Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels

Tong Zhou
School of Computer Science
Georgia Institute of Technology
USA
tz@gatech.edu

Jun Shirako
School of Computer Science
Georgia Institute of Technology
USA
shirako@gatech.edu

Anirudh Jain
School of Computer Science
Georgia Institute of Technology
USA
anirudh.j@gatech.edu

Sriseshan Srikanth
AMD Research*
USA
seshan@gatech.edu

Thomas M. Conte
School of Computer Science
Georgia Institute of Technology
USA
conte@gatech.edu

Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology
USA
richie@cc.gatech.edu

Vivek Sarkar
School of Computer Science
Georgia Institute of Technology
USA
vsarkar@gatech.edu

Abstract

Major simultaneous disruptions are currently under way in both hardware and software. In hardware, "extreme heterogeneity" has become critical to sustaining cost and performance improvements after Moore's Law, but poses productivity and portability challenges for developers. In software, the rise of large-scale data science is driven by developers who come from diverse backgrounds and, moreover, who demand the rapid prototyping and interactive-notebook capabilities of high-productivity languages like Python.

We introduce the Intrepydd programming system, which enables data scientists to write application *kernels* with high performance, productivity, and portability on current and future hardware. Intrepydd is based on Python, though the approach can be applied to other base languages as well. To deliver high performance, the Intrepydd toolchain uses ahead-of-time (AOT) compilation and high-level compiler optimizations of Intrepydd kernels. Intrepydd achieves portability by its ability to compile kernels for execution on different hardware platforms, and for invocation from Python or C++ main programs.

An empirical evaluation shows significant performance improvements relative to Python, and the suitability of Intrepydd for mapping on to post-Moore accelerators and architectures with relative ease. We believe that Intrepydd represents a new direction of "Discipline-Aware Languages" (DiALs), which brings us closer to the holy grail of obtaining productivity and portability with higher performance than current Python-like languages, and with more generality than current domain-specific languages and libraries.

CCS Concepts: • **Software and its engineering → Context specific languages**.

Keywords: Python, compilers, application kernels

* The work for this paper was done while at Georgia Institute of Technology.

Onward! '20, November 18–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8178-9/20/11…$15.00
https://doi.org/10.1145/3426428.3426915

ACM Reference Format:
Tong Zhou, Jun Shirako, Anirudh Jain, Sriseshan Srikanth, Thomas M. Conte, Richard Vuduc, and Vivek Sarkar. 2020. Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '20), November 18–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3426428.3426915

1 Introduction

This paper addresses the problem of how to advance programming systems in the face of two shifts toward heterogeneity, one in hardware and the other in software. In hardware, *what* we need to program is changing. Previously, we enjoyed the "free ride" of automatic improvements in the efficiency and cost of general-purpose computing that came from Moore's Law and Dennard Scaling, which demanded few if any changes to the hardware model or software interface. In contrast, future improvements will require specialized, post-Moore accelerators and radically different computer architectures [33]. In software, *who* is programming is also changing. Software development is no longer the exclusive activity of computer science professionals trained in software engineering techniques and tools. Rather, a diverse mix of "citizen developers,"[1] who span physical science, engineering, business, medicine, social science, and the humanities, are engaging more and more with computing, armed with data, their domain knowledge, and tools from statistical data analysis, text mining, graph analytics, image and signal processing, computer vision, natural language, and machine learning. These simultaneous trends are hastening the search for a "unicorn" programming system that can deliver sustained productivity and performance for the future.

The current approach is to start with general-purpose, productivity-oriented languages such as Python [2], R [3], or Julia [28], and augment them with high-performance implementations of commonly used functions in native libraries, e.g., functions in the NumPy [6, 71] and SciPy [5] libraries for Python. However, fixed library interfaces and implementations do not adapt well to the needs of new applications. For example, Google and Facebook are changing the standard interfaces of linear algebra for scientific computing to better match the data structures and numerical precisions of machine learning [36, 37, 75], as is the graph algorithms community for combinatorial graph primitives [34, 48].

Domain Specific Languages (DSLs) show promise within their target domains, e.g., PyTorch and TensorFlow for neural networks [8, 15], Halide for image processing computations [62], GraphIt for graph computations [77], and TACO for tensor kernels [49]. However, it is widely appreciated that the decision to develop or adopt a DSL involves a complex set of tradeoffs in productivity, integration into existing software systems, extensibility, and ultimately, development costs [57]. This state of affairs suggests that, in practice, developers will need a spectrum of solutions including, but not limited to, DSLs.

Within that context, our proposed approach is to broaden the scope of domain specific languages to *Discipline-Aware Languages* (DiALs), as illustrated in Figure 1. Specifically,

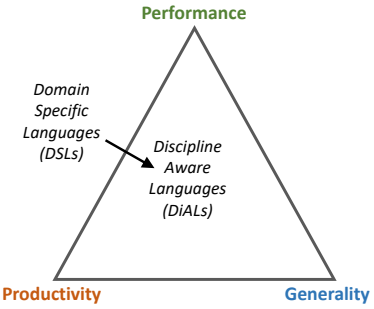

Figure 1. Moving from Domain Specific Languages (DSLs) to Discipline Aware Languages (DiALs)

we introduce a data-scientist-oriented programming model, Intrepydd, which aims to deliver high performance, productivity, and portability for writing kernels in different domains in the data science discipline while also making it possible to easily target post-Moore hardware. Intrepydd's design is based on Python syntax, given its broad popularity.[2] However, Intrepydd differs from Python in significant ways. First, it is a language for only writing computational kernels, and not for writing complete applications. Second, code generated from Intrepydd programs can be invoked from a Python application (with the `-host=python` mode) or from a C++ application (with the `-host=cpp` mode) thereby enabling a larger degree of *portability* than Python, especially for accelerators and newer hardware platforms that do not support the Python runtime environment. Third, Intrepydd is designed for ahead-of-time (AOT) compilation and high-level compiler optimizations of Intrepydd kernels, thereby delivering higher *performance* relative to base Python, and also to popular toolchains for AOT and JIT optimization of Python programs, such as Cython [7, 26] and Numba [11, 51] (Section 4). Finally, the primitives supported in Intrepydd are designed for data science applications, thereby delivering *productivity* but with more *generality* than is typically available in DSLs.

Intrepydd makes the following major contributions:

1. *Intrepydd language definition:* Intrepydd adopts Python syntax with a type system and language constructs designed for efficient AOT compilation. (Section 3)
2. *End-to-end programming system:* Our prototype tool chain supports C++ code generation for Intrepydd kernels with two different modes for the host language. (Section 4)
3. *Use of ahead-of-time compilation to enable high-level whole-module optimizations:* The Intrepydd compiler integrates extensions of well-known AOT optimizations including loop invariant code motion (LICM), fusion of element-wise operations on dense arrays, optimization of sparse array operations, optimization

[1] https://www.gartner.com/en/information-technology/glossary/citizen-developer

[2] After JavaScript, Python is the second most popular language used in GitHub projects [19].

of array allocations, and explicit parallelization via a `pfor` construct with automatic privatization analysis for scalar variables. (Section 5)

4. *Broad empirical evaluation:* We show significant performance improvements in single-core performance for Intrepydd relative to Python, Cython, and Numba, for six benchmark applications from the data analytics discipline. Furthermore, we demonstrate multicore scalability for four of the six benchmarks that use Intrepydd's `pfor` construct. We also compare the performance of Julia and Intrepydd on four benchmarks, three of which were drawn from a commonly used set of Julia microbenchmarks. (Section 6)
5. *Easy access to post-Moore accelerators/architectures:* Beyond mainstream multicore CPUs, we show that Intrepydd can also be used to target post-Moore accelerators and novel architectures with relative ease by using two examples: the memory-centric Emu Chick [39] migratory thread architecture and the MetaStrider sparse reduction accelerator [67]. (Section 7)

These results show the potential for advancing the state of the art in high productivity languages like Python and Julia by using the Intrepydd approach for computational kernels.

The importance improving the performance of Python-like high-productivity programming has been underscored by many, including in the 2018 Turing Lecture by John Hennessy[3] and David Patterson. The remainder of our paper elaborates on aspects of the Intrepydd vision that have already been realized toward such a goal and outlines where we hope to go next.

2 Background

In this section, we briefly discuss background information on frameworks and tools in the Python ecosystem that are most relevant to this paper. We review related work in more detail in Section 8.

2.1 NumPy Arrays

NumPy is a core library for scientific computing in Python [6]. A NumPy array `ndarray` is an N-dimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its shape, which is a tuple of N non-negative integers that specify the sizes of each dimension. The array data is stored in a contiguous block of memory using primitive data types.

2.2 Cython

Cython is a compiled language that is typically used to generate CPython extension modules [7]. Annotated python-like code is compiled to C or C++ automatically wrapped in interface code, so as to produce extension modules that can be loaded and used by regular Python code using the import statement. Parallel execution of Cython code is limited by the Python global interpreter lock (GIL) unless the GIL is explicitly disabled. However, Python data structures, such as Lists and Dicts, cannot be modified when the GIL is disabled, making it necessary for the programmer to have working knowledge of C and C++ data structures, thereby reducing Cython's programmability. Unlike Numba (see below), Cython does not try to compile any library calls like NumPy or apply any fusing optimizations. Instead, it just invokes the Python interpreter to handle the calls.

2.3 Numba

Numba [11] is a Just-in-Time (JIT) compiler, implemented as a Python library, that translates a subset of Python and NumPy code into fast machine code. Python kernels that are to be accelerated are annotated with the `@jit` keyword, and, optionally, the type information for arguments and the return type. Numba, by default, follows a lazy compilation strategy where the kernel is compiled to machine code and interfaced with the Python code the first time it is called. Numba can operate in two modes, `object` mode, which handles all values as python objects by utilizing the Python C API, and `nopython` mode, which does not access the Python C API. The `nopython` mode produces more performant code. However, it requires that all types in the kernel be inferable by the Numba compiler. Numba's parallelism is limited by the same GIL issue as Cython.

2.4 Ahead-of-Time Compilation

Ahead-of-Time (AOT) compilation statically translates a source language program into a target language program prior to program execution. AOT compilation usually allows for longer compilation times than JIT compilation, and can thereby apply more complex and advanced program transformations. The rationale for this approach is that AOT compilation time is typically assumed to be amortized over multiple runs of the compiled program, whereas JIT compilation time is added to the execution time of each run. The classical drawbacks of AOT compilation relative to JIT compilation are two-fold: it is harder to exploit runtime information, and AOT's longer compile-times can affect productivity in interactive compile-debug scenarios.

2.5 Domain Specific Languages

A Domain-Specific Language (DSL) is a programming language specialized to an application domain. Within their target domains, DSLs provide higher-level abstractions with simpler syntax than general-purpose languages and enable high performance on multiple computer architectures. DSL compilers can perform high-level program optimizations by taking advantage of the domain-specific knowledge that is

[3] At 38m0s in the lecture, John Hennessy stated the following related to optimization of Python programs: "I've worked on compilers ... where a factor of 2x makes you a star ... if you get ... a factor of 23x, you'll be a hero."

specified by programmers (i.e., domain experts) via DSL language constructs. However, as mentioned earlier, the use of a DSL has multiple limitations when it needs to be integrated with existing software systems [57].

3 Language Definition

To address the goals and motivations discussed in Section 1, Intrepydd builds on an AOT-compilable subset of Python with extensions for matrix, tensor, graph, and deep learning computations that expose opportunities for high level compiler optimizations.

Intrepydd is intended for writing kernel functions, not complete or main programs, thereby limiting the compile-time overhead of AOT compilation. Instead, code generated from Intrepydd programs can be integrated within a Python application (with the `host=python` mode) or a C++ application (with the `host=cpp` mode).

A brief summary of the Intrepydd language definition is shown in Table 1. (Section 4 indicates which of these languages are fully supported by our prototype tool chain, and which are still in progress.) Intrepydd supports the following statement types, which are familiar to all Python and C/C++ programmers: assignment statements, return statements, sequential for and while loops with break and continue statements, conditional if / elif / else statements, and calls to user-defined and built-in functions. Intrepydd also supports the following operators:

- arithmetic operators:[4] `+`, `-`, `*`, `/`, `//`, `**`, `@`
- comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- logical operators: `and`, `or`, `not`
- membership operators: `in`

As indicated at the top of Table 1, an important difference between Intrepydd and Python is the requirement that function definitions include types for parameters and return values. This is one of the key features that enable AOT compilation. The following subsections discuss Intrepydd's key features in more detail.

3.1 Data Types

Intrepydd currently supports the following data types, selected for their suitability for data analytics applications and efficient AOT compilation:

- **Boolean types.** A Boolean (`bool`) value can be either `True` or `False`.
- **Numeric types.** 32/64-bit integers (`int32`, `int64`); 32/64-bit floating-point numbers (`float32`, `float64`).
- **Sequence types.** `List(type)` is a mutable sequence of homogeneously typed items designed for efficient execution when the sequence size can vary. Note that this is different from Python's List which can store heterogeneous items. `Array(type)` is a mutable indexed collection of homogeneously typed items designed for efficient execution when the sequence size is fixed.
- **Sparse Types.** A SparseMat (`SparseMat(type)`) is a 2-D sparse matrix of homogeneously typed items.
- **Array shapes**: As with NumPy arrays, `Array(type)` and `SparseMat(type)` have `shape` attributes that specify an array's rank and dimension sizes. For array `A`,

[4] The last three are floor-div, power, and matmult operators.

Table 1. Intrepydd Language Summary

Typed function definition	
def *func*(*prms*) -> *rt*: *body*	• *func*: Function name • *prms*: Parameters with types • *rt*: Type annotation for return value • *body*: Statements for function body
Optional type annotations for variable declarations	
var: *ty* *var*: **List**(*ty*) *var*: **Dict**(*ty*) *var*: **Array**(*ty*, *n*) *var*: **SparseMat**(*ty*) *var*: **Heap**(*ty*)	• *var*: Variable • *ty*: Data type annotation • List, Dict: List and hash table • Array: Dense *n*-dimensional array • SparseMat: Sparse matrix • Heap: Support push/pop/peek
Control flow statement	
for *tar* **in** *iterable*: *body*	• *tar*: Variable to store iterable item • *iterable*: List/Array of items to iterate • range(*st*, *ed*, *sp*) for numeric List • *body*: Statements for loop body
while *cond*: *body*	• *cond*: Condition to continue loop • *body*: Statements for loop body
if *cond*: $body_T$ **else**: $body_F$	• *cond*: Condition to be evaluated • $body_T$: Statements when *cond* is True • $body_F$: Statements when *cond* is False • else part is optional
break / continue	• Terminate loop / skip iteration
Parallel statement	
pfor *tar* **in** *iterable*: *body*	• Parallel loop • *tar*, *iterable*, *body*: Same as for loop
async: *body*	• Asynchronous task • *body*: Statements for task body
finish: *body*	• Finish statement • *body*: Statements for finish scope
isolated: *body*	• Isolated statement • *body*: Statements for isolated scope
Assignment and function call statement	
var = *expr*	• *var*: Variable to store result of *expr* • *expr*: Expresson to be evaluated • Conform to standard Python
expr	• *expr*: Expression (function call)

```
1 # Double each element of array A
2 pfor i in range(A.shape[0]):
3     temp = 2 * A[i]
4     A[i] = temp
```

Listing 1. An example of using pfor (parallel for)

functions `ndim(A)` and `shape(A,n)` are provided to obtain `A`'s dimensionality and the size of the `n`-th dimension, respectively. In addition, inter-node data distributions (as in the Chapel [30] and X10 [31] languages) can be specified in the `shape` attribute for mapping on to distributed-memory parallel machines.

- **Mapping Types.** `Dict(type)` maps a hashable key to a value. Currently, only boolean and numeric types are supported as keys. Any Intrepydd type is permitted as values.
- **Heap Types.** `Heap(type)` is a tree-based data structure in which the tree is a complete binary tree. Both Max-Heaps and Min-Heaps are provided. Heaps are commonly used in graph analytics.

3.2 Type Inference

The Intrepydd data types listed above are inferred automatically for local variables and expressions, based on the type declarations required for function parameters and return values. A simple form of type inference is performed by observing the following rules:

1. Type inference begins with a set of constants such as boolean or numeric constants.
2. Assignment unifies the types of its left- and right-hand side values.
3. Information discovered in the program later can be used to retroactively fill in placeholders in previously unresolved types.
4. The number of dimensions for dense and sparse arrays is inferred together with element type when statically analyzable.

If a variable's type cannot be inferred, the Intrepydd compiler will generate a compile-time error message requesting the programmer to add explicit type declarations for the variable by using Python's PEP 484 type annotation with the "`var: Type`" syntax. As mentioned in Table 1, these annotations are optional in general; some programmers may choose to add them for improved documentation. However, they can occasionally become necessary in the cases when type inference fails.

3.3 Parallel Statements

Intrepydd introduces a parallel-for (`pfor`) loop statement, which is not available in Python or C++. A simple example appears in Listing 1. Observe that sequential for-loops that are eligible for parallelization can be converted to parallel loops in Intrepydd by replacing "`for`" by "`pfor`". Doing so is far simpler than using Python's multiprocessing package [4] to execute loop iterations in parallel. Further, the programming system takes care of creating private copies of scalar variables like `temp`, without requiring the programmer to add special annotations (as is necessary in most parallel programming systems, e.g., OpenMP [59]).

The main conditions for a `pfor`-loop to be eligible for parallel execution are as follows.

- The loop should have no cross-iteration dependences on array variables. For example, if `2*A[i]` in the above loop is replaced by `2*A[i-1]`, the loop will no longer be eligible for parallelization since there can be a race condition between (say) the write of `A[0]` in iteration `i=0` and the read of `A[0]` in iteration `i=1`. The user must verify this condition.
- The loop should have no read-after-write (flow) cross-iteration dependences on scalar variables. Listing 1 has no cross-iteration dependences on scalar variable, temp, since the value of temp is written and read in the same iteration. The compiler will check this condition.

In addition, Intrepydd borrows the `async` and `finish` constructs for structured task parallelism from the X10 language [31], which are also not available in Python or C++. The `async` S_1 statement creates an asynchronous task that can execute S_1 in parallel with computations that follow the `async` statement. A `finish` S_2 statement performs all computations within S_2 and only completes when all of S_2's computations, including all `async` tasks created within S_2, have completed. Both `async` and `finish` statements can be arbitrarily nested. We assume that each Intrepydd function has an implicit finish scope for the function body so that all its asynchronous tasks must complete before the function can return. Finally, the `isolated` construct supports mutual exclusion among `async` tasks [41].

3.4 Library Functions

Intrepydd currently supports basic built-in functions and limited classes of numerical library functions, which will be extended in future versions. The majority of Intrepydd library functions are wrappers for standard Python libraries, such as NumPy [6], while the current implementations do not include much platform-specific optimization, as one might see in OpenBLAS or BLIS [21, 72].

As a part of Intrepydd language specifications, all library functions include type annotations and the def/use relations among input arguments and return variables. For element-wise operations and a class of linear-algebra functions, per-element dataflow and computational information are also provided as function summary information. For instance, 2-D matrix-matrix multiply `matmult` has the following summary:

- Arguments – A_1, A_2: Array(double, 2)
- Return – R: Array(double, 2)
- Dataflow – Use($A_1[i, *]$, $A_2[*, j]$) to Def($R[i, j]$)

Table 2. Intrepydd library functions for numerical computations

Reduction	
$b = reduce(A)$	• sum, prod, min, max, argmin, argmax, any, all, allclose, where • A: Array; b: Scalar
Element-wise unary function	
$B = unary(A)$	• abs, minus, isnan, isinf, not, sqrt, exp, cos, sin, tan, acos, asin, atan, copy • A, B: Array
Element-wise binary function	
$C = binary(A, B)$ $C = binary(A, b)$ $C = binary(a, B)$	• add, sub, mul, maximum, pow, div, log, eq, neq, lt, gt, le, ge, logical_and, logical_or, logical_not • A, B, C: Array; a, b: Scalar
Linear algebra	
$C = la(A, B)$ $C = la(A)$	• transpose, matmult (general), innerprod (1D×1D), matvect (2D×1D), vectmat (1D×2D), matmat (2D×2D), syrk, syr2k, symm, trmm, lu, ludcmp • A, B, C: Array
Sparse function	
$X = sparse(S, A)$ $X = sparse(S, a)$ $X = sparse(A, S)$ $X = sparse(a, S)$	• spm_add, spm_mul, spmv, spmm, spmm_dense (return dense array) • S: SparseMat; A: Array; a: Scalar • X: SparseMat or Array

- Computation – $R[i, j] = \Sigma_k A_1[i, k] \times A_2[k, j]$

The function summary information is used for type inference, dependence analysis, and static optimizations in the Intrepydd compiler (Section 5.2).

Table 2 shows a brief summary of Intrepydd library functions for numerical computations. Note that scalar versions of all element-wise functions are also supported, e.g., `b = sqrt(a)` for scalar variables `a` and `b`.

4 Compilation Pipeline

We have built an end-to-end compilation pipeline for Intrepydd, as illustrated in Figure 2. Since the Intrepydd language is based on Python, the toolchain for this pipeline is built on top of the Python Typed AST package [18]. In this prototype toolchain, the Intrepydd source file (.pydd) is parsed to the `typed_ast.ast3` Abstract Syntax Tree (AST) with type annotations, on which the following compiler analyses and optimizations are performed in the static optimization phase:

1. Type and dimensionality inference
2. Array operator conversion into library calls
3. Dataflow and dependence analysis
4. Loop invariant code motion (LICM, Section 5.1)
5. Dense/sparse function fusion (Section 5.2)

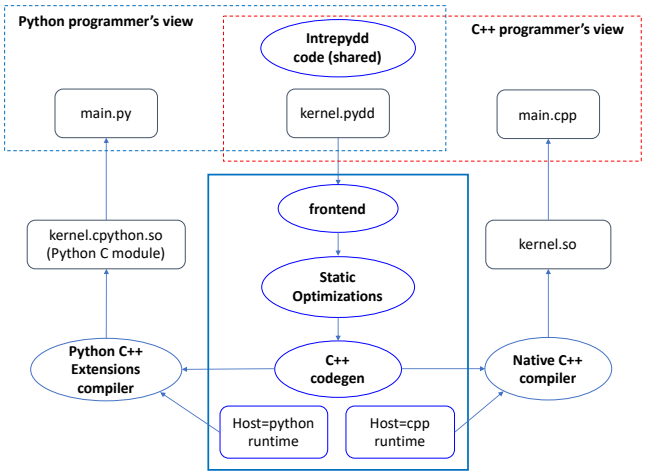

Figure 2. Design of the Intrepydd compilation pipeline

6. Array allocation and slicing optimization (Section 5.3)
7. Loop parallelization (Section 5.4)
8. LICM during C++ code generation (Section 5.1)

In Step 1, the type and dimensionality of local variables and expressions are inferred based on type annotations on the function parameters (Section 3.2). The conversion in Step 2 is performed on-the-fly as an extension to Step 1. Analogous to Python's magic methods [1], arithmetic, comparison, and logical operators can be performed on both scalar and array types. When type inference (Section 3.2) detects an array type in an operator's arguments, the operator is converted into its corresponding library calls (Section 3.4), e.g., arithmetic: `a + b` into `add(a, b)`, comparison: `a == b` into `eq(a, b)`, and logical: `a and b` into `logical_and(a, b)`.

The program dataflow and dependence analyses (Step 3) provide legality constraints for loop and statement reordering transformations including privatization analysis for parallelization, and the precision of these analyses depends on the precision of their underlying def/use analysis. Analogous to Python, the Intrepydd language has simple def/use semantics for local variables, while def/use information of library function arguments is represented as per-function dataflow relations as part of the language specification (Section 3.4). After Step 3, the toolchain performs a sequence of high-level code optimizations which are described in Sections 5.1–5.4. In the final step, C++ code is generated from the optimized AST, and compiled into a binary module that can be called by the host program.

4.1 Intrepydd Constructs Currently Unsupported by the Prototype Toolchain

Our prototype toolchain supports most, but not all, of Intrepydd's language constructs described in Section 3. In this section, we briefly summarize the Intrepydd constructs that are not supported by the current prototype used to obtain the experimental results reported in Section 6. However,

full support for these constructs is currently active work in progress:

- The following operators are currently not supported for arrays. The workaround that can be used right now is to manually convert these array operators into their corresponding library calls in the source code, instead of relying on the Intrepydd compiler to do the conversion.
  - arithmetic operators: `//` (floor-div), `**` (pow)
  - comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
  - logical operators: `and`, `or`, `not`
  - membership operators: `in`
- The toolchain currently uses the compressed sparse row (CSR) format [38, 65] for all SparseMat instances, though the language definition allows for the use of other formats as well.
- With respect to parallel constructs, the current prototype supports `pfor` in `host=cpp` mode , but support for `async`, `finish` and `isolated` is still in progress as is support for `pfor` in `host=python` mode.

4.2 Python Host

In this mode, the host program is written in Python and the `pydd` source file is eventually compiled to a Python C module which can directly be imported in the host program.

Intrepydd's data structures are backed by the `pybind11` framework [14], so all argument passing policies and limitations are inherited. For example, when passing arguments from Python code to Intrepydd code, Lists and Dictionaries are passed by value and NumPy arrays are passed by reference. This implies that a copy operation is performed when passing a List or a Dictionary from the Python code to the Intrepydd code. The same conversion also takes place when returning a List or Dictionary from Intrepydd to the Python host. Primitive types are also passed by value.

Third-party Python packages other than NumPy are not yet supported. A data structure from these packages must be convertible to NumPy arrays to be accessible in Intrepydd code. For example, during our benchmarking we used this mechanism to pass a `scipy.sparse` matrix into Intrepydd using its underlying NumPy array representations.

4.3 C++ Host

This mode caters to C++ host programs, and the `.pydd` source file is compiled to a static or shared library which the host program can link with. There is no conversion between Intrepydd's data structures and C++ data structures. Instead, the generated C++ code for Arrays, Dictionaries and Lists relies on pure C++11 implementations built on top of the C++ Standard Template Library without any dependence on Python headers. The lack of Python dependence makes Intrepydd suitable for performant and portable application implementations on experimental architectures that may lack Python support. Additionally, this mode supports the `pfor` parallelization construct by generating OpenMP `parallel for` code [59]. Code generation support for `async` and `finish` is currently under development.

In [14]:
```
%%writefile opt.pydd
# opt.pydd

def update_centers(k: int64, X: Array(float64, 2), y: Array(int64)) \
                -> Array(float64, 2):
    m = shape(X, 0) # type: int64
    d = shape(X, 1) # type: int64
    centers = zeros((k, d), float64())
    counts = zeros(k, int64())

    # Sum each coordinate for each cluster
    # and count the number of points per cluster
    for i in range(m):
        c = y[i] # type: int64
        counts[c] += 1
        for j in range(d):
            centers[c, j] += X[i, j]

    # Divide the sums by the number of points
    # to get the average
    for c in range(k):
        n_c = counts[c] # type: int64
        for j in range(d):
            centers[c, j] /= n_c
    return centers

# eof
```

```
Overwriting opt.pydd
```

In [15]:
```
!../pyddc opt.pydd # Compile using Intrepydd
```

In [16]:
```
import opt
update_centers = opt.update_centers
kmeans(points, k, starting_centers=points[[0, 187], :], max_steps=50, ve
rbose=True)
```

```
iteration 1 WCSS =  417330.20399161614
iteration 2 WCSS =  241221.83691647876
iteration 3 WCSS =  227001.55370897506
iteration 4 WCSS =  225040.6824369842
iteration 5 WCSS =  224791.5775773802
iteration 6 WCSS =  224751.636111681
iteration 7 WCSS =  224744.82214507577
iteration 8 WCSS =  224743.13807689323
iteration 9 WCSS =  224742.7389964789
iteration 10 WCSS =  224742.6256260888
iteration 11 WCSS =  224742.5827842403
iteration 12 WCSS =  224742.5683493433
iteration 13 WCSS =  224742.56518699724
iteration 14 WCSS =  224742.5632137702
iteration 15 WCSS =  224742.5629567264
iteration 16 WCSS =  224742.56250036048
iteration 17 WCSS =  224742.56192824699
iteration 18 WCSS =  224742.56161207613
iteration 19 WCSS =  224742.56124801026
iteration 20 WCSS =  224742.5608175248
iteration 21 WCSS =  224742.56051294535
iteration 22 WCSS =  224742.560381938
```

Out[16]:
```
array([0, 0, 1, ..., 0, 1, 1])
```

Profile run. Let's re-run the profiler to see if the bottleneck sped up at all.

In [17]:
```
%lprun -f kmeans kmeans(points, k, starting_centers=points[[0, 187], :],
max_steps=50)
```

In [18]:
```
update_centers, compute_d2 = update_centers__0, compute_d2__0
t_0 = %timeit -o kmeans(points, k, starting_centers=points[[0, 187], :],
max_steps=50)

update_centers, compute_d2 = opt.update_centers, compute_d2__0
t_opt = %timeit -o kmeans(points, k, starting_centers=points[[0, 187],
 :], max_steps=50)

print("\n==> Speedup is ~{:.2f}x".format(t_0.average / t_opt.average))
```

```
1.26 s ± 53 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
635 ms ± 20.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

==> Speedup is ~1.98x
```

Figure 3. Example of using Intrepydd in a Jupyter notebook. Intrepydd interoperates with standard tools, such as profilers and timers as shown here.

4.4 Interactive Mode

4.4.1 Using Intrepydd in Jupyter Notebooks.

In the `host=python` mode, an Intrepydd program can work with

standard Python tools like Jupyter notebooks [50]. Taking inspiration from scientific laboratory notebooks, a Jupyter notebook is an interactive and programmable document format that allows end-user analysts to mix documentation, code, and the output of code (e.g., visualizations). Its goal is to help analysts create and easily share reproducible computational workflows.

Jupyter notebooks are an especially important target environment for Intrepydd because of its widespread use in science and industry. For instance, astronomers working on the Large Synoptic Survey Telescope (LSST), which will produce terabytes of data every night after its planned launch in 2022, are doing much of their frontend development with Jupyter notebooks.[5] Also, commercial companies like Netflix are using and investing heavily in Jupyter, as well as designing extensions that allow their analysts the ability to seamlessly transition interactive notebooks into production pipelines.[6] And growth and interest have been increasing rapidly. There were 2.5 million individual Jupyter notebooks on GitHub in 2018, and it has been observed that during the four-year periods from 2016 to 2019, the number of GitHub repositories for which Jupyter is the primary "language" has doubled every year.[7]

Thus, interoperability of Intrepydd with Jupyter has the potential to impact many end-user analysts. Figure 3 shows how easy it is to use Intrepydd in a Jupyter notebook. The Jupyter code cell labeled `In [14]` creates an Intrepydd function and writes it to a file. Cell `In [15]` compiles the Intrepydd function. There are two parts to compiling an Intrepydd function (discussed in Section 6.6): 1) translating Intrepydd to C++, which is relatively quick (under 0.5 seconds for the examples that we evaluated), and 2) compiling the generated C++ code, which incurs the usual overhead of invoking a C++ compiler (6 to 12 seconds for the examples that we evaluated). Cell `In [16]` shows how the Intrepydd module can be imported, and `update_centers` in the host Python program can be set to refer to the compiled Intrepydd code. The rest of the Python program can then invoke `update_centers` as though it was a Python function. Lastly, cells `In [17]` and `In [18]` invoke standard Python profiling and benchmarking tools, which reinforces how Intrepydd maintains interoperability with the tools Python programmers already know.

4.4.2 Debug Support. As mentioned in Section 2, a drawback of AOT compilation (as used by the Intrepydd toolchain) is that its long compile-times can hurt productivity in interactive compile-debug scenarios. To that end, Intrepydd

```
# Original Intrepydd code
# def foo(a: Array(), b: Array(), c: Array()) -> Array():
#     d = sqrt(abs(a)) / b * c.T
#     return d

# Automatically generated import statements
from pydd_types import Array
from numpy import sqrt, abs

# Original Intrepydd code is now executable as Python code
def foo(a: Array(), b: Array(), c: Array()) -> Array():
    d = sqrt(abs(a)) / b * c.T
    return d
```

Listing 2. Converting an Intrepydd code into Python code for debug support.

provides a mode to convert an Intrepydd program to a functionally equivalent Python version, which facilitates source-level debugging and tracing. In this mode, the Intrepydd function is executed as a normal Python program. Though this mode does not provide the performance benefits of C++ code generation from Intrepydd programs, it enables the user to debug their code on a smaller dataset without having to explicitly maintain both Python and Intrepydd versions of the code. The main task performed by the Intrepydd compiler to do this conversion is to examine the library functions used in the Intrepydd function, and generate import statements to map them to their NumPy/SciPy counterparts. If no such counterpart exists, our own Python implementation is imported and used. An example of this conversion is shown in Listing 2.

Another useful capability for performance debugging is unparsing the AST to Intrepydd source code after different stages in the Intrepydd toolchain. With this capability, a user can view the transformed code after specific optimizations are performed by the Intrepydd compiler.

5 High Level Code Optimizations

A major advantage of enabling Ahead-Of-Time (AOT) compilation for Intrepydd programs is that traditional scalar optimizations can be extended so as to be applied to aggregate operations on higher-level data structures, thereby resulting in significant execution time improvements. We describe three optimizations for sequential Intrepydd code in Sections 5.1–5.3, and also summarize the compiler support needed for Intrepydd's parallel `pfor` construct in Section 5.4. The techniques described in Sections 5.1–5.4 have been implemented in the current version of the Intrepydd compiler used to obtain the performance results presented in Section 6.

5.1 Loop Invariant Code Motion

Loop invariant code motion (LICM) is a classical code optimization technique that hoists invariant code fragments out of loops, so that they can be evaluated once instead of

[5] See https://www.nature.com/articles/d41586-018-07196-1, checked on August 13, 2020.
[6] https://hub.packtpub.com/how-everyone-at-netflix-uses-jupyter-notebooks-from-data-scientists-machine-learning-engineers-to-data-analysts/
[7] See https://octoverse.github.com/, checked on August 13, 2020.

```
1  while frontier.shape(0) > 0:
2      r_prime = copy(r)
3      for i in frontier:
4          p[i] += ((2.0 * alpha) / (1.0 + alpha)) * r[i]
5          r_prime[i] = 0.0
6
7      for i in frontier:
8          for j in range(indptr[i], indptr[i+1]):
9          dst_idx = indices[j]
10         update = (((1.0 - alpha) / (1.0 + alpha))
11                   * r[i] / degrees[i])
12         r_prime[dst_idx] = r_prime[dst_idx] + update
13     r = r_prime
14     frontier = where(r >= degrees*epsilon and degrees > 0)
```

Listing 3. Intrepydd code fragment extracted from `PR-Nibble` benchmark

```
1  def foo(xs: Array(double, 2)) -> double:
2      ...
3      for i in range(shape(xs, 0)):
4          for j in range(shape(xs, 1)):
5              a = xs[i, j]
6              ...
```

Listing 4. Example of 2-D array access in nested loop

in every iteration of the loop. While the LICM transformation is straightforward to perform on scalar variables, it can be challenging to identify a loop invariant computation on aggregate data structures implemented using pointers and arrays in a standard back-end optimizer like LLVM due to the inherent imprecision of pointer alias analysis. In contrast, the Intrepydd language defines clear semantics with respect to which aggregate variables are referenced/modified at each program point, including in library calls. Since the Intrepydd compiler is implemented using a high-level intermediate representation (IR) at the AST level, this semantics can be leveraged to enable aggressive LICM optimization of operations on aggregate data structures. We extended a standard LICM algorithm to make high-level operations (e.g., matrix operations) amenable to LICM in Intrepydd programs, so as to enable both statement-level and subexpression-level code motion. Further details of the algorithm can be found in Algorithm 1 in Appendix A[8].

Consider the Intrepydd code fragment in Listing 3, which was extracted from the `PR-Nibble` benchmark (discussed later in Section 6). In this example, the entire statement, "`update = (... * r[i] / degrees[i])`" (line 10), can be safely hoisted out of the enclosing `for j` loop. Likewise, the library calls to "`mul(degrees, epsilon))`" and "`gt(degrees, 0)`" (line 14, 15) can be hoisted out of the outermost `while` loop. These LICM opportunities cannot be realized when code optimization is performed at the C++ source code level, or at the LLVM IR level used by Numba.

[8]All appendices are provided in the supplemental document that accompanies this article in the Digital Library.

```
1  Array<double>* foo(Array<double>* xs) {
2    ...
3    for (int i = 0; i < pydd::shape(xs, 0); i += 1) {
4      for (int j = 0; j < pydd::shape(xs, 1); j += 1) {
5        a = xs.data()[i*pydd::shape(xs, 1)+j];
6        ...
```

Listing 5. Baseline C++ code generated from listing 4

```
1  Array<double>* sum_2d(Array<double>* xs) {
2    double* data = xs.data();
3    int shape1 = pydd::shape(xs, 1);
4    double a;
5    for (int i = 0; i < pydd::shape(xs, 0); i += 1) {
6      for (int j = 0; j < pydd::shape(xs, 1); j += 1) {
7        a = data[i*shape1+j];
8        ...
```

Listing 6. C++ code generated from listing 4 after LICM. Computation of `shape1` and `data` are hoisted out.

In addition to LICM on operations in an Intrepydd program, the Intrepydd compiler also performs LICM on the C++ source code that is generated from the Intrepydd program. For example, consider the code fragment in Listing 4 with a simple example of accessing a 2-D array in a nested loop. While the "`a=xs[i, j]`" statement does not contain a loop-invariant computation at the Intrepydd level, it can result in loop-invariant subcomputations at the C++ level. A direct translation of this program generates the baseline code shown in Listing 5. In this code, `pydd::shape(xs, i)` returns the `i`-th shape of array `xs`. The expression `xs.data()` returns the base pointer of the array. Both of these calls are loop invariants if the array is not resized of reshaped inside the loop, but the C++ compiler will not be able to determine that these library calls are loop invariant in this case. Instead, the Intrepydd compiler checks that the array has not be resized or reshaped inside the loop, and, if so, generates the code shown in Listing 6 which hoists `xs.data()` and `pydd::shape(xs, 1)` as loop invariants.

5.2 Optimization of Dense and Sparse Array Computations

A single line of Intrepydd code can contain multiple aggregate operations on dense/sparse arrays. These computations can be optimized as an extension of loop fusion [56] to enhance data locality and reduce loop iteration overheads. Similar array operator/library optimizations have been proposed and demonstrated in past work on DSL compilers [52, 54, 61, 69]. Further as part of loop fusion, sparse array operations bring additional opportunities for partial dead-code elimination, i.e., skipping computations that are later multiplied by zero elements of sparse arrays, as seen in systems like TACO [27, 49, 64]. Sections 5.2.1 and 5.2.2 respectively summarize the optimizations performed for dense and sparse array operations implemented at the AST level

```
# Original statement
# d = sqrt(abs(a)) / b * c.T

v1 = c.T
compatibility_check(a, b, v1)
d = empty([a.shape(0), a.shape(1)], float64())
for i1 in range(a.shape(0)):
    for i2 in range(a.shape(1)):
        # Scalarized loop body
        d[i1,i2] = sqrt(abs(a[i1,i2])) / b[i2] * v1[i1,i2]
```

Listing 7. Code generation for dense operation fusion on input Intrepydd statement, `d = sqrt(abs(a)) / b * c.T` to improve data locality.

in the Intrepydd compiler. We are unaware of any past work that has explored these optimizations for Python-related languages like Intrepydd.

5.2.1 Dense Element-wise Operation Fusion. The goal of this optimization is to fuse element-wise operations that operate on arrays with the same rank (# dimensions) and shape. We briefly summarize our algorithm here, and the complete algorithm can be found in Algorithm 2 in Appendix A. Given an AST statement node of interest, our algorithm first infers the data type and # dimensions of all child nodes, based on library information (Section 3.4). Then it performs tree traversal to identify fusible sub-trees, which satisfy legality and profitability conditions. The code generation step converts sub-trees into explicit loop nests with scalarized loop bodies.

Listing 7 shows the code generated by this optimization pass for the input Intrepydd statement, "`d = sqrt(abs(a)) / b * c.T`". First, the array shapes are checked for compatibility by a runtime call (`compatibility_check`) and then explicit loops are generated for this expression. Evaluation of `c.T`, the transpose of `c`, is split as an unfusible expression, which is assigned to a new variable `v1`.

5.2.2 Sparsity Optimizations. The major benefit of dense optimization is data locality improvements and reduction of loop overheads, while sparse optimization can also remove a large amount of computations arising from (partial) dead-code elimination, e.g., by skipping computations of terms that will be multiplied by zero elements in sparse arrays. Our current approach focuses on optimizing sequences of three classes of operations: 1) sparse element-wise multiply operations (`spm_mul`), 2) dense element-wise operations, and 3) additional linear algebra functions (`matmult`, `syrk`, `syr2k`). The extension to general sparse array operations is a subject for future work and can be addressed by extending techniques used in existing optimization frameworks, such as TACO [49].

We summarize our algorithm in this section, and the complete algorithm can be found in Algorithm 3 in Appendix A. Overall, our sparse optimization algorithm is analogous to the dense algorithm, while it also analyzes the sparsity (i.e., shape of sparse arrays) for all child nodes of given statement. The inferred sparsity information is used in the legality and profitability conditions to identify fusible sub-trees during the tree traversal and to create the loop nests that iterate over non-zero elements during the code generation step.

```
# Original statement
# t = spm_mul(s, a @ b / x * y)

compatibility_check(s, a, b, x, y)
t = empty_spm([s.shape(0), s.shape(1)], float64())
for (row, col, val) in s.nonzero_elements():
    # Scalarized loop body
    v1 = 0.0
    for i1 in range(a.shape(1)):
        v1 += a[row,i1] * b[i1,col]
    spm_set_item(t, val*v1 / x[row,col] * y[col], row, col)
```

Listing 8. Code generation for sparsity optimization of Intrepydd statement, `t = spm_mul(s, a @ b / x * y)`. Redundant computations that multiply 0 are eliminated.

As an example, Listing 8 shows an example of the sparsity optimization, where `s` is a sparse array and the result is stored in `t`. In the original statement, sparse matrix `s` multiplies a compound expression `a @ b / x * y` which stands for the `matmult` of 2-D array `a` and `b` divided by 2-D array `x` and then multiplying 1-D array `y`. A naive evaluation of this expression will evaluate `a @ b / x * y` first and then evaluate the multiplication by `s`, which can introduce many redundant computations when `s` is sparse. However, our optimized code in Listing 8 only computes `a[row,i1] * b[i1,col]` products for nonzero (`row,col`) elements in `s`.

5.3 Memory Optimization

We perform two kinds of memory optimization. The first is an allocation optimization, where we try to reuse an allocated array across multiple array variables. Listing 9 shows an example of such an optimization. Note that for cases where the original allocation zeros out the allocated memory, an additional call will need to be inserted to fill the memory space with zeros before reusing the space. Eliminating redundant allocations also reduces reference count management overhead. This optimization is performed at the Intrepydd AST level. Further details of the algorithm can be found in Algorithm 4 in Appendix A.

The second memory optimization focuses on a "strength reduction" that replaces slice objects used as arguments to library calls by raw pointers when possible to do so to avoid the overhead of creating an object for the slice. This optimization assumes the availability of a variant of the library call in C++ that takes a pointer as an argument instead of the slice object. To perform this optimization, our compiler generates code to compute the data pointer and the shape of the slice and pass them to the variant of the library call which accepts raw pointers and shapes. Note that this optimization is performed during C++ code generation.

```
# Original code
# while it < max_iter:
#     a = b + c  # all 2D arrays
#     d = zeros_like(a)
#     ...

a = empty_like(b)
d = empty_like(a)
while it < max_iter:
    add(b, c, out=a)
    fill(d, 0)
    ...
```

Listing 9. Code generation example for allocation optimization for original Intrepydd code shown at the top. Now allocated memory is recycled for every loop iteration instead of allocating every time.

5.4 Loop Parallelization

As described in Section 3, the Intrepydd programmer can use the `pfor` construct to indicate that there are no loop-carried dependences on non-scalar variables. For convenience, dependence checking and privatization for scalar variables are automated in the Intrepydd compilation framework. Parallel code is generated for `pfor` loops via OpenMP [59] pragmas in the C++ codegen phase, and the results of scalar privatization analysis are used to identify private variables for correct parallel execution. We implement standard dataflow analyses to 1) check for live-in (upward exposed) uses of scalar variables, which indicate a loop-carried dependence thereby disabling parallel code generation for the `pfor` loop, and 2) identify variables for inclusion in OpenMP's `private` clause and live-out variables for inclusion in OpenMP's `lastprivate` clause. According to OpenMP semantics, each thread works with local copies of variables specified in the `private` and `lastprivate` clauses. If a variable is specified in a `lastprivate` clause, the final value from the thread executing the last loop iteration is copied back to the original variable on exit from the `pfor` loop.

Looking at the simple `pfor` example in Listing 1, our data flow analysis for scalar variable `temp` can determine that it's neither live-in nor live-out, which implies that the loop can execute in parallel if `temp` is added to the private clause:

```
#pragma omp for private(i, temp)
```

6 Experimental Results

In this section, we present an empirical evaluation of our implementation of the Intrepydd tool chain. Based on the results and their explanations included below, we conclude that the Intrepydd approach can deliver notable performance benefits relative to current approaches available to Python programmers for improved performance, namely Cython and Numba; further, these benefits are especially significant for Python applications with execution times that are not dominated by calls to native libraries. We also claim that there is no loss in productivity relative to current approaches, since Intrepydd's requirement for declaring types of function parameters and return values is also shared by Cython (which imposes additional limitations as well) and by Numba. Finally, we claim that Intrepydd offers more portability than current approaches, since different code versions can be generated from the same Intrepydd source code using the `host=python` and `host=cpp` modes.

6.1 Benchmarks

Given the focus of Intrepydd on the data analytics area, we examined a set of publicly available data analytics applications written by data scientists in Python, and selected for use in a recent DARPA program. Since the primary contribution of the current Intrepydd implementation is to generate improved code, rather than to improve libraries, we expect significant performance improvements primarily for applications with execution times that are not dominated by calls to native libraries. We selected five of these applications for use as benchmarks (bigCLAM, changepoint, ipnsw, PR-Nibble, sinkhorn-wmd). For completeness, a sixth benchmark (ISTA) was also selected which spends significant time in NumPy libraries, even though we do not expect significant performance improvements for such programs.

The benchmarks are summarized in Table 3 (which includes citations with links to publicly available source code repositories), and span three different domains in the data analytics area – graph analytics, machine learning, and natural language processing. We are unaware of any single DSL that covers all three domains, where as Intrepydd can be used for all of them given its goal to be a DiAL that covers the data analytics area more broadly.

6.2 Experimental Setup

All single-core experiments were performed on a dedicated machine with dual Intel Xeon Silver 4114 CPUs with 10 physical cores per CPU socket, running at a constant 2.2 GHz with 192 GB of main memory. Intel Turbo Boost was disabled. Our baseline for benchmark comparison is Python version 3.7.6. All C/C++ code generated by Intrepydd (in `host=python` mode) and Cython (version 0.29.15) was compiled with GCC 7.5.0 using optimization level O3. The Numba (version 0.48) and Cython benchmark source codes were written with programmability similar to that of Intrepydd, without any additional hand optimizations. For Cython benchmarks, all kernel functions were annotated with `@cython.boundscheck(False)` and `@cython.wraparound(False)`, while for Numba, all functions were annotated with `@jit(nogil=True, nopython=True)`. Each benchmark was compiled once and executed 11 times (with each run being launched as a new process, which included dynamic optimization only in the case of Numba). The execution times reported for each data point is the average latency of the later 10 runs from the set of 11 (the first run was omitted in the average). The primary

Table 3. Benchmarks, with references to GitHub repositories, domains, source lines of code (SLOC) counted using CLOC tool, and descriptions.

Benchmark	Domain	SLOC	Description
bigCLAM [63]	Graph analytics	57	Overlapping community detection in massive graphs using the BIGCLAM algorithm.
changepoint [43]	Graph analytics	100	Finds change points of a graph based on node properties
ipnsw [44]	Graph analytics	91	Non-metric Similarity Graphs for Maximum Inner Product Search.
ISTA [45]	Machine learning	69	Local graph clustering (LGC) through Iterative shrinkage-thresholding.
PR-Nibble [45]	Machine learning	73	Uses Page Rank algorithm to find local clusters without traversing the entire graph .
sinkhorn-wmd [46]	NLP	52	Computes Word Movers Distance (WMD) using Sinkhorn distances.

Table 4. Average single core execution times (in seconds), and standard deviation of 10 runs (in parentheses) as a percentage of average execution time, for the baseline Python application and for the primary kernel written in each of baseline Python, Cython, Numba, unoptimized Intrepydd, and optimized Intrepydd. Kernel execution times include time of native calls from the main Python program to the kernel function.

Benchmark times		Primary Kernel execution times (seconds)				
Benchmark	Python (Baseline)	Python (Baseline)	Cython	Numba	Intrepydd (Unoptimized)	Intrepydd (Optimized)
bigCLAM	12.69 (2.25%)	12.36 (1.21%)	11.54 (1.31%)	4.157 (0.47%)	2.56 (3.60%)	1.09 (0.89%)
changepoint	20.89 (0.71%)	16.37 (0.65%)	119.64 (1.13%)	16.46 (0.48%)	1.47 (0.22%)	1.47 (1.23%)
ipnsw	21.07 (0.90%)	17.44 (1.61%)	17.03 (2.43%)	3.21 (0.59%)	1.68 (0.32%)	0.79 (0.18%)
ISTA	30.29 (0.28%)	27.37 (0.06%)	26.93 (0.09%)	30.62 (0.10%)	79.36 (0.06%)	18.51 (0.33%)
PR-Nibble	53.42 (1.23%)	52.84 (1.18%)	5.04 (0.45%)	3.27 (0.36%)	0.83 (0.84%)	0.006 (0.42%)
sinkhorn-wmd	48.17 (0.26%)	46.44 (0.08%)	46.82 (0.20%)	47.03 (0.05%)	47.61 (0.09%)	1.22 (0.40%)

kernel execution times were measured at the Python invocation sites, and include any overheads of native function calls. For single core experiments, each process was pinned to a single-core in order to maintain a steady cache state and reduce variations between runs. The normalized standard deviation for the averaged runs varied between 0.06% and 3.60%.

6.3 Comparison with Python, Numba, Cython for Single-core Execution

Table 4 summarizes the results of our performance measurements of different single-core executions of five versions of each benchmark: baseline Python, Cython, Numba, unoptimized Intrepydd, and optimized Intrepydd. The first Python column includes execution times for the entire application to provide a perspective on the fraction of the application time spent in its primary kernel, which is reported in all successive columns. All Intrepydd executions were performed using the `host=python` mode, i.e., they all started with a Python main program which calls the automatically generated code from Intrepydd as an external library. The optimized Intrepydd version was obtained by applying our implementation of the LICM, dense/sparse optimizations, and memory optimizations, described in Sections 5.1, 5.2 and 5.3. As mentioned earlier, the execution times for the five versions are for the primary kernel in each benchmark.

Figure 4 shows the performance improvement ratios for the primary kernels relative to the baseline Python version, using the data in Table 4. When compared to baseline Python, the optimized Intrepydd version shows improved performance ratios between $11.1\times$ and $8809.8\times$ for the five non-library-dominated benchmarks and $1.5\times$ for the library-dominated, ISTA benchmark. These are more significant improvements than those delivered by Cython and Numba, thereby supporting our claim that Intrepydd delivers notable performance benefits relative to current approaches available to Python programmers. Further, optimized Intrepydd did not show a performance degradation in any case, though Cython and Numba showed degradations in two cases (changepoint and ISTA). We also note from Table 4 that all but one cases of unoptimized Intrepydd match or exceed the performance of Python. The exception is ISTA, for which the unoptimized Intrepydd version runs $2.9\times$ slower than the Python version. As mentioned earlier, ISTA's performance is dominated by libraries, and this gap arises from the fact that the performance of the Intrepydd libraries has not been tuned like the performance of the NumPy libraries. However, this gap is more than overcome when static optimizations are enabled for the Intrepydd code.

Finally, the impact of progressively adding the optimizations from Sections 5.1-5.3 is shown in Table 5, and demonstrates that all three optimizations can play significant roles

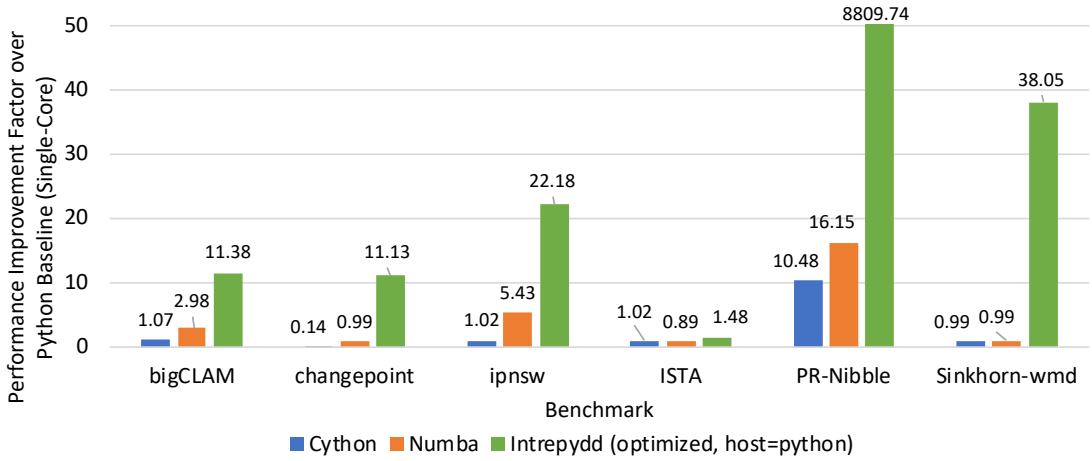

Figure 4. Single core performance improvements for primary kernels relative to Python baseline (derived from data in Table 4). Intrepydd offers an order of magnitude performance improvement over Cython and Numba on average.

Table 5. Average single core execution times (in seconds) for the primary benchmark kernels written in Intrepydd and compiled with increasing sets of optimizations: unoptimized, +LICM (Section 5.1), +Sparse/Dense Array optimizations (Section 5.2), +memory optimizations (Section 5.3).

Primary Kernel execution times (seconds)				
Bench-mark	**Intrepydd**	**Intrepydd (+LICM OPT)**	**Intrepydd (+Array OPT)**	**Intrepydd (+Memory OPT)**
bigCLAM	2.558	2.557	1.541	1.086
changepoint	1.472	1.469	1.466	1.471
ipnsw	1.679	0.786	0.786	0.786
ISTA	79.362	18.732	18.473	18.509
PR-Nibble	0.831	0.114	0.106	0.006
sinkhorn-wmd	47.612	47.395	1.225	1.220

in contributing to performance improvements. For instance, ipnsw and ISTA benefit mostly from LICM. PR-Nibble benefits significantly from both LICM (0.831sec → 0.114sec), and even more significantly from memory optimization (0.106sec → 0.006sec) which eliminates a large number of dynamic array allocations by instead reusing existing arrays. The major computation in sinkhorn-wmd is later multiplied by a sparse array with sparsity that is less than 1%. Thus, the sparse array optimization efficiently removed several computations that would have been multiplied by zero, thereby resulting in a large reduction in kernel execution time (47.395sec → 1.225sec).

6.4 Impact of Parallelization

Intrepydd supports parallelization in the form of user-specified `pfor` loops with compiler-supported privatization and code generation. Given the known challenges of multithreading within a single Python process, our current Intrepydd implementation only supports the `pfor` construct in the `host=cpp` mode. In this section, we present results obtained from parallelization of four of the six benchmarks from the previous section that were able to use the `pfor` construct – changepoint, ipnsw, ISTA, and PR-Nibble. In addition to studying the impact of parallelization, the evaluation in this section demonstrates how different code versions can be generated from the same Intrepydd source code using the `host=python` mode for results in Section 6.3 and the `host=cpp` mode for results in this section.

For parallelization experiments, we used a dual socket Intel Xeon E-2680 V4 (14 cores per socket) server running at 2.40 GHz with 128 GB main memory. Intel Hyperthreading and Turbo-boost were disabled. Each benchmark was run using 1, 2, 4, 8 and 16 cores. As before, the timing measurements were obtained for the primary kernels in the benchmarks following the methodology described in Section 6.2.

Figure 5 shows a summary of optimized parallel Intrepydd `host=cpp` performance compared to single core optimized Intrepydd `host=python` on 1, 2, 4, 8 and 16 cores. The performance of the 1-core `host=cpp` version was observed to be between 0.9x to 1.5x faster than that of the single core `host=python` performance (with different C++ code generated in the two cases due to different target libraries).

The benchmarks with outermost level parallelization opportunities (changepoint and ISTA) showed near linear performance scaling with increasing core count; going from a single core performance of 1.5× and 0.9× respectively to 8× over `host=python` code when using 16 cores. The ipnsw benchmark also shows performance gains when using more cores. However, the impact of parallelization in this kernel is not as prominent. In the case of PR-Nibble, the performance increases from 0.9× when using 1 core to 2.1× when using 8 cores. However beyond 8 cores, the thread creation overhead

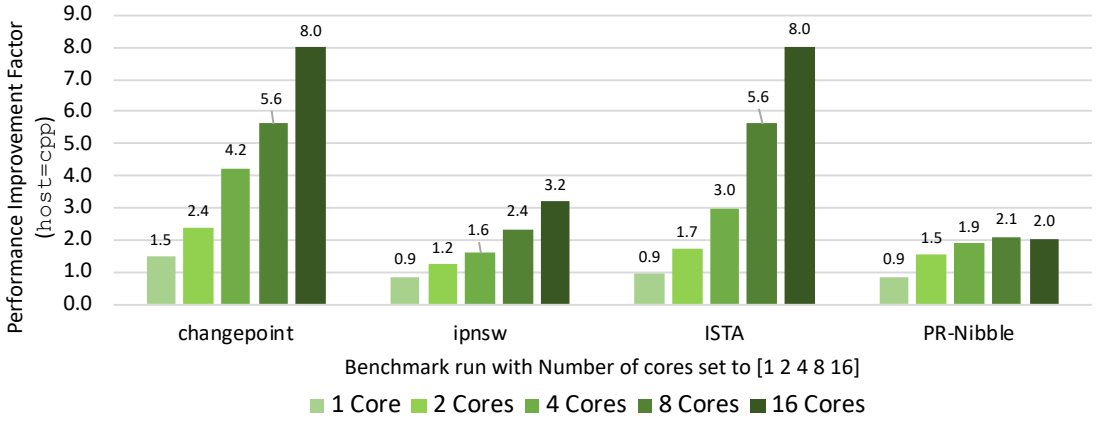

Figure 5. Multi core scalability of Intrepydd for a subset of the benchmarks in `host=cpp` mode. The Performance Improvement Factor is relative to the single-core execution time in `host=python` mode. Intrepydd implementations of parallelizable applications benefit from near linear scalability.

exceeds the amount of available work, resulting in a slight slowdown when compared to the 8 core performance.

6.5 Comparison with Julia

A comparison against Julia has been provided using micro-benchmarks from Julia's official website [16]. The Intrepydd versions are directly adapted from the Python versions provided by Julia with additional type annotations. Results (using Julia 1.4.1) for the three non-library dominant benchmarks, i.e. `fibonacci`, `quicksort` and `pi-sum` are shown in table 6.

The micro-benchmark results are comparable to Julia and show negligible performance improvements over Julia. This is attributed to their rather simple nature and the lack of typical complex algebraic expressions that can benefit from Intrepydd's optimizations. To illustrate the benefits of Intrepydd's optimizations, the sinkhorn-wmd benchmark was converted to idiomatic Julia. Optimized Intrepydd shows two orders of magnitude improvement over Julia (last row of Table 6). This improvement can be attributed to the automatic sparsity optimizations applied by Intrepydd, which largely removed the zero-multiplied computations as discussed in Section 6.3. The Intrepydd implementation of sinkhorn-wmd yielded 12.61× fewer instructions according to Hardware Performance Monitor measurements. As mentioned in Section 1, we believe that the Intrepydd approach for computational kernels is applicable to both Python and Julia, thereby implying that this performance gap can be reduced if Julia were to also adopt the ahead-of-time optimizations demonstrated by Intrepydd.

6.6 Compile Time

In this section, we summarize the overhead of compiling Intrepydd kernels for use by `host=python` main programs. A similar methodology as in Section 6.2 (i.e., similar to the single core results) was used to collect these results. As shown in Table 7, the compile time for an Intrepydd program is broken into the time taken by the Intrepydd compiler (implemented in Python) to generate C++ code, and the time taken by the underlying C++ compiler to generate the CPython module. The time taken by the Intrepydd compiler to apply AST optimizations and generate the C++ code was found to range between 0.2 and 0.5 seconds, a fraction of the time required by the underlying C++ compiler (6.08s to 12.05s) to generate a CPython module for the benchmarks evaluated in this paper. The standard deviation for all benchmarks was within 1% of the means reported in Table 7.

Table 6. Average single core execution times (in seconds) for Python, Julia and Intrepydd versions of four benchmarks.

Benchmark	Execution Time (seconds)		
	Python	Julia	Intrepydd
fibonacci	3.75	0.07	0.06
quicksort	1.76	0.93	0.80
pi-sum	0.57	0.01	0.01
sinkhorn-wmd	46.44	108.60	1.22

7 Mapping Intrepydd to Post-Moore Accelerators and Architectures

In this section, we show that Intrepydd is also a promising option for future post-Moore hardware, in addition to the performance benefits for current multicore CPU processors shown in Section 6. The International Roadmap for Devices and Systems (IRDS) tracks seven application areas that will drive post-Moore technologies for the semiconductor industry. All seven application areas are limited by memory

Table 7. Average compile times (in seconds) for the benchmarks broken down into Intrepydd and C++ compile times for `host=python` mode.

Benchmark	Compile Time (seconds)	
	AST Optimizations	Generated C++
bigCLAM	0.30	7.93
changepoint	0.21	5.73
ipnsw	0.48	12.05
ISTA	0.43	7.33
PR-Nibble	0.33	6.08
sinkhorn-wmd	0.21	7.20

latency [20], and future near-memory processors and accelerators hold promise to address this "memory wall" challenge. However, such processors and accelerators require specialized programming, which may be beyond the reach of the data scientists we are targeting with Intrepydd. In the following two subsections, we discuss how the Intrepydd approach can be used to enable generation of performance-portable code for two examples of forward-looking hardware, the Emu Chick processor [39] and the MetaStrider [67] accelerator.

7.1 Emu Chick Near-Memory Processor

The Emu Chick is a novel architecture built on the concept of migratory memory-side processing. A key building block in its hardware design is a "nodelet" which consists of a Narrow Channel DRAM (NCDRAM) memory unit and multiple lightweight Gossamer cores. This co-location of a memory unit with processor cores is what makes Emu a near-memory processing system. Any thread can access any memory location in the system via the partitioned global address space (PGAS) model [31]. If the accessed (target) location is in a different nodelet from the thread performing the access, the Emu Chick migrates the thread to the nodelet containing the datum being accessed. This obviates the need for a traditional, elaborate cache coherence mechanism.

The benefits of near-memory processing with the Emu Chick have been demonstrated in past work, e.g., [32, 76]. As shown in Figure 6, the results in [76] used a pointer chasing microbenchmark with a range of block sizes to show that Emu Chick is not only able to utilize a larger fraction of its available memory bandwidth but also does so more consistently for a larger range of block sizes when compared to a commodity Xeon processor.

The Emu processor uses an extension of the Cilk multithreaded programming model [29] for its hardware. Cilk is based on the C language, and Cilk threads are created and synchronized using the `cilk_for`, `cilk_spawn`, and `cilk_sync` C-API calls. These constructs are subsumed by higher-level language constructs in Intrepydd, namely `pfor`, `async` and `finish`. In addition to thread creation, programming for the Emu requires careful distribution of data across nodelets as well as judicious use of remote memory atomic operations. These primitives are also supported at a high level in Intrepydd with extended array shape and `isolated` constructs, thereby making it possible for the Intrepydd programmer to easily access the capabilities of a near-memory processor like the Emu Chick.

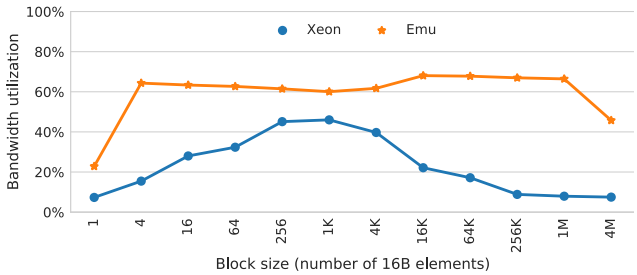

Figure 6. Bandwidth utilization of pointer chasing microbenchmark for Sandy Bridge Xeon and Emu processors (from Figure 8 of [76]).

7.2 MetaStrider Accelerator for Sparse Reductions

The end of Moore's Law has sparked a resurgence of interest in hardware accelerators for sparse matrix computations, most notably for sparse matrix multiplication (SpGEMM), given its central role in a large number of data analytics applications. In this section, we demonstrate how Intrepydd can be used to target a research accelerator for *sparse reductions*, MetaStrider [67]. (Sparse reductions account for over half of the execution time for SpGEMM.)

Given a sparse data stream of *key-value* pairs, a sparse reduction is performed on values that share the same key. To motivate the opportunity for hardware acceleration of sparse reductions, consider that memory arrays are organized as rows and columns, and are accessed most efficiently when subsequent requests are served by the same row [68]. This is typically the case in applications that operate on dense data structures. However, since the keys in sparse data streams exhibit low spatio-temporal locality, conventional cache and memory systems suffer from poor performance. In the context of main memory (DRAM), sparse reductions suffer from a high number of DRAM row activations with poor row utilization.

MetaStrider supports multiple instructions at the assembly code level to support sparse reductions. One such instruction is a three-operand asynchronous reduction operation of the form `RED` R_k, R_v, `base`, where R_k and R_v are registers containing the key and value of the record to be reduced, and `base` refers to the target array to be reduced. Such key-value pairs are asynchronously offloaded by the CPU to the accelerator so that it may efficiently perform in-situ reduction in the cache and memory hierarchy. The Intrepydd compiler

```
# A and B may be stored in conventional sparse formats or in the
# accelerator format, abstracted as Intrepydd sparse matrix type.
def SpGEMM_OuterProduct(A: SparseMat, B: SparseMat) -> SparseMat:
    C = empty_spm(A.shape(0), B.shape(1))
    Y = A.getNonZeroColumnIndices()    # Set of valid column indices
    Z = B.getNonZeroRowIndices()       # Set of valid row indices
    X = sorted(Y & Z)                  # Sorted list of valid indices
    finish:
        for x in X:
            colA = A.getCol(x) # List of non-zeros (index, value)
            rowB = B.getRow(x) # List of non-zeros (index, value)
            for (i, A_ix) in colA:
                for (j, B_xj) in rowB:
                    partial = A_ix * B_xj
                    async:
                        isolated:
                            C[i,j] += partial
    return C
```

Listing 10. Programmer annotations to leverage hardware acceleration of sparse reductions in SpGEMM.

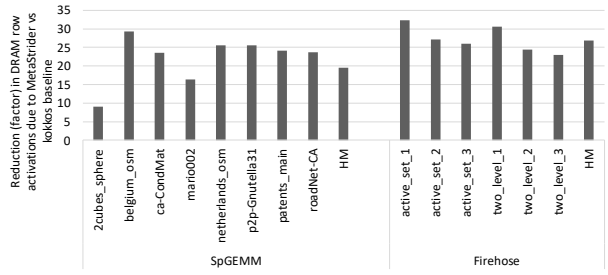

Figure 7. MetaStrider is able to significantly reduce the number of DRAM row activations for sparse reductions (adapted from Figure 8a of [67]).

can be extended so that it can automatically generate RED instructions from code with finish-async-isolated constructs, as shown in Listing 10.

In addition to generating a RED instruction for the "`C[i,j] += partial`" statement in Listing 10, the Intrepydd compiler can generate an INIT instruction (to initialize the accelerator) at the start of the finish scope, and an END instruction at the end of the finish scope which waits for all pending RED instructions to complete.

The potential benefits of MetaStrider performance were evaluated in [67] via cycle-accurate simulation, and the results are shown in Figure 7. This evaluation studied the benefits of MetaStrider for SpGEMM workloads and cybersecurity workloads (Firehose [23]). The baseline for comparison was set to be a state-of-the-art hashmap-based reducer from the Kokkos framework [35] executed on a standard memory system without the MetaStrider accelerator. As shown in Figure 7, MetaStrider, reduces the number of DRAM row activations by significant factors (approximately, 9× - 32×) relative to the baseline. Listing 10 shows that the capabilities of MetaStrider can be accessed by high level primitives in Intrepydd, instead of using C-based assembly-level intrinsics which is the standard programming interface for such accelerators.

8 Related Work

The work most closely related to Intrepydd falls roughly into two broad categories: compilers for Python and tools targeting high-performance data analytics.

Compiling Python. There are several efforts to compile Python (or some subset), most notably Numba [11], Cython [7], PyPy [17], Nuitka [10], Shed Skin [12], and Pyston [13]. Of these, Cython, Nuitka, and Shed Skin are source-to-source translators from Python to C/C++. PyPy and Numba use JIT compilation to improve program performance. PyPy is an implementation of Python in Python. However, it is known to be slow in running NumPy operations since it emulates the CPython C API. There is an attempt to enable loop-aware optimizations in PyPy's Tracing JIT [24, 25]. By a pre-processing based on loop peeling, it makes existing forward-passing optimizers applicable to loops, including common subexpression elimination (and hence LICM for given loop) and allocation removal. Note that these optimizations are performed on the traces of PyPy's interpreter execution while Intrepydd applies AOT optimizations including LICM, allocation removal, and dense/sparse array operator optimizations to the typed Python AST, which keeps the simple program structure and high-level abstractions of source Intrepydd programs. Numba, implemented as a Python library, dynamically translates a subset of Python code into machine code. Numba is the closest to Intrepydd in terms of goals, but since it is a JIT compiler, it limits the amount of time allocated for code optimization. Additionally, since Numba is a Python-based JIT compiler, it cannot be used in environments where Python is not supported.

While all these efforts aim to improve performance through generation of native code, to the best of our knowledge, none of them has leveraged high level semantics to perform the kinds of code optimizations described in Section 5 for Intrepydd.

Tools for High-performance Data Analytics. The most important abstraction in data analytics applications is arguably that of linear algebra. In this context, there has been significant prior work on languages [3, 28, 55], libraries [42, 73, 74], and compilers [49, 64, 66, 70] for optimizing dense and sparse linear algebra via loop transformations that can optimize loop nests along with other optimizations. While some of the library approaches provide a collection of manually optimized kernels [42], they may still leave performance on the table since they are not able to optimize across different operations. In contrast, the Intrepydd compiler can use a more global context to fuse sparse and dense operations across loops. Other library approaches and DSLs [49, 73] provide frameworks to express compound expressions, and automatically generate optimized native code for these expressions. These approaches are not as flexible and convenient as the approach taken by Intrepydd, which can generate optimized code for tasks beyond just dense and sparse tensor

algebra while maintaining a Python-like syntax and being compatible with both Python and C++ host environments.

Julia [28] is a high-level, high-performance, dynamic programming language designed for high-performance numerical analysis and computational science which, akin to Intrepydd, can be used as a package callable from Python code. However, Julia at present has a smaller adoption rate for data analysis applications compared to Python, and differences between Julia and Python syntax may be a barrier for mainstream data analytics developers. Further, the code optimization framework in Julia is also based on JIT compilation using LLVM, and faces similar limitations as those discussed for Numba. The proprietary commercial package, MATLAB [55], and the open-source functional-language, R [3], are also popular languages used by data scientists, but as high-level dynamic languages they lack sufficient performance [28].

In summary, it is worth noting that Intrepydd's AOT compilation approach for data science kernels can be used in conjunction with any of the above tool chains. In some cases (e.g., Python, Julia), the NumPy interfaces supported by Intrepydd can be used directly; in other cases (e.g., MATLAB, R), Intrepydd's interfaces will need to be extended to support the desired tool chain.

9 Conclusions and Future Work

This paper described the Intrepydd programming system, which is designed to enable data scientists to write application kernels with high *performance*, *productivity*, and *portability*. It does so through a combination of contributions that span language design, a prototype tool chain, and high-level compiler optimizations. The single-core performance results show significant improvements for Intrepydd relative to Python, Cython, and Numba. The improvements relative to Python range from one to nearly four *orders of magnitude* (11.1× to 8809.5×) for the five benchmarks that are not library dominated, and even show an improvement of 1.5× for one library-dominated benchmark, ISTA. We also compared the performance of Julia and Intrepydd on four benchmarks, three of which were drawn from a commonly used set of Julia microbenchmarks with execution times under 1 second for which Julia and Intrepydd showed comparable performance. For the fourth longer-running benchmark (`sinkhorn-wmd`), the Intrepydd version ran 89.0× faster than the Julia version. Finally, we showed multicore scalability results for Intrepydd, and also discussed the ease with which Intrepydd programs can be mapped to post-Moore accelerators and novel architectures. Overall, the Intrepydd approach has demonstrated the potential for significantly advancing the performance and portability that can be obtained from high productivity languages like Python and Julia.

There are many interesting directions for future work, including: completing the implementation for `async`, `finish`, and `isolated`; extending the optimizations performed by the Intrepydd compiler; adding support for other libraries that are used in data analytics such as SciPy [5], PyTorch [8], and scikit-learn [9]; completing support for post-Moore accelerators and novel architectures; and, extending Intrepydd to target Python-friendly parallel and distributed runtime frameworks such as Ray [58]. In addition, our approach complements, rather than precludes, other DSL-oriented and annotation-oriented techniques, and integration with such methods is another promising line of work. For instance, implementations of DSLs can use DiALs for acceleration, following the abstraction layering methods of systems like Devito [53], FEniCS [22], and telescoping languages [47]. Similarly, annotation systems with even higher levels of abstraction than DSLs, which have targeted performance and verification issues [40, 47, 60], could employ DSLs or DiALs "under-the-hood." However, perhaps the most rewarding direction for future work would be the adoption of the approach outlined in this paper in a context where it can have a production-level impact on data science application developers, for any programming system that is prevalent in data science applications (with Python and Julia being the most promising candidates).

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement Nos. HR0011-17-S-0055 and HR0011-20-9-0020. We would like to thank Prithayan Barua, Armand Behroozi, Akihiro Hayashi, Ankush Mandal, Sri Raj Paul, Caleb Voss and Mang Yu for their help with developing libraries and test cases for the Intrepydd implementation. We are also grateful to Todd Anderson for his input on the current capabilities of the Numba JIT compiler. Finally, the authors acknowledge the valuable feedback from co-PIs on both DARPA-supported projects – Deming Chen, Wen-mei Hwu, Taesoo Kim, Scott Mahlke, Sukarno Mertoguno, Viktor Prasanna, Alexey Tumanov — and their research group members, as well as participants in an evaluation of an early version of the Intrepydd system during the summer of 2019.

References

[1] [n. d.]. 3.3.8. Emulating numeric types. https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types.
[2] 1991. Python. https://www.python.org/.
[3] 1993. The R Project for Statistical Computing. https://www.r-project.org/.
[4] 2001. multiprocessing – Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html.
[5] 2001. SciPy. https://www.scipy.org/.
[6] 2006. NumPy. https://numpy.org/.
[7] 2007. Cython. https://cython.org/.
[8] 2007. PyTorch. https://pytorch.org/.
[9] 2007. scikit-learn. https://scikit-learn.org/stable/.
[10] 2012. Nuitka. https://nuitka.net/pages/overview.html.

[11] 2012. Numba. https://numba.pydata.org/.
[12] 2012. Shed Skin. https://shedskin.github.io/.
[13] 2014. Pyston. https://blog.pyston.org/.
[14] 2015. pybind. https://pybind11.readthedocs.io/en/stable/.
[15] 2015. TensorFlow: an end-to-end open source machine learning platform. https://www.tensorflow.org/.
[16] 2018. Julia Micro-Benchmarks. https://julialang.org/benchmarks/.
[17] 2019. PyPy. https://pypy.org/.
[18] 2019. Python Typed AST Package. https://pypi.org/project/typed-ast/.
[19] 2019. Top languages. https://octoverse.github.com/.
[20] 2020. Applications Benchmarking chapter, IEEE International Roadmap for Devices and Systems. https://irds.ieee.org.
[21] 2020. OpenBLAS: An optimized BLAS library. https://www.openblas.net/, Version 0.3.10.
[22] Martin Alnaes, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS Project Version 1.5. *Archive of Numerical Software* 3, 100 (12 2015). https://doi.org/10.11588/ans.2015.100.20553
[23] Karl Anderson and Steve Plimpton. 2015. *FireHose Streaming Benchmarks.* Technical Report. Sandia National Laboratory.
[24] Håkan Ardö, Carl Friedrich Bolz, and Maciej FijaBkowski. 2012. Loop-Aware Optimizations in PyPy's Tracing JIT. *SIGPLAN Not.* 48, 2 (Oct. 2012), 63–72. https://doi.org/10.1145/2480360.2384586
[25] Håkan Ardö, Carl Friedrich Bolz, and Maciej FijaBkowski. 2012. Loop-Aware Optimizations in PyPy's Tracing JIT. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/2384577.2384586
[26] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science Engineering* 13, 2 (March 2011), 31–39. https://doi.org/10.1109/MCSE.2010.118
[27] James Bergstra, Olivier Breuleux, Frederic Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in python. In *Proceedings of the 9th Python in Science Conference*. 3–10.
[28] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *CoRR* abs/1209.5145 (2012). arXiv:1209.5145 http://arxiv.org/abs/1209.5145
[29] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958
[30] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (2007), 291–312.
[31] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 519–538. https://doi.org/10.1145/1094811.1094852
[32] Prasanth Chatarasi and Vivek Sarkar. 2018. A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System. In *Proceedings of the Workshop on Memory Centric High Performance Computing (MCHPC '18)*. Association for Computing Machinery, New York, NY, USA, 37–44. https://doi.org/10.1145/3286475.3286481
[33] T. M. Conte, E. P. DeBenedictis, P. A. Gargini, and E. Track. 2017. Rebooting Computing: The Road Ahead. *Computer* 50, 1 (2017), 20–29.
[34] Timothy A Davis. 2018. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.
[35] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 693–702.
[36] Marat Dukhan. 2020. Indirect deconvolution algorithm. In *Proceedings of the IPDPS'20 Workshop on Parallel AI and Systems for the Edge (PAISE)*. https://doi.org/10.1109/IPDPSW50202.2020.00154
[37] Marat Dukhan and Artsiom Ablavatski. 2020. Two-pass softmax algorithm. In *Proceedings of the IPDPS'20 Workshop on High-Performance Big Data and Cloud Computing (HPBDC)*. https://doi.org/10.1109/IPDPSW50202.2020.00074
[38] S.C. Eisenstat, M.C. Gursky, M.H. Schulz, and A.H. Sherman. 1977. *Yale Sparse Matrix Package: I. The symmetric codes.* Technical Report RR-112. Yale University. https://apps.dtic.mil/dtic/tr/fulltext/u2/a047724.pdf
[39] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy. 2018. An Initial Characterization of the Emu Chick. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 579–588.
[40] J. Hückelheim, Z. Luo, F. Luporini, N. Kukreja, M. Lange, G. Gorman, S. Siegel, M. Dwyer, and P. Hovland. 2017. Towards Self-Verification in Finite Difference Code Generation. In *Proceedings of Correctness'17: First International Workshop on Software Correctness for HPC Applications (Correctness'17)*. ACM, New York, NY, USA. https://doi.org/10.1145/3145344.3145488
[41] Shams Imam, Jisheng Zhao, and Vivek Sarkar. 2015. A Composable Deadlock-Free Approach to Object-Based Isolation. In *Euro-Par 2015: Parallel Processing*, Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 426–437.
[42] Intel. 2018. SIGPLAN Empirical Evaluation Checklist. https://software.intel.com/mkl.
[43] Ben Johnson. 2019. graph-changepoint. https://github.com/bkj/graph-changepoint.
[44] Ben Johnson. 2019. ipnsw. https://github.com/prog-eval/prog-eval/tree/master/ipnsw.
[45] Ben Johnson. 2019. lgc. https://github.com/prog-eval/prog-eval/tree/master/lgc.
[46] Ben Johnson. 2019. sinkhorn-wmd. https://github.com/prog-eval/prog-eval/tree/master/sinkhorn_wmd.
[47] Ken Kennedy, Bradley Bloom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnson, John Mellor-Crummey, and Linda Torczon. 2001. Telescoping Languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing (JPDC)* 61, 12 (12 2001), 1803–1826. https://doi.org/10.1006/jpdc.2001.1724
[48] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.
[49] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901
[50] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks–a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and*

Agendas. IOS Press. https://doi.org/10.3233/978-1-61499-649-1-87

[51] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[52] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[53] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman. 2016. Devito: Towards a generic Finite Difference DSL using Symbolic Python. In *Proceedings of the Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE, 67–75. arXiv:1609.03361.

[54] Guoping Long, Jun Yang, Kai Zhu, and Wei Lin. 2018. FusionStitching: Deep Fusion and Code Generation for Tensorflow Computations on GPUs. *CoRR* abs/1811.05213 (2018). arXiv:1811.05213 http://arxiv.org/abs/1811.05213

[55] MATLAB. 2019. *version 9.7 (R2019b)*. The MathWorks Inc., Natick, Massachusetts.

[56] Nimrod Megiddo and Vivek Sarkar. 1997. Optimal Weighted Loop Fusion for Parallel Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. Association for Computing Machinery, New York, NY, USA, 282âĂŞ291. https://doi.org/10.1145/258492.258520

[57] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)* 37, 4 (12 2005). https://doi.org/10.1145/1118890.1118892

[58] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 561–577. http://dl.acm.org/citation.cfm?id=3291168.3291210

[59] OpenMP [n. d.]. OpenMP Specifications. http://openmp.org/wp/openmp-specifications.

[60] Dominic Orchard, Mistral Contrastin, Matthew Danish, and Andrew Rice. 2017. Verifying spatial properties of array computations. In *Proc. OOPSLA*. https://doi.org/10.1145/3133899

[61] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *8th Biennial Conference on Innovative Data Systems Research (CIDR'17)*.

[62] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 12. https://doi.org/10.1145/2491956.2462176

[63] Rob Romijnders. 2017. bigclam. https://github.com/RobRomijnders/bigclam.

[64] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 247–259. https://doi.org/10.1145/2967938.2967943

[65] Nobou Sato and W.F. Tinney. 1963. Techniques for exploiting the sparsity or the network admittance matrix. *IEEE Transactions on Power Apparatus and Systems* 82, 69 (12 1963), 944–950. https://doi.org/10.1109/TPAS.1963.291477

[66] Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2581122.2544155

[67] Sriseshan Srikanth, Anirudh Jain, Joseph M Lennon, Thomas M Conte, Erik Debenedictis, and Jeanine Cook. 2019. MetaStrider: Architectures for Scalable Memory-centric Reduction of Sparse Data Streams. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–26.

[68] Sriseshan Srikanth, Lavanya Subramanian, Sreenivas Subramoney, Thomas M Conte, and Hong Wang. 2018. Tackling memory access latency through DRAM row management. In *Proceedings of the International Symposium on Memory Systems*. 137–147.

[69] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 74–85. http://dl.acm.org/citation.cfm?id=3049832.3049841

[70] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[71] S. van der Walt, S. C. Colbert, and G. Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering* 13, 2 (March 2011), 22–30. https://doi.org/10.1109/MCSE.2011.37

[72] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. http://doi.acm.org/10.1145/2764454

[73] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/2503210.2503219

[74] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*. IEEE Computer Society, Washington, DC, USA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096

[75] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine learning at Facebook: understanding inference at the edge. In *Proceedings of the 2019 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2019.00048

[76] Jeffrey S. Young, Eric Hein, Srinivas Eswar, Patrick Lavin, Jiajia Li, Jason Riedy, Richard Vuduc, and Tom Conte. 2019. A Microbenchmark Characterization of the Emu Chick. *Parallel Comput.* 87 (2019), 60–69. https://doi.org/10.1016/j.parco.2019.04.012

[77] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491

A Optimization Algorithms

A.1 Loop Invariant Code Motion Algorithm

For completeness, Algorithm 1 summarizes the complete LICM algorithm discussed in Section 5.1.

Algorithm 1: Loop Invariant Code Motion Algorithm

Input : AST loop node N and its def/use information

```
1  begin
2      vars_ro := N's read-only variables
3      vars_l := N's local variables (defined w/o expose use)
4      stmts_inv := ∅
5      for each C ∈ { N's loop body } do
6          if C.vars_use ∈ vars_ro and C.vars_def ∈ vars_l then
7              candidates := { C }
8              for each S ∈ { C's successors } do
9                  mod := S.vars_def | S.vars_maydef
10                 if mod & C.vars_def ≠ ∅ then
11                     if subject_to(S, C) then
12                         candidates += { S }
13                     else
14                         candidates := ∅
15                         break
16             if candidates ≠ ∅ then
17                 stmts_inv += candidates
18                 vars_ro := vars_ro | C.vars_def
19     for each C ∈ stmts_inv do
20         Move C from loop body of N to the location immediately before N
21     for each C ∈ { N's loop body } do
22         exprs_inv := traverse C and collect sub-trees whose used variables are subset of vars_ro
23         for each expr_i ∈ exprs_inv do
24             Replace expr_i by new variable v_i in C
25             Create assign statement ast.Assign(v_i, expr_i) and insert immediately before N
26 def subject_to(S, C):
27     check_def := S.vars_def ∈ C.vars_def
28     check_use := S.vars_use ∈ (vars_ro | C.vars_def)
29     return check_def and check_use
```

Output: AST loop node N after LICM

A.2 Dense Element-wise Operation Fusion Algorithm

For completeness, Algorithm 2 summarizes the dense element-wise operation fusion algorithm discussed in Section 5.2.1.

The goal of this dense array optimization is to fuse element-wise operations that operate on arrays with the same rank (number of dimensions) and shape. Although possible, our algorithm does not fuse operations with different number of dimensions to avoid duplicating computations. On the other hand, an element-wise operation can accept arrays with different number of dimensions by broadcasting the one with fewer number of dimensions, which is implemented as array subscript expression in the generated code (e.g., `b[i2]` in Listing 7).

Given an AST statement node, N, our algorithm first infers the data type and number of dimensions of all child nodes C, annotated as $C.type$ and $C.ndims$[9] (line 3). For nodes corresponding to function calls, these annotations include information about return values. Then, it recursively identifies fusible sub-trees, consisting of multiple dense element-wise operations,

[9] $ndims = -1$ when the number of dimensions is statically unanalyzable.

Algorithm 2: Dense element-wise function fusion

```
   Input  : AST statement node N
1  begin
2      // Annotate all sub-nodes in N with type and ndims.
3      type_and_dimension_inference(N)
4      trees := ∅  // Table (key: variable, value: sub-tree)
5      split_subtrees_for_fusion(N, trees)
6      dense_codegen(N, trees)
7  def split_subtrees_for_fusion(N, trees):
8      if N is assign statement then
9          trees[N.target] := N.value
10         split_subtrees_for_fusion(N.value, trees)
11     else if N is function call then
12         for each A ∈ N's arguments do
13             if A is function call then
14                 if to_split(N, A) then
15                     v := new variable(A.type, A.ndims)
16                     trees[v] := A
17                     Replace A by v in N's arguments
18                 split_subtrees_for_fusion(A, trees)
19 def to_split(N, A):
20     ewf_1 := N is elem-wise function and N.ndims ≥ 1
21     ewf_2 := A is elem-wise function and A.ndims ≥ 1
22     return ewf_1 ≠ ewf_2 or ewf_1 = ewf_2 = True and N.ndims ≠ A.ndims
23 def dense_codegen(N, trees):
24     for each v ∈ trees as LIFO order do
25         stmt := create statement ast.Assign(v, trees[v])
26         if trees[v] is nested element-wise function and trees[v].ndims ≥ 1 then
27             n := trees[v].ndims
28             b := scalarize stmt by n dimensions
29             Create n nested loops with b as innermost loop body and insert immediately before N
30             Insert runtime call to allocate v
31             Insert runtime call to check compatibility of argument dense arrays (leaf nodes of N)
32         else
33             Insert stmt immediately before N
   Output : AST statement nodes generated by fusion
```

scalar expressions, and dense arrays with known number of dimensions. These are recorded in a hash table, *trees* (line 5). When a sub-tree includes non-fusible functions, each of them is split as a different sub-tree with a unique variable v and handled as an individual statement that defines v (lines 15–17). The code generation step iterates over *trees* in LIFO order. For each variable and sub-tree pair, it generates an assignment statement to v for each non-fusible sub-tree; or a nested loop with the scalarized body and the runtime compatibility check of argument arrays for each fusible sub-tree.

A.3 Sparsity Optimization Algorithm

For completeness, Algorithm 3 summarizes the sparsity optimization algorithm discussed in Section 5.2.2.

In addition to type and dimension inference for all child nodes of N, sparsity inference is performed on the sparse function nodes S that accept and return sparse arrays (i.e., `spm_mul`, `spm_add`, and `spmm` calls), identifying which arguments contribute to that the shape of S's returning sparse array (line 3). Analogous to the algorithm in Section 5.2.1, this algorithm recursively identifies sub-trees for sparsity optimizations, which consist of `spm_mul`, dense element-wise functions, supported linear algebra

functions, scalar expressions, and dense/sparse arrays with known number of dimensions. (line 5). A current limitation in the code generation phase is the shape of all sparse type nodes must be determined by a single sparse node *source*. Otherwise, they are handled as different sub-trees and optimized separately. Note that *source* can be the variable v that stores the result of other sub-trees recorded in $trees[v]$. The code generation step is also similar to that of the dense algorithm except that it generates loop nests that iterate over non-zero elements, which are obtained from the *source* node.

Algorithm 3: Sparsity optimization

```
   Input  : AST statement node N
1  begin
2      type_and_dimension_inference(N)
3      sparse_shape_inference(N)
4      trees := ∅
5      split_subtrees_for_sparse(N, trees, False)
6      sparse_codegen(N, trees)
7  def split_subtrees_for_sparse(N, trees, in_sparse_tree):
8      if N is assign statement then
9          trees[N.target] := N.value
10         cand := is_candidate(N.value)
11         split_subtrees_for_sparse(N.value, trees, cand)
12     else if N is function call then
13         for each A ∈ N's arguments do
14             if not in_sparse_tree then
15                 split := cand := is_candidate(A)
16             else
17                 cand := is_candidate(A) or to_fuse(N, A)
18                 split := not cand
19             if split then
20                 v := new variable(A.type, A.ndims)
21                 trees[v] := A
22                 Replace A by v in N's arguments
23             split_subtrees_for_sparse(A, trees, cand)
24 def is_candidate(N, tar_spm):
25     Check if 1) N is spm_mul; 2) N.ndims ≠ −1; and 3) N has a sparse argument that defines N's shape
26 def to_fuse(N, A):
27     Check if 1) A is not function; or 2) A is matmult, syrk, syr2k, or element-wise function and N is not matmult, syrk,
         or syr2k (cannot be nested)
28 def sparse_codegen(N, trees):
29     Similar to dense_codegen, but generate loops to iterate only non-zero elements
   Output : AST statement nodes generated by sparse opt
```

A.4 Allocation Hoisting Algorithm

For completeness, Algorithm 4 summarizes the allocation hoisting algorithm discussed in Section 5.3.

Allocation hoisting is similar to LICM as they both try to hoist invariant code outside of the loop. Compared to LICM, allocation hoisting is different in that it does not concern the value of the variables but concerns the liveness of the allocated memory. If we know that the previously allocated memory is no longer referenced at the current allocation site, then this memory space can be recycled. In case of a loop, the allocation can be hoisted out, and the allocated memory space will be recycled inside the loop thereby leading to a significant performance improvement.

We present an algorithm that identifies such allocation sites that can be recycled for a loop. The allocation sites we consider have the following form.

```
arr = empty(shape, dtype)
```

We assume `dtype` is a constant, and `shape` is a variable. In order to hoist the allocation outside, there are two conditions:

1. `shape` is a loop invariant.
2. Reference `arr` and all of its aliases are dead at this program point (right before this statement).

Condition 1 requires a loop invariant analysis pass, and some special handling for some known functioin calls may be needed to handle cases like `a = empty(shape(b))` where `b` has definition in the loop.

Condition 2 requires alias analysis to determine all references that could point to this memory region. Note this includes any pointer pointing in this memory range. For slicing operations like `subarr = arr.slice(s, e)`, `subarr` would also be considered an alias if `slice` creates reference by default instead of copying.

Also for nested loops, it is possible that an allocated memory can be reused in the inner loop but not in the outer loop. Starting from inner loops will make the iteration faster. Note that since `S`'s only dependence is `shape`, and the one of the allocation hoisting conditions is that `shape` must be a loop invariant, `shape` will need to be hoisted first.

Algorithm 4: Allocation Hoisting

Input : AST function node F and its variable liveness information

begin
  $aliased\_vars$:= set of array variables in F that are copied by reference
  repeat
    for each loop L in the function **do**
      for each $S : var = empty(shape)$ in $L.body$ **do**
        if *shape* is a loop invariant **and** *var* and all its aliases are not alive before S **then**
          move S immediately before $L.body$
  until *all allocation sites are hoisted as much as possible and data dependences are preserved*;

Output : AST loop node N after allocation hoisting

A pre-processing pass is needed to convert other allocation statements to `empty`, such as `empty_like` and `zeros`. Specifically `zeros` calls are converted an `empty` followed by `fill(arr, 0)`. During the hoisting pass, the `fill` calls will not be hoisted. Implicit allocations can also be handled by a pre-processing pass. For example, `c = a + b` where all `a`, `b`, and `c` are arrays can be converted to `c = empty_like(a)` followed by `add(a, b, out=c)`.

B Benchmark Kernel Codes

This section shows the Python, Julia, and Intrepydd source codes for the sinkhorn-wmd benchmark kernel that were used for the comparison in Section 6.5. Listing 11 shows the primary kernel code in Python. Listing 12 shows the primary kernel code in Julia. Listing 13 shows the primary kernel code in Intrepydd. As can be seen below, three versions are functionally equivalent despite variations in syntax.

```
    it = 0
    while it < max_iter:
        u = 1.0 / x
        v = c.multiply(1 / (K.T @ u))
        x = (1 / r) * K @ v.tocsc()
        it += 1
    u = 1.0 / x
    v = c.multiply(1 / (K.T @ u))
    return (u * ((K * M) @ v)).sum(axis=0)
```

Listing 11. Primary sinkhorn-wmd kernel in Python.

```
function kernel(K::Array{Float64,2}, M::Array{Float64,2},
                r::Array{Float64,2}, x::Array{Float64,2},
                max_iter::Int64, row::Array{Int32,1},
                col::Array{Int32,1}, data::Array{Float64,1},
                m::Int64, n::Int64)
    c = sparse(row, col, data, m, n)
    it = 0
    while it < max_iter
        u = 1.0 ./ x
        v = c .* (1.0 ./ (transpose(K) * u))
        x = ((1.0 ./ r) .* K) * v
        it += 1
    end
    u = 1.0 ./ x
    v = c .* (1.0 ./ (transpose(K) * u))
    return sum(u .* ((K .* M) * v), dims=1)
end
```

Listing 12. Primary sinkhorn-wmd kernel in Julia.

```
def kernel(K, M, x: Array(float64, 2),
           r, values, columns, indexes: Array(float64, 1),
           max_iter, ncols_c: int32) -> Array(float64, 1):
    c = csr_to_spm(values, columns, indexes, ncols_c)
    it = 0
    while it < max_iter:
        u = 1.0 / x
        v = c.spm_mul(1 / (K.T @ u))
        x = spmm_dense((1 / r) * K, v)
        it += 1
    u = 1.0 / x
    v = c.spm_mul(1 / (K.T @ u))
    return (u * spmm_dense(K * M, v)).sum(0)
```

Listing 13. Primary sinkhorn-wmd kernel in Intrepydd.