optimize across disjoint analytics libraries automatically end-to-end. Although Weld is only one possible design point and has its tradeoffs and limitations (e.g., we chose to focus on physical data movement optimizations using a functional programming-based IR instead of an IR with domain-specific operators), we believe the results are promising for several reasons:

1. Weld produces code competitive with domain-specific compilers like XLA [59] and HyPer [42], but with a more general IR.
2. Weld achieves speedups of up to 80× on eight threads in unmodified analytics applications written using existing, widely used APIs [1, 4, 39, 43], showing that it is possible to automatically make significant optimizations under existing library APIs.
3. Weld only requires moderate effort to integrate into existing libraries and can be integrated incrementally, requiring a similar effort to accelerating specific operators using languages like CUDA and C (which many libraries do [1,39,43]). Compared to CUDA or C, Weld provides similar single-operator performance, and enable *cross-operator* optimizations that current libraries do not provide unless they implement a compiler [59].

Perhaps most importantly, our work highlights the *interface for composing software libraries* as a key area to work in to enable high performance and high developer productivity at the same time. Today, most systems either feature a domain-specific, high-level interface (such as SQL) that enables rich optimization but requires all of the computation to be understood by a single engine, or low-level interfaces (such as function calls) that can be invoked from anywhere but do not offer end-to-end optimization. Weld's IR aims to give library developers enough freedom to implement their own computations, while still supporting the most performance-critical optimizations, such as pipelining. This lets users have the benefits of a wide library ecosystem and database-like end-to-end optimization at the same time. Weld is open source at `https://weld.rs`.

2. WELD OVERVIEW

This section gives an overview of Weld, covering background on its IR and runtime API from [48] as well as several new extensions we made since that first version to enable additional optimizations.

2.1 Goals and Non-Goals

Weld aims to accelerate in-memory applications that compose data-parallel computations on a single machine, taking advantage of multicore and SIMD processing while minimizing memory movement. It does not currently aim to generate distributed code or handle disk spilling, although it can be integrated into distributed big data systems to accelerate per-node computation. As shown in previous work, many distributed frameworks can be CPU- or memory-bound [4, 12, 47] (as opposed to I/O-bound), so Weld can improve their performance. For example, we show in §8 that Weld can accelerate Spark SQL workloads by up to 6.5×.

Weld optimizes computations across functions and libraries by using a common intermediate language and runtime. There are many possible designs for such a system, ranging from one with built-in primitives for every domain (e.g., machine learning, graph algorithms or relational algebra) to low-level languages such as OpenCL that give library developers full control over how to implement their computations. With Weld, we designed a minimal IR and runtime that enables the most impactful cross-library optimizations while being expressive enough to allow libraries to represent their algorithms. Specifically, Weld was designed to focus 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 index maintenance, reordering linear algebra expressions or reordering SQL joins still need to be implemented within each library outside of Weld, but many systems already perform these. For example, a relational library could first optimize a logical plan by ordering joins, pushing down predicates, etc. and then use Weld for physical plan optimization. For already-optimized code such as BLAS or for domain-specific tasks such as using an index to load on-disk data, Weld supports calling existing C functions.

We expect libraries to integrate Weld in two ways. First, many libraries, such as Pandas and NumPy, already implement performance-sensitive functions in low-level languages such as C. Developers can port individual functions in these libraries to use Weld's functional IR instead, thus automatically benefiting from cross-function optimizations. These libraries often already have compact in-memory data representations (e.g., NumPy arrays [43]), so Weld can work

directly against their in-memory data at no extra cost. Second, some libraries, such as Spark SQL and TensorFlow, already perform code generation beneath a lazily evaluated API. For these libraries, Weld offers both the ability to interface efficiently with other libraries and a systematic way to JIT code. For example, much of the complexity in code generators for databases comes from operator fusion logic that requires transforming a tree of operators into a single loop [2, 42]. With Weld, each operator can emit a separate loop over its inputs, and our optimizer will automatically fuse them.

2.2 Weld IR

The first component in Weld is its intermediate representation (IR), a language that libraries use to describe their computations. The IR plays a similar role to relational algebra in a database, but aims to also support other analytics workloads (e.g., linear algebra).

In order to support a wide range of computations while still enabling key optimizations, Weld's IR is based on monad comprehensions [23], a low-level representation for functional programs or relational algebra that facilitates expressing fusion optimizations. The key idea is to provide a *parallel loop* operator that can read one or more collections of data and updates one or more declarative *builders* to produce results, such as computing a sum or constructing a hash table. Weld imposes a number of constraints on the IR to make it amenable to analysis and optimization: the IR is purely functional, meaning variables are immutable once assigned, and virtual function calls are disallowed. We described the IR in [48], but provide an overview and discuss extensions here.

Data Types. Weld's basic data types are scalars (e.g., `int`, `float`), structures (denoted `{T1,T2,...}`), variable-length vectors (`vec[T]`), and dictionaries (`dict[K,V]`). These types are nestable to support complex data such as JSON or XML. We chose these types because they appear commonly in analytics applications and in low-level data processing code (e.g., dictionaries appear in hash joins).

Computation. Weld's IR contains sequential operators for arithmetic, dictionary lookups, indexed lookups on vectors, sequential `while` loops, sorting, and calling external C functions. In addition, the IR has two parallel constructs: a parallel `for` loop and an abstraction for constructing results called *builders*.

Weld's parallel loops can be nested arbitrarily, which allows complex library function composition (e.g., calling NumPy on each row of a table) and workloads that perform an irregular amount of work per item (e.g., processing an array of variable-length strings). Each `for` 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. As examples, a `vecbuilder[T]` takes values of type `T` and builds a vector of merged values, and a `merger[T,op]` combines values of type `T` into a single result using an associative operation `op`. In the database context, builders are used to construct the result of a query while consuming data. For example, a `merger` captures aggregations, a `vecbuilder` captures selections, and the dictionary-based builders can be used for hash joins or grouping. Builders are *declarative types* since they only specify a high-level operation rather than specific hardware implementation. This allows Weld to change builder implementations adaptively based on the data and hardware.

Builders support three basic operations. The `merge(b, v)` operation adds a new value `v` into the builder `b` and returns a new builder object to represent the result. Merges into builders are associative, enabling parallel execution. The `result(b)` operator destroys the builder `b` and returns its final result: no further operations are allowed on it after this. Finally, Weld's parallel `for` loop updates one or more builders in parallel. `for(vector, builders, func)` applies a function of type `(builders, index, elem)=> builders` to each element of a vector in parallel, then returns the updated builders. Each call to `func` receives an element of a vector and its index, and may merge zero or more values into each builder. Although our original IR [48] only allowed iterating over one vector at a time in a loop, we have also extended it to iterating over multiple vectors together, as we discuss in §2.3.

Table 1: Builder types in Weld.

Builder Types	
`vecbuilder[T]`	Builds `vec[T]` by appending merged values of type `T`.
`merger[T,op]`	Builds a `T` by merging values using an associative operation `op`.
`dictmerger[K,V,op]`	Builds `dict[K,V]` by merging `{K,V}` pairs using an associative operation.
`vecmerger[T,op]`	Builds `vec[T]` by merging `{index,T}` pairs into specific cells in the vector using an associative operation.
`groupbuilder[K,V]`	Builds `dict[K,vec[V]]` by merging `{K,V}` pairs and grouping by key.

```
// Merge two values into a builder.
b1 := vecbuilder[int]
b2 := merge(b1, 1)
b3 := merge(b2, 2)
result(b3) // returns [1, 2]

// Use a for loop to merge multiple values.
data := [1, 2, 3]
b1 := merger[int, +]
b2 := for(data, b1, (b,i,x) => merge(b, x))
result(b2) // returns 6
```

Listing 1: Examples of using builders.

Weld places two additional 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. Second, functions passed to `for` must return builders derived from their arguments. These restrictions allow Weld to safely implement builders using mutable state.

Weld also provides macros that implement common functional operators such as `map`, `filter` and `reduce`. These operators all map into loops and builders.

User Defined Functions. Weld supports calling user-defined C functions by name as part of its IR. Users pass values from the Weld program to the C function and specify the Weld return type of the function. Each Weld type has a standard C-compatible data layout (§2.4). UDFs can be used to call compute-optimized kernels such as Level-3 BLAS routines or to access state outside of Weld.

Benefits of Weld's IR. As described in [48], Weld's IR can express many common data processing tasks, including relational operators, functional APIs like Spark, graph computations and linear algebra. Moreover, the explicit separation of loops and builders also makes it possible to describe *fused* computations that update multiple builders in a single loop. For example, suppose that an application wants to compute two results based on a sequence of numbers. In the Weld IR, it is possible to write a single loop over the data that produces both results in two different builders, as shown in Listing 2. Weld's optimizer analyzes the builders and loops to apply these optimizations automatically.

```
// Compute two results from same sequence.
data := [1, 2, 3]
squares := map(data, x => x*x)
doubled := map(data, x => 2*x)

// Program above converted to a single loop.
b1, b2  := vecbuilder[int], vecbuilder[int]
res = for(data, {b1, b2},
   (bs, i, x) => {merge(bs.0, x*x), merge(bs.1, 2*x)})
squares, doubled := result(res.0), result(res.1)
```

Listing 2: Computing two results in a single loop.

2.3 Weld IR Extensions

To better support more workloads, we extended the Weld IR presented in [48] with a more general concept of loops. First, some libraries, such as NumPy, require *strided* access over part of an array (e.g., accessing just one column in a matrix). Second, many operations require iterating over multiple collections together. As an example, many Pandas [39] functions require operating over two separate data columns in a DataFrame. A limited version of this could be achieved in the IR from [48] by using indices to look up elements in multiple arrays, but we wanted to make this more explicit to enable more optimizations. We thus extended the `for` construct in Weld to accept multiple *iterators* as input, where an iterator is defined as a vector of data, a start index, an end index, and a stride. The function passed to the `for` loop receives the corresponding element from each iterator and can process elements from each iterator together. This extension enables automatic identification of a broader class of redundant computations, even in applications using complex access patterns such as matrix operations.

2.4 Weld Runtime API

Weld's second major component is its runtime API. The API is inspired by interfaces like OpenCL and CUDA, which allow libraries submit code that runs on parallel hardware. However, unlike these two interfaces, Weld's API is *lazily evaluated*. As libraries call the API and submit fragments of IR code, Weld remembers the fragments and only executes them when a special function is called to force evaluation. This creates the opportunity to run our optimizer (§3). Once Weld has optimized the provided IR code, our system compiles it and executes it against the application's in-memory data.

As an example, the Python program in Listing 3 takes a Pandas DataFrame `data`, filters values from a column, and sums the result with NumPy. In the current versions of these libraries, each step would proceed eagerly as separate loops over the data. With Weld's API, the filtering and aggregation computations are submitted lazily, and Weld's optimizer will fuse and vectorize them together for enhanced performance to just compute the final result `sum`.

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

Listing 3: A sample Python program using Pandas an NumPy.

Developers integrate Weld into their libraries using an interface called `WeldObject`, which represents either a lazily evaluated subcomputation or some in-memory data in the application (e.g., a NumPy array). A `WeldObject` may depend on other `WeldObjects` (possibly from other libraries), forming a DAG for the whole program where leaf nodes in the DAG refer to in-memory data. Table 2 summarizes the main API functions for manipulating `WeldObjects`.

Table 2: Weld API for lazily composing and evaluating functions.

API Summary	
`NewWeldObj(data,ty)`	Creates a `WeldObject` representing an in-memory value `data` with type `ty`.
`NewWeldObj(deps,expr)`	Creates a `WeldObject` with the Weld IR expression `expr`, which may depend on any of the `WeldObjects` in `deps`.
`Evaluate(obj)`	Evaluates the `WeldObject` `obj` and returns its value to the application.

Developers create `WeldObjects` using the `NewWeldObj` call. This call has two variants: one to register in-memory data in the application with Weld and one to define an object computed from other `WeldObjects`. To encapsulate in-memory data, developers pass a pointer to the data dependency and the Weld type of the dependency (e.g., `vec[int]` for an integer array). Weld's runtime uses a standard binary format for data types that allows the system to operate over many existing in-memory formats without marshalling. Scalar types (`int`, `float`, etc.) and structs follow C packed structure layout, and vectors `vec[T]` are represented as a pointer and a length. If necessary, developers can provide "encoder" and "decoder" functions for conversions to and from the Weld format.

To define `WeldObjects` that depend on other `WeldObjects`, developers pass a list of the parent `WeldObjects` (`deps`) and Weld IR code representing a computation. The IR code must reference only the objects declared in `deps`. Internally, each `WeldObject` has a unique name; using an existing `WeldObject` in an IR fragment substitutes the name into the fragment. Listing 4 shows an example defining a function to square a number passed as a `WeldObject`:

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

Listing 4: A simple function that squares an argument using Weld.

The `Evaluate` call evaluates a `WeldObject` instance and returns a result. Libraries choose when to evaluate an object. 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.

2.5 Runtime API Extensions

We developed two API extensions beyond the system described in [48] to support larger applications:

Grouped Evaluation. We updated `Evaluate` to allow evaluating multiple Weld objects in one computation, which is useful in applications that produce multiple results. Our optimizer automatically identifies shared subcomputations across these results.

Memory Management Functions. We added functions allowing libraries to control Weld's memory usage. Libraries can set a cap on how much memory a call to `Evaluate` may use, and the call will fail if it needs to allocate more memory at runtime. This is useful in engines that already perform their own memory management, such as Spark SQL: these engines can call Weld-optimized code on batches of data and spill to disk when they grow past a certain size.

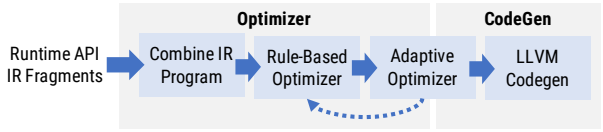


Figure 2: Architecture of the Weld optimizer. IR code submitted to Weld is combined, optimized via rules, and transformed to generate adaptive optimization code for decisions based on runtime statistics. The optimized IR code is compiled to assembly using LLVM.

3. AN OPTIMIZER FOR WELD

The main contribution we evaluate in this paper is an automatic optimizer for Weld. Optimizing Weld computations is conceptually similar to relational query optimization, but two aspects of Weld's "common runtime" setting pose unique challenges:

1. IR code passed to Weld is generated out of fragments from different library functions that have no knowledge of each other, even though they may depend on the same data or subcomputation. This contrasts with many queries to a DBMS, where the full SQL query is generated as one program or even written by one user. Weld thus needs to identify and eliminate many forms of *redundancy* that arise from imperative data analytics libraries. We addressed this challenge by performing a wide range of redundancy elimination transformations from the database and compiler literature (e.g., pipelining [22] and fusion transformations across multiple loops [30], which are similar to scan sharing [52] and some types of multi-query optimization [55]). Our loop-and-builder IR and the extensions for loops with multiple inputs (§2.3) also help with this task.
2. In the most general target setting for Weld (ad-hoc data analysis in a language such as Python), there are no precomputed data statistics, so Weld needs to make any data-dependent decisions adaptively at runtime. Moreover, each query might only run once, so any data sampling to compute statistics must be fast. In this paper, we focused on designing an optimizer for this setting to show that effective optimization is still possible. However, our optimizer can also use application-provided statistics instead of measuring them at runtime.

We designed an optimizer using a combination of transformation rules and adaptive decisions that combines ideas from both the database and compilers literatures (Figure 2). The optimizer begins with rule-based optimizations from the database and compiler literature, then moves on to an adaptive optimization phase where key data-dependent decisions are considered (such as whether to use branching or predication based on a filter's selectivity [29]). For these decisions, our optimizer generates code to sample the relevant property of the data at runtime and switch between two possible execution plans. Both plans are represented in Weld IR and are passed through the rule-based optimizer again. Finally, given a Weld IR program that contains both the sampling code and the possible execution branches, we apply a code generator based on LLVM to JIT-compile the code. This code then runs on a multithreaded, memory-managed runtime. §4–6 describe the rule-based, adaptive, and code generation optimizations in turn.

4. RULE-BASED OPTIMIZATIONS

The first step of the optimizer is to apply rule-based optimizations on the combined IR program. Like other database optimizers and compilers [33, 56], we group our rules into *phases* and run each phase to a fixpoint. Moreover, each rule produces a new abstract syntax tree (AST) in the Weld IR, making it easy to combine rules

```
filtered= df[df['country'] == 'usa']
prices = filtered['price'] * filtered['quantity']
outp= prices.sum()
```

Pandas to Naïve Data Parallel Operators

```
tmp0 = filter(country, x => x == 'usa')
tmp1 = map(zip(price, quantity), (x) => x.price * x.quantity)
tmp2 = filter(zip(tmp0, tmp1), (x) => x.tmp0)
outp = reduce(tmp2, 0, (x, y) => x.tmp1 + y)
```

Disjoint operators to fused Weld Program

```
result(for(zip(price, quantity, country),
  merger[int,+],
  (b,i,x) => if (x.country == 'usa',
            merge(b, x.price * x.quantity), b)
 ))
```

Figure 3: Optimizing a simple Pandas program that filters and aggregates data. While Pandas repeatedly loops over data, Weld integration enables pipelining.

in an arbitrary order. After each pass, Weld also applies a number of *simplification transformations* to remove redundant computation, such as inlining variables that are only used in one expression, common subexpression elimination, constant propagation, and standard algebraic simplifications. These rule-based optimizations help eliminate redundancies caused by composing independently-written functions and libraries. We elaborate on key optimization passes.

4.1 Fusion

The fusion pass performs the main data movement optimizations in Weld, merging data-parallel operations from different libraries and functions into a single parallel `for` loop and removing redundant iteration over the same data or materialization of results. "Fusion" is our general term for several transformations including pipelining `for` loops whose output is directly consumed by another loop (as in many database engines [22, 42]) and fusing independent loops that read the same input data (similar to some forms of multi-query optimization [55] and scan sharing [52] in the database literature, or loop fusion [30] in the compilers literature). This pass is critical because independently called libraries will produce separate loops, even if these loops are running over the same data. For example, in Pandas and NumPy, every operator used to build an expression (e.g., `vec1+vec2+vec3`) produces a new array or data column for its intermediate result. Figure 3 shows an example of a simple Pandas program that benefits from fusion.

We perform two types of fusion: *pipelining* and *horizontal fusion* of loops over the same data. Listings 5 and 6 give examples of both.

```
// Before pipelining.
v1 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b, x+1)))
v2 := result(for(
    v1, vecbuilder[int], (b,i,x) => merge(b, x*5)))

// After pipelining.
v2 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b, (x+1)*5)))
```

Listing 5: Pipelining expressed using Weld.

In the pipelining transformation, Weld fuses the body of loops whose result is consumed in just one other loop. Rather than traversing (and materializing) vectors after each operation, pipelining enhances memory locality by loading data into the cache or registers

and applying all operations at once. While DBMS engines usually pipeline operators in the Volcano query processing model, without Weld, independently written libraries have no common substrate to allow for such optimizations. Compared to code-generating databases like HyPer [42], which pipeline operators via a produce/-consume API and a complex translation to imperative code to fuse individual operators, Weld's approach is to use a higher level IR that facilitates identification and fusion of pipelinable loops while simultaneously capturing parallelism. For example, in Listing 5, Weld identifies that `v1` is only used in one downstream expression, and fuses it into that loop. The optimized loop still executes in parallel and is represented in the same IR, allowing further optimizations.

```
// Before horizontal fusion
v1 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b,x+1)))
v2 := result(for(
    v0, merger[int,+], (b,i,x) => merge(b,x)))
{v1, v2}

// After horizontal fusion
tmp := for(v0, {vecbuilder[int], merger[int,+]},
    (bs,i,x) => {merge(bs.0, x+1), merge(bs.1, x)}
)
{result(tmp.0), result(tmp.1)}
```

Listing 6: Horizontal fusion of two independent loops.

In horizontal fusion, loops over the same input data that produce different results are fused to loop over data just once, and return a tuple of results. Like pipelining, horizontal fusion enhances memory locality. The transform helps when repeatedly computing multiple results from the same input data: for example, the Pandas API creates a new vector from each operator on a data frame column (e.g., `col+1`), as shown in Figure 3.

4.2 Size Analysis Optimizations

Weld contains two optimizations based on knowledge of the size of a vector (`vec[T]`) at optimization time. *Loop unrolling* [36] allows replacing small loops with a sequence of statements to reduce control flow, while *preallocation* allows allocating memory for data structures in advance instead of growing them dynamically.

When Weld's optimizer is invoked, it knows the size of the input vectors to the computation. Moreover, libraries can mark the size of intermediate results in IR code with an annotation (e.g., if a library knows it works with 3-element vectors). The optimizer propagates this information to other intermediate results when possible (e.g., a loop that merges exactly one value per iteration into a `vecbuilder` will produce a result of the same size as the input).

Using this information, Weld first unrolls `for` loops with a simple body to run sequentially if they are sufficiently small. This can yield better pipelining within the CPU by eliminating control flow and branching logic, and also reduce overheads from Weld's multi-threaded runtime by running small loops sequentially.

Second, Weld attempts to preallocates memory for data structures that might otherwise be resized. For example, the `vecbuilder` supports merging a variable number of records, and is implemented using a dynamically-growing array by default. However, if the loop merging values into a `vecbuilder` adds in exactly one value per iteration (i.e., each control path in its body has just one `merge`), the optimizer can pre-allocate a vector of the same size as the input.

Unrolling and preallocation have a large impact in numerical code especially, such as NumPy code working with arrays of fixed dimensions. While these transformations also exist in traditional compilers, they are more powerful in Weld because the optimizer has access to size information that is only available at *runtime*.

4.3 Vectorization

The vectorization pass changes `for` loops to use SIMD instructions when possible to take advantage of modern CPU execution capabilities. Specifically, the optimizer change loops over elements of type `T` to loops over `simd[T]`, where `simd[T]` is a fixed-length SIMD data type. This pass only vectorizes loops without branches; our adaptive predication optimization (§5.1) determines whether to vectorize branches because the benefit depends on selectivity.

Although LLVM also provides a vectorizer, we found that it was easier to get consistent results by vectorizing at the Weld IR level. This is because Weld's IR is purely functional, with no side-effects, and is thus simpler to analyze than lower-level IRs. These optimizations are harder to apply in a lower-level IR such as LLVM due to the need to analyze pointer aliasing (whether two addresses in mutable memory might point to the same data) [37]. Even in cases where the aliasing analysis is successful, applying these passes on Weld IR is beneficial because it reduces compile time—an important factor because Weld needs to JIT-compile code.

5. ADAPTIVE OPTIMIZATIONS

Certain optimization decisions are data-dependent. In a DBMS, statistics catalogs and cost-based optimizers guide whether such optimizations should be applied to a plan, but the libraries Weld integrates with generally do not have these tools a priori. We therefore designed a set of *adaptive optimizations* that can occur at runtime based on observed properties of the data. In particular, we consider two adaptive optimizations that we found have the greatest impact: *adaptive predication* and *adaptive data structures*.

5.1 Adaptive Predication

Given a Weld program with a branch (i.e., an `if` statement), *predicating* the branch evaluates the condition, on-true expression, and on-false expression unconditionally, and then uses a hardware `select` instruction to choose between the true or false expression depending on whether the condition is `true` or `false`. The key advantage of predication is that it enables vectorization: modern CPUs contain vectorized `select` instructions, whereas loops with branches or other control flow are difficult to vectorize. The tradeoff is that both the true and false expression are always evaluated (whereas in a branched expression, only one of the two is evaluated). If the branch is highly predictable and computing the true and false expressions is expensive, predication may hurt performance. The choice of whether to predicate thus depends on (*i*) the selectivity of the branch (a data-dependent factor) and (*ii*) the branch target costs [29].

Adaptive predication proceeds as follows. Given a loop with a branch that can be vectorized, the optimizer replaces the loop with three loops. The first loop samples the input vector and evaluates the condition of the branch on each sample to estimate the branch selectivity. The second loop is a copy of the original branched loop with scalar instructions, and the third loop is a vectorized and predicated version of the original loop. We generate code that uses the measured selectivity to choose between these two loops using a cost model. The cost model determines the costs of the true and false expressions to determine whether, at the given selectivity, evaluating *both* expressions unconditionally with SIMD will provide a speedup.

We use a simplified version of existing database cost models [29, 38] to make this decision. These cost models take as input a conditional of the form `expr = if(cond, merge(b, body), b)` and the measured selectivity of `cond`. The memory accesses in branched expressions can be separated into two sets: the set of accesses in the

condition, C, and accesses in the *body*, B. In branched code, we assume that memory accesses in the condition are *sequential*, and accesses in the body are *random* (i.e., they will not be prefetched). This is because accesses in the condition are performed on every loop iteration. We assume sequential accesses are prefetched by the CPU, and the access time is thus limited by the memory bandwidth. Columns accessed only in the body will not always be prefetched; therefore, the memory latency (computed as in [38]) dominates the access time. Therefore, we compute the cost for branched code as:

$$\sum_{e \in C} \frac{\texttt{sizeof}(e)}{T} + s \times \sum_{e \in B} L, \tag{1}$$

where T is the memory throughput in bytes per second, L the memory latency, and s the measured selectivity.

In predicated code, we treat accesses in both the condition and body are sequential, since both execute on every input row. For simplicity, we captured the expected speedup from vectorization in predicated code using a constant multiplier v to reflect the reduced number of instructions, but we validated that the results were similar to modeling instruction costs and memory access time separately. This gives us the following expression for the predicated cost:

$$v \times \sum_{e \in C \cup B} \frac{\texttt{sizeof}(e)}{T} \tag{2}$$

The generated code evaluates both of these expressions based on the measured selectivity and chooses the plan with the lower estimated cost (i.e., lower expected runtime). We show in §8 that predication can either increase or decrease performance depending on the data, so choosing whether to apply it *adaptively* is important.

5.2 Adaptive Data Structures

Weld's builders are declarative data types that can choose an implementation without specifying one in the IR. An important data-dependent decision Weld makes for builders using dictionary-like data structures (e.g., hash tables for `groupBy` operations) is determining when to use thread-local dictionaries that are merged when `result` is called, and when to use a single global dictionary shared by all threads. The local strategy performs better at lower cardinalities because it avoids the synchronization overhead of the global dictionary, but uses more memory due to keys appearing in multiple dictionaries. The global strategy naturally performs better at large key cardinalities due to its more efficient use of memory.

Our builder implementations for dictionaries thus use an adaptive design that comes close to the best of both worlds: for each thread, we use a local dictionary until the dictionary reaches a certain size threshold, then switch to using a global dictionary for all new keys that are not already in the local one. Thus, hot keys can be accessed in the local dictionary without synchronization overhead, while keeping memory usage and the risk of page faults and TLB misses in check. This design adapts to both small and large cardinalities in the event that Weld does not have access to statistical information (e.g., cardinality estimates) at optimization time.

6. CODE GENERATION AND RUNTIME

The final phase of Weld's optimizer is code generation. Weld JIT-compiles its optimized IR code into multithreaded assembly code using LLVM, and then executes it on a custom runtime. Our runtime supports dynamic load balancing between threads using a work-stealing mechanism inspired by Cilk [7], allowing Weld to support workloads with irregular parallelism such as graph algorithms.

The runtime manages one worker thread per core, where each thread can run *tasks* from a local queue (a task being a function pointer and argument). Each task may spawn additional tasks onto its local queue. When a worker thread is idle, it steals a task from a random worker, which can be shown to offer near-optimal load balancing and high locality [7]. In addition, new tasks can have a set of *parent tasks* specified that must complete before they become runnable, creating a dependency DAG. The parent tasks will update a counter on the child task when they complete to ultimately determine when it has become runnable. Weld generates code by generating one task for the outermost expression in the AST, and recursing down into the expressions needed to compute it. Whenever the code generator encounters a `for` loop (the only construct that can spawn parallel work), it generates a new LLVM function for running the loop's body, which takes a start and end iteration index and any external variables the loop body depends on as arguments. The runtime splits loops into multiple tasks by passing different iteration indices into the function dynamically, similar to [7].

Builders support merging values from multiple tasks. Internally, each builder type follows an API that includes an initialization step, a function to "promote" a local builder (running in just one task) to a "global" builder (when its task gets split), and a function for merging values that also receives the iteration index of the value being merged. For example, the `merger` type will store one value per thread and combine them in its `result` operation, while the `vecbuilder` will keep inserted items in loop iterator order.

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

Library	Glue LoC	Per-Operator LoC
NumPy	Py: 84, C++: 201	avg: 16, max: 50
Pandas	Py: 416, C++: 284	avg: 22, max: 64
Spark SQL	Py: 5, Scala: 300	avg: 23, max: 63
TensorFlow	Py: 175, C++: 652	avg: 22, max: 85

7. LIBRARY INTEGRATIONS

We integrated Weld into four popular libraries: Spark SQL, TensorFlow, NumPy, and Pandas. Each integration was performed *incrementally* and supports porting only a subset of the operators to Weld while interoperating with non-ported ones. Each integration 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 3.

Spark SQL. We integrated Weld with Spark SQL [4] to accelerate its local computations on each node, which can be a bottleneck on modern hardware [47]. Spark SQL already has a lazy API to build an operator graph, and already performs Java code generation using a similar technique to HyPer [42], so porting it to use Weld was straightforward: we only needed to replace the generated Java code with Weld IR via Weld's API. Spark SQL's existing Java code generator uses complex logic [2] 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 [1] 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 Weld programs, 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 created a `WeldObject` -based subclass of its array type called `weldarray`. This routes the NumPy *ufuncs* (C-based implementations of low-level operators such as addition, dot product, or element-wise logarithms) through the `weldarray` class and allows us to easily offload unsupported operations to NumPy (thus enabling incremental integration). Our integration supports most NumPy *ufuncs* and partially supports other NumPy features such as reductions and broadcasting. It also accesses existing NumPy arrays directly without copying memory, because they already stored as packed arrays of primitive types that we can pass to `NewWeldObj`.

To work around NumPy's eagerly evaluated interface, each operation adds a dependency or computation to the `weldarray` `WeldObject`, and evaluation occurs only when necessary. Our evaluation points are similar to Bohrium's [8], another effort at building a lazily evaluated NumPy. Specifically, we evaluate a `weldarray` when the data is accessed (e.g., by `print`) or when the values are needed for operators we have not ported to Weld. Like Bohrium, our port requires minimal changes to NumPy applications (just importing a different package). `weldarray` is less than 1000 lines of Python code, and did not require extensive familiarity with internals.

NumPy also supports indexed access into vectors and matrices to access specific elements in an array or to view "slices" of the array (i.e., a range of values across a set of axes). When a user performs indexed access into a `weldarray`, our port evaluates the array and defers indexing to NumPy. This also allows Weld to support NumPy's advanced indexing features [44]. Notably, indexed access removes opportunities for lazy evaluation, and as in native NumPy and Bohrium, they can be slow due to the overhead of running Python code. In general, in NumPy it is considered best practice to avoid indexed access in loops [24].

We ported Pandas in a similar way, by creating wrapper objects around the Pandas `DataFrame` and `Series` classes. We ported Pandas' filtering, sorting, predicate masking, aggregation, groupby, merge, per-element string slicing, `getUniqueElements`, and pivot table operations to Weld. Pandas represents DataFrame columns as NumPy arrays, so we reuse code from our NumPy port to pass pointers to this data to Weld. We additionally added custom encoding functions for string data, because Pandas stores them as an array of pointers to Python string objects. We copy these strings to arrays of characters managed by our code when we need to access them from Weld.

8. EVALUATION

Our evaluation seeks to answer the following questions:

1. Does Weld speed up realistic data science workloads that compose functions from existing analytics libraries?
2. Which optimizations have the greatest impact on performance?
3. Can Weld provide benefits when integrated incrementally?
4. How does Weld compare to specialized systems for domains such as relational algebra, linear algebra, and graph analytics?

Unless otherwise noted, we ran experiments on an Amazon EC2 `r4.8xlarge` instance, with 16 Intel Xeon E5-2686v4 cores (32 hyperthreads) and 244GB of memory. We used LLVM 3.8 for compilation. Each result is an average of five runs, and all end-to-end Weld runtimes include Weld optimization and LLVM compilation times, sampling time, and data encoding and decoding. We present results on one and eight threads. We compare against NumPy v1.13.1, Pandas v0.19.2, TensorFlow v1.2, and Spark v2.2.

Table 4: Workloads used in our evaluation. An operator represents a single library API call (e.g., a pivot table construction or groupby in Pandas or an element-wise sum or logarithm in NumPy).

Workload	Libraries	Description (# Operators)
Data Cleaning [16]	Pandas	Cleans a DataFrame of 311 requests [53] by replacing `NULL`, broken, or missing values with `NaN`. (8)
Crime Index	Pandas NumPy	Computes an average "crime index" score, given per-city population and crime information. (16)
Black Scholes	NumPy	Computes the Black Scholes [21] formula over a set of vectors. (19)
Haversine	NumPy	Computes Haversine Dist. [26] from a set of GPS coordinates to a fixed point. (18)
N-Body	NumPy	An n-body simulation that uses Newtonian force equations to determine the position/velocity of stars over time. (38)
Birth Analysis [35]	Pandas NumPy	Given a dataset of number of births by name and year, computes the proportion of names starting with "Lesl" grouped by gender and year-of-birth. (12)
MovieLens	Pandas NumPy	Given the MovieLens dataset [25], finds the movies that are most divisive between male and female viewers. (13)
Log. Reg.	NumPy TensorFlow	Whitens and normalizes MNIST [41] images in NumPy, and then evaluates a logistic regression classifer on the whitened images in TensorFlow. (14)
NYC Filter	Pandas	Counts the number of taxi rides [45] which occur outside of Manhattan, have zero cost, and zero distance. (12)
Flight Delays	Pandas NumPy	Computes the mean delay, the unique tail numbers, the unique carriers, and the total number of flights [19] from any of four airports in NYC to Seattle. (13)

8.1 Workloads and Datasets

We evaluate Weld primarily on ten real data science workloads we found from various online sources such as tutorials or cookbooks for specific libraries, popular GitHub repositories, and Kaggle competitions (Table 4). Each workload uses one or more of our ported libraries (§7). We ran the workloads with no code changes (modulo importing our versions of the libraries) when possible, except for two workloads where we also evaluated adding a grouped evaluation call across multiple results (§2.5): Black Scholes and Flight Delays.

8.2 End-to-End Performance

In this section, we study the end-to-end performance of our ten workloads using Weld compared to the native versions of the libraries. Because NumPy and Pandas are single-threaded, we report results on both one and eight threads, to show that Weld improves

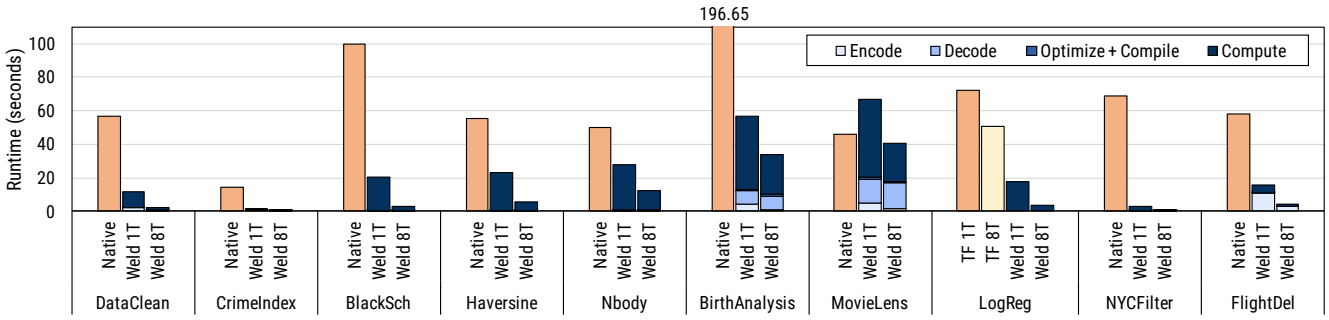


Figure 4: Performance of each workload on 1 and 8 threads, compared vs. native libraries. Weld's compile time includes data sampling.

single-thread efficiency too. §8.3 details the optimizations that impact performance in each workload.

NumPy Workloads: Black Scholes, Haversine, and N-Body. These workloads perform numerical vector computations using NumPy. Figure 4 shows the results. Overall, Weld's optimizer improves performance over native NumPy by 2.5–5× across the three workloads, even though individual operators in NumPy are implemented in C and BLAS [34]. In Black Scholes and Haversine, Weld fuses every vector math operation into a single loop to reduce memory allocation and data movement. On a single thread, we found this improves performance by reducing page faults caused by demand paging [17] by removing allocation of intermediate results. N-Body uses NumPy's indexing features to set the diagonal of a matrix to 0; this forces Weld to evaluate a partial computation and prevents fusion of loops that occur before and after the index operation. In all workloads, NumPy implements vector operations using L1 and L2 BLAS calls, but Weld still accelerates them by reducing data movement. Weld automatically parallelizes each workload as well, increasing the speedup over the single-threaded NumPy versions to 4–30×. Data encoding time in these workloads is negligible, since NumPy internally represents vectors as C arrays and encoding only involves a pointer copy.

Pandas Workloads: Data Cleaning and NYC Filtering. These workloads use Pandas to filter, normalize and clean a DataFrame by dropping `NULL` values, selecting values that pass predicates, and replacing broken values (e.g., short zip codes) with placeholders. These workloads represent common preliminary tasks seen in data science and are often bottlenecked by memory movement.

Although most individual Pandas operators are implemented in C, NumPy or Cython [15], Weld still provides speedups as shown in Figure 4. The speedups come from fusing every loop into a single one: the NYC workload in particular benefits greatly from lazy evaluation and fusion because the final output is a scalar, so after loop fusion Weld does not need to allocate any extra memory. The NYC workload shows a 23× speedup on one core, while the data cleaning workload shows a 5× speedup. Weld also provides further transparent multicore speedups over Pandas on eight threads.

Multi-Library Workloads. The remaining data science workloads from §8.1 combine operators from multiple libraries. Figure 4 shows the results for each workload, compared against its native implementation. We describe each result in more detail below.

Birth Analysis and MovieLens: These workloads construct a pivot table and filter columns using Pandas in a preprocessing stage, and then compute aggregate statistics using NumPy in an analysis phase. Weld accelerates the birth analysis workload by fusing loops in the analysis portion. The MovieLens workload is dominated by hash table performance, which is similar in both Pandas and Weld (both use an optimized C hash table). Weld does not speed up the pivot table construction substantially for this reason. Weld matches the *runtime* performance of Pandas here on one thread, showing that its general IR can capture complex workloads effectively. However, Weld performs worse than Pandas *end-to-end* on the MovieLens workload due to decoding time, since C-strings returned by Weld must be marshalled into Python strings. Furthermore, since the Python C API is not thread-safe, the decoding step cannot be parallelized on eight threads. In addition, Weld's merging of thread-local dictionaries into a final result is also serial. This is an area for future improvement and can improve scalability. Overall, Weld accelerates the end-to-end birth analysis workload by 3.5× and is within 25% on MovieLens on one thread. Weld speeds up MovieLens by 1.25× on eight threads, with 40% of the end-to-end time spent in the *serial* decode step. Efforts toward common data layouts such as Apache Arrow [5] can prevent expensive data conversion steps.

Flight Delay and Crime Index: The flight delay workload produces multiple results, each of which are computationally inexpensive to compute (e.g., a filtered column using Pandas or a mean using NumPy). Weld again fuses every loop in the workloads here, producing a single pass over the input to compute multiple results. Weld accelerates this workload by 3.7× on one thread and 14× on eight threads. Weld improves performance of the crime index workload for similar reasons by fusing the dot products used to compute the crime index into a single loop and using vectorization, by 9× and 32× on one and 8 threads respectively.

Logistic Regression: This workload, which combines NumPy and TensorFlow to normalize (whiten) images and evaluate a logistic regression model over them, shows a 4× performance improvement via Weld over NumPy and TensorFlow *with* XLA, on a single thread. This performance improvement increases to 13× over the native library implementations with eight 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 eight cores, the speedup is due to parallelizing the whitening computation; in the native library implementation of the workload with eight 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; we omit them for brevity.

Optimization and Sampling Times. Weld's optimization times (including IR optimization and LLVM code generation) ranged from 62ms to 257ms (mean 126ms) across all workloads. The overhead of adding sampling for adaptive predication similarly ranged from 100–250ms. Since we expect real analytics workloads to run for many seconds to minutes (as in our experiments), we believe that these times are acceptable. Our optimizer is thus able to produce high-quality optimization decisions quickly in ad-hoc workloads with no statistics.

Experiment	All	-Fuse	-Unrl	-Pre	-Vec	-Pred	-Grp	-ADS	-CLO
DataClean	1.00	1.82	1.05	1.06	1.05	1.05			
CrimeIndex	1.00	15.86	3.17	1.02	1.01	1.01			2.95
BlackSch	1.00	2.72		1.00	1.85		1.51		
Haversine	1.00	2.19		1.06	1.01	1.01			
Nbody	1.00	1.69		1.50	1.12	1.02			
BirthAn	1.00	1.07		1.05	0.98				1.00
MovieLens	1.00	1.07		1.00	0.98				1.01
LogReg	1.00	2.51		0.99					1.25
NYCFilter	1.00	11.21		0.99	1.40	4.45			
FlightDel	1.00	2.02		1.01	1.00	1.00	4.59		2.16
NYC-Sel	1.00	62.68	0.99		1.03	0.99			
NYC-NoSel	1.00	10.27	0.99		1.60	1.49			
Q1-Few	1.00	1.38							
Q1-Many	1.00	1.08							
Q3-Few	1.00	1.23							
Q3-Many	1.00	1.10							
Q6-Sel	1.00	1.45	0.97	0.95	1.00	1.05			
Q6-NoSel	1.00	10.00	1.01	1.02	2.69	2.49			

Figure 5: Slowdown from removing each optimization (while applying all others) on one thread. Fuse = Fusion, Unrl = Loop Unrolling, Pre = Preallocation, Vec = Vectorization, Pred = Adaptive Predication, Grp = Grouped Eval, ADS = Adaptive Data Structures, CLO = Cross-Library Optimization.

Experiment	All	-Fuse	-Unrl	-Pre	-Vec	-Pred	-Grp	-ADS	-CLO
DataClean	1.00	2.44	0.97	0.99	0.98	0.95			
CrimeIndex	1.00	195	2.04	1.00	1.02	0.96			3.23
BlackSch	1.00	6.68		1.44	1.95		1.64		
Haversine	1.00	3.97		1.20	1.02				
Nbody	1.00	1.78		2.22	1.01				
BirthAn	1.00	1.02		0.97	0.98				1.00
MovieLens	1.00	1.07		1.02	0.98				1.09
LogReg	1.00	20.18		1.00					2.20
NYCFilter	1.00	9.99		1.20	1.23	2.79			
FlightDel	1.00	1.27		1.01	0.96	0.96	5.50		1.47
NYC-Sel	1.00	32.43	1.29		0.96	0.93			
NYC-NoSel	1.00	6.16	1.02		1.26	1.17			
Q1-Few	1.00	2.60						3.75	
Q1-Many	1.00	1.13						1.12	
Q3-Few	1.00	1.86						2.56	
Q3-Many	1.00	1.10						0.97	
Q6-Sel	1.00	1.45	1.00	1.00	0.99	0.98			
Q6-NoSel	1.00	10.04	0.99	0.99	2.44	2.66			

Figure 6: Slowdown from removing each optimization on 8 threads.

8.3 Effects of Individual Optimizations

In this section, we study the effects that *individual* optimizations have on the runtime of the ten workloads. We specifically consider the optimizations discussed in §4–6: fusion, vectorization, loop unrolling, buffer preallocation, the runtime API's grouped evaluation feature (§2.5), adaptive predication, adaptive data structures, and cross-library optimization. For each workload, we turn each optimization off *one at a time* and measure its performnace impact.

Figure 5 shows the results of this study on a single thread. Each box shows the relative slowdown of turning the optimization off, compared to having all optimizations on (i.e., numbers close to 1.0 mean the optimization had little effect, while larger numbers mean a large effect). Blank entries mean the optimization did not apply, and colors show the scale of the effect. Figure 6 shows the same experiment with eight threads. To show the effects of adaptive predication and data structures, we also ran additional versions of some workloads with different selectivity shown below the line (the Sel and NoSel versions), as well as several TPC-H queries where we varied selectivity and number of keys; we discuss these below.

Fusion: The fusion transformations have the most impact across our workloads. This shows that optimizing memory allocation and data movement has a substantial cost in these workloads. Fusion optimizations affect every real workload except MovieLens (which is dominated by one operation) by up to 15× on one thread. Optimizing memory movement is especially important on on eight threads, where there is less memory bandwidth per thread. Data science libraries will thus need some form of cross-operator fusion to achieve optimal performance, and would benefit greatly from lazy runtime APIs like Weld's (indeed, many newer libraries use lazy APIs and optimizers on top of them to tackle this problem [1,4]).

Vectorization: Vectorization also shows significant impact in several compute-heavy workloads. In some cases, vectorization can only be applied if predication is used, which requires adaptivity.

Grouped Evaluation: This API feature takes lazy evaluation allows users to submit multiple results to evaluate at once at the cost of requiring a small code change. It helped in two workloads where, without grouping, Weld had to recompute a common subexpression that would otherwise be shared across the evaluation of two results.

Preallocation, Loop Unrolling: These optimizations impact CPU efficiency by reducing memory allocations or other overheads. They help most in NumPy workloads with fixed-size arrays.

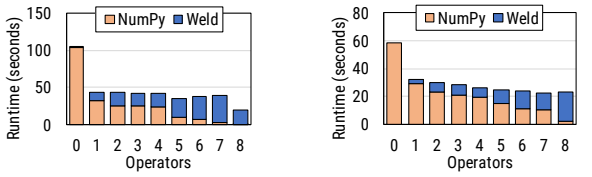
Adaptive Predication: To show the effects of adaptive predication on different selectivities, we ran the NYC Taxi workload on synthetic data with both high and low selectivity. Turning adaptive predication off always runs the branched version of the code. The `NYC-Sel` and `NYC-NoSel` entries in Figures 5 and 6 show the results. In `NYC-Sel`, most data is filtered by the first predicate in the workload, so adaptive predication chooses not to predicate the code and removing adaptive predication does nothing. `NYC-NoSel` shows the opposite effect: the filter always passes, so removing adaptive predication (and thus vectorization) results in a slowdown over the adaptively chosen plan, which would be to apply predication. We also show the same experiment on TPC-H Q6, which performs a number of filters and performs an aggregation, to the same effect.

Adaptive Data Structures: We ran a similar benchmark to show the effect of a non-adaptive dictionary on modified versions of TPC-H Q1 (an aggregation) and TPC-H Q3 (which contains a hash join). We again ran two versions of each experiment, varying the number of unique keys in the hash table. Figure 6 shows the results (`Q1-Few/Many` and `Q3-Few/Many`, referring to the number of distinct keys). The `Few` setting used 1024 distinct keys, and the `Many` setting used 2^{28} keys. We used 2^{31} total records in all cases. The comparison in Figure 6 is between the adaptive dictionary and the worse of the two simple dictionary implementation strategies: global and thread-local. We observe up to a 3.75× performance difference between the adaptive dictionary and the worst alternative.

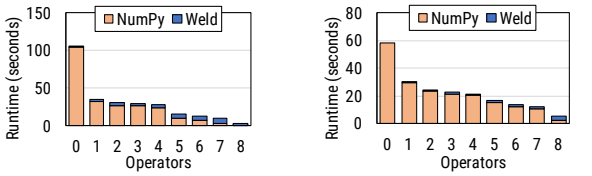
Cross-Library Optimizations: We evaluate the effect of cross-library optimizations (CLOs) by forcing Weld to evaluate computations at library boundaries. Disabling CLO for Crime Index, Flight Delays, and Logistic Regression both prevents loop fusion and causes Weld to allocate temporary buffers for intermediate values, resulting in slowdowns. Birth Analysis and MovieLens predominately use Pandas operators, and only use NumPy for a small subset of the data after calling a top-K operator in Pandas over the pivot table columns. These workloads are also primarily bottlenecked by hash table operations in Pandas, so disabling CLO has a relatively small effect. Overall, the results show that CLO provides up to 3× *further* speedups even after optimizing computations in individual libraries.

8.4 Incremental Integration

To show that Weld can be integrated into libraries incrementally, we ran the Black Scholes and Haversine workloads and incremen-



(a) 1 Thread Black Scholes **(b)** 1 Thread Haversine Distance



(c) 8 Thread Black Scholes **(d)** 8 Thread Haversine Distance

Figure 7: Incremental Integration on two of the NumPy workloads.

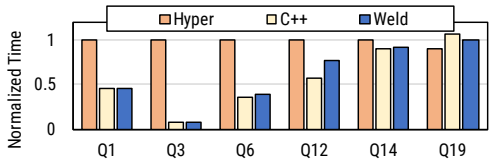
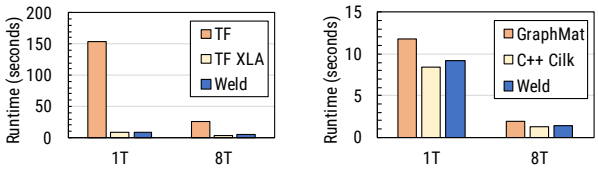


Figure 8: TPC-H microbenchmarks on 8 threads, compared against Hyper and handwritten C++ code using OpenMP for threading.



(a) Logistic Regression **(b)** PageRank

Figure 9: Specialized systems for linear algebra and graphs.

tally integrated one operator at a time to use Weld. Operators that were not Weld-enabled used native NumPy. The workloads both use eight unique operators; we ported these one by one to Weld in order of which operator took the most CPU cycles in each workload.

Figure 7 shows the results. On one thread, implementing the first operator in Weld gives a 1.5× speedup, and implementing half the operators gives a 2.5× speedup over native NumPy, by providing vectorized implementations for functions that NumPy runs sequentially. Interestingly, adding operators in Black Scholes slightly regressed performance in two cases, due to an extra `Evaluate` call, which causes some recomputation. On eight threads, the speedups at each step are significantly higher because Weld accelerates even the operators that can't yet be fused with other nearby ones by multithreading them. These results indicate that library developers can add Weld incrementally into the most widely used operators to start enjoying speedups without fully porting their libraries.

8.5 Comparison to Specialized Systems

We also evaluated our optimizer's code generation on x86 CPUs by comparing performance on several workloads against several state-of-the-art, domain-specific compilers and systems.

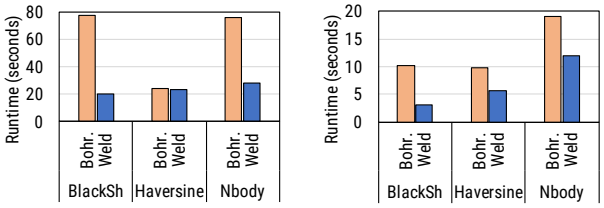
TPC-H Queries. Figure 8 show the results for a subset of the TPC-H queries (scale factor 10) on eight threads, compared against the HyPer [42] database and a hand-optimized C baseline using Intel's AVX2 intrinsics for vectorization and OpenMP [46] for parallelization. We chose these queries because they cover the major join types, scans, predicates, and aggregations and do not include complex string operations. HyPer generates LLVM IR for SQL queries, which is then compiled to machine code before execution. For Weld, we used the same physical plan as HyPer from its web interface [28] but wrote each operator in Weld. Execution time was competitive with HyPer across the board. Weld outpeformed HyPer on Q6 and Q12 because it applied predication and generated vectorized LLVM code. Results on one thread were similar.

Linear Algebra. Figure 9a shows Weld's performance on training a binary logistic regression classifier on the MNIST [41] dataset, classifying each digit as either zero or non-zero. We compare against TensorFlow with and without its specialized XLA compiler [59]. Unlike the logistic regression example in §8.2, the computation here is fully exposed to XLA for optimization, allowing it to achieve its best performance. Weld and XLA outperform standard TensorFlow by fusing operators, but perhaps surprisingly, Weld also matches XLA's performance even though XLA is built specifically for TensorFlow's linear algebra operators. We also compared Weld against Bohrium, an open-source lazy NumPy. Figure 10 shows the results; Weld outperforms Bohrium by vectorizing more operators and using grouped evaluation to eliminate redundant computation.

Finally, we benchmarked Weld's performance on dense matrix multiplication (DMM). This workload is both compute-bound and heavily optimized by existing linear algebra libraries. We compare Weld's generated code against a C baseline and MKL [40], the fastest DMM implementation we are aware of on Intel CPUs. We implemented the C and Weld kernels using (*i*) naive, triply nested loops and (*ii*) blocked loops for improved cache performance. The MKL implementation is proprietary but highly optimized and written in assembly to control aspects such as instruction scheduling. On a 8192×8192 matrix, we found that Weld's generated code matches C, but MKL outperforms both even against the blocked algorithm by 10×. Weld imposes no overhead when calling MKL as a UDF compared to a C program calling MKL. In short, like in systems designed for linear algebra [1, 14], users should call existing DMM kernels by using UDFs rather than relying on code generation to achieve the best performance.

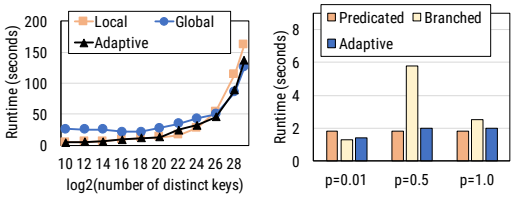
Graph Analytics. Figure 9b shows results for a PageRank implementation in Weld, compared against the GraphMat [58] graph processing framework. GraphMat had the fastest multicore PageRank implementation we found. Weld's per-iteration runtime for both serial and parallel code outperforms GraphMat, and is competitive with a hand-optimized Cilk [7] based C++ baseline.

Spark SQL. To illustrate Weld's benefits in a distributed framework, we evaluate a partial integration of Weld in Spark SQL's execution engine. Spark SQL natively generates Java bytecode, and uses a HyPer-like code generation process to pipeline code from different operators. In our Weld integration, we updated each Spark SQL operator to emit single Weld IR fragment encapsulating its computation, and relied on Weld's optimizer to fuse and co-optimize these fragments (§7). We tested the 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). We chose these two queries because they only contain scans, filters, and aggregations that our current Weld integration supports. Data was read from Spark's in-memory cache. We observed that Weld provides a 6.2× speedup for TPC-H Q1 and 6.5× for Q6, with Weld's speedup coming largely from its ability to generate low-level, vectorized x86 code, since the JVM did not vectorize Spark's code.



(a) 1 Thread **(b)** 8 Threads

Figure 10: Performance of Weld vs. Bohrium.



(a) Adaptive Dictionaries **(b)** Adaptive Predication

Figure 11: 11a. Performance of TPC-H Q1 with three different dictionary implementation strategies. The experiments all used 16 threads. **11b.** adaptively predicating the branch in TPC-H Q6.

Memory Usage. In each experiment, Weld's runtime memory usage matched the memory usage of the optimized C baseline implementations. In summary, Weld produces machine code competitive with existing systems on a variety of workloads, demonstrating both the expressiveness of its IR and the effectiveness of our optimizer.

8.6 Adaptive Optimization Microbenchmarks

In this section, we evaluate the quality of the adaptive transformations across a variety of inputs. For the dictionary, we plotted the runtime of a global-only, local-only, and adaptive dictionary implementation while varying the number of distinct keys in TPC-H Q1. We used 2^{31} total records in all cases. Figure 11a shows our results. In all cases, the adaptive dictionary achieves the performance of the better of the two dictionary strategies.

Figure 11b plots the runtime of a predicated, branched, and adaptively predicated version of TPC-H Q6 at various selectivities. Once again, adaptive predication matches the better of the branched and predicated strategies in the three cases. As the selectivity of the branch increases, the adaptive optimizer determines that evaluating a conditional branch has a lower cost than unconditionally evaluating both branch targets, and disables predication.

9. RELATED WORK

Both Weld and our optimizer build on ideas in several fields, including database optimizers, compilers, and hardware-efficient data processing. Unlike most existing systems, however, our setting is different because we aim to optimize across *diverse, independently written libraries* while retaining their user facing API. This required an optimizer that can identify and eliminate the redundancies in operations called by imperative APIs like NumPy and Pandas, and that runs quickly enough to be usable in ad-hoc interactive analytics.

RDBMS engines like HyPer [42], LegoBase [32], DBLAB [56] and Voodoo [49] perform runtime code generation, but these systems only support SQL workloads. Tupleware [12] is closer to our domain in that it also integrates LLVM-based UDFs into parallel workflows and includes a cost model for optimizing pipelining, vectorization and selection. However, Tupleware represents UDFs as sequential LLVM IR code and does not support more complex optimizations such as fusion of parallel loops, which are essential for Weld's setting. It also does not aim to integrate underneath existing, imperative analytics libraries. Work on analyzing UDFs in MapReduce [27] likewise does not consider parallel UDFs or integration under existing libraries.

OpenCL [57], CUDA [13] and SPIR [31] are languages for targeting parallel hardware that let users submit functions to run in parallel, but these systems do not aim to optimize across function invocations. Many existing libraries, such as NumPy, TensorFlow and Pandas, implement their operators in these languages or in C or Cython [15]. Nonetheless, our evaluation shows that Weld can significantly accelerate these libraries by optimizing across operators. Systems like XLA [59] and Bohrium [8] have also built cross-operator compilers but are limited to a single library.

Weld's IR is closest to monad comprehensions [10, 23] and Delite's multiloop construct [9,54], both of which support nested parallel loops that produce multiple results and loop fusion. However, to our knowledge, Weld is the first system to provide an adaptive optimizer for this type of IR and to evaluate integrating it under existing analytics libraries. Unlike Delite, Weld is also the first such system designed for incremental integration into existing libraries, and the first to evaluate evaluates optimizations *across* different libraries. LINQ [60], Emma [3] and NESL [6] also use functional or relational IRs but do not provide adaptive optimizers.

Finally, our optimizer leverages ideas from both databases (e.g., adaptive optimization [18] and memory cost models [29, 38]) and compilers [30,33], but adapts them to the novel setting of optimizing disparate, existing data analytics libraries. This setting creates new challenges including identifying and eliminating the highly redundant code produced by various libraries (e.g., through pipelining and horizontal fusion), efficiently representing and manipulating fused code (through Weld's IR), and making data-dependent optimizations in ad-hoc workflows with minimal time to collect statistics at runtime (through our adaptive optimizations). Our resulting design effectively accelerates real-world workloads while incurring minimal overhead for optimization, compilation and sampling.

10. CONCLUSION

In this paper, we showed that order-of-magnitude speedups are possible by optimizing across imperative data analytics libraries, and presented an optimizer that can achieve these speedups in Weld, a common runtime that can be added incrementally into existing libraries. Our optimizer includes rule-based and adaptive optimizations that work within and across popular libraries like Pandas, NumPy, TensorFlow and Spark SQL, and accelerates workloads by up to 80× on 8 threads with sub-second optimization overhead. We also showed that adding Weld *incrementally* into libraries still yields large speedups, quantified which optimizations affect each workload, and showed that our Weld optimizer is competitive with domain-specific systems in many cases despite its more general IR.

11. ACKNOWLEDGEMENTS

We thank the members of the Stanford DAWN lab for their invaluable feedback on this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project (Facebook, Google, Intel, Microsoft, NEC, Teradata, and VMware), by Amazon Web Services, and by NSF CAREER grant CNS-1651570 and NSF Graduate Research Fellowship 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.

12. REFERENCES

[1] 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*, pages 265–283, 2016.

[2] S. Agarwal, D. Liu, and R. Xin. Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop. `https://databricks.com/blog/2016/05/23/`, 2016.

[3] 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.

[4] 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*, pages 1383–1394, 2015.

[5] Apache Arrow. `https://arrow.apache.org/`, 2018.

[6] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-parallel Language. *SIGPLAN Not.*, 28(7):102–111, 1993.

[7] 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*, 37(1):55–69, 1996.

[8] Bohrium. `http://bohrium.readthedocs.io`, 2018.

[9] 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*, CGO 2016, pages 194–205. ACM, 2016.

[10] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1):87–96, March 1994.

[11] Cran. `https://cran.r-project.org`, 2018.

[12] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12):1466–1477, 2015.

[13] CUDA. `http://www.nvidia.com/object/cuda_home_new.html`, 2018.

[14] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A Science-oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[15] Cython. `http://cython.org`, 2018.

[16] Pandas Cookbook Chapter 7: Cleaning Up Messy Data. `https://github.com/jvns/pandas-cookbook/`.

[17] Demand Paging. `https://en.wikipedia.org/wiki/Demand_paging`, 2018.

[18] A. Deshpande, Z. Ives, V. Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[19] Flight Delays and Cancellations Dataset. `https://www.kaggle.com/usdot/flight-delays/data`.

[20] Gluon. `https://gluon.mxnet.io`.

[21] J. Goseme. Black Scholes Formula, 2013.

[22] G. Graefe. *Encapsulation of Parallelism in the Volcano Query Processing System*, volume 19. ACM, 1990.

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

[24] J. Hamrick. The Demise of for Loops. `https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html`.

[25] F. M. Harper and J. A. Konstan. The Movielens Datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[26] S. Heisler. A Beginner's Guide to Optimizing Pandas Code for Speed. `goo.gl/dqwmrG`, 2017.

[27] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J.-C. Freytag. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1292–1295. IEEE, 2013.

[28] HyPer Web Interface. `http://hyper-db.de/interface.html`, 2013.

[29] A. Kemper, F. Funke, H. Pirk, S. Manegold, U. Leser, M. Grund, T. Neumann, and M. Kersten. Cpu and cache efficient management of memory-resident databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 14–25, Washington, DC, USA, 2013. IEEE Computer Society.

[30] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer, 1993.

[31] J. Kessenich. An introduction to SPIR-V. `https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf`, 2015.

[32] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-level Language. *PVLDB*, 7(10):853–864, 2014.

[33] 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*, pages 75–86, 2004.

[34] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[35] W. Liu. Python and Pandas Part 4: More Baby Names. `http://beyondvalence.blogspot.com/2014/09/python-and-pandas-part-4-more-baby-names.html`, 2014.

[36] Loop Unrolling. `https://www.cs.umd.edu/class/fall2001/cmsc411/proj01/proja/loop.html`, 2001.

[37] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[38] S. Manegold, P. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 191–202. VLDB Endowment, 2002.

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

[40] Intel Math Kernel Library. `https://software.intel.com/en-us/mkl`, 2018.

[41] MNIST. http://yann.lecun.com/exdb/mnist/.

[42] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[43] NumPy. http://www.numpy.org/.

[44] NumPy Array Indexing. https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html, 2009.

[45] NYC Taxi Dataset. https://cloud.google.com/bigquery/public-data/nyc-tlc-trips.

[46] OpenMP. http://openmp.org/wp/.

[47] 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)*, pages 293–307, 2015.

[48] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia. Weld: A Common Runtime for High Performance Analytics. In *CIDR*, 2017.

[49] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo-A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.

[50] Pypi. https://pypi.python.org, 2018.

[51] Pytorch. http://pytorch.org, 2018.

[52] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory Scan Sharing for multi-core CPUs. *PVLDB*, 1(1):610–621, 2008.

[53] 311 Service Requests Dataset. https://github.com/jvns/pandas-cookbook/blob/master/data/311-service-requests.csv.

[54] 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.

[55] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[56] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1907–1922, New York, NY, USA, 2016. ACM.

[57] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[58] 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. *PVLDB*, 8(11):1214–1225, 2015.

[59] TensorFlow XLA. https://www.tensorflow.org/performance/xla/, 2018.

[60] 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*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[61] 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*, 59(11):56–65, October 2016.