Legate Sparse: Distributed Sparse Computing in Python

Rohan Yadav
rohany@cs.stanford.edu
Stanford University
USA

Wonchan Lee
wonchanl@nvidia.com
NVIDIA
USA

Melih Elibol
melibol@nvidia.com
NVIDIA
USA

Taylor Lee Patti
tpatti@nvidia.com
NVIDIA
USA

Manolis Papadakis
mpapadakis@nvidia.com
NVIDIA
USA

Michael Garland
mgarland@nvidia.com
NVIDIA
USA

Alex Aiken
aiken@cs.stanford.edu
Stanford University
USA

Fredrik Kjolstad
kjolstad@cs.stanford.edu
Stanford University
USA

Michael Bauer
mbauer@nvidia.com
NVIDIA
USA

ABSTRACT

The sparse module of the popular SciPy Python library is widely used across applications in scientific computing, data analysis and machine learning. The standard implementation of SciPy is restricted to a single CPU and cannot take advantage of modern distributed and accelerated computing resources. We introduce Legate Sparse, a system that transparently distributes and accelerates unmodified sparse matrix-based SciPy programs across clusters of CPUs and GPUs, and composes with cuNumeric, a distributed NumPy library. Legate Sparse uses a combination of static and dynamic techniques to efficiently compose independently written sparse and dense array programming libraries, providing a unified Python interface for distributed sparse and dense array computations. We show that Legate Sparse is competitive with single-GPU libraries like CuPy and achieves 65% of the performance of PETSc on up to 1280 CPU cores and 192 GPUs of the Summit supercomputer, while offering the productivity benefits of idiomatic SciPy and NumPy.

ACM Reference Format:
Rohan Yadav, Wonchan Lee, Melih Elibol, Taylor Lee Patti, Manolis Papadakis, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. 2023. Legate Sparse: Distributed Sparse Computing in Python. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607033

1 INTRODUCTION

Python is a widely used language for data science, machine learning, and scientific computing due to its ease of use and large ecosystem of numerical libraries. This ecosystem includes NumPy [13] for dense array-based computations and SciPy's [35] Sparse module for sparse matrix-based computations, both of which serve as foundations for numerous applications and frameworks.

Despite their widespread use, the canonical implementations of NumPy and SciPy target a single CPU node, with only select operations supporting multiple threads. As data set sizes and application computational demands continue to increase, there is a need to target resources more powerful than what a single CPU-only node can provide. Recent work has made great strides in this area for dense array programming systems [5, 22, 24, 29], but the automatic distribution and acceleration of SciPy-based sparse matrix programs has not yet been achieved.

SciPy or CuPy [22] (a single-GPU implementation of NumPy and SciPy) can be paired with a communication library like MPI or NCCL, or a task-based library like Dask Distributed [29] or Ray [21] to enable distributed execution. However, this composition requires the user to manually partition and communicate data, resulting in non-trivial code modification and necessitating distributed programming expertise. The industry-standard sparse linear algebra systems PETSc [3, 20] and Trilinos [34] expose Python wrappers around their low-level C/C++ APIs. While these APIs provide many high-level sparse matrix operations, they require programmers to reason about data distribution and data movement, a level of expertise many programmers do not have.

Our goal in this work is to develop a system that scales unmodified SciPy Sparse programs across distributed machines with good performance, and efficiently composes with cuNumeric [5], a distributed NumPy library. This system would provide the familiar dense and sparse array programming interfaces to allow users with and without expertise in distributed programming to rapidly prototype distributed applications and scale these applications to the size of machine needed to process their datasets. In this paper, we explore the large design space encompassed by these constraints, and demonstrate one design point that achieves our goals.

Achieving our goal of building a distributed and heterogeneous sparse array programming library that achieves both high performance as well as composability with an external dense array programming library requires solving a global problem of performance composability at multiple layers of the software stack. First,

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607033

unlike when developing a monolithic distributed library, operations launched by each separate library should compose with operations launched by other libraries at the distributed layer. This means that both libraries must share distributed data representations efficiently and perform only necessary synchronization and communication. Second, each library's operations must agree on the processor varieties targeted in a heterogeneous system. For example, if even one operation issued by a user's program does not have a GPU implementation, the data movement caused by falling back to a CPU implementation can significantly impact performance. Third, library operations must agree on the types of their data structures, especially when those data structures are sparse. When operations are not implemented on the program's requested sparse data structures, expensive format conversions to supported data structures can dominate program execution time and increase memory usage.

To compose at these three layers of the software stack, the implementation of each library needs to be flexible at each layer. At the distributed layer, each library must be flexible with the partitioning schemes used for individual operations to compose with how operations launched by the other library may partition data. And at the layers of processor varieties and types of data structures, the sparse library must support a myriad set of variants for each potential such combination, which also need be specialized to the chosen partitioning schemes.

We implement the flexibility and generality required of a distributed sparse array programming library that composes with cuNumeric through a careful separation of decisions made statically (during the implementation of Legate Sparse) and dynamically (during the execution of Legate Sparse programs). The combination of static and dynamic decisions is key to a successful implementation of Legate Sparse: we believe that a fully static approach is likely to sacrifice composability or implementation maintainability, while a fully dynamic approach is likely to sacrifice performance. Some decisions must be made statically to specialize kernels to processor kinds and sparse data structures, while other decisions must be postponed until runtime, where the specific interactions between different libraries are known. We divide the space of static and dynamic decisions in the implementation of Legate Sparse with the following key ideas:

- **Composable Distribution.** To compose distributed operations across libraries, we combine a constraint-based description of data distributions with a first-class representation of data partitions. This combinations allows for the compact encoding of potential distribution strategies for each operation statically. Then, we defer the decision of what concrete data partitions to use for each operation until runtime. We dynamically select partitions that satisfy the distribution constraints and align with existing data distributions, allowing (to the extent possible) for data to be operated on where it exists in the machine.
- **Compiler-Aided Kernel Generation.** To implement the large number of variants required to run each operation with each sparse data structure on each heterogeneous processor kind, we leverage the DISTAL [36, 37] sparse tensor algebra compiler. We utilized DISTAL to ahead-of-time generate distributed kernels specialized to each data format, processor variety and partitioning scheme, enabling Legate Sparse to dynamically dispatch across a wide set of statically-generated specialized kernels.
- **Dynamic Dependence and Communication Analysis.** Dynamic dependence and communication analyses enable independent libraries to launch work while ensuring precise synchronization and communication between the libraries. We implement Legate Sparse through a translation to the programming model of the Legion [6] runtime system, and leverage it to overlap computation and perform precise, data-dependent communication across library boundaries.

```
# Try to import cuNumeric and Legate Sparse,
# and fall back to NumPy and SciPy if not present.
try:
  import cunumeric as np
  import legate.sparse as sp
except ImportError:
  import numpy as np
  import scipy.sparse as sp

# Generate a random sparse matrix.
A = sp.random(n, n, format='csr')
# Make a positive semi-definite matrix from A.
A = 0.5 * (A + A.T) + n * sp.eye(n)

# Estimate the maximum eigen-value via the Raleigh quotient.
x = np.random.rand(A.shape[0])
for _ in range(iters):
  x = A @ x
  x /= np.linalg.norm(x)
result = np.dot(x.T, A @ x)
```

Figure 1: Legate Sparse and cuNumeric program that runs on a GPU cluster and falls back to SciPy and NumPy.

We have developed a prototype implementation of Legate Sparse that composes with cuNumeric to distribute and accelerate unmodified Python programs that use NumPy and SciPy, such as the eigenvalue estimation computation in Figure 1. Legate Sparse achieves the transparent distribution of SciPy and NumPy programs in a maintainable way: the implementations of Legate Sparse and cuNumeric are each unaware of the other library's implementation, and a large number of the kernels in Legate Sparse were automatically generated. Our prototype implements 35% of the SciPy Sparse API, which is sufficient to express complex computations from scientific computing and machine learning.

We evaluate the performance of Legate Sparse on the Summit supercomputer with a combination of NumPy and SciPy based workloads with varying complexity, ranging from 10 to 1000 lines of code. Our benchmark suite contains iterative linear solvers (conjugate gradient, geometric multi-grid), Runge-Kutta integration, and sparse matrix factorization. Our experiments show that, for a drop-in NumPy and SciPy replacement, Legate Sparse achieves good scalability to 1280 CPUs and 192 GPUs, achieving 65% of the performance of PETSc. We also show that Legate Sparse delivers comparable performance with CuPy on a single GPU and outperforms SciPy on a single CPU socket, while effectively scaling to larger numbers of processors.

2 BACKGROUND

In this section, we provide background on the SciPy Sparse module, and discuss components of the library relevant to this work. We then provide background on the Legion [6] runtime system, which both Legate Sparse and cuNumeric are built upon. Finally, we discuss how cuNumeric maps onto Legion's abstractions.

2.1 SciPy Sparse

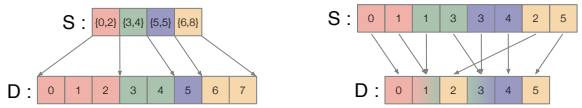
SciPy Sparse [2, 35] is a sub-module of the SciPy Python library that provides a high-level API for linear algebra operations over different types of sparse matrices. SciPy Sparse supports several common sparse matrix formats, including the CSR (compressed sparse rows), CSC (compressed sparse columns), DIA (diagonal) and COO (coordinate) formats, and supports format conversions and data reorganization operations between these formats. On these sparse matrices, SciPy Sparse supports a variety of basic mathematical operations, such as matrix-vector products, matrix-matrix products and diagonal computations, as well as higher-level linear algebra operations like iterative solves and eigenvalue computations. SciPy Sparse is directly composable with NumPy, as many operations within the API natively accept and return NumPy arrays. The standard implementation of SciPy implements the API with a combination of calling out to C operations and utilizing existing NumPy routines. Finally, the SciPy Sparse API has no notions of execution or distribution strategies, meaning that all parallelism performed by Legate Sparse must be implicit.

2.2 Legion

Legion [6] is a data-centric, task-based runtime system organized around *region* data structures that are *partitioned* into *sub-regions* and computed on by user-defined *tasks*. Legion performs dynamic analysis to extract parallelism from sequential user programs, represented as streams of tasks, by identifying tasks within the stream that operate on independent regions, and automatically inserting the necessary communication and synchronization to preserve the sequential semantics of the input program.

Data Model. All long-lived and distributed data is described through *regions*, which are multi-dimensional arrays. Regions are the underlying data structures that back both cuNumeric's distributed arrays and Legate Sparse's sparse matrices. Parallelism in Legion is expressed through the *partitioning* of regions into *sub-regions*. A partition P of a region R is a first-class object that represents the mapping from a set of *colors* to subsets of the indices of R. Partitions need not be disjoint, nor do they need to cover the whole index space: the sets of indices in P can overlap or *alias*, and their union does not need to cover all indices in R.

Legion supports *dependent partitioning* [33] operations for creating partitions from existing partitions. The most important dependent partitioning operation to this work is *image*, which operates on a source region that contains indices pointing into a destination region, and projects a partition of the source region onto the destination. Intuitively, given a partition of the source region, the image operation colors all indices in the destination region with the same color as the partition of each index in the source region. More precisely, consider a source region S and a destination region D, where elements in S are sets of indices in D. Given a partition P of S, the image of S to D is a partition P' of D such that $\forall c \in P, \forall i \in P[c], S[i] \subseteq P'[c]$.



(a) Image (by range) **(b) Image (by coordinate)**

Figure 2: Visualization of the image partitioning operation.

The image operation is demonstrated in Figure 2, where Figure 2a shows an image from a source region that contains ranges of indices, and Figure 2b shows a source region that names individual indices. Note that the partition of D created by the image in Figure 2(b) is aliased, because the 1st and 3rd elements of D are included in two sub-regions each. Image is a powerful operator that allows us to express co-partitioning of the indexing arrays that are often used to represent sparse matrices, and to capture the data-dependent communication patterns that arise in sparse computations.

Tasks. Tasks are the atomic unit of computation in Legion. Tasks are arbitrary, user-defined computations that operate on regions, and declare how they will use each region (read, write or reduce). Legion extracts dependencies between tasks, and inserts communication operations for the regions on which a task operates.

Mapping. Legion programs do not directly encode machine-specific decisions such as on what processors tasks should run and in what memories regions should be allocated. Instead, a separate *mapper* makes these decisions for the application at runtime, allowing the application to remain unchanged when porting to a new machine or during certain kinds of performance tuning.

2.3 cuNumeric

cuNumeric [5] is a distributed and accelerated drop-in replacement for NumPy. Like Legate Sparse, cuNumeric is implemented via a dynamic translation from the NumPy API to Legion. cuNumeric represents NumPy arrays as Legion's regions, partitions the regions for parallel processing, and launches tasks corresponding to NumPy operations. The original version of cuNumeric as described by Bauer et al. [5] selects partitions of regions for individual operations by maintaining a *key partition* for each region that tracks the latest written partition of the region. NumPy operations between multiple arrays choose partitions of arrays that keep the key partition of the largest region involved in an operation in place. Additionally, the original version of cuNumeric employs a specialized mapper that encodes heuristics for mapping NumPy programs, such as choosing when to distribute tasks, selecting the tile size to partition data into, and the mapping of tasks and regions onto processor and memories. To enable composability with Legate Sparse, we modify the partitioning strategies within cuNumeric to use the constraint-based system discussed in Section 4.1, and introduce composition-aware mapping algorithms to cuNumeric's mapper, as discussed in Section 4.2.

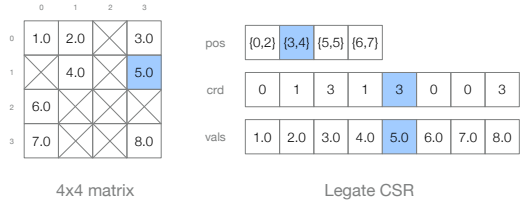


Figure 3: Legate Sparse's CSR sparse matrix encoding.

3 SPARSE DATA REPRESENTATION

The standard single-node representations of common sparse matrix formats store metadata about the indices of non-zero matrix entries and their values in packs of arrays. For example, the COO (coordinate) format stores three arrays, where the first two arrays store the row coordinate and column coordinate of each non-zero entry of the matrix, and the last array stores the value. The CSR (compressed sparse rows) format further compresses the COO format by implicitly representing the rows that contain non-zero entries: it maintains an array (often called `pos` or `indptr`) where the column coordinates and values for row i are stored within range $[\texttt{pos}[i], \texttt{pos}[i{+}1])$ of an array called `crd`. The CSC format is similar to CSR, but compresses the columns instead of the rows.

We use Legion's regions to extend these single-node representations into distributed sparse matrix data structures by mapping each of the arrays used to represent sparse matrices to regions. For instance, the row, column and value arrays in the COO format are represented directly as regions in Legate Sparse. Formats such as CSR and CSC are represented in a similar manner, but store the range of coordinates for a row or column i in a tuple at `pos[i]`, as depicted in Figure 3. This small variation from the standard representation allows us to directly employ Legion's image partitioning operation to relate partitions of the `pos` and `crd` regions with one another. We also use images to relate partitions of the `crd` region with referenced indices in dense vectors and matrices. For example, consider a distributed SpMV ($y = A \cdot x$), where A is stored as CSR. Performing an SpMV requires accessing the locations in x corresponding to the non-zero coordinates stored in A's `crd` region. We use an image from the partition of A's `crd` region to compute the referenced locations of x. An example of this operation is discussed in Figure 5 in Section 4.3. Images allow for the co-partitioning of the regions used to define sparse data structures, and to implement MPI-like scatter/gather operations in a high-level manner.

Our decision to represent sparse matrices as a set of regions instead of a collection of local sparse matrices per rank (as used by PETSc and Trilinos) has both benefits and downsides. Using a set of regions aligns more closely with the Legion programming model that we target, and careful choices of partitioning enables description of non-trivial communication patterns. Additionally, this choice allows for interoperation with Legate Sparse: since sparse matrices are constructed from regions, users can directly construct sparse matrices out of cuNumeric arrays, or extract and operate on the arrays that back a sparse matrix. A downside of this decision is that the partitioned pieces of the global sparse matrix passed to individual tasks are not valid sparse matrices from the perspective of external libraries like cuSPARSE. As a result, we pay a small performance penalty when reshaping these local pieces into formats accepted by these libraries when we use them. Our evaluation (Section 6) shows that our sparse matrix representation has low overhead while allowing for direct use Legion's API and close alignment with SciPy Sparse's programming model.

4 COMPOSABLE PARALLELIZATION

Our goal with Legate Sparse was to distribute and accelerate SciPy Sparse workloads while efficiently composing with cuNumeric. In this section, we describe the techniques employed to enable the performant composition of Legate Sparse with cuNumeric at the distributed layer of each library. We describe how Legate Sparse and cuNumeric partition the previously discussed sparse and dense data structures, and how each library launches parallel operations over the partitioned data using Legion. We show how abstractions built on top of Legion's partitions enable concise and composable parallel implementations of distributed operations, and describe how to map these operations onto physical hardware in a composable manner.

4.1 Constraint-Based Parallelization

Legate Sparse and cuNumeric distribute SciPy and NumPy programs by translating each operation into a set of task launches over partitioned regions. The selection of what partitions to use for each task launch has a significant impact on performance. For example, if two tasks t_1 and t_2 operate sequentially on a dense matrix M, t_1 selects a row-wise partition of M, and t_2 selects a column-wise partition of M, then a distributed transpose must be performed after t_1 has completed to put the data in the required distributed layout for t_2. To be performance-composable across operations, we need to re-use existing partitions whenever possible. However, to keep the implementation maintainable, we do not want every operation to have to explicitly consider all possible partitions of every input.

We resolve this tension by leveraging recent work in constraint-based automatic parallelization, introduced by Lee et al. [17]. We add a layer of indirection to task definitions and launches where, instead of describing the exact partitions that tasks should operate on, tasks describe what regions they will operate on and *constraints* on how those regions should be partitioned. Constraints can be simple, such as declaring that two regions must have aligned partitions (for an element-wise operation), or complex constraints that invoke dependent partitioning operations (such as relating the `pos` and `crd` regions in a CSR matrix by an image). We use a constraint solver inspired by Lee et al. [17] to select concrete partitions of each region that satisfy all of the declared constraints. The constraints are designed such that there is always at least one solution; if more than one solution is possible, the solver picks the solution that re-partitions the least amount of data. We refer to Lee et al. for a formal discussion of the constraint language and solving process.

We now discuss an example of the task launching process with constraints using the row-based distributed SpMV example in Figure 4. Upon execution of the task launching code in Figure 4, the `task` object contains the following partitioning constraints: `equals(y, pos)`, `image(pos, crd)`, `image(pos, vals)`, and `image(crd, x)`. Intuitively, these constraints mean the following: 1) the same partition must be selected for `y` and `pos`, 2) the partitions of `crd` and `vals` must be the result of an image from the

```
def spmv(self, A, x):
  # Compute y = A @ x and return y.
  y = cunumeric.zeros(A.shape[0])
  task = ctx.create_task(ROW_SPLIT_SPMV)
  # Add all regions to the task.
  task.add_output(y)
  task.add_input(A.pos, A.crd, A.vals, x)
  # Describe partitioning constraints.
  task.add_alignment_constraint(y, A.pos)
  task.add_image_constraint(A.pos, [A.crd, A.vals])
  task.add_image_constraint(A.crd, x)
  task.execute()
  return y
```

Figure 4: Python implementation of a row-based distributed CSR SpMV (adapted from DISTAL generated code).

selected partition of `pos`, and 3) the partition of `x` must be the result of an image from the selected partition of `crd`. The constraint solver realizes that the choices of partitions for `y` and `pos` are independent, while the partitions for `crd`, `vals` and `x` are dependent on choices for partitions for other regions. Then, the solver examines the existing partitions for `y` and `pos` and selects the existing partitions if they are aligned. Otherwise, it selects an existing partition that keeps the sparse matrix in place. Once these initial partitions have been selected, the solver uses Legion's image operation to construct partitions of `crd`, `vals` and `x` to satisfy the remaining constraints.

We developed Legate Sparse using the constraint system, and adapted the implementation of cuNumeric to use the same system. The constraint-based design is key to achieving performance composability at the distributed layer of our system for two reasons:

- **Partition reuse**. The constraint formulation enables reusing partitions across individual operations and libraries. Operations defined by Legate Sparse can consume partitions created by cuNumeric and vice-versa, avoiding unnecessary data movement when passing data between Legate Sparse and cuNumeric.
- **Localization of operation definitions**. Because each task only describes what partitions are possible to use, each task is defined independently. Existing operation implementations do not need to consider partitioning strategies defined in the future, and new operation implementations do not need to consider all possible existing partitioning strategies. The most important outcome of this design is that the cuNumeric and Legate Sparse implementations are *completely unaware of the other*. The lack of coupling streamlines development and is promising for the development of future libraries using the same strategy.

4.2 Composable Mapping

By representing sparse and dense arrays with regions and launching tasks using constraint-based parallelism, Legate Sparse and cuNumeric issue a stream of tasks to Legion. To execute this stream of tasks, Legate Sparse and cuNumeric must instruct Legion on which processor each task should run on, and in which memory to allocate each region of each task. Legate Sparse and cuNumeric communicate these decisions through separate *mapper* objects, which Legion queries before executing tasks. Proper mapping decisions are key to achieving high performance — if mapping decisions are not made in a coordinated manner between Legate Sparse and cuNumeric, performance degradation can occur due to unnecessary data movement. While the library implementations of Legate Sparse and cuNumeric are independent, we introduce a point of coupling at the runtime layer between the libraries by sharing mapper infrastructure and mapping policies between the two.

Legate Sparse and cuNumeric use the same strategy for mapping tasks to processors to ensure a consistent assignment. A consistent processor mapping strategy ensures that data does not thrash between operations launched by different libraries. For example, if Legate Sparse and cuNumeric launch an element-wise operation in series, using the same processor mapping strategy ensures no data movement occurs between the operations.

The more difficult aspect of composing mapping decisions across libraries is the mapping of regions onto memories in the machine. The key challenge involved in mapping regions is how to share allocations of distributed and partitioned data between libraries. Because interactions between libraries are unknown until execution, the partitions of regions created by libraries and the aliasing of those partitions are also not known until program execution. To minimize data movement and memory usage, the mapping strategy used by Legate Sparse and cuNumeric must reuse and resize region allocations across individual operations and library boundaries. We facilitate the reuse of region allocations by having the mappers for Legate Sparse and cuNumeric record all region allocations made in a shared store on each node, and having the mappers query the store on their local node before making new allocations.

However, reusing allocations that exactly match the extents of a region is not sufficient to achieve good performance, as tasks from different libraries may use multiple views of the same underlying region. As an example, consider a stencil computation, where a task reads from multiple tiles offset around a center tile. An efficient mapping for this computation would coalesce all offset tiles with the center tile into a single, larger allocation, reducing the total amount of memory and increasing cache locality.

To efficiently map multiple sub-regions into a single allocation, our shared mapping strategy employs a *coalescing* step before performing allocations. When selecting an allocation for a region, mappers examine the existing allocations for other sub-regions of the same parent region. If another sub-region has an intersection with the region being allocated, then the mapper has an option of merging the two views into a new, larger allocation with enough space for both regions. Tasks using the larger allocation then operate on slices of the allocation corresponding to the desired sub-region. We use a heuristic to drive coalescing decisions, where sub-regions are coalesced if the size of their overlapping components is sufficiently larger than their non-overlapping components. The coalescing step is key to reducing overall memory usage and eliminating redundant data movement. A concrete example is discussed in Section 4.3.

4.3 Execution Example

Figure 5 depicts the execution of the program displayed in Figure 1 with Legate Sparse and cuNumeric. Figure 5 shows how the sparse matrix-vector multiplication (SpMV) launched by Legate Sparse and the norm and division operations launched by cuNumeric share partitions and physical resources. The top half of the figure shows

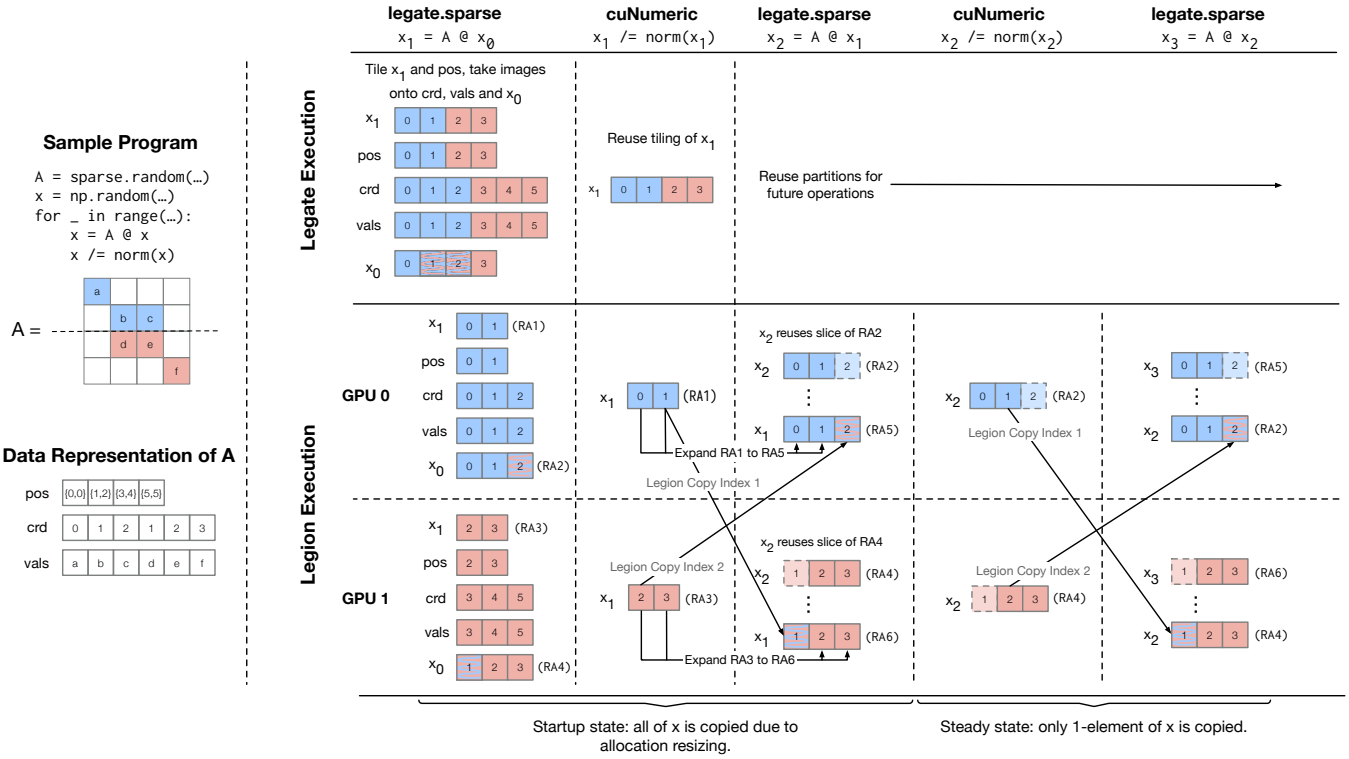


Figure 5: Execution of the program in Figure 1, with control flowing between Legate Sparse and cuNumeric. The left part of the figure contains an excerpt of the program and an example matrix A and its data layout. The right part of the figure is the execution; the top half depicts partitioning and launching of tasks in Legate, while the bottom half shows the Legion-level execution on the physical machine. In the right part of the figure, each region entry is labeled with the coordinate of the entry.

the execution of the Python Legate task launching logic, and the bottom half shows the physical execution with Legion on a 2 GPU system. An efficient implementation only performs one element halo exchanges of the x vector, and no other copies. We show how Legate Sparse and cuNumeric interoperate to achieve this strategy.

Throughout the figure, we refer to the versions of the vector x at each iteration i of the main loop with x_i. For example, the initial vector x is denoted x_0, and the resulting x after the first $x = A @ x$ operation is denoted as x_1. Next, all region entries in the figure are labeled with the coordinate of that entry within the region.

We first discuss the top half of the figure, which shows how Legate Sparse maps arrays onto regions and partitions these regions for distributed execution. The matrix A is organized in CSR as three separate regions, `pos`, `crd` and `vals`, as discussed in Section 3. The original vector x_0 is represented by a one-dimensional region. When the program launches the first SpMV, Legate Sparse creates a new region for the output vector x_1. Solving the constraints for SpMV described in Figure 4, Legate Sparse selects an aligned tiling of x_1 and `pos`. To satisfy the image constraints, Legate Sparse invokes Legion's image operation to create partitions of `crd`, `vals` and x_0 from the tiling. We use blue and red colors to show the resulting partitions of each region. Note how the image from `crd` into x_0 creates an *aliased* (colored blue and red) partition. Legate Sparse launches SpMV tasks over the partitions, dispatching the tasks to Legion. Next, control flows to cuNumeric for the norm and division operations, which we treat as a single operation for illustration purposes. These are element-wise operations without partitioning constraints, so cuNumeric selects the tiling of x_1 created by Legate Sparse. After cuNumeric launches the norm and division tasks, the loop repeats, and all partitions are reused by future iterations.

We now shift to the bottom half of the figure, which depicts the execution with Legion, and the mapping of logical operations onto physical resources. For all tasks launched, the Legate Sparse and cuNumeric mappers assign tasks and regions to each GPU and the corresponding framebuffer memory. The key to peak performance in this program is the mapper's choice of allocations for each region.

In the first iteration, the Legate Sparse and cuNumeric mappers make region allocations that correspond to the bounds of each region. The choices made in the second iteration of the program stress the importance of the compositional-awareness of the Legate Sparse and cuNumeric mapping strategies. When mapping the second SpMV operation, the mapper chooses new allocations (RA5 and RA6) for each piece of x_1, resizing the allocations RA1 and RA3 to account for the larger slice of x_1 required by each SpMV task. Resizing RA1 and RA3 requires a full copy of x_1, and a single element halo-copy between GPUs. Next, Legate Sparse sees that x_0 has gone out of scope, and chooses to reuse the allocations RA2 and RA4 by coalescing them into the requested sub-regions for x_2. Since

the allocations have been coalesced, the SpMV tasks only operate on a slice of the allocation, and similarly with the norm and division tasks. The elements outside of these tasks' slices are denoted by a faded color. At the start of the third iteration, the application hits a steady state where the existing allocations are large enough to re-use without additional resizing, causing only the single-element halo-copy to occur. Without the mapper's coalescing step, the full vector copy executed in the first iteration would be executed in each iteration, resulting in a significant loss of performance.

This example demonstrates how the use of constraint-based parallelization enables logically isolated implementations of operations to compose, and how information can be shared during mapping to extract efficient communication patterns.

5 LIBRARY KERNEL IMPLEMENTATION

Having described the abstractions with which distributed operations are defined in Legate Sparse, we now discuss our process for implementing the SciPy Sparse API. Our prototype supports the COO, CSR, CSC and DIA sparse matrix formats, and of the estimated 492 functions in SciPy Sparse, our prototype implements 176 (35%) functions; 14 were implemented by using the DISTAL compiler, 156 were ported from existing SciPy or CuPy implementations, and 6 had to be handwritten. In this section, we discuss these three cases, as well as the portions of the API that we have not yet implemented.

5.1 Generating Kernels with DISTAL

We used the DISTAL [36, 37] compiler to generate implementations for components of the SciPy Sparse API that perform tensor algebraic computations. These functions are performance critical (such as SpMV or SpMM), and require custom code tailored to the specific operation, sparse matrix formats and target hardware. This custom code is tedious and difficult to write; despite DISTAL being used to generate implementations of only 14 functions in the SciPy Sparse API, the generated code accounts for 46% of the total C++ and CUDA in Legate Sparse (2854/6135 LOC) and 12% of the total Python in Legate Sparse (697/5748 LOC). By generating this performance sensitive code, we enhance the maintainability of Legate Sparse, and allow developer time to be spent elsewhere when optimizing the library. We give an overview of DISTAL, and how it was used to generate code for Legate Sparse.

DISTAL compiles a tensor algebra domain specific language (DSL) into C++ code targeting the Legion runtime. DISTAL allows for the separate specification of 1) desired tensor computation, 2) sparse data format of each operand, 3) the distributed algorithm to use, and 4) the data distribution of the operands. This flexibility allows for the high level description of many kernels of interest within SciPy. The constraint solver discussed in Section 4.1 considers the existing data distributions of regions, so we only use the first 3 input languages of DISTAL. DISTAL generates code directly targeting the Legion API, so we perform slight manual modifications to the generated code to target our higher-level abstractions; these changes could be automated, but we have not found the manual work to be burdensome for developing our prototype.

DISTAL code to generate a distributed and multi-threaded CPU SpMV is found in Figure 6, the generated C++ task body is found in

```
// Runtime parameters: input sizes and processors.
Param n, m, procs;
// Define the tensor operators.
Tensor<double> y({n}, {Dense}), x({m}, {Dense});
Tensor<double> A({n, m}, {Dense, Compressed});
// Describe the desired computation.
IndexVar i, j, io, ii;
y(i) = A(i, j) * x(j);
// Schedule the computation.
DISTAL::compile(y.schedule()
                 .divide(i, io, ii, procs)
                 .distribute(io)
                 .communicate(io, {y, A, x})
                 .parallelize(ii, CPUThread));
```

Figure 6: Distributed, multi-threaded CSR SpMV in DISTAL.

```
void CSRSpMVTask(vector<Region> regions) {
  auto y = regions[0];
  auto pos = regions[1];
  auto crd = regions[2];
  auto vals = regions[3];
  auto x = regions[4];
  #pragma omp parallel for
  for (int i = y.bounds.lo; i <= y.bounds.hi; i++) {
    auto val = 0.0;
    for (int jA = pos[i].lo; jA <= pos[i].hi; jA++) {
      val += vals[jA] * x[crd[jA]];
    }
    y[i] = val;
  }
}
```

Figure 7: DISTAL-generated C++ task for row-based, multi-threaded CSR SpMV, with minor modifications.

Figure 7, and the constraint based task launching code in Figure 4 is the result of adapting DISTAL-generated C++ task launching code. The DISTAL C++ code declares some runtime parameters, initializes the tensor operands, describes the desired computation, and then schedules the computation for the target machine. The algorithm specified by the scheduling language distributes the rows of the matrix across all processors, and then parallelizes execution across the rows between CPU threads. To achieve peak performance on GPUs, we hand-modified the DISTAL-generated CUDA code to make calls into cuSPARSE when applicable. In our experience, this aspect was the most error prone step in developing the sparse linear algebra kernel implementations and could be made easier in the future with better compiler support for external library interaction, such as in the Mosaic system [4].

5.2 Porting SciPy and CuPy Implementations

The largest subset (156/176 functions) of our implementation of SciPy Sparse was done by porting existing implementations of the API in SciPy and CuPy. While developing Legate Sparse, we found that many functions in SciPy Sparse were implemented using parallel NumPy operations and previously defined SciPy Sparse kernels. By focusing our system design on composability with cuNumeric, we were able bootstrap our library with itself and

cuNumeric to obtain distributed and accelerated implementations of these functions without any distributed programming.

The classes of functions that we were able to directly port varied in complexity. The simplest of these functions were non-zero preserving, element-wise, unary operations on sparse matrices that are implemented by using the corresponding NumPy operation on the array storing the values of the sparse matrix. Some more complicated ported functions include computing sums across different axes of sparse matrices, and format conversions between sparse matrix formats. The most complex operations that we directly ported to Legate Sparse were higher-level operations such as solves and integrations. We ported several iterative linear solvers (CG, CGS, BiCG, BiCGSTAB, GMRES), Runge-Kutta integration and eigensolvers from SciPy and CuPy implementations to distributed implementations using Legate Sparse and cuNumeric.

5.3 Hand-Written Implementations

The final group of functions in Legate Sparse were those that required completely hand-written implementations. These functions include sorts and auxiliary operations that are implemented within SciPy with calls to C/C++ or Python loops that directly index into NumPy arrays. For these operations, we developed distributed and accelerated implementations using the constraint-based parallelism framework discussed in Section 4.1 paired with C/C++ and CUDA code for tasks adapted from the SciPy implementations.

5.4 Unimplemented Components

Having covered how we implemented components of SciPy Sparse, we now discuss the remaining portions of the API and the path forward to implementing them. Out of the 316 remaining functions in SciPy Sparse, 116 are defined on sequential matrix formats (list-of-lists and dictionary-of-keys) used for matrix assembly in shared memory, which we do not plan to support. 72 of the remaining 200 functions are defined on the BSR (block sparse rows) sparse matrix format, which we plan to support, and are able to use DISTAL to generate kernels for. This leaves 128 functions in SciPy Sparse defined on sparse matrix formats that we support in Legate Sparse (CSR, CSC, DIA, COO). Of these functions, we believe there is a path forward to a nearly complete implementation: 8 are possible to generate with DISTAL, 44 are possible to port from SciPy, 60 require a combination of porting and hand-writing, and 14 are specific to SciPy's implementation. The functions that require hand-writing cover different components of the API, including sparse matrix reshaping operators, operators that slice and update pieces of sparse matrices, and functions that call to external libraries like SuperLU.

6 EVALUATION

Experimental Setup. We evaluated the performance of Legate Sparse on the Summit supercomputer. Each Summit node has a 40 core dual socket IBM Power9. Each socket has three NVIDIA Volta V100s connected by NVLink 2.0, for a total of six GPUs per node. Each node is connected by an Infiniband EDR interconnect. We compile all code using GCC 9.3.0 and CUDA 11.0.2. Legion was configured with GASNet 2022.9.0 for inter-node communication.

Overview. We evaluate the performance of Legate Sparse by testing it on a set of SciPy programs from the scientific computing and machine learning domains. The set of benchmarks range from twenty-five to nearly a thousand lines of code and from microbenchmarks to full applications, displaying the complexity of the programs Legate Sparse is able to execute. All of the benchmarks use cuNumeric, and stress the interaction with Legate Sparse.

We measure Legate Sparse's performance running in both CPU-only and GPU-only settings, allowing us to compare against systems that only support CPUs or GPUs. On a single node, we compare against the standard implementations of SciPy and NumPy for CPUs, and CuPy for GPUs. CuPy provides a drop-in replacement for the SciPy and NumPy APIs, but can only utilize a single GPU. On multiple nodes, we compare (when a hand-tuned baseline exists) against the industry-standard PETSc sparse linear algebra library, which supports both CPUs and GPUs. PETSc provides a C API with high-level linear algebra operations similar to SciPy, but requires users to both specify low-level details about partitioning and distribution and hand-write many distributed NumPy-like array computations. For all experiments, we collect 12 runs of data points, drop the fastest and slowest runs, and then average the results of the remaining 10 runs.

6.1 Weak Scaling Experiments

In this section, we evaluate the weak-scaling performance of Legate Sparse, emulating a usage where users increase the size of their machine to scale to larger data sets. For all benchmarks but the quantum simulation, we compare the performance between one socket of CPUs and the three GPUs connected to that socket. However, we start the weak-scaling at one GPU to compare performance with CuPy. We plot throughput on a log-log plot due to the order-of-magnitude difference in performance between various systems.

SpMV Microbenchmark. Our first experiment is a microbenchmark for the scaling of the SpMV operations on banded sparse matrices. This benchmark is trivially parallel with no communication, and Figure 8 shows that both Legate Sparse and PETSc achieve perfect weak scaling. Most SciPy operations are single-threaded and cannot benefit in performance from additional cores or memory bandwidth of an additional CPU socket, resulting in no weak-scaling. As discussed in Section 3, our choice of using a global sparse matrix representation in Legate Sparse incurs some overhead from reshaping operations to the local partitions of the sparse matrices before passing the resulting local matrices to cuSPARSE, resulting in the slight performance differences between Legate Sparse versus CuPy and PETSc.

CG Solver. We implemented a conjugate-gradient iterative linear solver for a 2-D Poisson problem, with the results displayed in Figure 9. As with the previous experiment, we compare both modes of Legate Sparse to the same code run in SciPy and CuPy, and a comparable implementation in PETSc. As seen in the SpMV microbenchmark, Legate Sparse's CPU mode outperforms SciPy due to being multi-threaded. Legate Sparse and PETSc achieve nearly perfect weak scaling on CPUs, with PETSc slightly outperforming Legate Sparse, as Legion reserves some CPU resources for runtime work. On GPUs, CuPy, Legate Sparse and PETSc have similar performance on a single GPU, with Legate Sparse achieving 85%

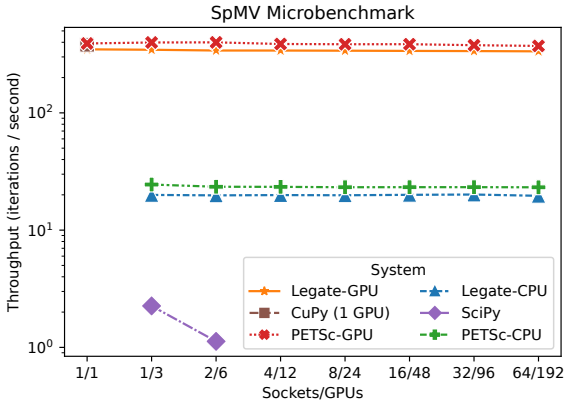


Figure 8: Weak-scaling of an SpMV microbenchmark.

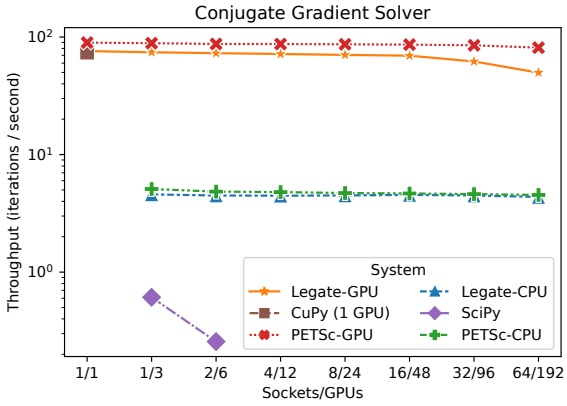


Figure 9: Weak-scaling of a Conjugate Gradient solver.

percent of the performance of PETSc. PETSc and Legate Sparse then weak-scale from a single GPU, where PETSc achieves nearly perfect weak scaling, starting to fall off slightly at 192 GPUs. Legate Sparse also scales well, but experiences some performance drop-off at 32 nodes due to the fast GPU kernels exposing overheads in Legion's all-reduce implementation[1], causing the dot-product communication in the CG solve to affect Legate Sparse's performance at a smaller processor count than PETSc. At 192 GPUs, Legate Sparse achieves 65% percent of PETSc's performance.

Multi-grid Solver. We implement a two-level geometric multi-grid conjugate gradient solver, which uses the injection restriction operator and a weighted Jacobi smoother[2]. Multi-grid methods are known to be relatively challenging to implement correctly and efficiently on distributed machines — our implementation is 300 lines of Python. We do not have a distributed reference implementation, so we compare Legate Sparse's CPU mode to SciPy, Legate Sparse's GPU mode to CuPy, and then weak-scale to larger machines. Figure 10 contains the weak-scaling results for the geometric multi-grid solver. As with prior experiments, Legate Sparse's

[1]The Legion developers are aware of this issue, and plan to address it in the future.
[2]This benchmark was inspired by, but is not directly comparable to HPCG [14].

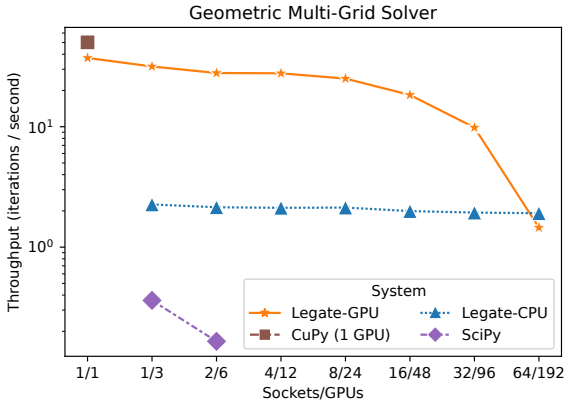


Figure 10: Weak-scaling of a Geometric Multi-Grid solver.

CPU version significantly outperforms SciPy and has good weak-scaling to 64 sockets. On a single GPU, CuPy is 30% faster than Legate Sparse's GPU version. This performance difference is caused by overheads in the Legate library due to its Python implementation. During the V-cycle of the multi-grid method, the application launches several tasks small enough to expose overheads in Legate's task launching and metadata management. Legate Sparse's GPU version starts off weak-scaling well, but has kernels that run fast enough to expose overheads in Legion that could be fixed in the future with tracing [18] and task fusion [32]. Similar performance on a preconditioned CG solver was seen by Bauer et al. [5]. Despite the imperfect weak scaling, Legate Sparse is able to execute the Python multi-grid solver on accelerated hardware much faster and on larger problem sizes than SciPy.

Quantum Simulation. We develop a Legate Sparse quantum simulation of Rydberg atom arrays. The simulation can be used to solve Maximum Independent Set (MIS) problems, as pioneered by the group of Mikhail D. Lukin and QuEra Computing [10]. Like previous implementations [1], we significantly reduce the memory footprint of the simulation by including only states that are allowed by the Rydberg blockade mechanism [19]. Likewise, the interactions between states are rather sparse, as they only permit transition between states in adjacent excitation manifolds and otherwise identical excitation structure. Competing quantum dynamics, namely the energy terms stemming from laser detuning of the system, are inherently sparse due to their diagonal action. Nevertheless, the exponential growth of the quantum state space is only partially stymied by exploiting inherent problem structure, so the simulation remains memory hungry. This application was developed in Python without any expectation that it would be eventually executed in a distributed fashion; the algorithms used in the simulation could be tuned to achieve more scalable performance. We aimed to maximize scale of the Python application as-is, and were able to achieve the exact simulation of the full wave function of up to 50 qubits at time-scales consistent with deep circuits or high entanglement.

The core computational component of this benchmark is an 8th-order Runge-Kutta integration. Similarly to the GMG benchmark, we compare against SciPy and CuPy. Due to the nature of the

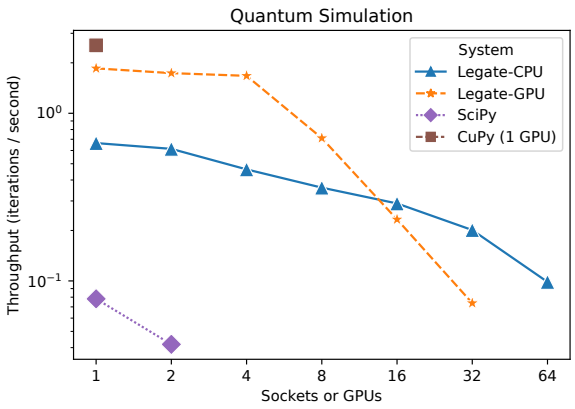


Figure 11: Weak-scaling of a quantum simulation.

application, we were unable to exert fine-grained control over the input size: we could only approximately double the problem size. Therefore, we utilize 4 of the 6 GPUs on each Summit node for this benchmark to directly compare weak-scaling performance between CPUs and GPUs. We stress that Legate Sparse can successfully utilize all 6 GPUs per node for standard runs of the simulation.

The weak-scaling results are found in Figure 11. As with prior experiments, Legate Sparse significantly outperforms the standard implementation of SciPy. On a single GPU, CuPy achieves a 40% speedup over Legate Sparse, for a similar reason as the GMG benchmark — several small tasks launched in the integration expose overheads in Legate. The simulation experiences a loss in weak-scaling efficiency as the number of processors increases. This fall-off is expected due to the communication pattern of the application: the sparse matrices that describe the atomic relationships have a very high bandwidth (the coordinates in a row reference a wide range of columns). Our profiling shows that the algorithms used by the application require every processor to exchange tens to hundreds of megabytes of data with at least half of the other processors in the system, almost an all-to-all communication pattern.

At 1 to 4 GPUs, Legate Sparse's GPU version significantly outperforms the CPU version, due to utilizing the higher-bandwidth NVLink. Once inter-node communication over Infiniband is required after 4 GPUs, the GPU version has similar performance as the CPU version, even dropping below the CPU performance at 16 GPUs. This drop is due to the ratio of communication to effective bandwidth available between each experiment: at 16 GPUs, Legate Sparse's GPU version is utilizing 4 nodes of network hardware, while Legate Sparse's 16-socket CPU version is using 8 nodes to exchange the same amount of data, thus having double the network bandwidth available to communicate through. Finally, the large halo regions present in the application result in imperfect weak scaling of the memory usage per processor, causing Legate Sparse's 64 GPU version to run out of memory.

Dataset	CuPy	Legate Sparse	
	Samples/sec	Samples/sec	Min Req. Resources
ML-10M	197156	69648	1 GPU
ML-25M	28590	55288	2 GPUs
ML-50M	X	33857	6 GPUs
ML-100M	X	6907	12 GPUs

Figure 12: Sparse Matrix Factorization Performance

6.2 Sparse Machine Learning

To evaluate the potential of Legate Sparse as a high-level programming model for sparse machine learning applications, we implement the sparse matrix factorization algorithm with bias [15]. We optimize our model with mini-batch SGD [28], and use a closed-source sparse autograd procedure to generate Python source code for the gradient, which we hand-optimized to remove redundant computations and to exploit sparsity patterns. We compare against CuPy, and measure training throughput in terms of samples per second on the 10 million (10m), 25 million (25m), 50 million (50m) and 100 million (100m) MovieLens datasets [12]. The 50m and 100m datasets were derived from the 20m dataset using randomized fractal expansions [7]. The training loop loads the input dataset into host memory, shuffles the training data before each epoch, and constructs batches of sparse matrices from samples of the training data to update the model parameters. Our implementation falls within 99.7% of SOTA prediction performance for the 10m dataset [26, 27]. The results for these experiments are found in Figure 12.

A key optimization in our implementation is the use of the SDDMM (sampled dense-dense matrix multiplication) operation to avoid materializing dense matrices in expressions of the form $A \odot (B \cdot C)$, where A is sparse and B, C are dense. We generated a high-performance distributed SDDMM implementation using DISTAL, and exposed cuSPARSE's SDDMM kernel for CuPy to use, since CuPy did not support SDDMM out-of-the-box.

We ran each dataset with CuPy on a single GPU, and found that it could only fit the 10m and 25m datasets without running out of memory. In contrast, Legate Sparse can scale to the larger datasets without code modifications by simply adding more GPUs, handling the 50m and 100m datasets with 6 GPUs and 12 GPUs respectively.

CuPy achieves a 2.8x speedup over Legate Sparse on the 10m dataset. Similar to prior experiments, this performance difference arises from overheads in Legate exposed by small tasks launched by the application. Next, CuPy processes the 25m dataset on a single GPU, but achieves nearly half the throughput of Legate Sparse. CuPy runs close to the GPU memory limit on the 25m dataset, and Legate Sparse is unable to do the same due to reserved GPU memory for Legion and external CUDA libraries. We saw that cuSPARSE's SDDMM kernel was inefficient compared to DISTAL's kernel: the SDDMM began to dominate CuPy's execution time on the 25m dataset, while remaining a small percentage of total execution time for Legate Sparse, leading to the speedup on 2 GPUs. Finally, while Legate Sparse can execute the 50m and 100m datasets, it experiences some performance degradation at larger scales. This is due to all-to-all communication patterns inherent in the factorization algorithm, which performs several dense matrix transpose operations in the gradient computation. The effect is more noticeable on the 100m

dataset: 12 GPUs is two nodes of Summit, so many communications go through the lower bandwidth Infiniband instead of NVLink.

7 RELATED WORK

Distributed Sparse Linear Algebra and Tensor Algebra Libraries. Distributed sparse linear algebra has received tremendous attention from the community. The industry-standard sparse linear algebra packages PETSc [3, 20] and Trilinos [34] are long-lasting results of this research. These systems contain a wide variety of sparse linear algebra operations, many of which have been ported to GPUs. However, these systems offer a lower-level API than SciPy (and Legate Sparse), and exist within an explicitly-parallel, message-passing based environment, requiring some expertise in parallel and distributed programming. Additionally, is it not always straightforward to integrate these large systems with external libraries.

The Cyclops Tensor Framework (CTF) [30, 31] is an explicitly-parallel library for distributed dense and sparse tensor algebra that provides a Einstein summation notation-based API, similar to DISTAL. CTF has a flexible API for tensor computations, it lacks composability with a NumPy-like dense array programming library.

Accelerated and Distributed NumPy. Replacing the NumPy and SciPy API is a common approach to accelerating these programs. Several systems exist that accelerate and distribute the NumPy API that we will discuss, but we are not aware of any existing system that successfully distributes the SciPy sparse matrix APIs.

We first discuss systems that target a single node. CuPy accelerates both NumPy and SciPy code on a single GPU by offloading NumPy and SciPy calls to corresponding kernels on the GPU. CuPy can execute in a multi-GPU environment, but requires users to manage data movement and synchronization between the GPUs. Grumpy [25] and Bohrium [16] are systems that lazily evaluate NumPy programs and then generate optimized code for CPUs and a GPU. Weld [23] is a composability-focused system (like Legate Sparse) that provides a drop-in replacement for NumPy and Pandas programs targeting CPUs and a GPU. Similiarly to Grumpy and Bohrium, Weld lazily evaluates the input program and performs cross library optimizations like fusion to increase efficiency.

Dask [29] is a popular library for distributed computing in Python with a high-level array library similar to NumPy. NumS [11] is a distributed NumPy replacement built on top of the Ray [21] task-based runtime system targeting clusters of CPUs. Jax [9] is a drop-in replacement for NumPy with support for vectorization, automatic differentiation and fusion. Jax can target distributed machines, but has restrictions on the kinds of partitioning and distribution it can perform. cuNumeric [5] (formerly known as Legate NumPy) is a library that provides a drop-in, distributed backend for NumPy, targeting both clusters of CPUs and GPUs. cuNumeric shares a similar architecture as Legate Sparse, and our work focuses on maintainable and performant composability with cuNumeric. These systems target NumPy computations, and are not able to execute SciPy operations on sparse matrices, unlike Legate Sparse.

DaCe [38] accelerates annotated Python and NumPy programs onto distributed clusters of CPUs and accelerators by translating them into a high level representation called Stateful Dataflow Multigraphs (SDFGs) [8] and performing a series of optimizations on this representation. While the SDFG representation allows for optimizations such as reordering and fusing computation, DaCe requires both code changes to use and explicit partitioning and message passing between memories. As such, DaCe inhabits a different part of the design space than the part targeted by Legate Sparse.

8 CONCLUSION

We have introduced Legate Sparse, a system that distributes and accelerates unmodified SciPy Sparse programs while composing with cuNumeric. Developing Legate Sparse involves solving composability problems across the software stack; we integrate the libraries at the distributed layer through a constraint-based partitioning scheme and a dynamic runtime system, and use the DISTAL compiler to generate kernel variants for different sparse data structures and heterogeneous processors. Moving forward, the strategy used in Legate Sparse provides a model that others may use to develop high-performance distributed libraries. We believe the ideas in Legate Sparse form a path towards an ecosystem of distributed libraries that compose and share data like the standard Python computing ecosystem.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their valuable comments that helped us improve this manuscript. We thank Olivia Hsu, Scott Kovach, Shiv Sundram, Bobby Yan, AJ Root, Manya Bansal, Praneeth Kolichala, Pat McCormick and Torsten Hoefler for their comments and discussions on early stages of this manuscript. We thank Steven Dalton for his help with developing prototype linear solvers in Legate Sparse. Rohan Yadav was supported by an NSF Graduate Research Fellowship, and part of this work was done while Rohan Yadav was an intern at NVIDIA Research. This work was supported by the Advanced Simulation and Computing (ASC) program of the US Department of Energy's National Nuclear Security Administration (NNSA) via the PSAAP-III Center at Stanford, Grant No. DE-NA0002373, by the Department of Energy's Office of Advanced Scientific Computing Research (ASCR) under contract DE-AC03-76SF00515, and by NSF grant CCF-2216964.

REFERENCES

[1] 2023. Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture. https://github.com/QuEraComputing/Bloqade.jl/

[2] SciPy Authors. 2022. `scipy.sparse` documentation. https://docs.scipy.org/doc/scipy/reference/sparse.html. https://docs.scipy.org/doc/scipy/reference/sparse.html

[3] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2022. PETSc Web page. https://petsc.org/. https://petsc.org/

[4] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (jun 2023), 26 pages. https://doi.org/10.1145/3591236

[5] Michael Bauer and Michael Garland. 2019. Legate NumPy: Accelerated and Distributed Array Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings*

of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 1–11. https://doi.org/10.1109/SC.2012.71

[7] Francois Belletti, Karthik Singaram Lakshmanan, Nicolas Mayoraz, Walid Krichene, Yi fan Chen, John Anderson, Taylor Robie, Tayo Oguntebi, Amit Bleiwess, and Dan Shirron. 2019. *Scaling Up Collaborative Filtering Data Sets through Randomized Fractal Expansions*. Technical Report. https://arxiv.org/pdf/1901.08910.pdf

[8] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs. *CoRR* abs/1902.10345 (2019). arXiv:1902.10345 http://arxiv.org/abs/1902.10345

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[10] S. Ebadi, A. Keesling, M. Cain, T. T. Wang, H. Levine, D. Bluvstein, G. Semeghini, A. Omran, J.-G. Liu, R. Samajdar, X.-Z. Luo, B. Nash, X. Gao, B. Barak, E. Farhi, S. Sachdev, N. Gemelke, L. Zhou, S. Choi, H. Pichler, S.-T. Wang, M. Greiner, V. Vuletić, and M. D. Lukin. 2022. Quantum optimization of maximum independent set using Rydberg atom arrays. *Science* 376, 6598 (2022), 1209–1215. https://doi.org/10.1126/science.abo6587

[11] Melih Elibol, Vinamra Benara, Samyu Yagati, Lianmin Zheng, Alvin Cheung, Michael I. Jordan, and Ion Stoica. 2022. NumS: Scalable Array Programming for the Cloud. https://doi.org/10.48550/ARXIV.2206.14276

[12] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (dec 2015), 19 pages. https://doi.org/10.1145/2827872

[13] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[14] Michael Allen Heroux and Jack. Dongarra. 2013. Toward a new metric for ranking high performance computing systems. (6 2013). https://doi.org/10.2172/1089988

[15] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37. https://doi.org/10.1109/MC.2009.263

[16] Mads Kristensen, Simon Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. 2013. Bohrium: Unmodified NumPy Code on CPU, GPU and Cluster.

[17] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. 2019. A Constraint-Based Approach to Automatic Data Partitioning for Distributed Memory Execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 24 pages. https://doi.org/10.1145/3295500.3356199

[18] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-Based Runtimes. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 441–453. https://doi.org/10.1109/SC.2018.00037

[19] Jongseok Lim, Han-gyeol Lee, and Jaewook Ahn. 2013. Review of cold Rydberg atoms and their applications. *Journal of the Korean Physical Society* 63, 4 (2013), 867–876.

[20] Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, Alp Dener, Matthew Knepley, Scott E. Kruger, Hannah Morgan, Todd Munson, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Junchao Zhang. 2021. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Comput.* 108 (2021), 102831. https://doi.org/10.1016/j.parco.2021.102831

[21] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. https://doi.org/10.48550/ARXIV.1712.05889

[22] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf

[23] Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2017. Weld: Rethinking the Interface Between Data-Intensive Applications. *CoRR* abs/1709.06416 (2017). arXiv:1709.06416 http://arxiv.org/abs/1709.06416

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[25] Mahesh Ravishankar and Vinod Grover. 2019. Automatic acceleration of Numpy applications on GPUs and multicore CPUs. *CoRR* abs/1901.03771 (2019). arXiv:1901.03771 http://arxiv.org/abs/1901.03771

[26] Steffen Rendle. 2012. Factorization Machines with libFM. *ACM Trans. Intell. Syst. Technol.* 3, 3, Article 57 (May 2012), 22 pages.

[27] Steffen Rendle, Li Zhang, and Yehuda Koren. 2019. On the Difficulty of Evaluating Baselines: A Study on Recommender Systems. https://doi.org/10.48550/ARXIV.1905.01395

[28] Herbert E. Robbins. 1951. A Stochastic Approximation Method. *Annals of Mathematical Statistics* 22 (1951), 400–407.

[29] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*. Citeseer.

[30] Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *CoRR* abs/1512.00066 (2015). arXiv:1512.00066 http://arxiv.org/abs/1512.00066

[31] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190. https://doi.org/10.1016/j.jpdc.2014.06.002 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[32] Shiv Sundram, Wonchan Lee, and Alex Aiken. 2022. Task Fusion in Distributed Runtimes. In *2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*. 13–25. https://doi.org/10.1109/PAW-ATM56565.2022.00007

[33] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent Partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/2983990.2984016

[34] The Trilinos Project Team. 2020 (acccessed May 22, 2020). *The Trilinos Project Website*. https://trilinos.github.io

[35] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[36] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3519939.3523437

[37] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 59, 15 pages.

[38] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, Portability, Performance: Data-Centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 95, 13 pages. https://doi.org/10.1145/3458817.3476176