BUILDING COMPOSABLE DISTRIBUTED AND ACCELERATED SOFTWARE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rohan Yadav
March 2026

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Fredrik Kjolstad) Principal Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Kunle Olukotun)

Approved for the Stanford University Committee on Graduate Studies

Abstract

Modern high-performance computing systems are organized as distributed collections of heterogeneous computing elements connected by hierarchical networks. Extracting the peak performance from these machines requires programmers to navigate a deep software stack that spans programming an individual accelerator to orchestrating communication and synchronization across a cluster. Mastering this multi-level software stack is challenging for computing experts, and almost impossible for the domain scientists that need to use these high-performance computing systems to advance their research.

Abstraction of low-level details and composition of independent modules are the standard tools for controlling software complexity. While effective in the world of sequential programming, composable distributed and accelerated software is rarely found. Most independently written distributed software cannot be composed without myriad correctness issues. Once correctness has been established, composition is often at odds with achieving the best performance; significant performance is lost at module boundaries, either through the cross-module coordination itself or the missed optimizations across modules.

This thesis describes the Legate library ecosystem and runtime system. The Legate library ecosystem is a collection of high-level libraries that mimic the standard interfaces of libraries like NumPy and SciPy while scaling to clusters of accelerators while seamlessly composing. The Legate runtime system is a multi-level runtime system that introduces several technologies to support the correct and efficient composition of independently written Legate libraries. The Legate runtime introduces analyses and program representations to support the efficient composition of both

distributed data and distributed computation. These analyses are performed dynamically, and thus can incur overheads that impede scalability and absolute performance; Legate also introduces techniques to control these overheads as they arise in the composition of independent modules.

Lastly, this thesis describes the experience of developing a Legate library using the abstractions exposed by the Legate runtime system. In particular, we describe the development of Legate Sparse, a distributed drop-in replacement for the SciPy Sparse library, enabling the automatic scaling and acceleration of idiomatic sparse applications developed in Python. We show how Legate's abstractions enable both abstraction and composability, and free the developer from the responsibility of reasoning how the library is used within the context of a larger application.

Altogether, this thesis describes the architecture and analyses required to develop composable software that achieves high performance on distributed and accelerated computing systems.

Acknowledgments

I would like to thank...

1. alex fred michael mike
2. kunle (and whoever else is on the reading committee)
3. undergraduate advisors (Sam, Umut)
4. nvidia legate team, especially wonchan, manolis and shriram j
5. Acknowledge all collaborators on all papers I've written, not just the ones in this thesis.
6. fred group and alex group
7. Mentees (Joseph, Rohan, Ahmad)
8. stanford friends
9. cmu/sf friends
10. family

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Software Composition 2

1.1.1 Composition is (Often) Not Performance Preserving 3

1.1.2 Composition of Distributed and Accelerated Software 4

1.2 Dissertation Overview 7

1.3 Collaborators and Publications 8

2 Legate Overview **9**

2.1 Legate Library Ecosystem 9

2.2 Legate Runtime 11

3 Composing Distributed Data **15**

3.1 Legate Front-End 15

3.1.1 Data Model 16

3.1.2 Computational Model 17

3.1.3 Co-Partitioning Distributed Data 17

3.2 Legion 24

3.2.1 Legion Program Representation 25

3.2.2 Composition Through Implicit Parallelism 26

3.2.3 Sharing Physical Data With Composable Mapping 27

3.2.4 Execution Example 29
3.3 Results 31
3.3.1 Conclusion 37

4 Composing Distributed Computation 38
4.1 Legate Middle End 39
4.1.1 Data Model 39
4.1.2 Computational Model 43
4.2 Distributed Task Fusion 43
4.2.1 Dependencies 44
4.2.2 Fusion Algorithm 46
4.2.3 Proof of Correctness 48
4.2.4 Discussion 49
4.3 Task Fusion Optimizations 50
4.3.1 Temporary Store Elimination 50
4.3.2 Memoization of Fusion Analysis 52
4.4 Kernel Fusion 53
4.4.1 MLIR Background 55
4.4.2 Generator Functions 55
4.4.3 Compilation Pipeline 56
4.4.4 Qualitative Benefits 56
4.5 Experimental Results 57
4.5.1 Weak Scaling Experiments 59
4.5.2 Compilation Time 62
4.5.3 Conclusion 63

5 Controlling Overheads (Dependence Analysis) 64
5.1 Background and Motivation 65
5.1.1 Motivating Example for Automatic Tracing 66
5.2 What Are Good Traces? 68
5.3 Trace Identification 70
5.3.1 A Stream of Tokens 71

5.3.2 Finding Traces With High Coverage . . . 71
5.3.3 Recognizing and Replaying Candidate Traces . . . 78
5.3.4 Achieving Responsiveness and Quality . . . 79
5.4 Automatic Tracing Implementation Concerns . . . 81
5.4.1 Distributing the Analysis . . . 81
5.4.2 (The Lack of) Speculation . . . 82
5.4.3 Non-Idempotent Traces . . . 83
5.5 Evaluation . . . 84
5.5.1 Experimental Setup . . . 84
5.5.2 Weak Scaling . . . 84
5.5.3 Strong-Scaling . . . 90
5.5.4 Overheads of Automatic Tracing . . . 92
5.5.5 Trace Search . . . 93
5.6 Conclusion . . . 94

6 Controlling Overheads (Task-Based Execution) 95

7 Implementing a Legate Library 96
7.1 SciPy Sparse . . . 96
7.2 Sparse Data Representation . . . 97
7.2.1 Partitioning Constraint Interaction . . . 98
7.3 Library Kernel Implementation . . . 100
7.3.1 Generating Kernels with DISTAL . . . 101
7.3.2 Porting SciPy and CuPy Implementations . . . 102
7.3.3 Hand-Written Implementations . . . 103
7.3.4 Unimplemented Components . . . 103

8 Related Work 104

9 Conclusion 106

Bibliography 107

List of Tables

List of Figures

1.1 Comparison of high-performance machines from the 2010's and now. Include a figure here comparing a machine like BlueGene/Q to a B200 SuperPOD . . . 2
1.2 Python Library Ecosystem . . . 3
1.3 Example of ScaLAPACK distributed matrix-multiplication interface. . 5

2.1 Figure here of the many legate libraries, the legate runtime system itself (as a single blob), then dispatching to many different kinds of machines. . . . 10
2.2 Figure here of the multiple layers of the Legate runtime system stack. 11
2.3 Figure about a little bit of the data model in Legate. . . . 12
2.4 Figure about the different IR stages. . . . 13
2.5 Figure about how independent components from separate libraries are glued together by the legate runtime. . . . 14

3.1 Legate front-end program representation. . . . 16
3.2 Figure here about adding elementwise matrices. . . . 18
3.3 Figure here about adding elementwise matrices mismatch in a larger application. . . . 20
3.4 Figure here about adding elementwise matrices, but this time using constraints. . . . 21
3.5 Figure that shows constraint resolution. . . . 24
3.6 Legion program representation. . . . 25
3.7 Figure that is the Legion composition example. . . . 26

3.8 Execution of sample Legate program, with control flowing between Legate Sparse and cuPyNumeric. Turn RA (region allocation) into SA (store allocation) 29
3.9 Weak-scaling of a Conjugate Gradient solver. 33
3.10 Weak-scaling of a Geometric Multi-Grid solver. 34
3.11 Weak-scaling of a Runge-Kutta integration-based Quantum Simulation. 35

4.1 Legate's IR exposes a distributed data model and a model for distributed computation on distributed data. 40
4.2 Examples of Tiling partitions. Partitions maps points in the denoted domain to sub-stores. Each color refers to the sub-store associated with a each point in the domain. 42
4.3 Visualization of dependence maps $\mathcal{D}(T_1, T_2)$. 45
4.4 Fusion constraints employed by Legate to identify potential communication between index tasks. 47
4.5 Example of distributed temporaries. 51
4.6 Example of task stream memoization. 52
4.7 Fused MLIR kernel for three way element-wise addition traversing the compilation pipeline. The initial kernel is created by sequentially composing two of the generated task bodies in Figure 4.7b. 54
4.8 Tasks per iteration with and without fusion. Task count is not always whole as iterations may launch different tasks, or fusion occurs across iteration boundaries. Reported task granularities are from unfused single-GPU executions. Window size was selected by Legate. 58
4.9 Microbenchmark weak scaling (higher is better). 59
4.10 Weak scaling of linear solvers (higher is better). 61
4.11 Weak scaling of full applications (higher is better). 62
4.12 Warmup times on 8 GPUs. 62

5.1 Example cuPyNumeric program and the stream of tasks issued to the Legate (and then Legion) runtime. An intuitive trace around the main loop does not correspond to a repeated program fragment. 67

5.2 Example of a task stream and fixed trace set T with an invalid matching function f, and two matching functions with different $\mathsf{coverage}(T, f)$. 70
5.3 Visualization of Legion's automatic tracing analysis. 73
5.4 Execution of Algorithm 2 on "aabcbcbaa". The candidates for each suffix pair is shown between the pair. 77
5.5 Visualization of Legion's buffer sampling strategy on a buffer of size 8. After processing the i'th task, Legion mines the buffer slice labeled i. 80
5.6 Weak scaling of S3D on Perlmutter. Legion with automatic tracing ("auto") performs competitively with hand-annotated traces. 86
5.7 HTR-Solver (Perlmutter) 87
5.8 Weak scaling of HTR on Perlmutter. Legion with automatic tracing ("auto") performs competitively with hand-annotated traces. 87
5.9 Weak scaling of CFD on Eos, where Legion with automatic tracing ("auto") outperforms the untraced version. 88
5.10 Weak scaling of TorchSWE on Eos, where Legion with automatic tracing ("auto") outperforms the untraced version. 89
5.11 Strong scaling of FlexFlow on Eos. 91
5.12 Warmup iterations before Legion with automatic tracing reaches a replaying steady state. 93
5.13 Visualization of Legion finding traces in S3D. 94

7.1 Legate Sparse's CSR sparse matrix encoding. 98
7.2 Python implementation of a row-based distributed CSR SpMV (adapted from DISTAL generated code). 99
7.3 Distributed, multi-threaded CSR SpMV in DISTAL. 102
7.4 DISTAL-generated C++ task for row-based, multi-threaded CSR SpMV, with minor modifications. 102

Chapter 1

Introduction

Modern high-performance computing systems have continued to deliver increasing performance year-over-year, offering more compute as well as additional memory capacity and bandwidth. However, these gains in performance have come with rapid increases in complexity, where the development of high-performance software on these systems is becoming increasingly challenging. This complexity arises from increasing heterogeneity and hierarchy in these systems. Heterogeneity in modern systems arises from the collection of processors available on individual compute nodes, which contain multiple specialized accelerators like GPUs in addition to a general-purpose multi-core CPU. These heterogeneous compute nodes are organized into distributed machines with hierarchical networks for communication, where links of different latencies and bandwidths connect processors within a node, between nodes in a rack, and across racks. Achieving high performance on these distributed and accelerated systems is a full-stack problem, requiring fast kernels that run on the individual computing elements, efficient orchestration software that coordinates these kernels while moving data across the machine, and algorithms that expose sufficient parallelism and minimize communication.

While high-performance machines are increasing in complexity, enabling developers to program them productively is a critical challenge with immediate impact on scientific progress. High-performance computing is critical to workloads across computational science (including physics, chemistry, biology and climate science),

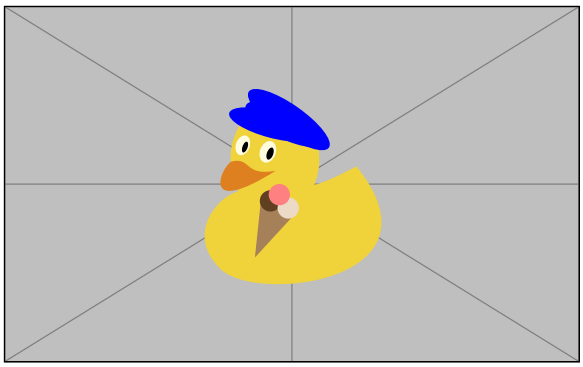


Figure 1.1: Comparison of high-performance machines from the 2010's and now. Include a figure here comparing a machine like BlueGene/Q to a B200 SuperPOD

machine learning and data analytics. Leveraging the performance improvements modern machines offer can enable scientists to run larger or higher resolution simulations, train larger machine learning models, or analyze more data collected by physical sensing devices. However, the increasing complexity of programming modern machines directly blocks the forwards progression of science. Most experts in scientific domains are not additionally experts in high-performance parallel computing, meaning that leveraging a modern high-performance system is either extremely challenging or even impossible.

In this thesis, I describe how to control the complexity of programming modern high-performance machines through the development of *composable* software that enables end users to develop programs built from separate high-level, domain-specific libraries that execute in conjunction with the performance of a low-level, highly-tuned implementation. The artifacts of this thesis is the Legate ecosystem of libraries that provide friendly interfaces to scientists, like NumPy [26] and SciPy [52], and the Legate runtime system that enables the efficient execution of Legate programs.

1.1 Software Composition

The tried and true approach to managing complexity in computer science is through abstraction. Computer scientists hide complex reasoning and implementation details behind simple interfaces that expose functionality to other software components. All

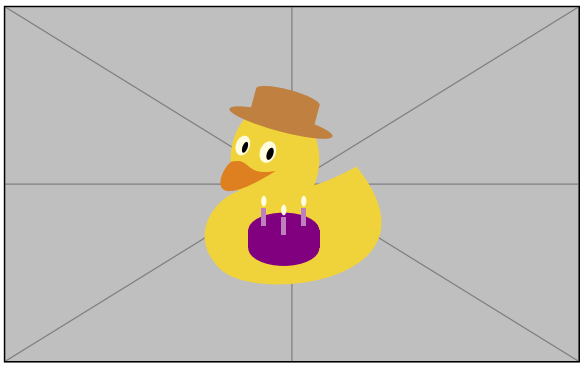


Figure 1.2: Python Library Ecosystem

of modern software engineering is based upon these principles of abstraction: correct modules are composed to produce new correct modules. Composition enables large software ecosystems to thrive, such as the large data processing and scientific computing library ecosystems in Python (Figure 1.2). Domain scientists develop sophisticated applications by mixing-and-match components from these library ecosystems, leaving the implementation details to computer science experts.

1.1.1 Composition is (Often) Not Performance Preserving

While these large ecosystems exist and enable the development of sophisticated software in the sequential programming world, the composition of performant, independent modules often does not yield the same performance as a bespoke implementation tuned for a specific purpose. These performance differences can arise from a variety of sources. In some cases, these differences can arise from inefficient use of hardware features like a tiered memory hierarchy when composing programs from high-level interfaces such as found in NumPy [7, 39]. The difference in performance can even be asymptotic [31], as additional domain knowledge about the end-to-end application may enable optimizations beyond what is possible in the purview of an individually optimized module. Performance degradation at module boundaries may be acceptable on small machines, and developers often pay this price to develop prototype programs more quickly.

1.1.2 Composition of Distributed and Accelerated Software

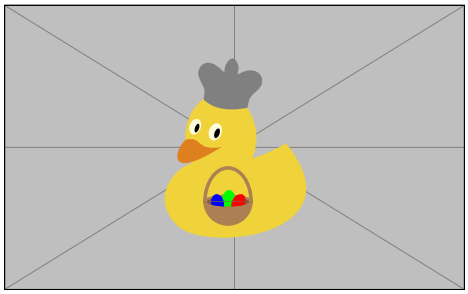
While composition preserves correctness and sometimes sacrifices performance in the sequential programming world, neither property is maintained by the current status quo of distributed and accelerated software. Concretely, most independently written modules of distributed and accelerated software are often not even correct under composition, and if correctness is achieved, the end-to-end performance is often far from what is achieved by a hand-tuned implementation. These effects make it difficult for large library ecosystems that mirror those found in the sequential programming world to exist for distributed and accelerated machines, which in turn, makes it difficult to develop new applications for these systems.

Correctness

Standard distributed and accelerated software built through bulk-synchronous, message-passing interfaces like MPI place the burden of managing correctness during composition onto the programmer performing the composition. For example, consider the interface to a distributed matrix-matrix multiplication operation within the distributed linear algebra library ScaLAPACK [16], shown in Figure 1.3. The interface accepts a variety of information about the operand distributed matrices, including the backing pointers for the locally held data by each process involved in the operation, as well as descriptors that inform ScaLAPACK about how each sub-component relates to the globally distributed sparse matrix. While the interface is sensible, it exposes several unsavory aspects around correctness when it is when composed into a larger application. First, the descriptors understood by ScaLAPACK expect a certain subset of data distributions of the underlying matrices (1-D or 2-D block-cyclic distributions); users must be able to either correctly describe their distributed data coming from another source in this manner, or are responsible for coordinating to transform their data into the distributed orientation required by ScaLAPACK. Doing this transformation correctly is a burden on the user, who then also must transform the distributed data back into whatever distributed orientation is required by downstream operations. Second, by speaking in raw pointers, the user is subject to race

```
1  void pdgemm(
2    // Transpose information.
3    char *transa, char *transb,
4    // Global matrix sizes.
5    int *m, int *n, int *k,
6    double *alpha,
7    // Locally-stored matrix A information.
8    double *a, int *ia, int *ja,
9    // Distribution descriptor for A.
10   int *desca,
11   // Locally-stored matrix B information.
12   double *b, int *ib, int *jb,
13   // Distribution descriptor for B.
14   int *descb,
15   double *beta,
16   // Locally-stored matrix C information.
17   double *c, int *ic, int *jc,
18   // Distribution descriptor for C.
19   int *descc
20 );
```

(a) ScaLAPACK `PDGEMM` API.



(b) Sample Data Distribution

Figure 1.3: Example of ScaLAPACK distributed matrix-multiplication interface.

conditions due to potential concurrent interactions in the larger application: other components of the application may be concurrently reading from or writing to the operand matrices, or other communication that will populate the matrices has not yet completed. These concurrent interactions can increase in complexity when applications utilize asynchronous accelerators like GPUs, which operate on detached streams of control from the CPU and networking operations.

Performance

Moving beyond correctness in the composition of independent modules in distributed and accelerated programs, achieving high performance from the composition of individually optimized components is challenging. Unlike when running on small machines or developing prototypes, achieving high performance on distributed and accelerated machines is critical. These machines are expensive and a limited resource; they must be fully utilized to justify their use and to avoid wasting resources.

Many critical optimizations in a large distributed and accelerated application span decisions made across different logical components of the program, and discordant decisions in these components can result in significant performance penalties. Some of these decisions are optimizations of their own, while others are choices that interplay

between correctness and performance. I now discuss some of these performance opportunities lost at module boundaries in distributed and accelerated software, grouped by opportunities around having a coherent parallelism strategy and opportunities when efficiently coordinating logically separate application components.

When independently written modules are optimized, the developers make decisions about a parallelization strategy. When these strategies are chosen without knowledge of the larger application context that a module is used within, a locally optimal strategy can become globally sub-optimal. Consider the parallelization strategy described earlier used by ScaLAPACK, which accepted 2-D distributions of dense matrices. If this module is composed with a separate module that independently decides it requires 1-D distributions of matrices, a significant cost is paid shuffling data around the system between these distributions. This cost is completely unnecessary if later computations could be rewritten or reformulated to accept the 2-D distribution as is. Alternatively, consider a module that prefers using CPUs instead of GPUs for a particular computation — in the context of a larger application that keeps data resident on the GPUs, a locally slower GPU execution would avoid the costs of moving data before and after a CPU implementation of the module. Finally, a bespoke implementation may fuse the compute kernels of logically independent application components to better utilize processor memory hierarchies, but a modular application with separate implementations forgoes this optimization.

The composition of these independent modules then can lead to inefficient handling of the application at module boundaries, which often arises from the lack of information exposed by and the flexibility of individual modules. For example, consider again the ScaLAPACK matrix-multiplication implementation: if this implementation launched work onto asynchronous accelerators, the user may be responsible to ensure that other modules only read the results when the accelerators complete, and often insert overzealous synchronization to ensure application correctness. Finally, well-tuned distributed and accelerated applications heavily overlap compute and communication to find available work and keep machine resources busy at all times. In existing distributed software, finding and exploiting these opportunities for overlap between and across independent modules is the responsibility of the developer, and difficult to

realize without breaking down the abstractions provided by the modules themselves.

The goal of this thesis is to support the correct and efficient composition of independent software on distributed and accelerated machines. We focus on the composition of high-level, collection-based libraries like NumPy and SciPy which form the basis of the very popular scientific computing ecosystem in Python. Scientific and data analysis workloads developed in these systems have plentiful dynamic behavior, including dynamic control flow and data-dependent behavior, which render them incompatible to the advances in machine learning infrastructure [20, 40] that support domain scientists at scale on distributed and accelerated machines.

1.2 Dissertation Overview

To demonstrate the correct and efficient composition of independent software on distributed and heterogeneous machines, we have developed the Legate ecosystem of composable distributed libraries, and the Legate runtime system, which contains the technical pieces that enable the composition of Legate libraries. Chapter 2 begins by providing an overview of both the Legate library ecosystem and Legate runtime system, and enumerates the many components of the deep Legate technical stack. The remainder of the thesis walks through this stack of technology and discusses the key ideas that enable efficient and correct composition of Legate libraries. Chapter 3 discusses analysis and data representations within the Legate runtime system that enable the efficient sharing of distributed data across independent modules. Chapter 4 discusses program analysis and program representation that fuses computations across independent Legate libraries. Chapter 5 and Chapter 6 focus on different components of controlling the overheads that various layers of the Legate runtime system impose to enable efficient composition. Chapter 7 describes the processing of building a Legate library (Legate Sparse [59]) using the abstractions provided by the Legate runtime system. Finally, we describe related work in Chapter 8 and draw conclusions in Chapter 9.

1.3 Collaborators and Publications

This thesis discusses ideas that were the focus of several publications [57–60]. Work in this thesis was done in collaboration with Michael Bauer, Wonchan Lee, Manolis Papadakis, Shiv Sundram, Melih Elibol, Taylor Lee-Patti, Sean Treichler, Joseph Guman, David Broman, Michael Garland, Fredrik Kjolstad and Alexander Aiken. Many of the experimental results were achieved by building on the production-grade Legate implementation developed by the Legion, Legate and Realm teams at NVIDIA.

Chapter 2

Legate Overview

In this chapter, I give an overview of the Legate library ecosystem and Legate runtime system. I discuss the many different components involved in the Legate technical stack, as well as describing the different program representations used by each layer. These representations and the transformations between them are discussed in detail in later chapters.

2.1 Legate Library Ecosystem

The Legate library ecosystem is a collection of high-level distributed libraries that act as drop-in, distributed replacements to popular Python libraries in the scientific computing, data analytics and machine learning ecosystems. An explicit goal for libraries within the Legate ecosystem is composition; Legate libraries should correctly and efficiently compose just like their sequential counterparts. An overview diagram of the Legate ecosystem is shown in Figure 2.1. Legate libraries are developed on top of the Legate runtime system, which then enables their efficient composition as well as scaling from machines containing a single GPU to multi-node multi-GPU machines.

The Legate ecosystem has expanded far beyond the original prototype research libraries, and now has a team of engineers at NVIDIA that maintains existing libraries, develops new libraries, and engages with customers to help them utilize Legate. While I personally contributed to the development of a subset of these Legate libraries,

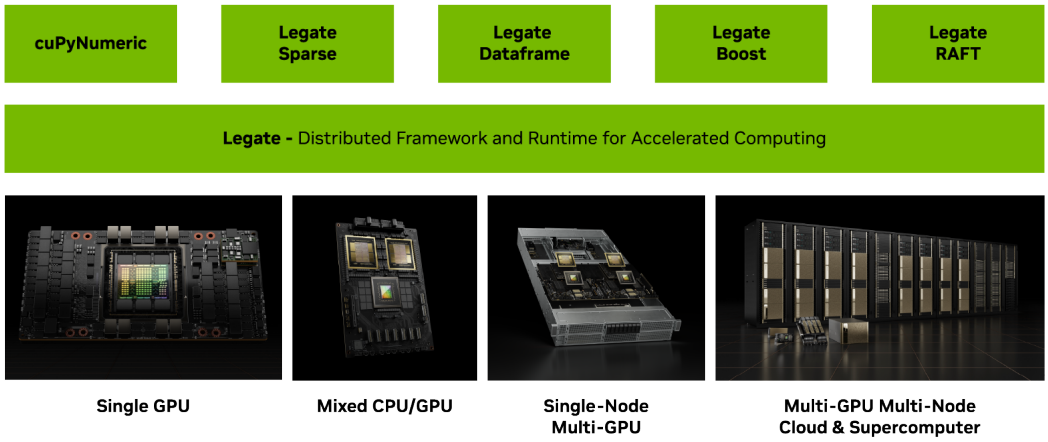


Figure 2.1: Figure here of the many legate libraries, the legate runtime system itself (as a single blob), then dispatching to many different kinds of machines.

others were developed by independent groups aiming to scale their own workloads. The first Legate library was cuPyNumeric [11] (originally Legate NumPy), developed by Mike Bauer, and is a drop-in replacement for NumPy. I developed Legate Sparse [59], a distributed implementation of SciPy Sparse, that acts a companion to the distributed dense array programming provided by cuPyNumeric. All of the Legate application results presented later in this thesis contain programs built from cuPyNumeric and Legate Sparse.

Beyond dense and sparse array programming, NVIDIA engineers have developed Legate libraries that focus on the data analytics and machine learning domains. Legate Dataframe [43] provides a Pandas-like interface to performing distributed dataframe and relational operations. Legate RAFT [44] provides computational primitives and algorithms for machine learning and information retrieval. Legate Boost [42] delivers a distributed implementation of gradient boosting for training machine learning classifiers. Legate IO [37] provides distributed file I/O support from scientific file formats like HDF5. Finally, Legate STL [22] is a lower-level C++ library that uses Legate to provide distributed implementations of C++ iterator-style collection operations.

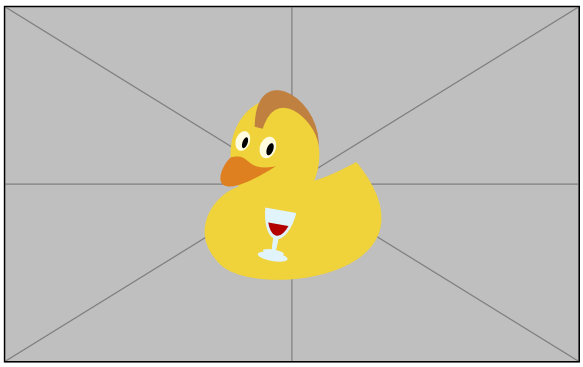


Figure 2.2: Figure here of the multiple layers of the Legate runtime system stack.

2.2 Legate Runtime

Legate libraries are developed on top of abstractions provided by the Legate runtime system, which enables Legate libraries to compose efficiently and scale to large machines. The multiple layers of the Legate runtime system are shown in Figure 2.2; these layers consist of the Legate front-end, Legion [15], and Realm [48]. The contributions of this thesis will span techniques across each of these runtime system layers.

At a high-level, the Legate runtime system exposes a *task-based* programming model with a strong *data model*, visualized in Figure 2.3. Legate's data model exposes distributed data as *stores*, which are multi-dimensional arrays. Libraries like Legate Sparse map distributed data structures onto collections of stores. Computations are defined through *tasks*, which are user-defined functions that operate on (subsets of) stores. Tasks may perform arbitrary computation on their argument stores, such as launch kernels onto GPUs. The different layers of the Legate runtime system perform analysis and lower these abstractions onto a physical execution of tasks and data movement across the target machine. Legate's task-based programming model undergoes several lowering steps from the user program before executing on the machine, as shown in Figure 2.4.

The Legate front-end exposes a programming model that is *implicitly-parallel*, *scale-free*, and *implicitly partitioned*. Concretely, this means that programmers express computations in a sequential manner (logically, a single thread of control issues tasks into Legate), and Legate is responsible for extracting parallelism from this

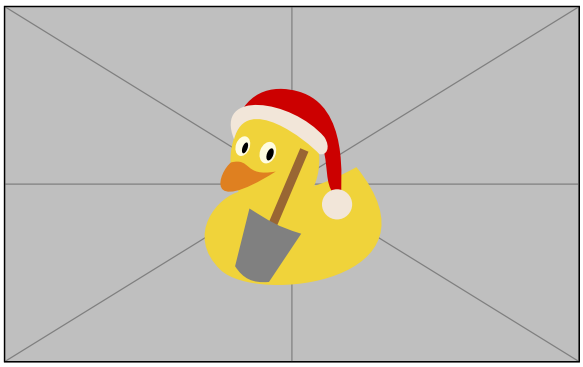


Figure 2.3: Figure about a little bit of the data model in Legate.

stream of tasks. Legate automatically partitions stores across the machine based on constraints that tasks declare on the stores they access (implicitly partitioned), and task descriptions do not specify concrete parallelization strategies (such as how many GPUs a task should be partitioned across). The flexibility exposed by the Legate front-end representation allows the Legate runtime system to make key decisions around data partitioning and parallelization that are coherent across multiple distributed libraries, which is critical for performance. These analyses are discussed further in Chapter 3.

The Legate front-end program representation is then lowered into a second intermediate representation (referred to as the Legate middle-end) that is implicitly-parallel, but is *explicitly partitioned* with *symbolic* scale. In this representation, tasks are associated with concrete partitions of stores to be accessed, and parallelization strategies have been assigned by Legate; the parallelization and partitioning strategies are represented in a compact manner, discussed more in Chapter 4. This representation enables the composition of computation across different distributed libraries, in particular the fusion of operations across library boundaries.

The Legate middle-end program representation is then dynamically translated into a Legion [15] program. Legion is an implicitly-parallel task-based representation with explicit partitioning and explicit scale. Legion performs dynamic analysis to extract parallelism from a sequential input stream of tasks from Legate. Legion determines what tasks are safe to execute in parallel and inserts dependencies between tasks based on the stores each task accesses. In addition to inserting dependencies

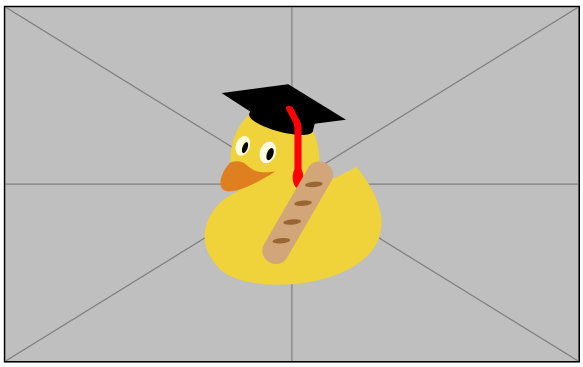


Figure 2.4: Figure about the different IR stages.

between tasks, Legion also inserts the data movement operations required to maintain coherence between stores as they are accessed by tasks running on different processors on different nodes. The exact mechanisms used by Legion to perform this dependence analysis is out of scope of this thesis and explored in prior work [12, 15]. However, this thesis contributes a component to Legion's analysis that enables these analysis to be performed efficiently in the context of a Legate program manipulating multiple independent libraries (Chapter 5).

Finally, as the result of dependence analysis, Legion generates task-based programs that target the Realm [48] runtime system. Realm is an explicitly-parallel, explicitly-partitioned task-based programming model with explicit scale, providing higher-level systems with a high-degree of control over how programs execute across a distributed and accelerated machine. Realm is the final layer of this software stack that is responsible for actually executing computations on target processors, performing the desired data movement, and ensuring that dependencies specified by higher layers of the system are respected. As with Legion, much of Realm is prior work [48] and developed by others. This thesis contributes an analysis and compilation strategy for Realm programs that dramatically lowers the overheads of executing Realm programs, which improves their absolute efficiency and strong scaling, bubbling up into future improvements for the whole Legate stack (Chapter 6).

rohany: There should be a paragraph here about how the the combination of these intermediate representations allow independent libraries to launch implicitly parallel tasks into the system, while the system weaves together all of these tasks while

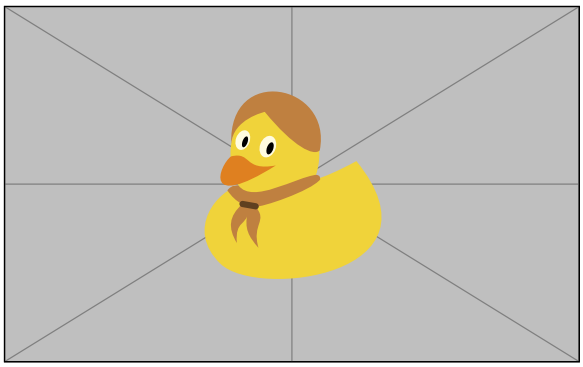


Figure 2.5: Figure about how independent components from separate libraries are glued together by the legate runtime.

performing separate analyses, finally turning the result into a completed computation graph that is scheduled to run on the target machine by Realm.

The remainder of this thesis discusses these individual layers of the Legate software stack in depth, but in the context of how the choices of representation and analysis enable composition of different performance critical components of high-performance applications. Chapter 3 discusses how representation and analysis in the Legate front-end and Legion enable independent modules to share distributed data efficiently. Chapter 4 discusses how the representation of the Legate middle-end enables fusion optimizations that efficiently compose distributed computations across modules. Finally, Chapter 5 and Chapter 6 discuss optimizations that control the overheads imposed by the dynamic analysis at various layers of this analysis stack; controlling these overheads enables strong and weak scaling comparable to hand-optimized implementations.

Chapter 3

Composing Distributed Data

This chapter discusses program representations and dynamic analysis within both the Legate front-end and Legion runtime system that enable distributed data structures to be effectively shared across multiple independently-written distributed libraries. We first focus on the Legate front-end, and introduce Legate's developer-facing programming model, which leverages a *constraint-based* representation and analysis to co-partition distributed data across multiple libraries. We then move into components of the Legion runtime system that support the the correct usage of distributed data across library boundaries, as well as the sharing of physical data allocations.

3.1 Legate Front-End

As discussed in Chapter 2, the Legate front-end programming model is implicitly-parallel, scale-free, and implicitly-partitioned. The Legate front-end exposes a *data model* as well as a *computational model* to describe distributed programs; we describe each in turn. A formal grammar containing the full front-end language (describing both components of the model) is shown in Figure 3.1, and is described in more detail in Sections 3.1.1 and 3.1.2.

Syntax

$$
\begin{array}{rrcl}
\textit{Unique ID} & id & & \\
\textit{Shape} & s & ::= & (\mathbb{Z}, \ldots) \\
 & & & \\
\textit{Store} & S & ::= & \mathsf{Store}(id, s) \\
\textit{Partition Constraint} & C & ::= & \mathsf{equal}(S, S) \mid \mathsf{broadcast}(S, \mathbb{Z}\ \mathsf{list}) \mid \mathsf{image}(S, S) \mid \ldots \\
 & & & \\
\textit{Privilege} & Pr & ::= & \mathsf{Read} \mid \mathsf{Write} \mid \mathsf{Reduce} \mid \mathsf{Read\text{-}Write} \\
\textit{Task} & T & ::= & \mathsf{Task}((S, Pr)\ \mathsf{list}, C\ \mathsf{list}) \\
\textit{Program} & P & ::= & T\ \mathsf{stream}
\end{array}
$$

Figure 3.1: Legate front-end program representation.

3.1.1 Data Model

Distributed data is represented in Legate through *stores*, which are multi-dimensional arrays. As shown in Figure 3.1, each store has an (internal) unique identifier and a multi-dimensional shape. Libraries like cuPyNumeric directly represent NumPy's arrays as stores, while libraries like Legate Sparse use collections of stores to represent distributed sparse matrices. Notably, concrete partitions of distributed data are absent from the data model; the language does not directly support the construction of the subsets of a store that might be needed on each processor for a distributed computation. Instead, the language provides *partitioning constraints* that instead describe a set of potential possible concrete partitioning choices. This construct is discussed in signficant detail in Section 3.1.3, and is a critical design choice to enable composing distributed data. The production implementation of the Legate front-end additionally contains operations that perform logical shape transformations on stores, such as shifting indices, or changing the dimensionality of the store. These transformations are necessary for a fully functional implementation, but we elide discussion of them in this thesis.

3.1.2 Computational Model

Distributed computation is represented in Legate through *tasks*, which are user-defined functions. Tasks explictly declare the stores that they will access, and additionally declare how they will access a store through *privilege annotations*. In addition to privilege annotations on argument stores, tasks also describe a list of partitioning constraints over the argument stores. These partitioning constraints describe assumptions inside the implementation of the task that require specific partitioning relationships between the task's store arguments.

The representation of a complete program in the Legate front-end is a stream of tasks issued to the system by the application (possibly from independent libraries). The Legate front-end is implicitly-parallel, as the application issues a sequential stream of tasks, and Legate is responsible for extracting parallelism from this stream while satisfying the semantics of the sequential program. The representation is also implicitly-partitioned, as only partitioning constraints are provided over stores (discussed in Section 3.1.3), and Legate is responsible for choosing the concrete manner in which stores are partitioned across the target distributed machine for different operations. Finally, the representation is scale-free, as no constructs in the language are tied to the size of the physical machine; Legate can freely choose over how many processors to distribute a computation over, and the size of this representation does not scale with the size of the target machine.

3.1.3 Co-Partitioning Distributed Data

The previous discussions of the Legate front-end's models for distributed data and computation emphasized the freedom for the Legate runtime to make decisions about how data and computation are partitioned and distributed across the machine. To understand the importance of this freedom, we first describe the current state of developing distributed software.

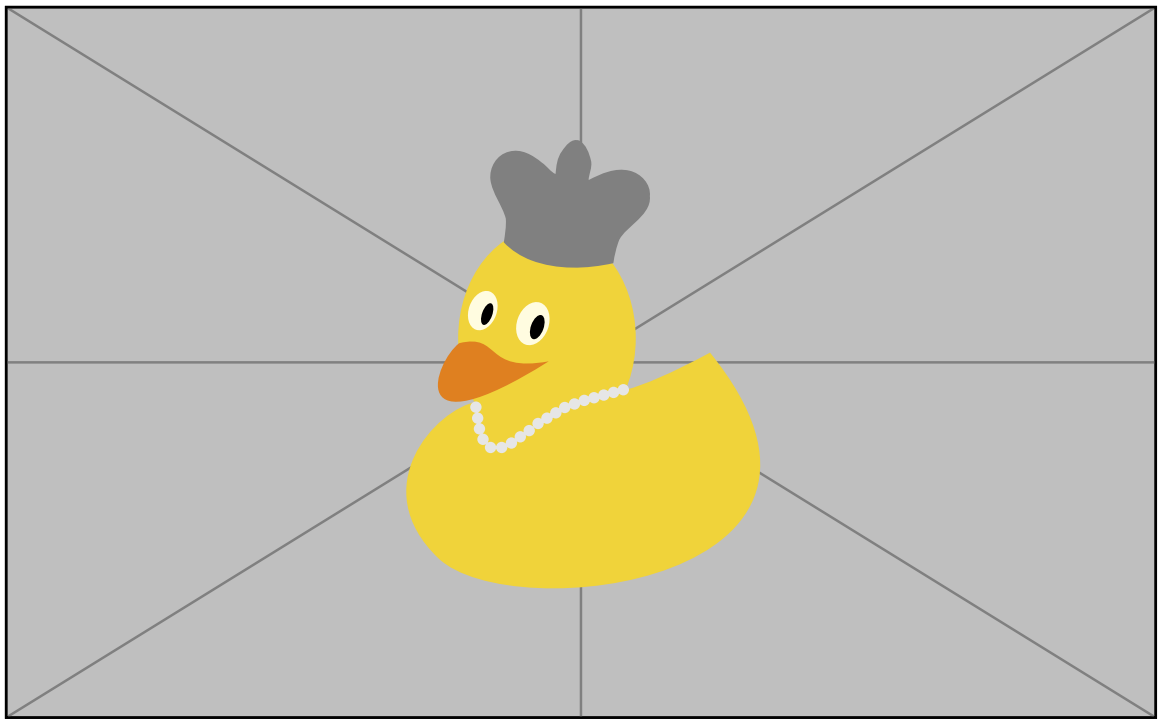


Figure 3.2: Figure here about adding elementwise matrices.

The Problems With Explicit Partitioning

Nearly all distributed libraries that we are aware of are *explicitly-partitioned*, meaning that the choice of how distributed data is partitioned across the target machine is uniquely specified in the target program. Explicit partitioning is separate from the underlying class of programming model, such as bulk-synchronous (MPI) or task-based (Legion); any explicitly-partitioned model is susceptible to challenges that inhibit efficient composition. Because the choice of how distributed data is partitioned is explicitly embedded in the source program, these decisions are made at the time at which a distributed library is developed, rather than *when the library is used*. These decisions can be locally optimal within the context of an individual distributed library, but suboptimal in the larger context of the application that the library is used within.

To make this composition problem concrete, consider the case of build a distributed NumPy library (like cuPyNumeric), and implementing the element-wise addition operation on two distributed arrays, as shown in Figure 3.2. An explicitly-partitioned system must commit to an arbitrary choice of data partitioning strategy For example, an MPI library would maintain each array in a particular partition across each rank, or a Legion library would maintain an internal `LogicalPartition` object of a particular structure to use for each array. Suppose the arbitrary partitioning choice was to maintain a 2-D tiling of each array. Problems arise when this hypothetical library is used in a larger application that first invokes a computation from different library on the same distributed data before invoking the elementwise addition operation. If the previous library independently chose a different partitioning strategy for the same distributed data, then either the user or the underlying runtime system is responsible to shuffle data between the two orientations of the distributed data. However, element-wise addition does not require a specific partitioning, and any partitioning strategy that aligns the partitions of the input and output arrays is sufficient for correctness. In this case, the resulting shuffles of distributed data cause a large amount of unnecessary data movement, and a significant cost to performance.

Implicit Partitioning With Constraints

Instead of forcing libraries to commit to partitioning strategies at library definition time, Legate's front-end representation uses a *constraint-based* parallelism strategy to enable the *late-binding* of partitioning decisions to the context within which a library is used. As discussed in Section 3.1.1, task definitions in the Legate front-end representation declare partitioning constraints over their input stores, instead of concrete partitions. The Legate runtime system then solves this system of constraints at program execution time to discover store partitions that align with how the stores are already partitioned within the context of the larger application. This enables distributed data to be reused in place whenever possible, and allows for reasoning about implementations of tasks within libraries and individual functions to be independent of the surrounding context. A visualization of the constraint-based scheme is shown in Figure 3.4.

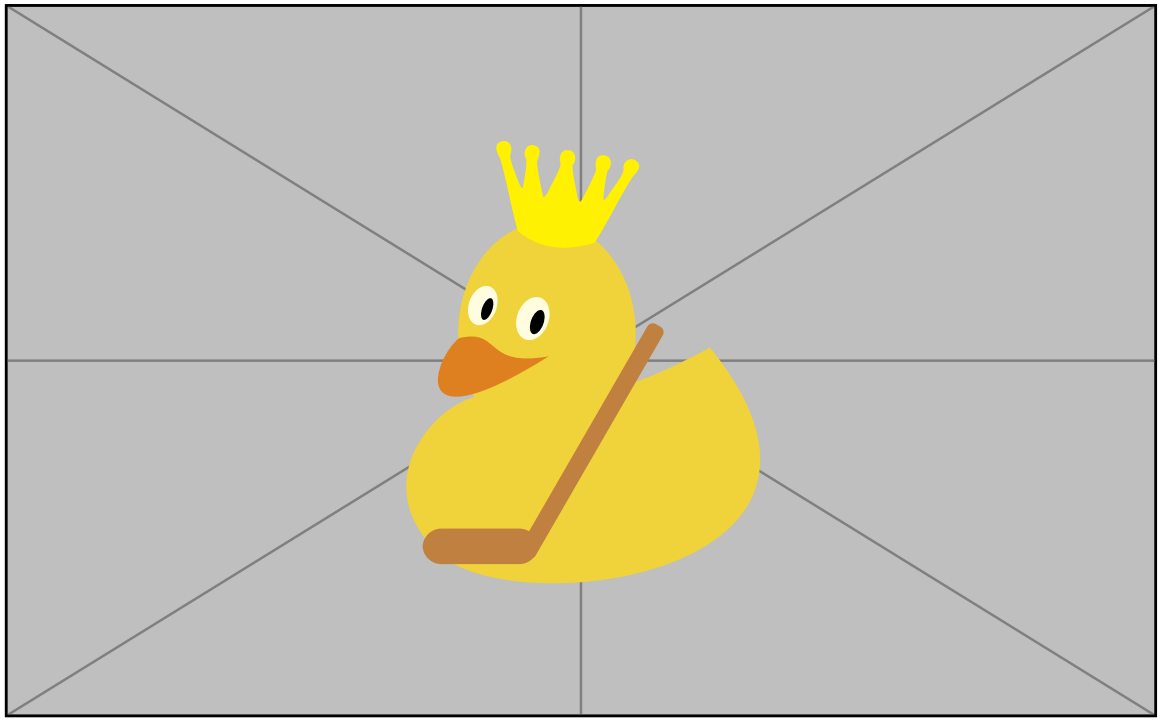


Figure 3.3: Figure here about adding elementwise matrices mismatch in a larger application.

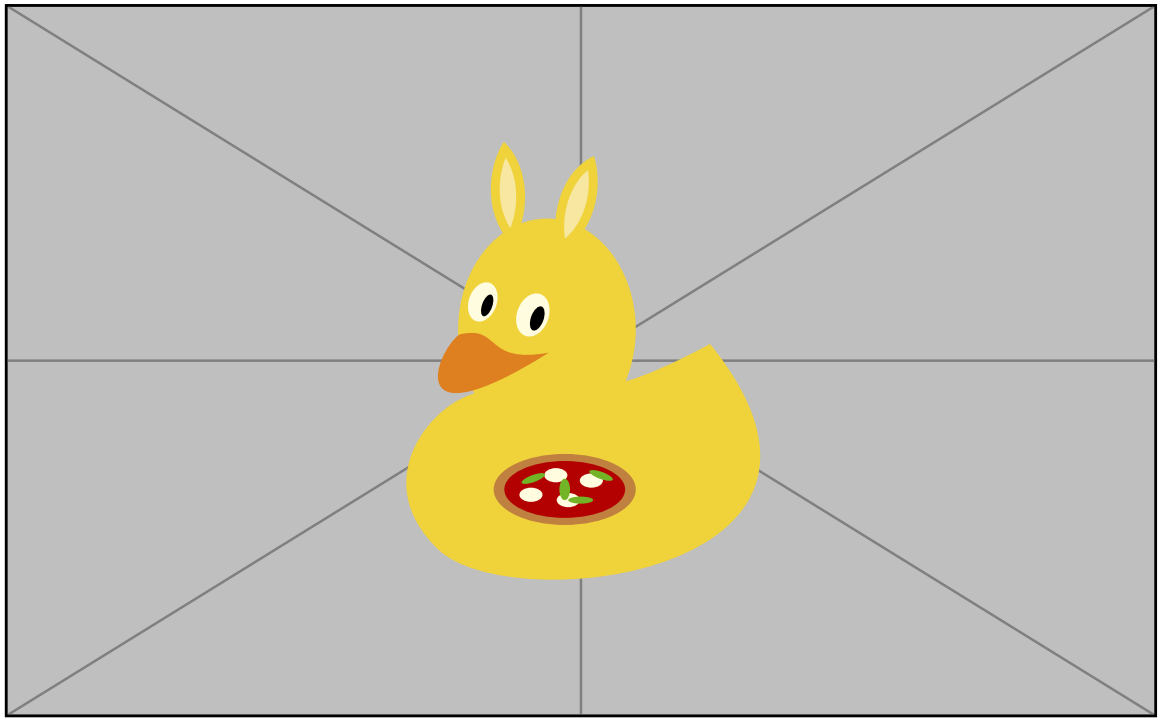


Figure 3.4: Figure here about adding elementwise matrices, but this time using constraints.

Constraints. The syntax presented in Figure 3.1 desribes three kinds of partitioning constraints, which are sufficient to describe non-trivial partitioning strategies. As discussed prior, the full partitioning implementation with the production implementation of Legate contains more partitioning constraints along with shape manipulation operations, but these are not necessary to discuss the technical core of the work. The $\mathsf{equal}(S_1, S_2)$ constraint captures partition equality between two stores, requiring the same partition choice to be made across both stores S_1 and S_2. The $\mathsf{broadcast}(S, I)$ constraint requires the dimensions in I (an integer list) to not be partitioned in a valid partitioning of S, allowing for partial replication of data. The final constraint is $\mathsf{image}(S, D)$, which relates the partitions of two stores through dynamically valued indirections. Intuitively, an image relationship between two stores projects a partition of the source store through the indirection values within the source, which contain pointers into the destination store. More formally, let a partition P be a mapping from a set of *colors* to a subsets of indices within a store, and let the elements of S be sets of indices in D. Then the $\mathsf{image}(S, D)$ constraint requires the partitions P_S and P_D of S and D, respectively, to satisfy the following relationship: $\forall c \in P_S, \forall i \in P_S[c], S[i] \subseteq P_D[c]$. The image partitioning operation itself was introduced in prior work [50], and is useful to support a wide variety of data-dependent partitioning patterns that occur in sparse matrix processing [56, 59, 61] and simulation [5, 23, 50].

As discussed previously, tasks declare a set of partitioning constraints upon their argument stores, and allow the Legate runtime system to solve the system of constraints into a concrete set of partitions. The more flexibility the application developer provides, i.e. declaring a partitioning constraint set with the largest possible number of solutions, the more options the Legate runtime system has to satisfy the constraints while minimizing the data movement at task boundaries. In-depth examples of using partitioning constraints are described in Chapter 7, where I describe the implementation of Legate Sparse using the Legate front-end.

Solving. The Legate runtime system solves systems of partitioning constraints through a dynamic program analysis, as tasks are issued by the application into

the runtime system. As the analysis is dynamic (and on the critical path of task launching), we favor speed of resolution over global optimality in the choice of solution. The Legate runtime system maintains as metadata with each store a *key partition*, which refers to the last partition that a store was written to over. The key partition represents an approximation of the current orientation of a store as it is distributed across the target machine; it is only an approximation because read-only operations over different partitions may have left additional copies of the store in other orientations across the target machine. The constraint resolution strategy uses these key partitions as anchors to avoid unnecessary movement of distributed data.

The constraint solver analyzes the partitioning constraints (which form a graph) to identify the *independent* and *dependent* partitions. Independent partitions can be arbitrarily chosen by the solver, while dependent partitions are computed from independent partitions. For example, in the set of constraints $\{\mathsf{image}(S_1, S_2), \mathsf{equal}(S_2, S_3)\}$, the partition of S_1 an independent constraint while the partitions of S_2 and S_3 are dependent. For each store S that can be independently partitioned, the solver examines if the existing key partition of S satisfies all constraints over S. If so, the key partition of S can be reused and all dependent partitions can be recomputed. Otherwise, the solver selects an arbitrary new partition for S that satisfies the considered constraints. A small visual example of constraint resolution is shown in Figure 3.5.

The independent partitions may also form a co-partitioned set, such as the partition constraint set $\{\mathsf{equal}(S_1, S_2), \mathsf{equal}(S_2, S_3)\}$. In this case, a partition must be chosen for all of S_1, S_2 and S_3 independently from other stores, but must be the same. In this case, the solver selects a key partition to use (if one exists) from the three stores that minimize the total amount of data movement.

More aggressive constraint solving strategies are possible, such as solution strategies discussed by Lee et al. [34] that solve for partitions over an entire graph of tasks at once. Investigating such approaches in the context of iterative Legate computations is interesting future work. However, we do not currently have non-contrived applications where more global partitioning strategies are required.

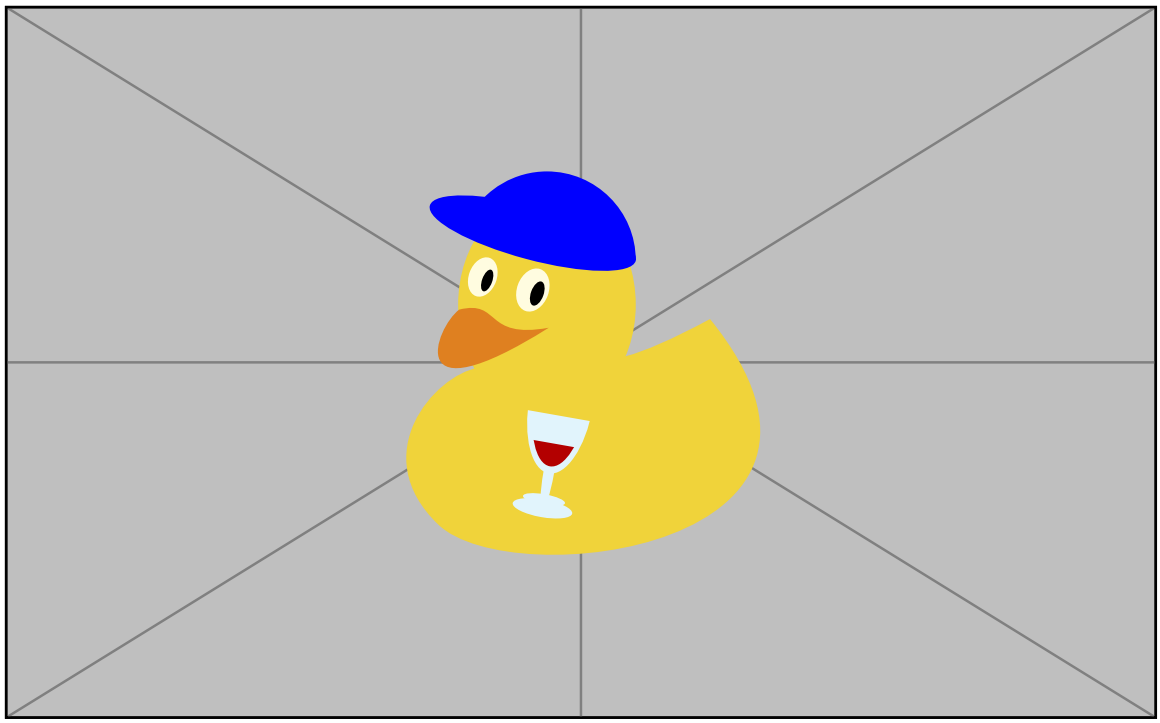


Figure 3.5: Figure that shows constraint resolution.

3.2 Legion

In the previous section, I discussed the Legate front-end's program representation, which used implicit scale and implicit partitioning to avoid unnecessary data movement caused by incoherent partitioning decisions made by independent libraries. The Legate front-end representation is critical in enabling independent libraries to compose with high performance. I now skip the Legate middle-end representation (discussed in Chapter 4) to focus on the Legion layer of the runtime stack, which enables both the correct and efficient composition of independent distributed libraries. This thesis does not focus on the internals or implementation of Legion, which are detailed in prior work [10, 13–15]. Instead, this thesis focuses on the abstractions exposed by Legion, and how those abstractions are used to enable the efficient composition of Legate programs.

Syntax

$$
\begin{array}{rlll}
\textit{Unique ID} & id & & \\
\textit{Point} & p & ::= & (\mathbb{Z}, \ldots) \\
\textit{Rect} & r & ::= & [p, p] \\
 & & & \\
\textit{Store} & S & ::= & \mathsf{Store}(id, r) \\
\textit{Partition} & P & ::= & \mathsf{map}(p, p\ \mathsf{set}) \\
 & & & \\
\textit{Privilege} & Pr & ::= & \mathsf{Read} \mid \mathsf{Write} \mid \mathsf{Reduce} \mid \mathsf{Read\text{-}Write} \\
\textit{Task} & T & ::= & \mathsf{Task}(p\ \mathsf{set}, (S, Pr, P)\ \mathsf{list}) \\
\textit{Program} & P & ::= & T\ \mathsf{stream}
\end{array}
$$

Figure 3.6: Legion program representation.

3.2.1 Legion Program Representation

As discussed in Chapter 2, Legion exposes an implicitly-parallel task-based programming model with explicit partitioning and explicit scale. Figure 3.6 presents a grammar that describes a simplified version of the Legion programming model. The structure of the Legion programming model is similar to the Legate front-end, as both layers represent programs as streams of tasks operating over stores. However, the differences arise in the representation of parallelism in computation and data.

Legion represents partitions as explicit sets of subsets of stores[1]. Partitions are represented as maps from a set of *colors* (points in a multi-dimensional index space) to a set of multi-dimensional points that represent a subset of a store. These partitions can be viewed as the output of the partitioning constraint solver in **??**, though in reality they go through an additional lowering process before conversion into Legion partitions (**??**). These subsets may be arbitrary, allowing for disjoint/aliased and complete/incomplete partitions. Scale is also represented explicitly in Legion. Each task in the program representation actually refers to a group of *point tasks*, one point task for each point in the argument set. The data each point task accesses is described by the point's index into the corresponding partition of each argument store. Despite this lower-level program representation (where decisions about scale

[1]Stores are referred to as *regions* in Legion-only publications.

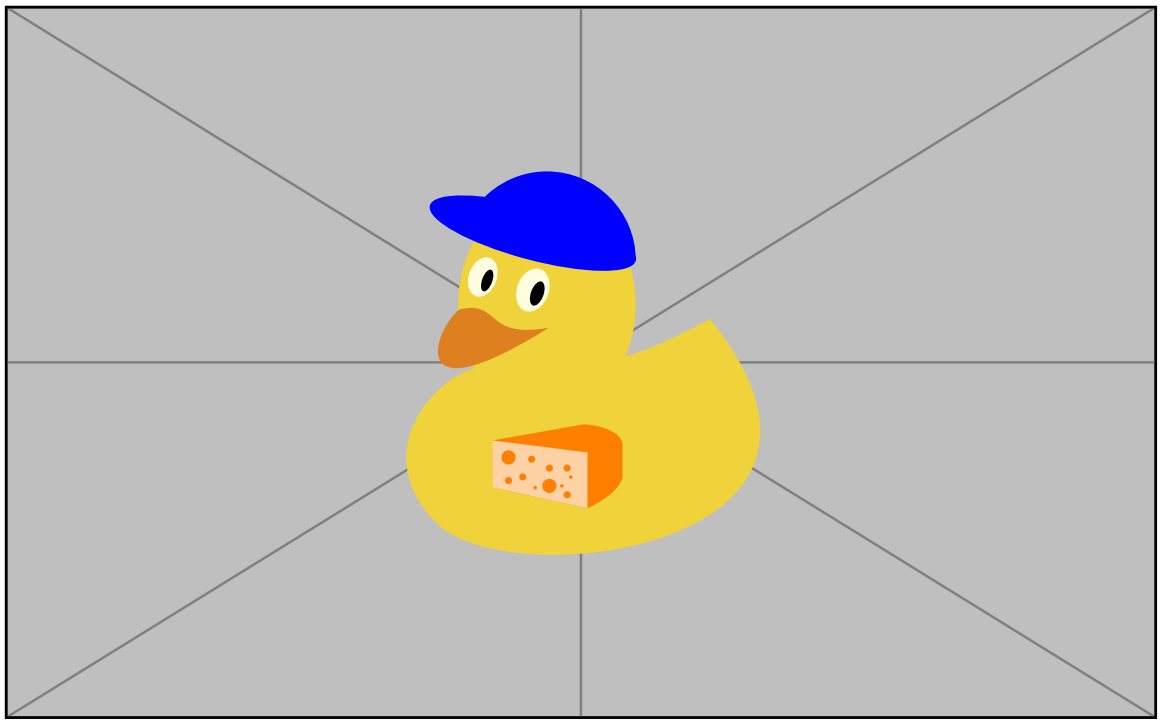


Figure 3.7: Figure that is the Legion composition example.

and data partitioning are concrete), parallelism is still implicit; Legion is responsible for extracting parallelism from tasks in the stream while maintaining sequential semantics between tasks in the stream.

3.2.2 Composition Through Implicit Parallelism

Legion's implicitly-parallel program representation and sequential semantics offer critical capabilities for correct and efficient composition of independent libraries, and logically isolates the development of these libraries. To extract parallelism from an implicitly-parallel, distributed program while maintaining sequential semantics, Legion is responsible for identifying and enforcing dependencies between all tasks a newly launched task, and inserting communication to ensure the data a task views is the most up-to-date version of the data.

By performing these two analyses, Legion acts as the "glue" between independent Legate libraries. As shown in Figure 3.7, Legate libraries issue tasks independently of each other, only declaring the stores they access and how they are accessed. The

Legate front-end selects partitions of stores for these tasks, but thanks to Legion, can focus only on the *policy* aspect of partitioning constraint resolution. The Legate front-end can select arbitrary choices of partitions to minimize data movement, and Legion is responsible for actually inserting the necessary data movement and synchronization (possibly across library boundaries) that result from those partitioning choices. Additionally, Legion can exploit parallelism available between libraries by finding pieces of independent work between libraries and running them concurrently, or overlapping the communication required to prepare data for one library with independent computation from a separate library. These optimizations are key pieces of tuned HPC applications that run efficiently on modern supercomputers; these applications utilize every available resource at all times to avoid exposing bottlenecks.

The kind of efficient composition enabled by Legion's implicit representation of parallelism and communication is difficult to achieve in explicitly-parallel programming models that commit to communication and synchronization with only the local program view. Since an explicitly-parallel library must describe at program development time the synchronization and communication patterns, these libraries are unable to adapt to the larger context a library is used in. For example, an MPI-based library that internally synchronizes after message passing would be unable to automatically overlap independent work from an unrelated library, or might send more data than necessary if the desired data was already available on the current node.

3.2.3 Sharing Physical Data With Composable Mapping

Independent Legate libraries describe computations over distributed data through tasks operating on stores. The Legate runtime system is responsible for actually executing these tasks on concrete processors and placing the partitioned pieces of stores into concrete memories for tasks to access. This process is called *mapping* within the Legion ecosystem; Legion exposes this functionality through *mapper* objects, which the Legion runtime system interacts with to answer policy decisions around where tasks should run and stores should be placed. The Legate runtime system as a whole implements and registers a mapper with Legion. Some policy decisions are exposed to

individual libraries (for example, Legate libraries may request tasks to run on certain kinds of processors), but others are coordinated across library boundaries through shared mapping infrastructure provided by Legate. Concretely, the Legate runtime controls the exact processors that tasks are mapped to, as well as the mapping of stores to physical allocations in concrete memories in the machine. Centralized control over the exact processors used by tasks avoids thrashing of data between different processors assigned to process the same partitions of data.

The more difficult aspect of composing mapping decisions across libraries is the mapping of stores to memories on the machine. The key challenge involved in mapping stores is how to share allocations of distributed and partitioned data between libraries. Because interactions betwen libraries are unknown until chosen by the partitioning constraint solver, the partitions of stores and the aliasing patterns of those partitions are also not known until program execution. To minimize data movement and memory usage, the mapping strategy must reuse and resize physical allocations across individual operations and library boundaries.

Legate maintains a record of all store allocations made on each node, and reuses physical allocations across as many tasks as possible. However, reusing allocations that exactly match the extents of a store is not sufficient to achieve good performance, as tasks from different libraries may use multiple views of the same underlying store. 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 partitioned pieces of a store into a single allocation, Legate employs a *coalescing* step before allocating. When selecting an allocation for a store, Legate examines the existing allocations for other partitions of the same store. If another store's partition has an intersection with the partition being allocated, then Legate may merge the two views of the data into a single, larger allocation with enough space for both views of the store. We use a heuristic to drive coalescing decisions, where store partitions are coalesced if the size of their overlapping components is sufficiently larger than their non-overlapping components.

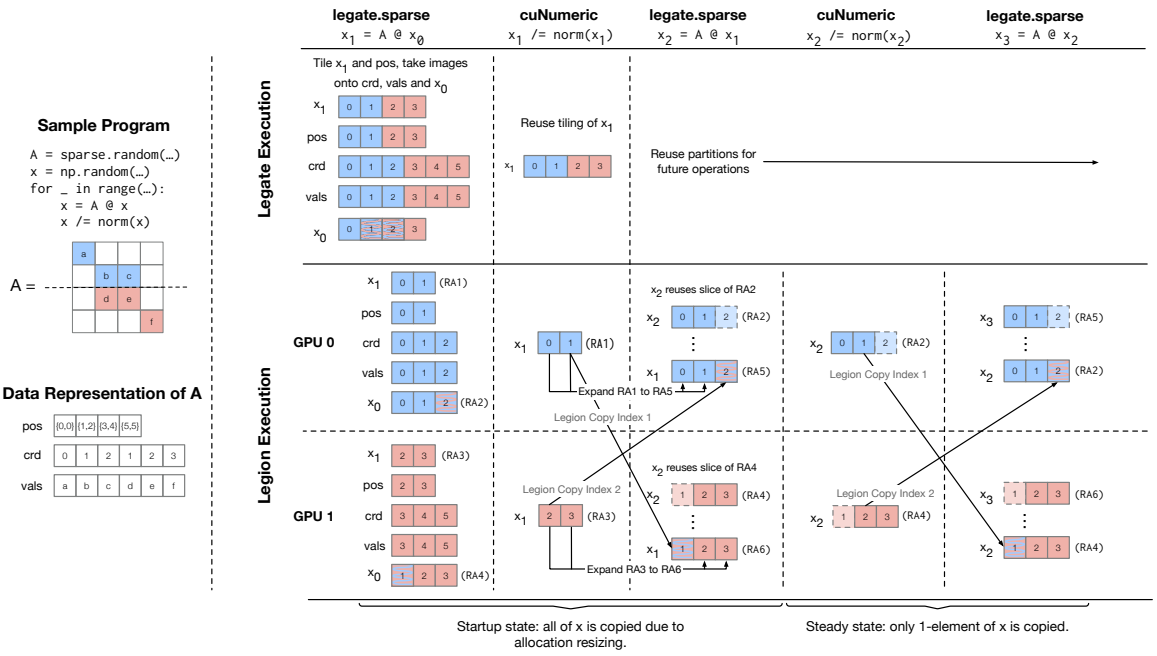


Figure 3.8: Execution of sample Legate program, with control flowing between Legate Sparse and cuPyNumeric. Turn RA (region allocation) into SA (store allocation)

3.2.4 Execution Example

Figure 3.8 visualizes the execution of a Legate program that combines operations from cuPyNumeric (vector norm and division) and Legate Sparse (sparse matrix-vector multiplication) on a two GPU machine. An efficient implementation of this main loop with global knowledge only performs one element halo exchanges of the x vector between GPUs, and no other data movement. This figure demonstrates how the design of the Legate runtime system enables the efficient sharing of partitions and physical allocations of distributed data across library boundaries to achieve this property in the resulting composed software.

The left part of Figure 3.8 contains the source program along with an example sparse matrix A and the stores used to represent it (see Chapter 7 for more details about distributed sparse matrices and their layouts). 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 store entry is labeled with the coordinate of the

entry, rather than the data value at that entry. 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.

We first discuss the top half of the figure, which shows how stores are partitioned for distributed execution. Legate Sparse's representation of sparse matrices and constraints used to partition them (discussed in Chapter 7) relate the `pos`, `crd` and `vals` stores with the input and output vectors. Legate solves the constraints to construct partitions (identified with blue and red colors) of each store on the first iteration, and creates a tiled key partition of x_1 and an aliased partition of x_0, based on the data dependent coordinates needed in the sparse matrix. Control flows to cuPyNumeric for the norm and division operations (which we treat as a single operation for illustration purposes). These are elementwise operations without partitioning constraints, so Legate reuses the key partitions created during the solving of Legate Sparse's constraints for cuPyNumeric. These partitioning choices are then reused for all future iterations of the main loop.

We now shift to the bottom half of Figure 3.8, which depicts the execution with Legion, and the mapping of logical operations onto physical resources. For all tasks launched, the Legate Sparse and cuPyNumeric mapping policies assign tasks and stores to each GPU and the corresponding GPU memory. The key to peak performance in this program is the Legate mapper's choice of allocations for each store.

In the first iteration, Legate creates allocations for each store (denoted with "SA") that correspond to the exact bounds of each store requested by each library's tasks. The choices made in the second iteration of the program stress the importance of the compositional-awareness of Legate's mapping strategy. When mapping the second SpMV operation, the mapper chooses new allocations (SA5 and SA6) for each piece of x_1, resizing the allocations SA1 and SA3 to account for the larger slice of x_1 required by each SpMV task. Resizing SA1 and SA3 requires a full copy of x_1, and a single element halo-copy between GPUs. Next, Legate sees that the store backing x_0 can be deleted, and chooses to reuse the allocations SA2 and SA4 by coalescing them with the requested partition of 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.

3.3 Results

The presented strategy and Legate runtime system subset (focusing on composing distributed data efficiently) is sufficient to implement non-trivial applications that achieve good performance and scale to large machines. Chronologically, the Legate Sparse publication [59] was first, and contained the presented core techniques to build independent libraries that efficiently share distributed data. I will discuss the initial applications developed using a combination of Legate Sparse and cuPyNumeric and evaluated on the (now decommissioned) Summit supercomputer. I will show that the presented techniques for composing distributed data enable complex applications to be built from independent components, while still achieving performance competitive with hand-tuned systems.

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.

Our initial evaluation of Legate consists of Python applications developed using a combination of SciPy Sparse and NumPy, and range from twenty-five to nearly a thousand lines of code. We measure Legate'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.

Weak Scaling

In this section, we evaluate the weak-scaling performance of Legate, 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.

CG Solver. We implemented a conjugate-gradient iterative linear solver for a 2-D Poisson problem, with the results displayed in Figure 3.9. As with the previous experiment, we compare both modes of Legate to the same code run in SciPy and CuPy, and a comparable implementation in PETSc. Legate's CPU mode outperforms SciPy due to being multi-threaded. Legate and PETSc achieve nearly perfect weak scaling on CPUs, with PETSc slightly outperforming Legate, as Legion reserves some CPU resources for runtime work. On GPUs, CuPy, Legate and PETSc have similar performance on a single GPU, with Legate achieving 85% percent of the performance of PETSc. PETSc and Legate then weak-scale from a single GPU, where PETSc achieves nearly perfect weak scaling, starting to fall off slightly at 192 GPUs. Legate 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[2], causing

[2]The Legion developers are aware of this issue, and plan to address it in the future.

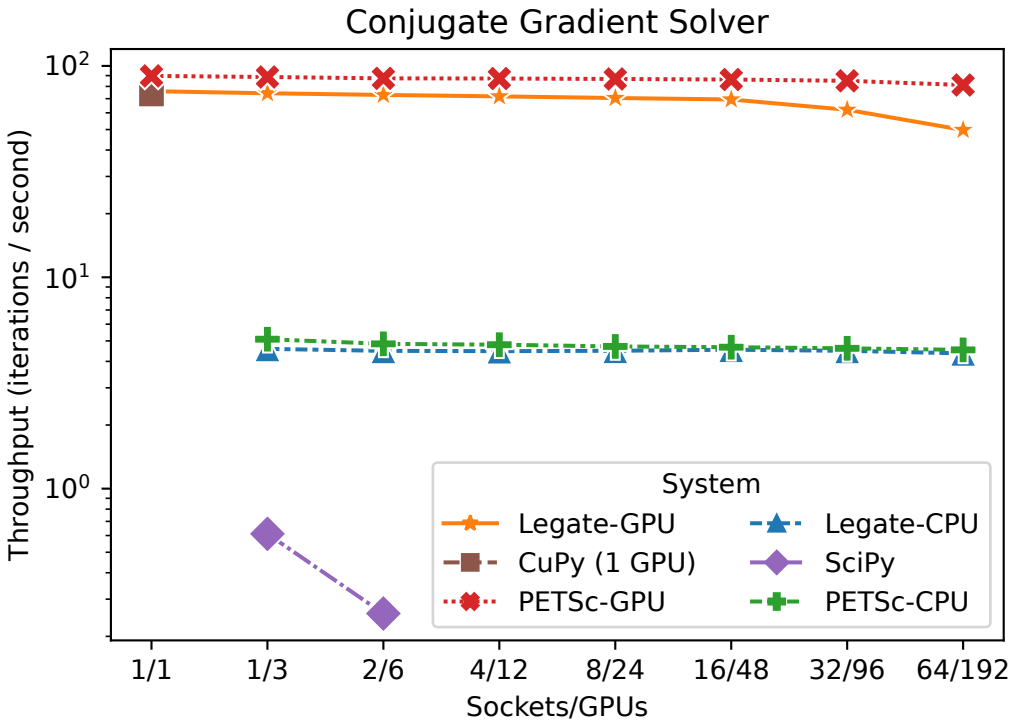


Figure 3.9: Weak-scaling of a Conjugate Gradient solver.

the dot-product communication in the CG solve to affect Legate's performance at a smaller processor count than PETSc. At 192 GPUs, Legate 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[3]. 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's CPU mode to SciPy, Legate's GPU mode to CuPy, and then weak-scale to larger machines. Figure 3.10 contains the weak-scaling results for the geometric multi-grid solver. As with prior experiments, Legate's CPU version significantly outperforms SciPy and has good weak-scaling to 64 sockets. On a single GPU, CuPy is 30% faster than Legate's GPU version. This performance difference is caused by

[3]This benchmark was inspired by, but is not directly comparable to HPCG [27].

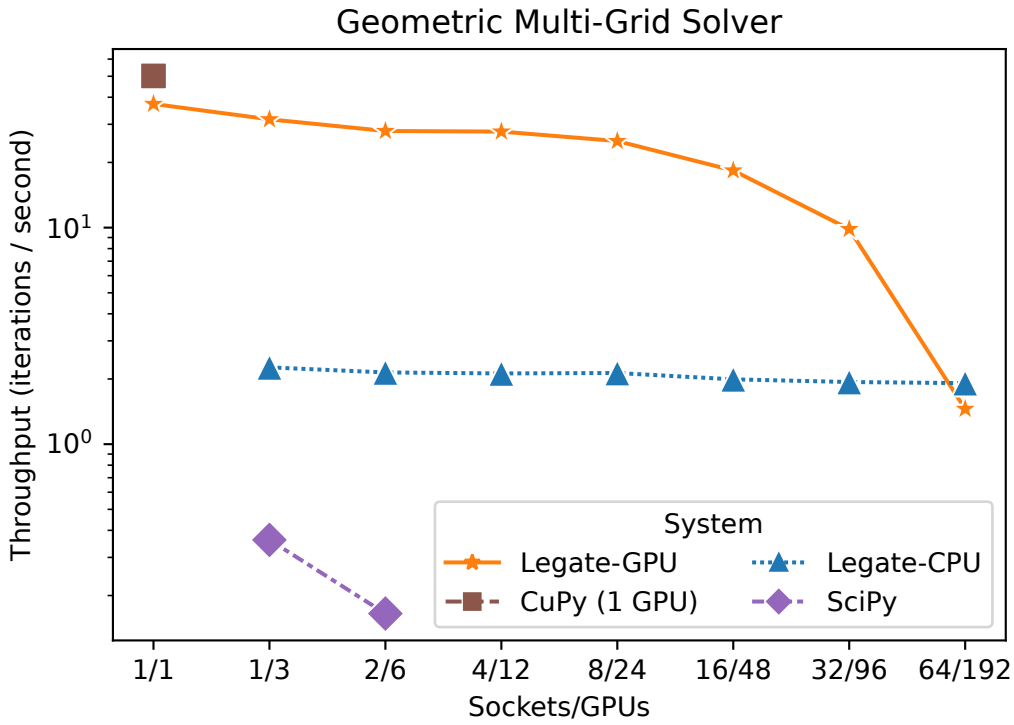


Figure 3.10: Weak-scaling of a Geometric Multi-Grid solver.

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's GPU version starts off weak-scaling well, but has kernels that run fast enough to expose overheads in Legion, which have been fixed since the original collection of these results. Benchmark results later in this thesis (in Chapter 4) show significantly improved strong scaling of the GMG code. Despite the imperfect weak scaling, Legate 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 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 [24]. Like previous implementations [41], we significantly reduce the memory footprint of the

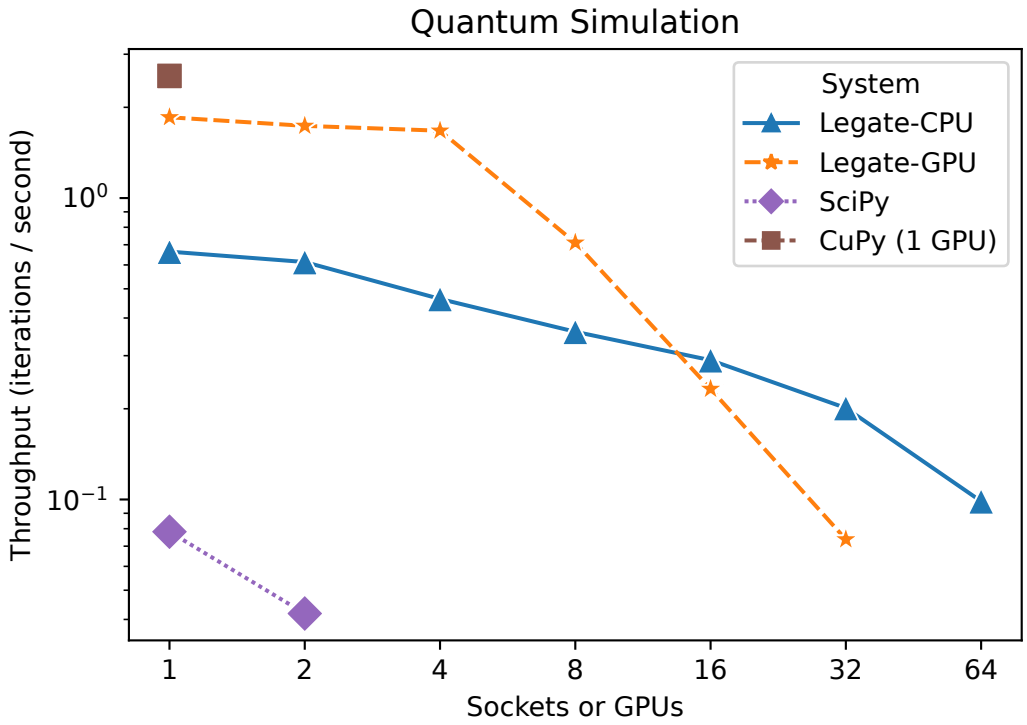


Figure 3.11: Weak-scaling of a Runge-Kutta integration-based Quantum Simulation.

simulation by including only states that are allowed by the Rydberg blockade mechanism [36]. 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. using the "DOP853" method, which we ported from directly from the implementation in SciPy. Despite being written with no expectation of distributed execution, Legate is able to efficiently parallelize and distribute this program, leveraging the efficient composition enabled by sharing distributed data across Legate Sparse and cuPyNumeric.

Similarly to the GMG benchmark, we compare against SciPy and CuPy. Due to the nature of the 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 can successfully utilize all 6 GPUs per node for standard runs of the simulation.

The weak-scaling results are found in Figure 3.11. As with prior experiments, Legate significantly outperforms the standard implementation of SciPy. On a single GPU, CuPy achieves a 40% speedup over Legate, 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'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's GPU version is utilizing 4 nodes of network hardware, while Legate'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's 64 GPU version to run out of memory.

3.3.1 Conclusion

Some sort of synthesis of this work and connection to the other pieces of the thesis.

Chapter 4

Composing Distributed Computation

The previous chapter presented the Legate front-end program representation and Legate's use of the Legion runtime system to allow independent libraries to efficiently and correctly share distributed data, and skipped the intermediate layer between the front-end and Legion. This chapter focuses on the program representation and analysis used within the Legate middle-end that enables the fusion of distributed computations across function and library boundaries. We show that careful choice of these representations and placement within the larger Legate architecture enables a scalable, efficient, and precise fusion analysis. Legate separates the problem of fusion into *task fusion*, which determines whether the bodies of tasks can be combined without communication, and *kernel fusion*, which fuses the many computational loops of fused tasks. This analysis allows for Legate to find optimization opportunities missed by application developers, and allows Legate applications to even exceed the performance of programs developed in specialized frameworks.

I elided the motivating example section from the asplos paper, but I'm wondering again if more motivation is necessary here, as i think somewhere we want to get across that the problem of safely identifying when fusion is possible is really the hard thing, and it isn't completely trivial either.

4.1 Legate Middle End

As discussed in Chapter 2, the Legate middle-end program representation is implicitly parallel (like the Legate front-end and Legion), but is explicitly partitioned with symbolic scale. In this representation, stores have been assigned concrete partitioning strategies, and the width of parallelism is represented symbolically. The symbolic representation means that even though data has been partitioned, the size of the program representation itself does not scale with the size of the target machine (unlike in lower-level models like Legion and Realm), enabling a fusion analysis that also is independent of the scale of the target machine. The Legate middle-end representation is shown in Figure 4.1a, visualized in Figure 4.1b, and discussed in detail next. The data and computational abstractions are then lowered onto the corresponding Legion abstractions after the analyses discussed in ??.

4.1.1 Data Model

Similarly to the Legate front-end and Legion, distributed data is represented as stores. We refer to the rectangular shape of a store as a *domain*, which is also used to describe the decomposition of data and compute across processors. Stores are partitioned across the target machine into *sub-stores*, which are the subsets of a store.

Partitions of stores are first-class objects in the Legate middle-end, unlike the constraint-based representation in the Legate front-end and the set-of-sets representation in Legion. A *partition* is a mapping from points in a domain to sub-stores, where each point in the domain represents a processor. This mapping is represented in a structured manner, breaking different kinds of mappings into different syntactic groups. For simplicity of presentation, we consider a limited set of partition kinds, sufficient to explore Legate's analyses. The production implementation of Legate supports more partition kinds with no additional technical insights. The main requirement on partitions is that two partitions of the same kind can be checked for inequality without examining each sub-store within each partition. This requirement is critical for a scalable analysis, as discussed in **??**.

The first partition kind None represents the replication of a store, where all points

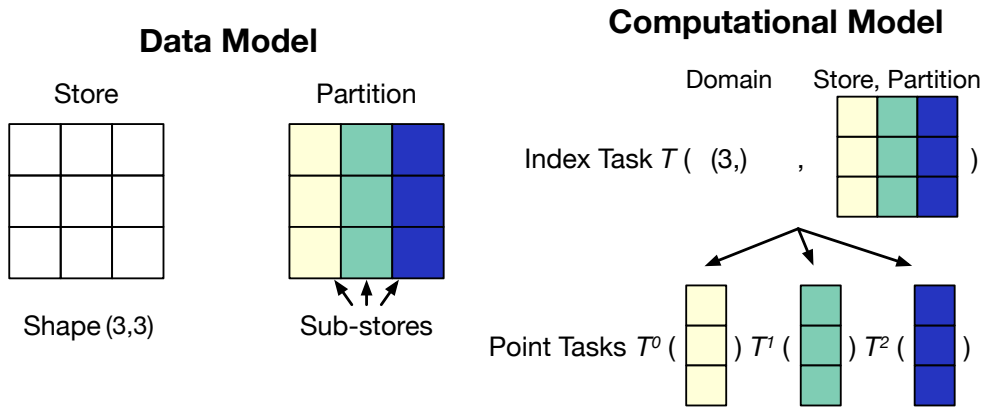
Syntax

Unique ID	id		
Point	p	::=	$(\mathbb{Z}, \ldots)$
Store	S	::=	Store(id, p)
Projection Function	F	::=	Projection$(id, \text{Point} \to \text{Point})$
Partition	P	::=	None | Tiling(p, p, F) | Image(S, S)
Privilege	Pr	::=	Read | Write | Reduce | Read-Write
Index Task	T	::=	IndexTask$(p, (S, P, Pr)$ list$)$
Task Window	W	::=	T stream

Constructs for Reasoning

Sub-Store	S^p	$\triangleq$	SubStore(S, P, p)
Point Task	T^p	$\triangleq$	PointTask$((S^p, Pr)$ list$)$

(a) Legate's intermediate representation. make this consistent with the other IR diagrams in figure 3.

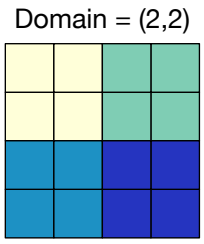


(b) Relationships between components of the Legate middle-end representation.

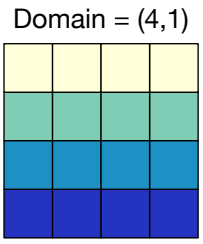
Figure 4.1: Legate's IR exposes a distributed data model and a model for distributed computation on distributed data.

in the partition's domain are mapped to the entire store. The second partition kind Tiling represents an n-dimensional affine tiling of a store. A Tiling contains an n-dimensional tile shape and an offset from the origin, which are used to compute the sub-store associated with each point in the partition's domain. For example, Figure 4.2a shows a tiling of a two-dimensional store using 2x2 tiles over a 2x2 domain, while Figure 4.2b shows a row-based tiling (i.e., tiles of size 1x4) of the same store over a 4x1 domain. Figure 4.2c shows a partition of a subset of the store beginning at the point $(1, 1)$. Tiling partitions also contain a *projection function* that applies a transformation to each point in the partition's domain before computing the subset with the tile size and offset. Projection functions enable Tiling partitions to express aliased and replicated data. For example, Figure 4.2d shows a vector tiled over a two-dimensional domain by a projection function that discards the second dimension of each point in the partition's domain, resulting in a partially aliased partition. The formula that defines the sub-store bounds for each point of a Tiling partition is shown in Figure 4.2e. The final partition kind is Image, which .... The representations of None and Tiling partitions are scale-free as the mapping of points to sub-stores is implicit in the partition, rather than explicitly storing the bounds of each sub-store in the partition. These partition kinds are the concrete partitions synthesized by the partitioning constraint resolution process described in Section 3.1.3.

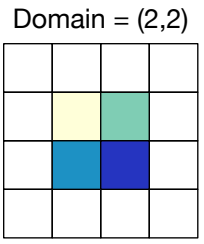
To reason about the sub-store referenced by each point of a partition, we include an explicit $\mathsf{SubStore}(S, P, p)$ construct, representing the sub-store associated with point p of store S using partition P. As a short-hand, we let $S[P, p] = \mathsf{SubStore}(S, P, p)$, and refer to S as the *parent* store of $S[P, p]$. The indices contained within the sub-store $S[P, p]$ are directly computable in cases when P is None or Tiling, but may depend on runtime values held by stores in when P is Image. Our later definitions assume that it is possible to find the intersection between two sub-stores, but our fusion algorithm in **??** does not require explicit computation of these intersections.



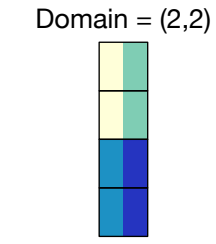
(a) 2x2 tiling of a 4x4 store.



(b) 1x4 tiling of a 4x4 store.



(c) Offset 1x1 tiling of a 4x4 store.



(d) Aliased blocking of a size 4 store.

$$\text{sub-store-bounds}(\text{Tiling}(\text{shape}, \text{offset}, \text{proj}), p) =$$

$$[\text{proj}(p) * \text{shape}, \text{proj}(p+1) * \text{shape}) + \text{offset}$$

(e) Function that computes a bounding-box within the store that a Tiling partition maps point p to.

Figure 4.2: Examples of Tiling partitions. Partitions maps points in the denoted domain to sub-stores. Each color refers to the sub-store associated with a each point in the domain.

4.1.2 Computational Model

The Legate middle-end represents programs as a stream of tasks. An $\mathsf{IndexTask}(d, A)$ represents a group of parallel tasks over points in a rectangular domain d, referred to as the *launch domain*. Stores are passed to tasks with privilege annotations just like in other layers of the Legate runtime system, and each parallel task within the group reads from, writes to, or reduces to the sub-stores referred to by the stores and partitions at each point. This representation is explicitly parallel as tasks are annotated with their launch domain and partitions of distributed data structures. However, the representation is scale-free as the size of the representation is independent of the degree of parallelism — only the symbolic size of the launch domain increases.

Similar to sub-stores, Legate's middle-end IR has a notion of a *point task*, which is one point in an index task's launch domain. Given an index task $T = \mathsf{IndexTask}(d, A)$, let T^p be the point task at point $p \in d$, operating on the list of stores $[(S[P, p], pr) : \forall\ (S, P, pr) \in A]$. Point tasks operate on the sub-stores corresponding to their index point.

We define the predicates $\mathsf{R}(T, (S, P))$, $\mathsf{W}(T, (S, P))$ and $\mathsf{Rd}(T, (S, P))$ to be true when the task T correspondingly reads from, writes to, or reduces to the store S using partition P. When (S, P) has the privilege Read-Write, both $\mathsf{R}(T, (S, P))$ and $\mathsf{W}(T, (S, P))$ are true. We also overload these predicates for point tasks and sub-stores, where $\mathsf{R}(T^p, S)$ is true when point task T^p reads sub-store S.

4.2 Distributed Task Fusion

rohany: improve the handling of "index tasks" nomenclature here. Legate leverages the middle-end representation to fuse distributed computations through task fusion, when then enables the fusion of computational kernels within the fused tasks (Section 4.4) yielding end-to-end speedup through reduced runtime overheads and faster kernels. Legate buffers the stream of issued tasks into a *window* of tasks to be analyzed before submission to Legion. A distributed task fusion algorithm finds a fusible prefix of tasks in the window, and replaces the prefix with a fused task. To be fusible,

the prefix of index tasks must be executable in sequence without cross-processor communication. We define when communication may occur between index tasks and describe when a sequence of index tasks is fusible. We then give an algorithm for finding fusible index task sequences.

4.2.1 Dependencies

Communication is required between point tasks that have a dependence. The dependence implies synchronization and potentially data movement between the point tasks. A dependency exists between two point tasks that access the same data unless both tasks read from or reduce to the data with the same associate and commutative operator. Recall that for an index task T, we refer to the point task at point p as T^p. We define $\mathsf{dep}(T_1^p, T_2^{p'})$ to be true if $T_2^{p'}$ depends on T_1^p.

Definition 1. Given point tasks T_1^p, $T_2^{p'}$ where index task T_1 is issued before index task T_2, $\mathsf{dep}(T_1^p, T_2^{p'})$ if $\exists$ sub-stores S, S' with the same parent such that $S \cap S' \neq \emptyset$ and either

true-dep:

$$\mathsf{W}(T_1^p, S) \wedge \Big(\mathsf{R}(T_2^{p'}, S') \vee \mathsf{W}(T_2^{p'}, S') \vee \mathsf{Rd}(T_2^{p'}, S')\Big)$$

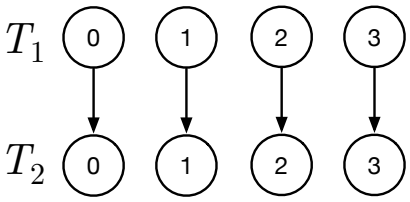
anti-dep:

$$\mathsf{R}(T_1^p, S) \wedge \Big(\mathsf{W}(T_2^{p'}, S') \vee \mathsf{Rd}(T_2^{p'}, S')\Big)$$

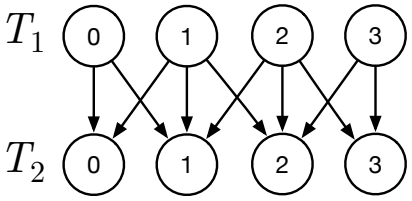
reduction-dep:

$$\mathsf{Rd}(T_1^p, S) \wedge \Big(\mathsf{R}(T_2^{p'}, S') \vee \mathsf{W}(T_2^{p'}, S')\Big).$$

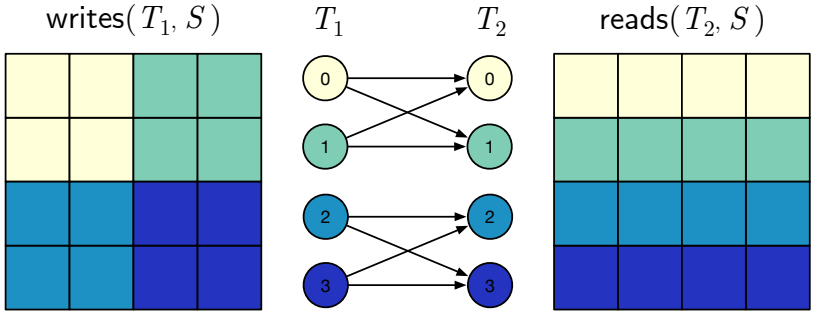
The dependencies between two index tasks T_1 and T_2 are defined by the pairwise dependencies of their point tasks. We capture these dependencies through a mapping between the points of T_1 and T_2 that represents all of the point tasks in T_2 that depend on point tasks in T_1. Figure 4.3 shows different dependence maps over the launch domain $(4,)$.



(a) Point-wise dependence map: $\mathcal{D}(T_1, T_2)[p] = \{p\}$



(b) Stencil dependence map: $\mathcal{D}(T_1, T_2)[p] = \{p-1, p, p+1\}$



(c) $\mathcal{D}(T_1, T_2)$ of writing to, then reading from different partitions.

Figure 4.3: Visualization of dependence maps $\mathcal{D}(T_1, T_2)$.

Definition 2. For two index tasks T_1 and T_2, the *dependence map* $\mathcal{D}(T_1, T_2)$ is a function of type $\mathsf{domain}(T_1) \to \mathcal{P}(\mathsf{domain}(T_2))$, where $\forall p \in \mathsf{domain}(T_1), \mathcal{D}(T_1, T_2)[p] = \{p' \in \mathsf{domain}(T_2) : \mathsf{dep}(T_1^p, T_2^{p'})\}$.

Having characterized the dependencies between two distributed index tasks T_1 and T_2, we can now define when fusion of T_1 and T_2 is valid. T_1 and T_2 may be fused if the only dependencies that exist between their point tasks are at most point-wise, as the processor that executes each point task does not need to communicate with any other processors.

Definition 3. Index tasks T_1 and T_2 are fusible if $\forall p, \mathcal{D}(T_1, T_2)[p] \subseteq \{p\}$.

While Definition 3 admits a simple dependency structure, there are several subtleties in what tasks are fusible and the identification of fusible tasks. First, tasks with at most point-wise dependencies is a broader set than just tasks that perform point-wise array operations. Point-wise dependencies allow for simultaneous reads and writes of different stores (Section 5.1.1) and multiple reductions to the same store. While task dependencies may be at most point-wise, the computations within

the tasks are arbitrary computations that may be more complex than point-wise operations. Next, identifying when at most point-wise dependencies exist between two index tasks is non-trivial as tasks operate on arbitrarily aliasing distributed data. Legate defines a framework, discussed next, to reason about fusion in this setting, allowing for fusion to be performed between components within and across libraries.

4.2.2 Fusion Algorithm

A naïve algorithm for fusion might fully materialize $\mathcal{D}(T_1, T_2)$ to check that the condition in Definition 3 holds. However, the computation required to materialize $\mathcal{D}(T_1, T_2)$ scales with the number of processors. Even lower-level systems like Legion do not materialize all of $\mathcal{D}$, but instead leverage sophisticated algorithms to compute only the portion of $\mathcal{D}$ needed by each node [14]. However, a key insight in Legate is that to perform distributed task fusion effectively, our analysis only needs to rule out cases where $\exists\, p, \mathcal{D}(T_1, T_2)[p] \not\subseteq \{p\}$. Legate's middle-end representation enables this analysis to be performed irrespective of the scale of the target machine. Our algorithm for distributed task fusion identifies when index tasks have point-wise dependencies through greedy application of a set of *fusion constraints* to identify a fusible prefix of the task window. We then build a fused task from the identified prefix. We describe each of these components in turn, and then sketch a correctness proof.

Fusion Constraints.

We define four constraints to identify when communication may occur between distributed index tasks, i.e., when $\exists p, \mathcal{D}(T_1, T_2)[p] \not\subseteq \{p\}$. These fusion constraints are sound, but not complete—for example, leveraging application knowledge could result in fusion opportunities that are out of scope. Figure 4.4 presents each of the constraints used by Legate by defining when a provided sequence of tasks satisfy the constraint.

Launch Domain Equivalence. The first constraint checks that the tasks to be fused have the same launch domain. The partitioning constraint resolution process may decompose computations across different launch domains, or applications may

$$
\begin{aligned}
&\mathsf{launch\text{-}domain\text{-}equivalence}([T_1, \ldots, T_n]) = \\
&\quad \forall i, \mathsf{domain}(T_i) = \mathsf{domain}(T_1) \\
&\mathsf{true\text{-}dependence}([T_1, \ldots, T_n]) = \\
&\quad \forall\ T_i \text{ s.t. } \mathsf{W}(T_i, (S, P)), \\
&\quad \nexists T_j \text{ s.t. } (\mathsf{R}(T_j, (S, P')) \vee \mathsf{W}(T_j, (S, P'))) \ \wedge i < j \wedge P \neq P' \\
&\mathsf{anti\text{-}dependence}([T_1, \ldots, T_n]) = \\
&\quad \forall T_i \text{ s.t. } \mathsf{R}(T_i, (S, P)), \\
&\quad \nexists T_j \text{ s.t. } \mathsf{W}(T_j, (S, P')) \ \wedge i < j \wedge P \neq P' \\
&\mathsf{reduction}([T_1, \ldots, T_n]) = \\
&\quad \forall T_i \text{ s.t. } \mathsf{Rd}(T_i, (S, P)), \\
&\quad \nexists T_j \text{ s.t. } (\mathsf{R}(T_j, (S, P')) \vee \mathsf{W}(T_j, (S, P'))) \ \wedge i \neq j
\end{aligned}
$$

Figure 4.4: Fusion constraints employed by Legate to identify potential communication between index tasks.

scope their computation directly to different subsets of the machine, requiring different launch domains. Data movement is generally required between these different decompositions.

True Dependence. The next constraint utilizes the partitions of stores and the privileges with which they are accessed to identify communication between index tasks caused by read-after-write dependencies. The true-dependence constraint checks that if a task T_i writes to a store S through partition P, then it cannot be followed by a task T_j that reads or writes to S with an aliasing partition P', as T_j requires communication of the updated values written by T_i. However, operating on the same partition P is permitted, preserving point-wise dependencies between T_i and T_j.

Our analysis relies on the ability to check whether two partitions alias, which Legate does through a constant-time equality check between partitions. Constant-time alias checking is possible through the Legate middle-end representation's symbolic representation of scale and the syntactic grouping of partitions into structured kinds. Legate does not need to compute pairwise intersections of the sub-stores accessed by the point tasks of considered index tasks, a computation that scales quadratically with the number of processors. Additionally, the alias analysis does not depend on the structure of the partitions, as the constraints are defined without knowing the syntactic kinds of each partition. Finally, this aliasing check is not too coarse, since partitions of different syntactic kinds nearly always alias in practice.

Anti-Dependence. The anti-dependence constraint ensures that $\mathcal{D}$ does not contain write-after-read dependencies. The constraint enforces that if a task T reads a store S, then any tasks that write to S must write to the same distributed view as the read to be fused with T. Thus, a fused task may read from multiple different distributed views of a store, but then cannot write to any of the views, as such an operation would require communication of the written data.

Reduction. The reduction constraint makes sure that viewing a partially reduced value is not allowed. It does not permit a task that reads from or writes to a store to be fused with a task performing a reduction to any view of the same store.

Fused Task Construction.

Legate's fusion algorithm greedily applies the fusion constraints on the input task window to find its longest fusible prefix. The true-dependence and anti-dependence constraints are verified through a forwards dataflow analysis on the task window. The analyses iterate through the candidate prefix of tasks, and track the effects that each task applies to its argument stores. Once a suitable prefix of the task window has been identified, Legate builds a fused task that has all store arguments and executes the same computation as the identified prefix of tasks. The fused task's store arguments are constructed by reading all stores read by tasks in the prefix, and similarly for the written to and reduced to stores. Stores that are both read from and written to are promoted to the Read-Write privilege. Legate constructs the body of the fused task by composing the bodies of each task in the prefix in program order—we further discuss this process in Section 4.4.

4.2.3 Proof of Correctness

We now show that our algorithm correctly fuses sequences of distributed index tasks. We prove the following statement:

Theorem 1. Given a window of tasks $[T_1, \ldots, T_n]$, our task fusion algorithm identifies a prefix $[T_1, \ldots, T_f]$ and produces a fused task F such that

1. $[T_1, \ldots, T_f]$ are fusible, and

2. F preserves all dependencies of the task sequence $[T_1, \ldots, T_f]$.

We provide a proof sketch for each component of the theorem. To prove that $[T_1, \ldots, T_f]$ are fusible, we must show that for each pair of tasks $T_i, T_j, i < j$ in $[T_1, \ldots, T_f]$, $\forall\ p,\ \mathcal{D}(T_i, T_j)[p] \subseteq \{p\}$. The launch-domain-equivalence constraint ensures that the dependence map is between points of the same dimensionality. For the sake of obtaining a contradiction, suppose $\exists\ p, p'$ such that $p \neq p'$ and $\mathsf{depends}(T_i^p, T_j^{p'})$. Then one of the three dependencies in Definition 1 must exist. Suppose that the condition for true-dep is satisfied, meaning that $\exists S, P, P'$ such that $S[P, p] \cap S[P', p']$ and $\mathsf{W}(T_i, (S, P))$ and one of $\mathsf{R}(T_j, (S, P'))$, $\mathsf{W}(T_j, (S, P'))$ or $\mathsf{Rd}(T_j, (S, P')$ is true. $\mathsf{R}(T_j, (S, P'))$ or $\mathsf{W}(T_j, (S, P'))$ are contradictions, as the true-dependence constraint would disallow fusion. $\mathsf{Rd}(T_j, (S, P'))$ is a contradiction due to the reduction constraint. Similar logic can be applied to other dependence cases. Here, we show that our algorithm is sound by identifying cases where fusion is possible—we do not claim completeness by proving the converse.

We have shown that all dependencies between index tasks are at most point-wise, so any T_j^p can only depend on T_i^p, where $i < j$. Since the fused task body is the composition of each task in $[T_1, \ldots, T_f]$ in program order, all dependencies in $[T_1, \ldots, T_f]$ are preserved.

4.2.4 Discussion

Performing fusion at the Legate layer between the application and Legion is key for a domain-agnostic analysis, and for analysis scalability as the size of the machine increases. We compare against fusion on high-level domain-specific libraries, and against fusion within lower-level runtime systems like Legion.

Domain-specific algorithms for fusion [2,20,51,55,56] are effective optimizations for individual distributed libraries. Approaches that perform fusion on a set of domain-specific computations use algorithms and analyses that are tied to the domain of computations being optimized, especially analyses related to distributed memory. As a result, these techniques do not readily generalize across libraries. Legate targets fusion in the more general case after computations have been decomposed into tasks

in a domain-specific manner, enabling domain-agnostic analyses to find optimizations across function and library boundaries. We expect that domain-specific techniques may be used in conjunction with the Legate's analysis.

While generality is lost when fusing operations within individual libraries, scalability becomes a concern when analyzing lower-level program representations. A key design decision in the Legate middle-end representation is its use of symbolic scale, as the representation of parallel task groups and partitions of distributed data are independent of the degree of parallelism. This design enables Legate to symbolically compute a conservative estimate of the aliasing relationships between distributed data structures through constant-time queries, which are heavily used when defining the fusion constraints in Figure 4.4. In contrast, Legion's lower-level representation of partitions that explicitly maps points to sets of indices scales with the number of pieces the data is partitioned into. These representations are more flexible than the Legate middle-end, but result in the aliasing relationship queries needed by a fusion algorithm to scale with the degree of available parallelism.

4.3 Task Fusion Optimizations

Having described our algorithm for task fusion, we now describe optimizations necessary for a practical implementation. We show how to eliminate temporary distributed data structures (Section 4.3.1) and how to memoize the fusion analysis (Section 4.3.2). Temporary elimination and memoization are widely applied optimizations; we discuss how to perform these optimizations in a distributed, task-based setting.

4.3.1 Temporary Store Elimination

Once Legate identifies a fusible prefix of tasks, stores that fusion has made temporary may be promoted into task-local data. Conversion of distributed data into task-local data is critical for realizing the benefits of fusion, as task-local data can then be optimized away (Section 4.4) to maximize reuse.

```
1 import cupynumeric as np
2 x, y = np.zeros(n), np.ones(n)
3 flush_window()
4 z = 2.0 * x
5 w = y + z
6 v = w ** 2
7 norm = np.linalg.norm(w[len(w)//2:])
8 del x, y, z, w
9 flush_window()
```

(a) cuPyNumeric code fragment.

```
1 # Partitions and launch
2 # domains excluded.
3 ---
4 MULT([(x, R), (z, W)])
5 ADD([(y, R), (z, R, (w, W)])
6 POW([(w, R), (v, W)])
7 ---
8 NORM([(w[len(w)//2:], R), (norm, Rd)])
```

(b) Emitted task stream.

Figure 4.5: Example of distributed temporaries.

To introduce when a store is temporary, consider the cuPyNumeric program in Figure 4.5a and the resulting task stream in Figure 4.5b. This example introduces some new operations, specifically `flush_window`, which sends all pending tasks through Legate's analysis into Legion, and the Python `del` operator, which drops references. The program creates the stores `x`, `y`, `z`, `w`, and `v`. Consider the program state after line 9: the tasks that initialize `x` and `y` have executed, as the first `flush_window` call sent those tasks through Legate. We note that there are no pending tasks outside the window, and future tasks are ones the application may launch once the call to `flush_window` returns. The fusion algorithm determines that the tasks issued by lines 4–6 can be fused, while the final `norm` must be excluded. First, `v` is not temporary because the application holds a reference to it, meaning that it could launch a task that reads `v` after the call to `flush_window()`. Next, while the application has deleted its reference to `w`, the `norm` task reads a piece of `w` and is still pending after the fused task, and thus must observe any effects performed on `w`, meaning that `w` is not temporary. The stores `x` and `y` are only read by the fused task, and thus are not temporary. Only `z` is temporary because it is produced entirely within the fused task and is not visible to the application or pending tasks. We formalize this intuition as constraints that must be satisfied for a store to be temporary.

Definition 4. Given tasks $[T_1, \ldots, T_f, \ldots, T_n]$, a store S is *temporary* in the fusion of $[T_1, \ldots, T_f]$ if

1. If $\exists\, T_j, P$ s.t. $\mathsf{R}(T_j, (S, P))$, $\exists\, T_i$ such that $i < j \wedge \mathsf{W}(T_i, (S, P)) \wedge \mathsf{covers}(S, P)$
2. $\nexists\, T_k, P$ s.t. $k > f \wedge \mathsf{R}(T_k, (S, P)) \vee \mathsf{Rd}(T_k, (S, P))$

```
1 T1([(S1,Read),(S2,Write)])
2 T2([(S2,Read),(S1,Write)])
3 T3([(S1,Read),(S3,Write)])
4 T4([(S3,Read),(S1,Write)])
```

```
1 T1([(S5,Read),(S6,Write)])
2 T2([(S6,Read),(S5,Write)])
3 T3([(S5,Read),(S7,Write)])
4 T4([(S7,Read),(S5,Write)])
```

```
1 T1([(S5,Read),(S6,Write)])
2 T2([(S6,Read),(S5,Write)])
3 T3([(S7,Read),(S7,Write)])
4 T4([(S7,Read),(S5,Write)])
```

(a) Two isomorphic task streams and one differing task stream.

```
1 T1([(0,Read), (1,Write)])
2 T2([(1,Read), (0,Write)])
3 T3([(1,Read), (2,Write)])
4 T4([(2,Read), (0,Write)])
```

```
1 T1([(0,R), (1,W)])
2 T2([(1,R), (0,W)])
3 T3([(2,R), (2,W)])
4 T4([(2,R), (0,W)])
```

(b) Canonical representations of isomorphic and differing streams.

Figure 4.6: Example of task stream memoization.

3. S has no live application references.

The function $\mathsf{covers}(S, P)$ is true when the partition P contains all points in the store S. The first two constraints check that the store's contents are entirely created within the fused task and not used by any other existing task; these conditions are checked through a forwards dataflow analysis of the task stream. The third constraint ensures that the application can no longer view any effects on a store, checked through a split reference counting scheme on stores in the Legate runtime. The split reference counting scheme separates references held by the application from references held by the Legate runtime. Temporary stores are demoted from a distributed allocation into a task-local allocation, as described in Section 4.4.

4.3.2 Memoization of Fusion Analysis

The final component of our distributed task fusion pipeline is memoization analysis and code generation (Section 4.4). The key challenge in memoization is allowing for the analyses to be replayed on *isomorphic* task streams rather than identical task streams. Consider the streams of tasks in Figure 4.6a, where partitions and launch domains are excluded.

Legate may reuse the analysis results from the left stream in Figure 4.6a on the middle stream, as the pattern of stores among tasks is isomorphic. In contrast, the right task stream in Figure 4.6a has a different pattern of stores across tasks,

particularly the use of `S7` in `T3`. We observe that this problem is identical to *alpha-equivalence*, where each store argument is a bound variable. We identify when two task streams are isomorphic within Legate through a conversion to and comparison on a canonical, De-Brujin index-like representation. This representation is shown in Figure 4.6b. A similar technique has previously been used to avoid enumerating instruction sequences equivalent up to register renaming [8].

In the automatic tracing section, need to carefully articulate why a similar analysis does not work in that setting, and why other machinery is needed.

- here, fusion constraints demarcate fused tasks that need to be memoized. We don't have to identify when a fused task might be seen again, or find a good amount of tasks to fuse when that isn't specified.
- Isomorphic trace replay is quite difficult to do in Legion due to the lack of mapping from physical instances in the trace to interactions of physical data from outside the trace; the trace isomorphism talks about what happens during the trace but not about what happens outside, making it difficult. Isomorphic trace replay might be a point solution for Legate (if it could work), but the automatic tracing system is more general.

4.4 Kernel Fusion

The final component of fusion in the Legate middle-end is a compilation stack to optimized fused tasks. A high-level program representation is required to both perform optimizations like loop fusion and to lower to different backends like GPUs and multi-threaded CPUs. We leverage the MLIR compiler stack, which is extensible and is pre-packaged with many common compiler analyses. We first provide background on MLIR, and then describe the code generation process and optimizations performed. We then discuss how the Legate architecture enables the separation of reasoning about distributed programs from the optimization of nested loops.

```
class Task {
  // Standard implementation.
  void gpu_variant(
    vector<pair<Store, Priv>> args,
  );
  // Opt-in MLIR task body generator.
  mlir::func::FuncOp generator(
    vector<pair<StoreDesc, Priv>> args,
  );
};
```

(a) API to define a Legate task amenable to fusion.

```
func.func @kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %c: memref<?xf64>) {
  %dim = memref.dim %c, 0
  affine.for %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %c[%i]
  }
}
```

(b) MLIR generated for an element-wise addition.

```
func.func @fused_kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %d: memref<?xf64>,
  %e: memref<?xf64>) {
  %dim = memref.dim %e, 0
  %c = memref.alloc %dim
  affine.for %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %c[%i]
  }
  affine.for %i = 0 to %dim {
    %0 = affine.load %c[%i]
    %1 = affine.load %d[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %e[%i]
  }
}
```

(d) After temporary elimination.

```
func.func @fused_kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %c: memref<?xf64>,
  %d: memref<?xf64>,
  %e: memref<?xf64>) {
  %dim = memref.dim %e, 0
  affine.for %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %c[%i]
  }
  affine.for %i = 0 to %dim {
    %0 = affine.load %c[%i]
    %1 = affine.load %d[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %e[%i]
  }
}
```

(c) Initial body of fused task.

```
func.func @fused_kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %d: memref<?xf64>,
  %e: memref<?xf64>) {
  %dim = memref.dim %e, 0
  affine.par %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    %3 = affine.load %d[%i]
    %4 = arith.addf %2, %3
    affine.store %2, %e[%i]
  }
}
```

(e) Fully optimized fused task.

Figure 4.7: Fused MLIR kernel for three way element-wise addition traversing the compilation pipeline. The initial kernel is created by sequentially composing two of the generated task bodies in Figure 4.7b.

4.4.1 MLIR Background

We leverage MLIR [33] to build a JIT compiler for Legate. MLIR is an extension of LLVM [32] that provides compiler infrastructure for program analyses on higher-level languages than three-address code. The most relevant component of this infrastructure to our work is the notion of a *dialect*, which is an intermediate representation that has user-defined semantics. A key aspect of dialects in MLIR is that a single MLIR program can contain types and operations from multiple dialects, enabling the composition of dialects with different semantics. Compilers built using the MLIR framework run passes over programs that either optimize the operations within a single dialect, or convert between dialects to perform progressive lowering. Legate leverages community-developed dialects and passes to optimize and lower task bodies into CPU and GPU code.

4.4.2 Generator Functions

Legate library developers implement tasks by providing task bodies that run on processors like CPUs or GPUs. To opt into Legate's kernel fusion, developers register a developers register a *generator* function for the implementation of a task that produces an MLIR fragment describing the task's computation. We found the integration effort of adding these generator functions to be modest, requiring 50–100 lines of C++ code per operation. Only library developers, not end users of Legate, must develop MLIR kernels for tasks. Additionally, the integration effort was incremental—as more tasks were implemented with MLIR generators, Legate could exploit more kernel fusion. An example generated MLIR fragment by cuPyNumeric for an element-wise addition operation is shown in Figure 4.7b.

The generated MLIR fragment in Figure 4.7b contains multiple dialects: 1) stores are mapped onto the `memref` dialect, which provides stronger aliasing guarantees than raw pointers; 2) dense iteration is mapped onto the `affine` dialect, a target for polyhedral compilation [18]; and 3) the computation itself is mapped onto the `arith` dialect, containing arithmetic operations. Using MLIR, other dialects can be used to express higher level operations, like dense and sparse tensor algebra with the `linalg`

and `sparse_tensor` dialects.

4.4.3 Compilation Pipeline

When Legate identifies that a sequence of tasks may be fused, it invokes each task generator and constructs an MLIR module containing the body of each task in the original program order. Figure 4.7c shows a fused task for the cuPyNumeric computation `c = a + b; e = c + d`, where all variables represent distributed vectors. This program originally has two index tasks (one for each add operation) which are fused into a single index task where the original task bodies (the MLIR in Figure 4.7b) appear sequentially in the fused task. Before optimization of the task body, Legate first promotes distributed data into task-local allocations, resulting in Figure 4.7d.

After elimination of temporary stores, we apply passes that fuse and parallelize nested loops, and remove task-local temporary allocations to yield the optimized code in Figure 4.7e. The generated kernel is the ideal implementation for the original program: the separate loops of the original task bodies have been combined into a single pass over the vectors, and the temporary `c` has been eliminated. The optimized kernel is then lowered to GPU kernel launches or OpenMP parallel regions.

In this work, we leverage polyhedral optimizations [18, 19] to perform fusion and parallelization of loops in kernels. However, with higher level dialects in MLIR, various domain-specific kernel fusion techniques (see **??**) could be leveraged within a fused task body. We consider the exact kernel fusion techniques used to be orthogonal to our work. rohany: Somewhere, either here or in the future work, describe the ecosystem of MLIR dialects that a system like Legate could host to enable plug-and-play composition across different libraries without even knowing about it, as well as the potential phase ordering problems that may come from such an approach.

4.4.4 Qualitative Benefits

We note several qualitative benefits of Legate's architecture in contrast to approaches that attempt to optimize distributed programs entirely through analysis of imperative code. A key design decision of Legate is to leverage a distributed data model

in a dynamic representation of computation with symbolic scale to enable cheap dependence analysis between distributed computations. Separating out the reasoning about distributed computation avoids intertwining loop optimizations with distributed communication analyses, allowing the loop optimizations to remain unaware of the distributed context. This separation also allows for information gained during the distributed analysis phase to be used in code generation: properties such as array non-aliasing are provided to the MLIR optimization passes to generate better code. Finally, the separation of distributed computation into tasks means that Legate does not need to identify optimizable fragments of static source code from the larger application source. Through the process of building a Legate library, developers extract the key computational kernels from code that does not need to be considered for analysis and optimization.

4.5 Experimental Results

We now evaluate the performance that the fusion of computation enabled by the Legate middle-end representation and analysis over the initial performance achieved by the data composition techniques in the rest of Legate. We performed these experiments on the (now decomissioned) NVIDIA Selene supercomputer, which is a cluster of NVIDIA A100 DGX SuperPOD nodes. Each Selene node has 8 A100 GPUs with 80GB of memory, connected by NVLink and NVSwitch connections, and a dual socket, 128 core AMD 7742 Rome CPU with 2TB of memory. Each node is connected via an InfiniBand connection through 8 NICs.

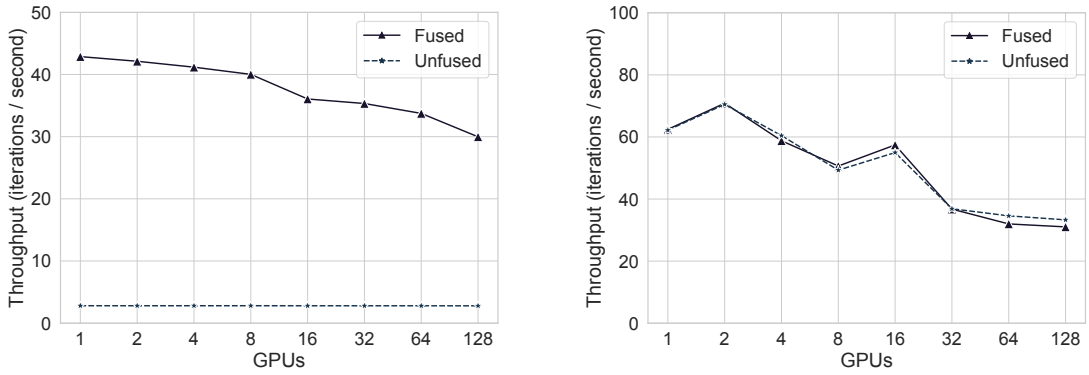
For each experiment, we perform a weak-scaling study, and report the throughput achieved per processor. Each reported value is the result of performing 12 runs, dropping the fastest and slowest runs, and then computing the average of the remaining 10 runs. In the weak-scaling experiments (Section 4.5.1), we exclude a set of warmup iterations from timing to measure the steady-state performance with and without fusion by Legate. We separately evaluate the overhead that Legate's fusion imposes due to compilation in **??**.

Benchmark	Tasks per Iteration	Tasks per Iteration (Fused)	Avg Task Length (ms)	Window Size
Black-Scholes	67	1	5.3	70
Jacobi	3	2	5.3	5
CG	12.1	4.1	1.9	10
BiCGSTAB	27.1	8.1	1.7	15
GMG	24.1	11.1	1.8	15
CFD	378	40.7	1.1	30
TorchSWE	276.5	152.8	1.4	30

Figure 4.8: Tasks per iteration with and without fusion. Task count is not always whole as iterations may launch different tasks, or fusion occurs across iteration boundaries. Reported task granularities are from unfused single-GPU executions. Window size was selected by Legate.

Overview. We evaluate fusion in Legate on unmodified cuPyNumeric and Legate Sparse applications that range from tens to thousands of lines of Python. An overview of each application is in Figure 4.8. We compare each application's performance when run with and without fusion — no changes to the application are necessary. For some applications, a suitable baseline written in the industry-standard PETSc [4] library for distributed sparse linear algebra already exists, and we compare against those baselines. For other applications, we compare against manually optimized implementations by the original authors. However, some full cuPyNumeric applications have no baseline other than when run without name—these applications are sufficiently complex that developing an independent high-performance distributed, multi-GPU implementation is not feasible. We show that when fusion opportunities are available, Legate can exploit them to find speedups in unmodified, distributed applications. Legate enables high-level programs to equal, and in many cases improve on, the performance of hand-optimized code.

We do not ablate on the optimizations in Section 4.3, as temporary elimination is essential for speedup with kernel fusion and memoization is a requirement for a practical implementation. The window sizes shown in **??** were selected automatically by the Legate runtime through a process that increases the window size when all tasks in the current window size were fused. As a result, these window sizes enable the maximum amount of fusion possible in each application.



(a) Black-Scholes (b) Jacobi Iteration

Figure 4.9: Microbenchmark weak scaling (higher is better).

4.5.1 Weak Scaling Experiments

Black-Scholes. The Black-Scholes option pricing benchmark is a trivially-parallel micro-benchmark that contains a sequence of 67 data-parallel, and thus fusible, operations. It is a micro-benchmark that provides a reference point on potential improvement when the entire application is amenable to fusion. **??** shows that Legate achieves a 10.7x speedup over the unfused program on 128 GPUs, as the fused program is a single task containing a single GPU kernel making one pass over the data, greatly increasing the arithmetic intensity of the computation.

Dense Jacobi Iteration. Unlike Black-Scholes, dense Jacobi iteration has negligible potential benefit from fusion. Jacobi iteration consists of a dense matrix-vector multiplication and two fusible vector operations that are negligible in runtime. This benchmark shows that our analyses do not have a significant negative impact on performance when there is no fusion. Legate achieves 0.93–1.08x of the performance of the unfused Jacobi iteration in **??**, where we believe the slight improvement is due to experimental variability.

Sparse Krylov Solvers. We evaluate sparse Krylov solvers implemented with cuPyNumeric and Legate Sparse, namely Conjugate Gradient (CG) and Bi-Conjugate Gradient Stabilized (BiCGSTAB). The PETSc benchmark implementations are implemented in MPI+C using PETSc's API. To perform a controlled comparison against PETSc, we modify Legate Sparse to perform a similar optimization as PETSc, where

the non-zero coordinates in each sparse matrix partition are stored as 32-bit integers instead of 64-bit integers.[1] We note that this restriction has been lifted in current versions of PETSc, but was not fixed at the time these experiments were collected.

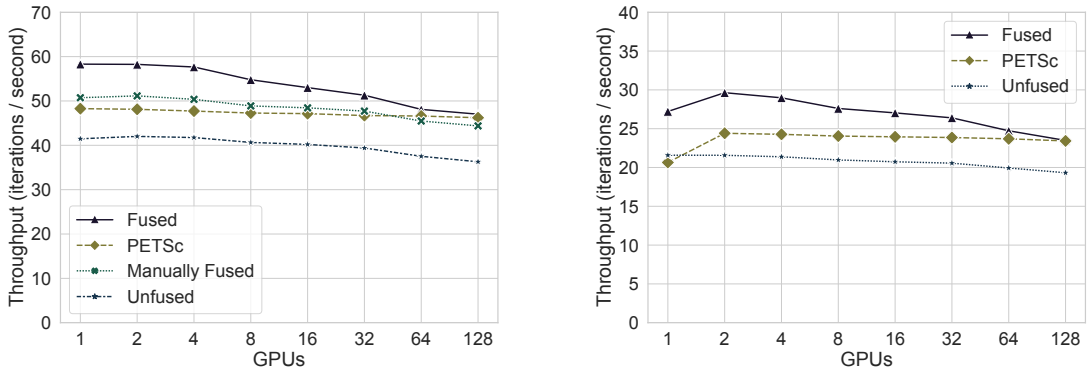
The original implementation of CG in Legate Sparse (as presented in Chapter 3) had been optimized manually to perform many of the optimizations that Legate does automatically with fusion. As a result, the implementation no longer resembled the high-level description of CG. We compare against this manually fused implementation, a naturally written implementation using cuPyNumeric and Legate Sparse, and PETSc. Figure 4.10a shows that Legate automatically optimizes the naturally written CG so that it runs faster than both the manually optimized version and PETSc. Legate finds additional fusion opportunities by fusing AXPY's and dot-products from different iterations.

We implement an unfused version of BiCGSTAB in cuPyNumeric and Legate Sparse and compare against PETSc. Figure 4.10b shows that Legate accelerates the high-level implementation of BiCGSTAB to outperform the unfused version by 1.31x on average (geo-mean) and PETSc by 1.15x on average (geo-mean). PETSc exposes several fused kernels to users for use in building iterative solvers, but these kernels can quickly become complicated and esoteric[2]. In contrast, Legate enables users to write high-level programs in cuPyNumeric and Legate Sparse and then derives optimized kernels for efficient execution.

Geometric Multi-Grid Solver (GMG). Moving from smaller benchmarks to full applications, we apply Legate's fusion to a Geometric Multi-Grid (GMG) solver developed in Legate Sparse. The GMG solver is a CG-based iterative solver with a V-cycling preconditioner, the injection restriction operator, and a weighted Jacobi smoother. As with the previous benchmarks, using Legate with the more complex solver required no changes to user-facing code, and results in a 1.2x speedup over the original implementation, as seen in Figure 4.11a. Additionally, unrelated performance challenges within the Legion runtime system were resolved since the original

[1]PETSc stores coordinates in 32-bit integers even when 64-bit integers are requested at build time, affecting the performance of the SpMV kernel.

[2]Such as `VecAXPBYPCZ` in BiCGSTAB (`https://petsc.org/main/manualpages/Vec/VecAXPBYPCZ/`).



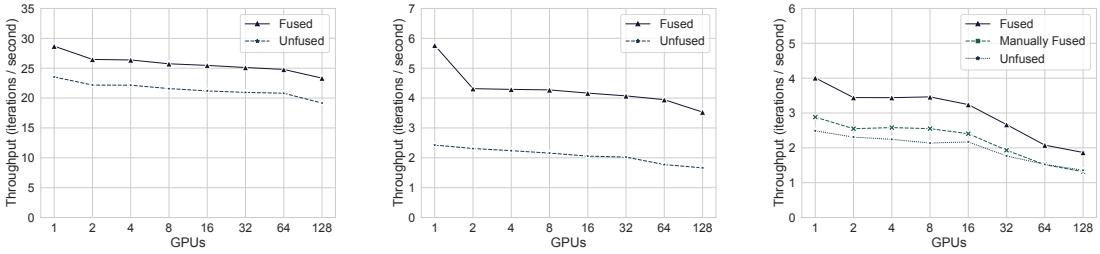
(a) CG (b) BiCGSTAB

Figure 4.10: Weak scaling of linear solvers (higher is better).

development of Legate Sparse, yielding better weak scaling than originally presented in Chapter 3.

Computational Fluid Dynamics (CFD). CFD is a cuPyNumeric application that solves the Navier-Stokes equations for 2D channel flow [9]. The application performs element-wise operations on aliasing slices of distributed arrays, exposing opportunities for fusion. Legate finds between 1.8x–2.3x speedup over the original implementation, as shown in Figure 4.11b. Legate achieves higher speedup on a single GPU than on multiple GPUs. On a single GPU, data is not partitioned, enabling longer sequences of tasks to satisfy fusion constraints. On multiple GPUs, the dependencies caused by aliasing data reduce the opportunities for fusion.

Shallow Water Equation Solver (TorchSWE). Our final benchmark application is also our most complex: the cuPyNumeric port of the TorchSWE shallow-water equation solver [21]. We compare against the original cuPyNumeric port, as well as a version that the cuPyNumeric developers manually optimized using `numpy.vectorize`. The `vectorize` utility JIT-compiles a user-defined element-wise operator, doing some of the optimizations that the Legate middle-end performs automatically. **??** shows the performance of TorchSWE with Legate compared to these baselines. Legate achieves a 1.61x speedup on average (geo-mean) over the unfused TorchSWE, and a 1.35x speedup on average (geo-mean) over the manually vectorized version (labeled with "Manually Fused" in **??**). Since Legate analyzes the entire application, it finds fusion



(a) GMG (b) CFD (c) TorchSWE

Figure 4.11: Weak scaling of full applications (higher is better).

Benchmark	Standard (s)	Compiled (s)	Breakeven Iterations
Black-Scholes	0.38	0.06	N/A
Jacobi	0.53	0.43	N/A
CG	0.67	1.30	99.44
BiCGSTAB	1.26	2.19	80.43
GMG	0.49	1.38	118.75
CFD	5.10	10.89	25.21
TorchSWE	0.97	8.82	43.88

Figure 4.12: Warmup times on 8 GPUs.

opportunities missed by developers optimizing the program by hand.

4.5.2 Compilation Time

We measure the overhead that the Legate middle-end's compilation imposes on the overall application runtime. When evaluating our benchmarks, we compute the throughput after warmup iterations have concluded. To measure the effect of compilation, we measure the warmup time with and without compilation, using the window sizes reported in **??**. We then compute the number of iterations required for the fused version to be faster than the unfused version of the application when including the warmup compilation time. The results are shown in Figure 4.12; Legate's compilation times are modest, requiring 25–119 iterations to amortize the cost of compilation. The fused Black-Scholes computation is so much faster than the unfused version that a single iteration is sufficient to amortize compilation. For Jacobi, compilation time was overlapped with expensive dense matrix-vector multiply kernel, and thus not exposed

in the warmup. As seen in Figure 4.9b, due to experimental variation, the fused and unfused versions of Jacobi are slightly faster or slower than each other on different GPU counts. These costs are especially reasonable as scientific applications like the ones we evaluated would be run in production for thousands to millions of iterations. In the future, a production-grade implementation of task and kernel fusion in Legate could maintain a cache of compiled kernels on disk, rather than in memory, and pay the compilation cost only the first time the application is run.

4.5.3 Conclusion

Some sort of synthesis of this work and connection to the other pieces of the thesis.

Chapter 5

Controlling Overheads (Dependence Analysis)

The previous chapters of this thesis focused on the intermediate layers within the Legate runtime system that enable the composition of distributed data and computation across library boundaries. These layers, along with the Legion and Realm beneath Legate, perform a variety of dynamic analyses to co-partition stores, fuse computations, perform dependence analysis, and schedule tasks from the source program. While these dynamic analyses are critical for the composed programs to achieve high performance, the cost of performing these dynamic analyses can impact the end-to-end performance of applications, especially as the task granularity of applications decreases. Controlling the impact of these overheads, especially in programs composed from independent pieces, is important to yield end-to-end performance at a variety of target problem sizes. This chapter and Chapter 6 discuss two key techniques in the Legate runtime system to control these overheads. This chapter focuses on amortizing dynamic analysis overheads caused by parallelism extraction and communication insertion in the Legion runtime system automatically, without user input. Legion's analysis is significantly more heavyweight than the corresponding analysis in Legate, and also can scale with the size of the target machine, which is why we focus specifically on analysis amortization within Legion. Then, Chapter 6 focuses on the bottom layer of the Legate runtime stack, tackling overheads at the Realm layer.

5.1 Background and Motivation

rohany: Not sure about the exact name of this section, but going to start to blend in parts of the paper here.

rohany: citations for the analyses used by Legion in parallelism extraction.

As discussed in Chapter 2 and Chapter 3, Legion exposes an implicitly-parallel task-based programming model. Programs are expressed as a sequential stream of tasks, where tasks describe which stores they access and how the stores will be accessed (with privileges). Legion then performs a *dependence analysis* to identify which tasks depend on each other, extract parallelism from the input sequential stream of tasks, and insert communication between tasks to ensure that tasks observe data consistent with the sequential program interpretation. Chapter 3 describes how this dependence analysis is a critical component for the efficient composition of independent Legate libraries.

Legion's dependence analysis, while extremely powerful, is the most heavyweight dynamic analysis performed in the entire Legate runtime stack. This analysis costs roughly 1 millisecond per task in the source program; as tasks drop below this threshold, which is becoming more common with higher performance GPUs, the analysis is unable to be amortized and dominates the performance of the application. To improve the performance of Legion's analysis, Lee et al. [35] introduced *dynamic tracing*, a memoization framework for Legion's dependence analysis. Tracing records the results of the dependence analysis for an issued sequence of tasks, and then replays the results of the analysis the next time an identical sequence of tasks is issued. Tracing has been shown to yield significant speedups by eliminating the cost of the dependence analysis on iterative programs, and reducing the overheads imposed by Legion to roughly 100 microseconds per task.

A significant limitation of tracing currently is that it requires the programmer to annotate repeatedly issued program fragments with stop/start markers for the runtime system. Explicit stop/start markers derail the correctness and performance of Legate programs (and directly in Legion) under composition. However, programmer introduced trace annotations do not correctly compose under composition, the correct

placement of trace annotations when composing complex software becomes unclear. Functions defined in independent Legate libraries may contain operations that cannot by traced by a practical tracing implementation, or may issue a different sequence of tasks on each invocation. Each of these cases result in errors, due to the incorrect trace annotations constructing an ill-formed sequence of operations. Furthermore, even simple programs constructed from an individual Legate library can have traces that do not correspond to syntactic loop structures in the source program, making it difficult to correctly place tracing annotations. We elaborate on such an example program later in Section 5.1.1.

To both improve programmer productivity and to enable the tracing of high-level programs composed from independent Legate libraries, we argue that implicitly-parallel task-based systems like Legion should automatically identify repeated sequences of tasks, memoize their dependence analysis results, and cheaply replay the analysis as needed. We call this the problem of *automatic tracing*, which is similar to Just-In-Time (JIT) compilation in the context of dynamic language implementations [25, 28, 38]. JIT compilers for dynamic languages interpret bytecode during program startup, and compile bytecode to native instructions as repeatedly invoked program fragments become hot. Following this architecture, Legion should interpret issued operations with a dynamic dependence analysis, and switch to an analysis-free compiled execution once repeated sequences of operations are encountered.

The key challenge of automatic tracing is the *identification* of repeated sequences of tasks produced by the source program. Unlike JIT compilers, the input to Legion is a stream of tasks that lacks information about control flow such as basic block labels or function definitions. As such, Legion cannot rely on these code landmarks or predictable execution flow to identify repeated sequences of operations.

5.1.1 Motivating Example for Automatic Tracing

We now describe how high-level, implicitly-parallel Legate programs can be difficult to add manual tracing annotations to, due to layers of abstraction over the concrete stream of tasks. These challenges arise as a result of the same properties that enable

```python
import cupynumeric as np
# Generate random system.
A = np.random.rand(N,N)
b = np.random.rand(N)
# Initialize solution and
# extract diagonal.
x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
# Jacobi iteration.
for i in range(iters):
  x = (b - np.dot(R, x)) / d
```

(a) Python source code.

```
DOT(R, x1, t1)
SUB(b, t1, t2)
DIV(t2, d, x2) # Iteration 1
DOT(R, x2, t1)
SUB(b, t1, t2)
DIV(t2, d, x1) # Iteration 2
DOT(R, x1, t1)
SUB(b, t1, t2)
DIV(t2, d, x2) # Iteration 3
DOT(R, x2, t1)
SUB(b, t1, t2)
DIV(t2, d, x1) # Iteration 4
```

(b) Main loop task stream.

Figure 5.1: Example cuPyNumeric program and the stream of tasks issued to the Legate (and then Legion) runtime. An intuitive trace around the main loop does not correspond to a repeated program fragment.

Legate programs to compose in the first place: the Legate runtime reduces operations from independent libraries into a single coherent task stream that can be analyzed and optimized. However, once the program has entered this flattened representation, recovering structure around looping patterns is a challenge.

Figure 5.1a is a small cuPyNumeric that performs Jacobi iteration. The corresponding stream of tasks issued by the main loop is shown in Figure 5.1a. As before, each cuPyNumeric array maps directly to a Legate store. For each task, we elide privilege annotations on store arguments, and let the first two arguments denote the source-level inputs, while the third argument is the source-level output.

To trace a program, the programmer must issue a `tbegin(id)` call (standing for "trace begin") before and a `tend(id)` call after the fragment. The first time Legion executes a trace with a particular `id`, it records the results of the dependence analysis, and then replays the results when executing the same trace `id` again [35]. For a trace to be valid, the sequence of tasks and their store arguments encapsulated by `tbegin(id)` and `tend(id)` calls must be exactly the same for a given `id`. The same store arguments must be used across trace invocations as the dependence analysis is affected by the usages of the stores and how they are partitioned.

A natural attempt to trace the program in Figure 5.1a would place the `tbegin` and `tend` around the body of the main `for` loop. However, this annotation results in an invalid trace, due to an implementation detail within cuPyNumeric and the

Legate runtime. The problem with this natural annotation is the loop-carried use of the Python variable `x`, which is bound to different cuPyNumeric arrays (stores) at different points of execution. Upon entering loop iteration i, `x` is bound to a store arbitrarily named `x1`, which is used as an argument for the first `dot` operation. As execution proceeds, cuPyNumeric allocates a new store `x2` for the result of the division with `d`, and binds the Python variable `x` to the region `x2`. Therefore, the next iteration $i+1$ issues a `dot` on `x2`, causing iteration $i+1$ to issue a different sequence of tasks than iteration i! Issuing a different sequence of tasks with the same trace `id` is a violation of the conditions to use tracing, and Legion may either choose to raise an error or fall back to the expensive dependence analysis.

Rohany: edit this paragraph when splicing it in. To correctly trace the program in Figure 5.1a, a programmer must either add trace annotations around every two iterations of the main loop, or use two different trace ID's for each different iteration's repetition pattern. This steady state of groups of two iterations is achieved because when `x` is assigned, the region it refers to can be collected and immediately reused by cuPyNumeric. Relying on this steady state is brittle, as the addition of more operations in the main loop or a change in cuPyNumeric's region allocation policy could perturb the way in which the necessary steady state for tracing is achieved. Instead, we argue that a dynamic analysis should analyze the application stream of tasks and automatically discover what fragments of the application should be traced, removing this concern from the programmer.

5.2 What Are Good Traces?

By identifying traces automatically, we aim to reduce the amount of time Legion spends performing dynamic dependence analysis. To maximally reduce this dynamic dependence analysis cost, we must define what are the properties of traces that Legion should automatically identify.

A simple model of a Legion's dependence analysis is that Legion spends time α analyzing each task. The first time a trace is issued, the dependence analysis results are memoized, so Legion spends time α_m (memoization time) on each task in the

trace, where α_m is slightly larger than α. Then, on subsequent executions of the trace, there is some constant c amount of overhead to replaying the trace, but every task in the trace only incurs an analysis cost of α_r (replaying time), where $\alpha_r \ll \alpha$.

Using this model of the Legion, we derive several properties of traces that we aim to automatically discover. First, the selected traces should maximize the number of traced operations to minimize the number of tasks that contribute an α to the overall analysis cost. Next, the selected traces should be relatively long so that the constant replay cost c does not accumulate. Finally, the set of selected traces should be small, so that Legion does not continually memoize new traces and pay α_m per task in each new trace. Intuitively, the ideal set of traces corresponds to the loops in the target program.

We now concretize the good traces that Legion should find as the solutions of a concrete optimization problem. Consider the sequence of tasks S constructed from a complete execution of the target program. A system for automatic trace identification must construct from S

- A set of traces T, containing sub-strings of S,
- A function $f : T \to \mathsf{interval\ set}$, mapping each $t \in T$ to a set of intervals in S that are *matched* by t,

that maximizes the *coverage* of f, defined by $\mathsf{coverage}(T, f) = \sum_{t \in T} \sum_{i \in f(t)} |i|$, subject to the constraints

1. $\forall t \in T$, t is longer than a minimum length,
2. $\bigcup_{t \in T} f(t)$ is a disjoint set of intervals.

Multiple solutions exist for this problem, so we prefer solutions that first maximize the number of matched intervals ($\sum_{t \in T} |f(t)|$), and then minimize the total number of selected traces ($|T|$). Maximizing $\mathsf{coverage}(T, f)$ directly minimizes the number of untraced tasks, and selecting a small set of traces that repeats many times minimizes the memoization cost of α_m per task. Finally, a minimum length is placed on traces to ensure that the constant replay cost c can be effectively amortized. We present a concrete problem instance and example solutions in Figure 5.2.

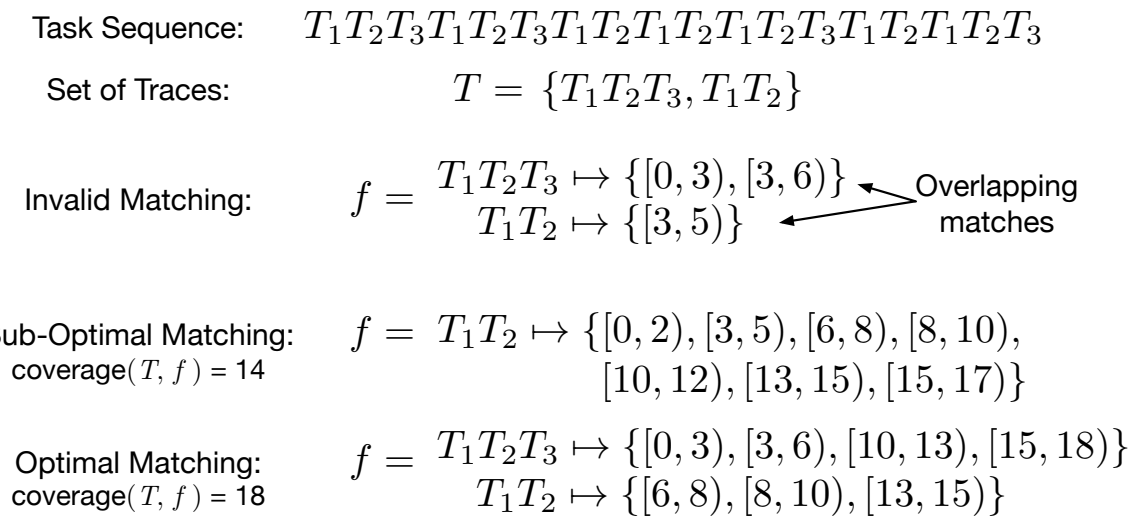


Figure 5.2: Example of a task stream and fixed trace set T with an invalid matching function f, and two matching functions with different coverage(T, f).

The presented optimization problem precisely defines the properties of traces that Legion should attempt to find, but it does not directly yield an algorithm to discover good solutions. Additionally, the optimization problem is structured in a post-hoc formulation, where an optimal solution is constructed from the results of the entire program execution. In practice, Legion must construct the solution (T, f) in an online manner, using the currently visible prefix of the sequence of tasks launched by the application. In the next section, we discuss algorithms for dynamically finding good solutions to this optimization problem through a set of string processing algorithms.

5.3 Trace Identification

Dynamically finding good traces requires processing information about the tasks seen so far, and then using that information to record and replay traces in the future. An overview of Legion's dynamic analysis procedure for automatic tracing is sketched in Algorithm 1. Legion has two components that correspond to the targets of the optimization problem in **??**. The *trace finder* constructs the candidate set of traces T by accumulating the tasks issued by the application into a buffer, and asynchronously mining the buffer to find candidate traces. The *trace replayer* then constructs the

matching function f by ingesting the candidate traces into a trie, and identifying candidate traces in the application stream by maintaining pointers into the trie that represent potential matches. The automatic tracing analysis intercepts calls to the application-exposed `ExecuteTask` function, and forwards a potentially different set of tasks and trace markers to the runtime. A concrete example of how Legion identifies a trace in an application is shown in Figure 5.3. We now describe each of these components in detail.

5.3.1 A Stream of Tokens

An insight is that automatic trace identification is inherently an online string analysis problem of finding repeated sub-sequences in the application's task stream. As seen in Figure 5.1b, the task stream is not just a list of identifiers—tasks have store arguments that must also be the same across iterations to be used in a trace. To capture all aspects of a task that can affect the dependence analysis, Legion constructs a hash from each task and its store arguments. Converting the input stream of tasks into a stream of hash tokens enables more direct application of string processing techniques, and straightforward handling of traceable operations that are not tasks, such as copies and fills.

5.3.2 Finding Traces With High Coverage

Legion's trace finder records tasks as they are issued by the application into a buffer (we describe a refinement to this scheme in Section 5.3.4). Once the buffer fills up, Legion launches an asynchronous analysis of the buffer to find a set of traces within the buffer that maximize the coverage of the buffer. We discuss previous ideas that are related to this goal, and then describe the solution used in Legion (We discuss more related work in **??**; I'm not sure exactly where this discussion of other related work should go..)

Algorithm 1: Legion's Dynamic Analysis for Automatic Tracing.

```
   /* Initialize token history buffer B and pending async analyses J.  */
1  B, J ← [], []
   /* Initialize trie of candidates C, potential current traces A, and
      pending tasks P.  */
2  C, A, P ← Trie(), [], []
   /* Discussed in Section 5.3.2.  */
3  TraceFinder (H)
4      B ← B + [H]
5      if ShouldAnalyzeHistory(B) then
           /* What subset of the history to analyze is discussed in
              Section 5.3.4.  */
6          B′ ← GetAnalysisSubset(B)
           /* Find repeated sub-strings.  */
7          j ← async FindRepeats(B′)
8          J ← J + [j]
9          B ← MaybeClearHistory(B)
   /* Discussed in Section 5.3.3.  */
10 TraceReplayer (T, H)
11     if ∃ j ∈ J, j is complete then
12         IngestCandidates(j, C)
13     P ← P + [T]
       /* Advance all potential traces by H in the trie if possible.  Remove
          impossible traces, and extract fully matched candidates.  */
14     A ← AdvanceActiveCandidates(C, A, H)
15     A ← FilterInvalidCandidates(C, A)
16     D, A ← FilterCompletedCandidates(C, A)
17     if |D| > 0 then
           /* Select one of the pending candidates to replay.  Execute any tasks
              before it, and issue a trace replay for the candidate.  */
18         R ← SelectReplayTrace(D, P, A)
19         P, A ← ExecuteAndReplay(R, P, A)
   /* Applications (or Legate itself) issue tasks through Legion's ExecuteTask
      function.  */
20 ExecuteTask (T)
21     H ← Hash(T)
22     TraceFinder(H)
23     TraceReplayer(T, H)
```

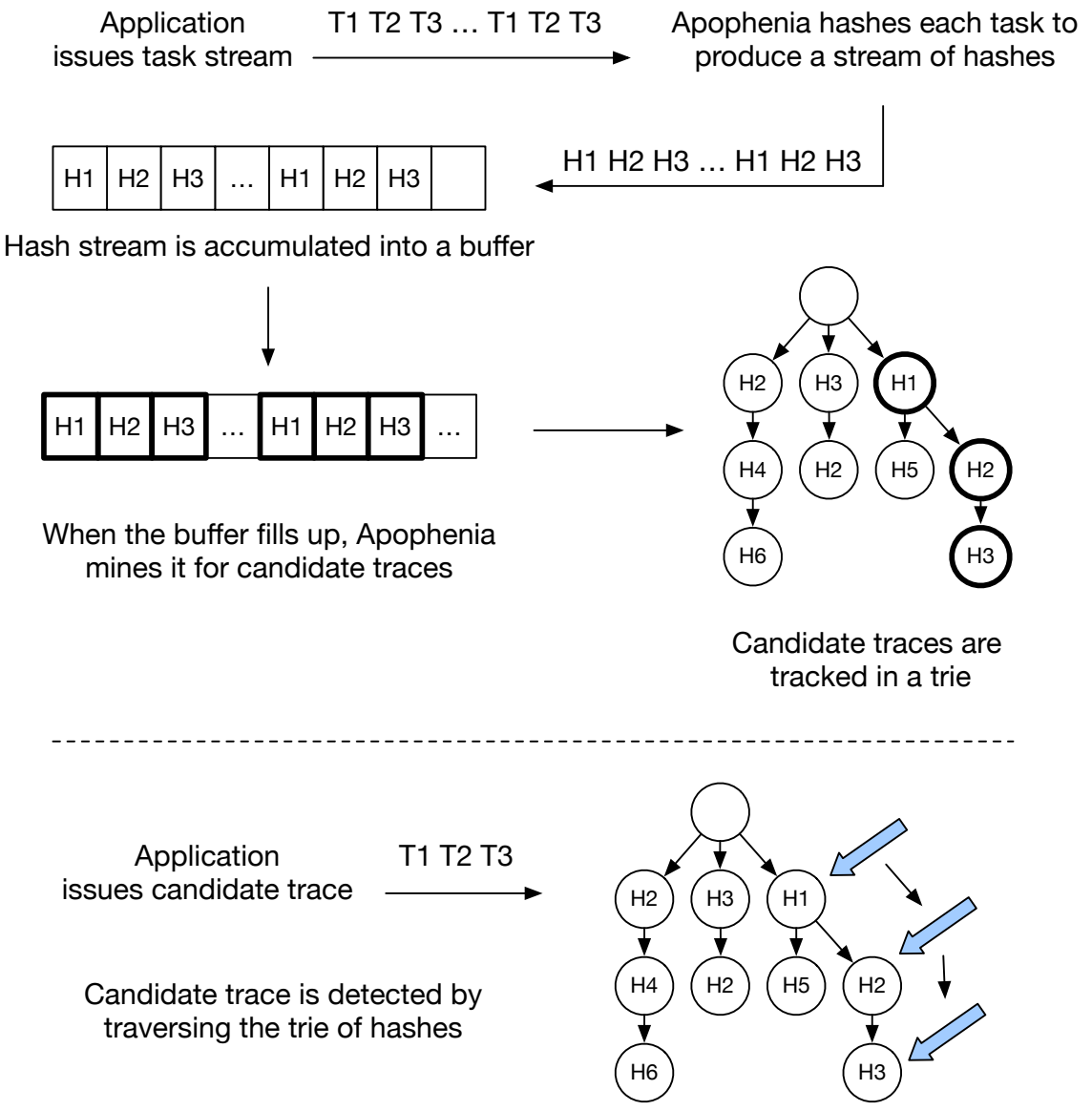


Figure 5.3: Visualization of Legion's automatic tracing analysis.

Existing Techniques

The Lempel-Ziv family of algorithms use repeated sub-strings for compression. Algorithms like LZ77 [46,62,63] maintain a sliding window of previous tokens to search for repeats in when encoding upcoming tokens. The LZW [53] algorithm avoids the use of a sliding window by only increasing the length of any candidate repeat by a single token at a time. While not directly finding a set of repeats with high coverage, similar algorithms that use a sliding window would need to maintain and search in a window the size of the analyzed buffer, resulting in quadratic time complexity. In order to recognize a trace of length n, an LZW-style algorithm would also need to encounter the trace $n-1$ times. We wanted an algorithm that is sub-quadratic in order to scale to large buffer sizes. Real-world applications we discuss in **??** have traces that contain more than 2000 tasks, requiring token buffers of at least twice that size to detect a single repeat.

Within the programming languages community, recent work by Sisco et al. [45] used a technique called *tandem repeat analysis* [47] to find loops in the netlists that result from compiling hardware description languages. A tandem repeat is a sub-string α that repeats contiguously within a larger string S, such that α^k is a sub-string of S, for some k. Despite the success that Sisco et al. had using tandem repeat analysis, we found that even simple real world cuPyNumeric programs did not contain enough tandem repeats for the analysis to reliably identify a trace set with high coverage. The reason is that while these real-world programs tended to have repetitive main loops, there would often be irregularly appearing computations such as convergence checks or statistics calculations that occur infrequently between loop iterations. As such, the strings that represented these programs would not contain tandem repeats, but instead repeated sub-strings separated by other tokens.

A relaxation of tandem repeat analysis is to search for non-overlapping repeated sub-strings, which removes the contiguity requirement on the repeats. Concretely, given the string *ababab*, *abab* is an overlapping repeat, while *ab* is non-overlapping. We could use non-overlapping repeated sub-strings to assemble a set of traces T and a disjoint mapping f that achieves high coverage. While there exist standard suffix-tree algorithms to find repeated sub-strings, we found that the natural extensions of these

algorithms to detect non-overlapping repeated sub-strings also resulted in quadratic runtime complexity.

Our Algorithm

We design a repeat finding algorithm that is directly aware of the optimization problem in **??** and runs in $O(n \log(n))$, where n is the size of the token history buffer. At a high level, our algorithm makes a pass through a suffix array constructed from the input buffer to collect a set of candidate repeats. It then greedily selects the largest repeated sub-strings that do not overlap with any previously chosen sub-strings. Psuedocode for our algorithm is in Algorithm 2[1], which takes a string S and returns a set of sub-strings that achieve high coverage of S. We assume that the reader is knowledgeable about suffix arrays and their structural properties. However, understanding the algorithm in Algorithm 2 is not required to understand its usage in Legion, as discussed in Section 5.3.3 and Section 5.3.4.

As a first step, we construct a suffix array and longest common prefix array from the input buffer of tokens. We then iterate through adjacent pairs of suffixes to construct a set of *candidate repeats*, which are tuples of sub-strings defined by their length, the repeated sub-string, and its starting position in S. These candidates are constructed based on whether or not the shared prefix between adjacent suffix array entries overlap. Once all of the candidates have been constructed, we sort the candidates to greedily select candidates in order of length, and select as many occurrences of a particular sub-string as possible. We only select candidates that do not overlap with any previously selected candidates, and then deduplicate the chosen set of candidates as the result. A sample execution of Algorithm 2 is shown in Figure 5.4.

Our algorithm can be implemented with time complexity $O(n \log(n))$. Linear time algorithms exist for suffix array and LCP array construction [30]. Two candidates are generated for each entry in the suffix array, so sorting the candidates takes $O(n \log(n))$ time. The interval intersection step can be reduced to constant time by leveraging

[1]We also present a standalone implementation of the algorithm available at `https://github.com/david-broman/matching-substrings`.

Algorithm 2: Algorithm to compute non-overlapping repeated sub-strings.

```
1  FindRepeats (S)
2      SA, LCP ← SuffixArray(S)
       /* Candidates are tuples of string length, the repeated sub-string, and
          starting position.  */
3      C ← []
4      foreach i ∈ [0, |SA| − 1) do
           /* Extract adjacent suffix array entries and their overlap length.
              */
5          s1, s2, p ← SA[i], SA[i + 1], LCP[i]
6          if [s1 : s1 + p) ∩ [s2 : s2 + p) = ∅ then
               /* S[s1 : s1 + p] and S[s2 : s2 + p] are repeated strings that do not
                  overlap in S, so they are candidates.  */
7              r ← S[s1 : s1 + p]
8              C ← C + [(p, r, s1), (p, r, s2)]
9          else
               /* S[s1 : s1 + p] and S[s2 : s2 + p] overlap in S.  Assume s2 > s1, the
                  other case is symmetric.  In this case, the overlap is a
                  collection of repeats of S[s1 : s1 + d], by the structure of the
                  suffix array.  */
10             d ← s2 − s1
               /* Break prefix into two chunks of repeated pieces of
                  S[s1 : s1 + d].  */
11             l ← (p + d)/2
               /* Remove trailing tokens.  */
12             l ← l − (l%d)
13             r ← S[s1 : s1 + l]
14             C ← C + [(l, r, s1), (l, r, s1 + l)]
       /* Sort the candidates by decreasing length and by increasing sub-string
          and start position.  */
15     Sort(C)
       /* Greedily collect sub-strings that do not overlap with previously
          chosen sub-strings.  */
16     I, R ← [], []
17     foreach (l, _, s) ∈ C do
18         if [s, s + l) does not intersect I then
19             I ← I + [[s, s + l)]
20             R ← R + [S[s : s + l]]
21     return R
```

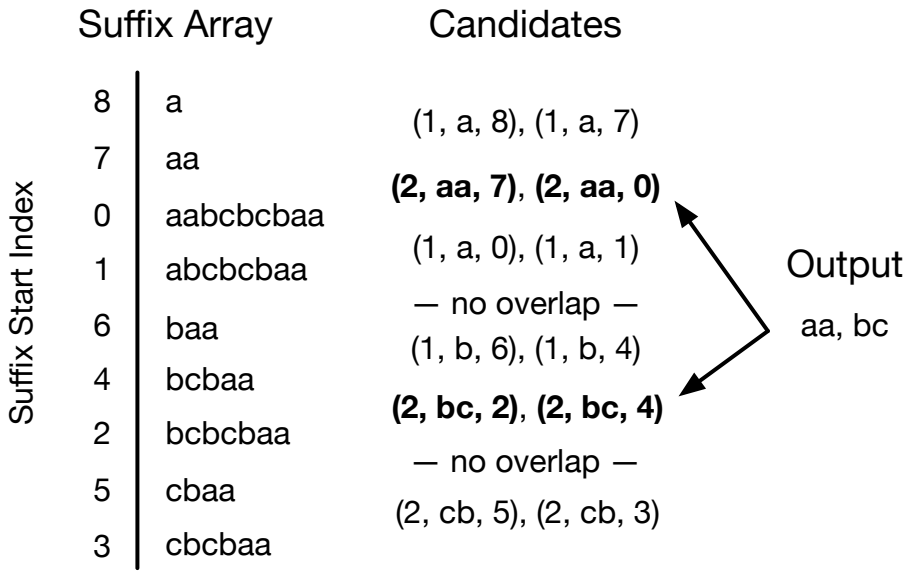


Figure 5.4: Execution of Algorithm 2 on "aabcbcbaa". The candidates for each suffix pair is shown between the pair.

the candidate iteration order, so the entire loop executes in $O(n)$ time. In particular, an array of length $|S|$ can be maintained, and as each candidate is selected, all positions covered by the candidate are marked. Then, as candidates are iterated over in decreasing length and increasing start position order, interval intersection can be checked by checking if the start or end of an interval is marked. Finally, the deduplication can be done by generating a unique ID for each candidate sub-string in the candidate generation phase, and adjusting the candidate representation to be a tuple of length, ID and starting position; using this sort order allows deduplication to be done at each iteration of the candidate selection loop.

Our algorithm aims to find good solutions to the optimization problem in **??** by identifying long repeated sub-strings and selecting as many as possible that do not overlap with each other. We trade off between an optimal solution to the optimization problem to instead find good solutions and maintain a lower asymptotic runtime. There are two such heuristics in our algorithm. First, when adjacent suffix array entries have a repetition, we consider only the maximal length repetition instead of all sub-strings of the repetition. Second, when we select which candidates to keep, we greedily choose the largest candidates instead of performing a bin-packing

computation. Our algorithm is guaranteed to find the longest repeated sub-string, but due to the second heuristic, we cannot provide theoretical guarantees about the other chosen sub-strings. We show in **??** that Legion using our algorithm is able to identify good traces in complex, real-world applications.

5.3.3 Recognizing and Replaying Candidate Traces

Legion's trace replayer uses Algorithm 2 to find candidate traces from the application's history of tasks. In this section, we discuss how Legion's trace replayer identifies and selects these candidate traces from the task stream to record and replay. Our design of the trace replayer has two major goals. First, the per-task overhead must be low, as it is imperative for performance for the application to issue as many tasks into the runtime as possible so that the runtime can either replay traces or perform dependence analysis ahead of execution. Slowing down the task launch rate would result in exposed latency from various sources in the runtime. Second, Legion must balance exploration and exploitation when selecting traces. As more information about the application is gained, Legion should switch to better traces as it finds them. However, Legion should not leave a steady state of replaying a particular trace until it is confident that performance can be improved, as memoization of the dependence analysis for new traces has a cost.

As discussed previously, Legion accumulates a history of tasks launched by the application and asynchronously uses Algorithm 2 to select candidate traces. Asynchronous analysis of task histories is important to avoid stalling the application by waiting for the analysis to finish before accepting the next task from the application.

When an asynchronous analysis completes, Legion ingests the results into a trie that maintains the current set of candidate traces. Along with this trie, Legion maintains a set of pointers into the trie that represent potential matched traces. As tasks are issued, Legion updates the set of pointers by creating new pointers for each new task, stepping any existing pointers down the trie if possible, and removing any pointers that are made invalid. Once a pointer reaches a leaf of the trie and has matched a trace, Legion has the option to record or replay the trace, by wrapping

the tasks with `tbegin` and `tend` calls.

Legion uses a scoring function to select which matched trace to replay when faced with multiple valid choices. The scoring function is based on the length of the candidate trace multiplied by a count of the number of times the trace has appeared. In calculation of the score, we impose a maximum value of the count that can be used, and exponentially decay the value of the count by how many tasks have been encountered since the trace last appeared. Finally, we increase the score slightly if a trace has already been replayed.

Our scoring function encodes heuristics about trace selection and aims to balance exploration and exploitation. We naturally prefer long traces over shorter ones, as longer traces have the potential to eliminate more runtime overhead. The capping of the appearance count allows for Legion to eventually switch from a trace that appeared early during program execution to a better trace that appears later in the execution. Next, decaying the appearance count ensures that a seemingly promising trace that occurs infrequently, does not eventually hit a threshold, and disrupts a steady state. Finally, since recording new traces has a cost, when faced with traces of a similar score, we bias Legion towards a trace it has already replayed.

5.3.4 Achieving Responsiveness and Quality

Legion's trace finder accumulates tasks into a buffer and mines the buffer for traces using Algorithm 2. An important question is what should the size of that buffer be? The size of this buffer trades off between responsiveness of Legion's trace identification and the quality of traces Legion is able to find. With a small buffer, Legion can identify traces early but will not be able to identify traces in programs with large loops. Meanwhile, a large buffer allows Legion to identify long traces in complex applications but introduces significant startup delay in smaller applications.

We did not want end users (especially non-expert users of high-level Legate libraries) to be required to continually adjust the buffer size parameter as their application changes. As such, some strategy to adapt the buffer size along this tradeoff space is necessary. We found that a strategy that attempts to dynamically resize

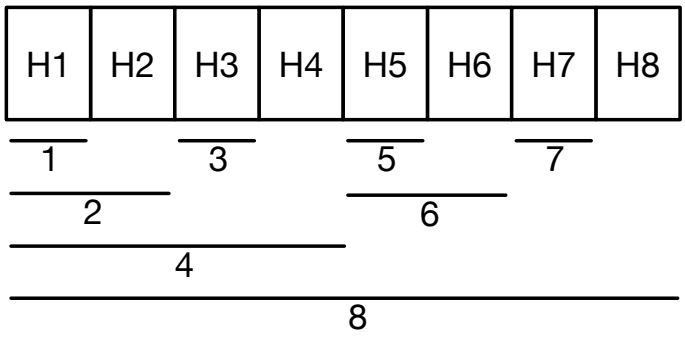


Figure 5.5: Visualization of Legion's buffer sampling strategy on a buffer of size 8. After processing the i'th task, Legion mines the buffer slice labeled i.

the buffer based on what traces to find is unsatisfactory, as the system is unable to differentiate between an application currently not repeating operations versus an application repeating a sequence of operations larger than the buffer size. Instead, we propose a strategy that selects a large fixed buffer size upfront, and then samples smaller pieces of the buffer in a principled manner to be responsive to the occurrence of short traces.

Legion samples from the buffer guided by the *ruler function* sequence [54], which provides a practically useful sampling strategy with provable guarantees. The ruler function counts the number of times a number can be evenly divided by two. Applying it to the sequence $1, 2, 3, 4, \ldots$ yields the sequence $0, 1, 0, 2, \ldots$. Raising the sequence to the power two yields $1, 2, 1, 4, \ldots$, which we can interpret as subsets of the buffer to analyze. For example, with a buffer size of four, as tasks arrive Legion would first analyze the first task, then the first two tasks, then the third task, and finally all four tasks. A visualization of this sampling policy is in Figure 5.5. This sampling policy lets Legion quickly react to changes in the application by analyzing recent pieces of the buffer while allowing larger traces to be found by infrequently analyzing longer components of the buffer. For example, sampling the full buffer in Figure 5.5 is required to find a trace that repeats in positions H2-H4 and H5-H7. In practice, we use the exponentiated ruler function as the multiples of a larger constant (such as 250) to sample the buffer with. Finally, given that our algorithm in Section 5.3 runs in $O(n \log(n))$, we show that our sampling strategy increases the total runtime complexity of processing the buffer by only an extra log factor, yielding a total of

$O(n \log^2(n))$. This technique enables all of the applications we evaluate (Section 5.5) to be run with the same buffer size configuration parameter.

5.4 Automatic Tracing Implementation Concerns

We now discuss important aspects of a realistic implementation of automatic tracing in Legion. In particular, we discuss the specifics of implementing automatic tracing in a distributed context, a conscious decision not to perform speculation when replaying traces, and the necessary extensions Legion's underlying tracing engine.

5.4.1 Distributing the Analysis

The automatic tracing analysis as presented in Section 5.3 is sequential, processing tasks as they are issued by the application or the higher-level Legate runtime system. In a distributed setting, automatic tracing leverages the interaction with Legion's *dynamic control replication* [13] to maintain acting as a sequential analysis, except for one component, which we discuss next. With control replication, the application executes on each node and Legion shards the dependence analysis and execution across nodes. The main restriction of control replication is that the application must issue the same sequence of tasks on every node. Our implementation of automatic tracing is a layer between the application and the rest of the Legion runtime system, meaning that standard calls into the Legion runtime are intercepted, and the automatic tracing infrastructure forwards a (possibly different) set of calls into the rest of Legion. Therefore, the automatic tracing analysis inherits the control replication requirements of the application. In particular, each node must agree on which traces to replay and when during program execution to record and replay the traces.

The only source of non-determinism in the automatic tracing analysis that could cause control divergence between nodes is the asynchronous processing of token buffers described in Section 5.3.2. An instance of Legion (and the automatic tracing analysis) exists on each node of the target machine, and each instance maintains a local history buffer of tasks to run asynchronous analyses on. Each issued asynchronous analysis

may complete earlier on one node than another, resulting in that node replaying a trace before another node has identified that trace as a candidate. However, making the analysis synchronous would result in stalling the application until analyses complete. We resolve this tension by having each node agree on a count of processed operations to issue before ingesting the results of an asynchronous analysis. If any node had to wait on an asynchronous analysis to complete, all nodes increase their count of operations to wait on for the next analysis. This strategy reaches a steady state where analysis results are ingested in a deterministic manner without stalling the application.

5.4.2 (The Lack of) Speculation

Speculation is a common technique in computer architecture to efficiently execute programs with data-dependent control flow. As automatic tracing in Legion has similarities to speculative components in architecture like trace caches (**??**), a natural design decision was if Legion should speculate on whether traces would be issued by the application. Concretely, if Legion decided to replay the trace $T_1T_2T_3T_1T_2T_3$ and has seen the tasks $T_1T_2T_3$ so far, should it wait until the second $T_1T_2T_3$ has been issued, or speculatively issue the trace $T_1T_2T_3T_1T_2T_3$ and roll back if the next three issued tasks differ from $T_1T_2T_3$? Our implementation of automatic tracing in Legion does not speculate and waits for the entirety of a trace to arrive before issuing the trace to the rest of the runtime system's analysis. The relative costs of different operations within the Legion runtime system made the potential upside of speculation not worth the implementation complexity.

Legion employs a pipelined architecture where a task flows through three stages: 1) the application phase, where the task is launched (into the automatic tracing analysis), 2) the dependence analysis phase, where the task is analyzed or replayed as part of a trace, and 3) the execution phase, where the task is executed. Depending on the cost ratio of the application and analysis phases, speculation may be beneficial as Legion waits for an entire trace to pass through the application phase. Legion's analysis phase is an order of magnitude more expensive than the application phase,

letting the application phase run far ahead of the analysis phase. Thus, waiting for an entire trace to be issued by the application rarely stalls the pipeline and gets exposed in the overall runtime. Therefore, we conclusded that designing a trace prediction algorithm and implementing a backup-rollback-recover scheme on speculation failures was not worth the complexity.

5.4.3 Non-Idempotent Traces

An important improvement to Legion that was needed for automatic tracing was support for *non-idempotent traces*; previously, Legion only supported *idempotent* traces. A trace is idempotent when its *preconditions* imply its *postconditions*. The preconditions of a trace capture the state of the dependence analysis when the trace was recorded, and the postconditions capture the state of the dependence analysis after the trace is executed. Intuitively, a trace may only be replayed if its preconditions are satisfied, i.e. at the start of the trace the dependence analysis is in the same state as when the trace was recorded.[2]

The impact of idempotency is that for back-to-back replays of an idempotent trace, the trace's preconditions are known to hold and do not need to be verified. For several technical reasons such as implementation complexity and the costs of checking trace preconditions, Legion only supported idempotent traces before our work. We found that it was necessary to relax this restriction and support non-idempotent traces. A consequence of modeling trace identification as a string analysis problem is that the conversion of task sequences into strings loses information about the pre- and post-conditions of a trace. As such, it is difficult to bias the string algorithms towards sub-strings with have semantic properties such as idempotency. Supporting non-idempotent traces involves recording multiple instances of the trace for each set of preconditions it may have (as the postconditions do not imply the preconditions), selecting the instance that is applicable in the dependence analysis state when replay is requested, and eagerly applying the postconditions to the dependence analysis state after replay.

[2]We refer readers to Lee et al. [35] for more information about trace conditions and idempotency.

5.5 Evaluation

5.5.1 Experimental Setup

We evaluate our implementation of automatic tracing on the largest and most complex Legion applications written to date, including production scientific simulations (some Legion-only and others in Legate) and a distributed deep learning framework. We perform a mixture of weak-scaling (**??**) and strong-scaling (**??**) experiments and show that our implementation of automatic tracing is able to match the performance of manually traced code when trace annotations already exist, and that Legion can identify traces in programs that were previously not traced. We then evaluate the overhead that automatic imposes on Legion applications (**??**) and visualize Legion's search process (Section 5.5.5).

We evaluated our implementation of automatic tracing on the Eos and Perlmutter supercomputers. Each node of Eos is an NVIDIA DGX H100, containing 8 H100 GPUs with 80 GB of memory and a 112 core Intel Xeon Platinum. Each node of Perlmutter contains 4 NVIDIA A100 GPUs with 40 GB of memory and a 64 core AMD EPYC 7763. Nodes of Eos are connected with an Infiniband interconnect, while Perlmutter uses a Slingshot interconnect. We compile Legion on Eos with the UCX networking module, and use the GASNet-EX [17] networking module on Perlmutter. We do not execute each application on both Perlmutter and Eos due to differences between the local environments on each machine. In our experiments, we evaluate the relative performance differences between traced and untraced programs, and comparisons between machines are not significant.

5.5.2 Weak Scaling

In this section, we discuss weak scaling results of applications using automatic tracing, as shown in **??** and Figure 5.10. In a weak scaling study, we increase the problem size as the size of the target machine grows to keep the problem size per processor constant. For each application, we perform a sweep over different sizes of the problem to vary the task granularity, thus affecting the impact of runtime overhead. These

different problem sizes are denoted in the graph by the "-s", "-m" and "-l" label suffixes which stand for small, medium and large. At smaller problem sizes, more runtime overhead can be exposed, while larger problem sizes make it easier to hide runtime overhead. In each weak-scaling plot, we report the steady-state throughput of each configuration and problem size after a number of warmup iterations (discussed in **??**). We report throughput in iterations per second achieved by each configuration, so within a particular problem size, higher is better; across problem sizes, the smaller problem sizes will achieve a higher iterations per second than the larger problem sizes.

S3D S3D [49] is a production combustion chemistry simulation code that has been developed over the course of many years by different scientists and engineers. The Legion port of S3D implements the right-hand-side function of the Runge-Kutta scheme, and interoperates with the legacy Fortran+MPI driver of the simulation. The integration between Legion and the legacy Fortran+MPI code leads to various constraints that the manual trace annotations interact with. For example, during the first 10 iterations, a hand-off between Legion and Fortran+MPI must occur every iteration, while after the first 10 iterations a hand-off is required only every 10 iterations. While not unmanageable, these interactions have led to relatively complicated logic to manually trace the main loop. We scale S3D on Perlmutter, and compare the performance of automatic tracing to manually traced and untraced versions of S3D. The results are shown in Figure 5.6. Even on a single node, tracing has a noticeable performance impact on the smaller problem sizes and affects the scalability of S3D. Legion with automatic tracing achieves within 0.92x–1.03x of the performance of the manually traced version, and between 0.98x–1.82x speedups over the untraced version. Manual annotations can slightly outperform the automatically collected traces by leveraging application knowledge to select traces that have lower replay overhead.

HTR HTR [23] is a production hypersonic aerothermodynamics application. HTR performs multi-physics simulations of hypersonic flows at high enthalpies and Mach numbers, such as for simulations of the reentry of spacecraft into the atmosphere. Like S3D, we evaluate the performance of Legion with automatic tracing on HTR

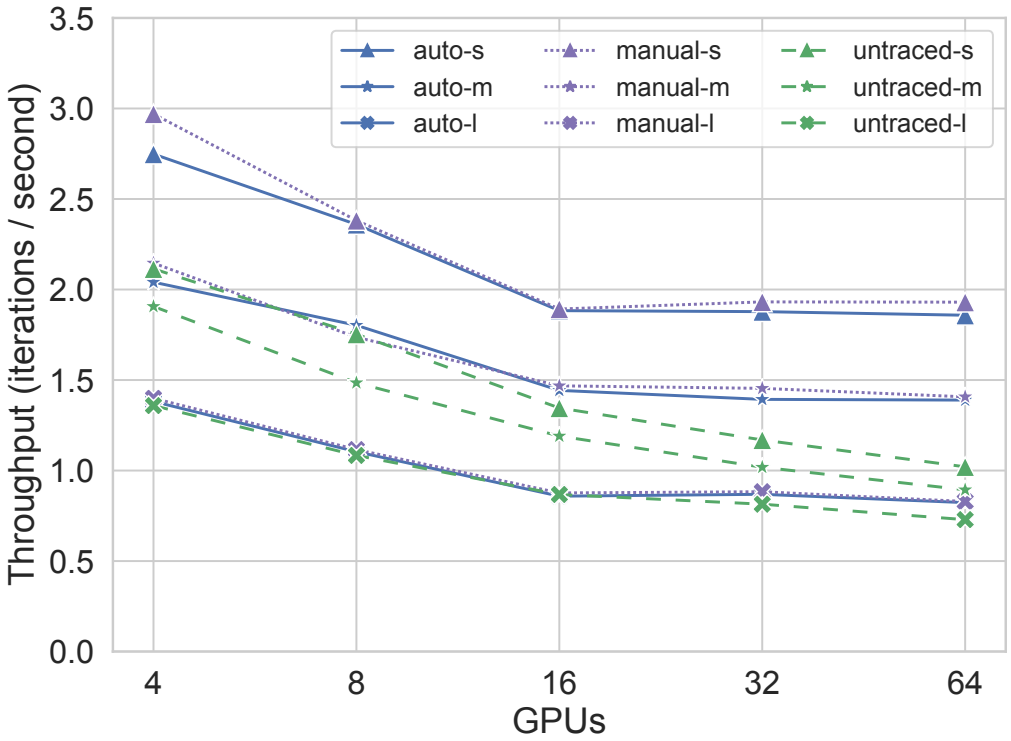


Figure 5.6: Weak scaling of S3D on Perlmutter. Legion with automatic tracing ("auto") performs competitively with hand-annotated traces.

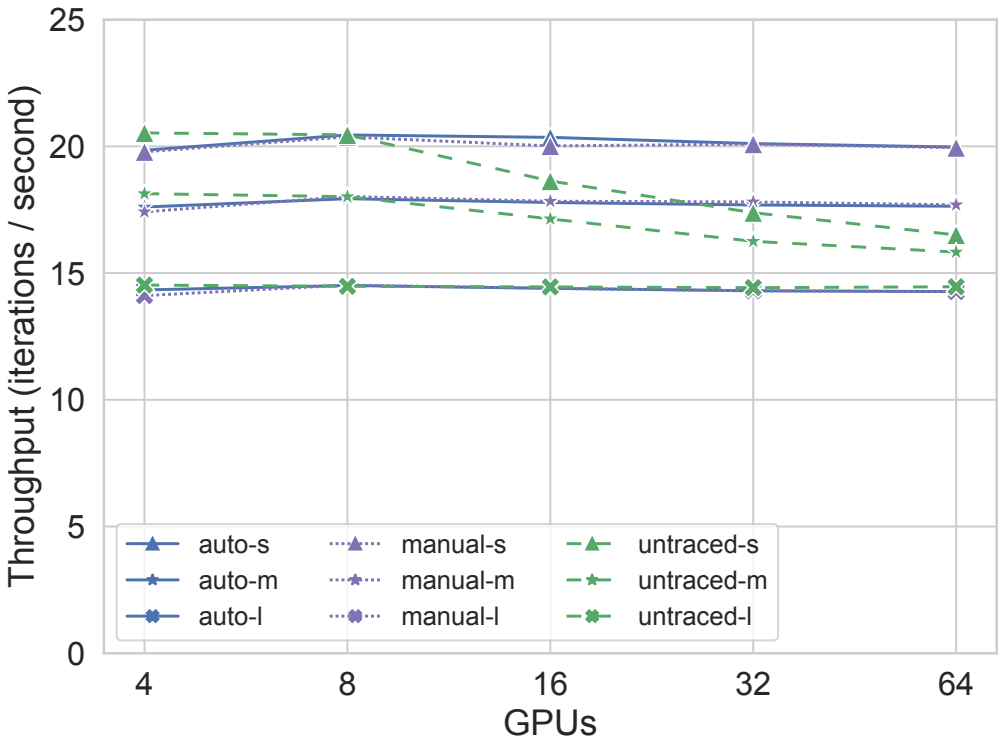


Figure 5.7: HTR-Solver (Perlmutter)

Figure 5.8: Weak scaling of HTR on Perlmutter. Legion with automatic tracing ("auto") performs competitively with hand-annotated traces.

on Perlmutter, and compare it against a manually traced version and an untraced version. While HTR without tracing performs competitively to the traced version at small GPU counts, Figure 5.7 shows that tracing is necessary for performance at scale. Legion with automatic tracing achieves within 0.99x–1.01x of the performance of the manually traced version, and between 0.96x–1.21x speedups over the untraced version.

CFD CFD is a cuPyNumeric application that solves the Navier-Stokes equations for 2D channel flow [9]. Unlike S3D and HTR, there is not a manually traced version of CFD, due to the difficulties around composition discussed in Section 5.1.1. Developing a manually traced implementation of CFD would require either rewriting

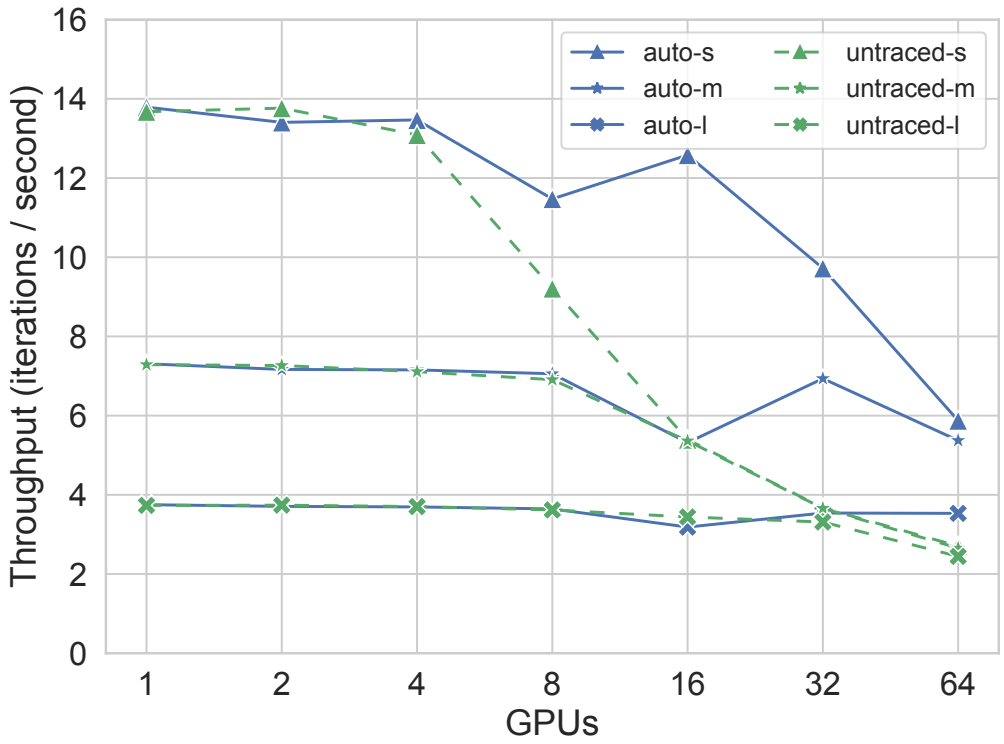


Figure 5.9: Weak scaling of CFD on Eos, where Legion with automatic tracing ("auto") outperforms the untraced version.

the application to remove any dynamic region allocation, or manual examination of allocator logs to find the number of iterations in the steady state. As a result, we compare CFD with automatic tracing to the standard untraced version on different problem sizes, which is the performance that cuPyNumeric users are able to achieve today.

Figure 5.9 shows weak scaling results for CFD on Eos. These results are similar to HTR, where leveraging tracing is necessary for performance at scale. On the smallest problem size, even though the tracing removes a large amount of runtime overhead, the tasks are too small to hide the communication latency at larger scales, leading to the observed fall off in performance. On larger problems, CFD with automatic tracing is able to maintain high performance while the untraced version falls off, yielding between 0.92x–2.64x speedups.

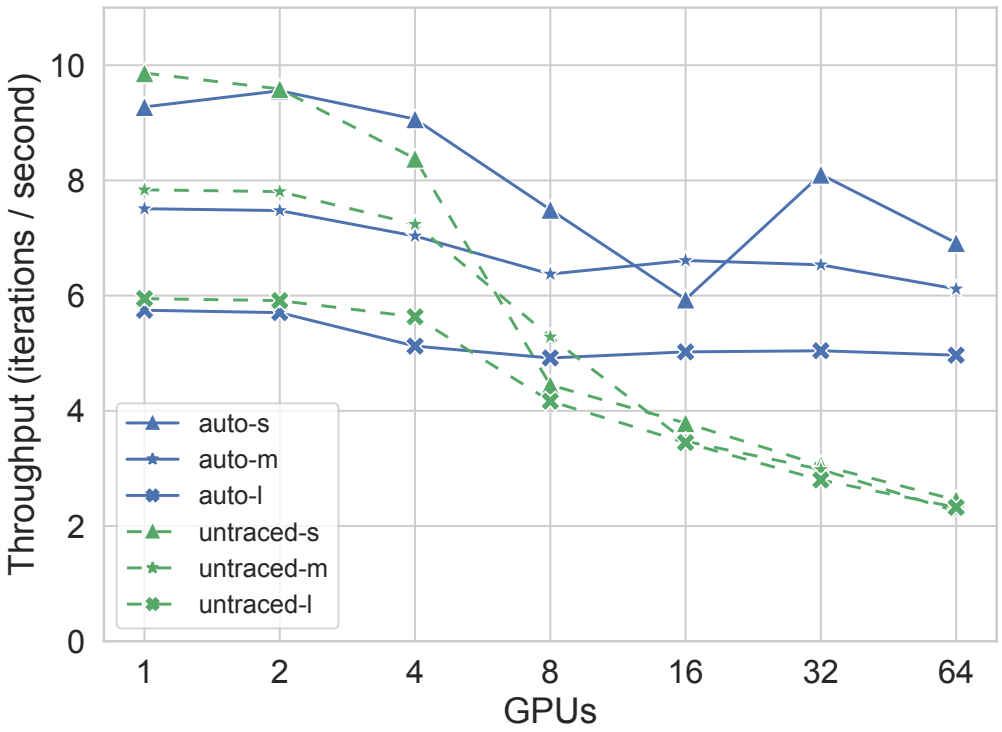


Figure 5.10: Weak scaling of TorchSWE on Eos, where Legion with automatic tracing ("auto") outperforms the untraced version.

TorchSWE TorchSWE is a cuPyNumeric port of the MPI-based TorchSWE [21] shallow-water equation solver, and is the largest cuPyNumeric application developed so far. Similarly to CFD, there is no manually traced version to compare to. However, unlike CFD, performing a rewrite of TorchSWE to enable manual tracing would be difficult, as TorchSWE contains an order of magnitude more lines of code. Weak scaling results for TorchSWE on Eos are shown in Figure 5.10, which show that TorchSWE is significantly impacted by Legion runtime overhead without tracing.

These results demonstrate that there does not exist a problem size for TorchSWE on Eos that can hide Legion's runtime overhead without tracing. Even the large problem size, which nearly reaches the GPU's memory capacity, exposes Legion runtime overhead at 8 GPUs. The reason for this is that TorchSWE maintains a large number of fields for each simulated point, and issues different array operations on each field.

The amount of data needed for each element in the simulation does not allow the task granularity to be easily increased to the untraced Legion minimum of $\tilde{1}$ms per task, as each new element added increases the memory footprint more than it increases the average task granularity. For such applications, leveraging tracing is a requirement, and automatic tracing enables complex applications like TorchSWE to do so automatically. TorchSWE itself contains enough task parallelism to hide communication latencies, but needs tracing to first lower runtime overhead. With automatic tracing, we are able to achieve between 0.91x–2.82x speedup on TorchSWE, achieving nearly perfect scalability on 64 GPUs.

5.5.3 Strong-Scaling

We now move from scientific simulation codes to distributed deep neural network training with FlexFlow [29, 51]. FlexFlow is a deep neural network framework that searches for hybrid parallelization strategies for different layers of the network. We perform a strong-scaling experiment with FlexFlow on Eos to train the largest (`pilot1`) network from the CANDLE [1] initiative[3]. A strong-scaling study fixes the problem size on a single processor, and increases the number of processors while keeping total problem size constant. To strong scale the training, we fix the batch size for single GPU, and then increases the number of GPUs available.

We compare the performance of FlexFlow with manual trace annotations, two configurations of Legion (discussed next), and no tracing. As seen in Figure 5.11, as FlexFlow scales up, the tasks become smaller and begin to expose Legion runtime overhead without tracing, leading to slowdowns when scaling up. The two configurations of Legion differ in the maximum trace length to be replayed (Legion's history buffer is the same, but recorded traces are broken into pieces of a given maximum size). The first (auto-5000) is the standard configuration with no maximum, as used in all other experiments, and the second (auto-200) has a maximum length of 200 tasks, which is similar to the length of the manually annotated trace. As FlexFlow strong scales, the cost of Legion issuing the trace replay starts to become exposed

[3]Due to engineering limitations in FlexFlow at the time of experiment collection, the network was parallelized only with data parallelism.

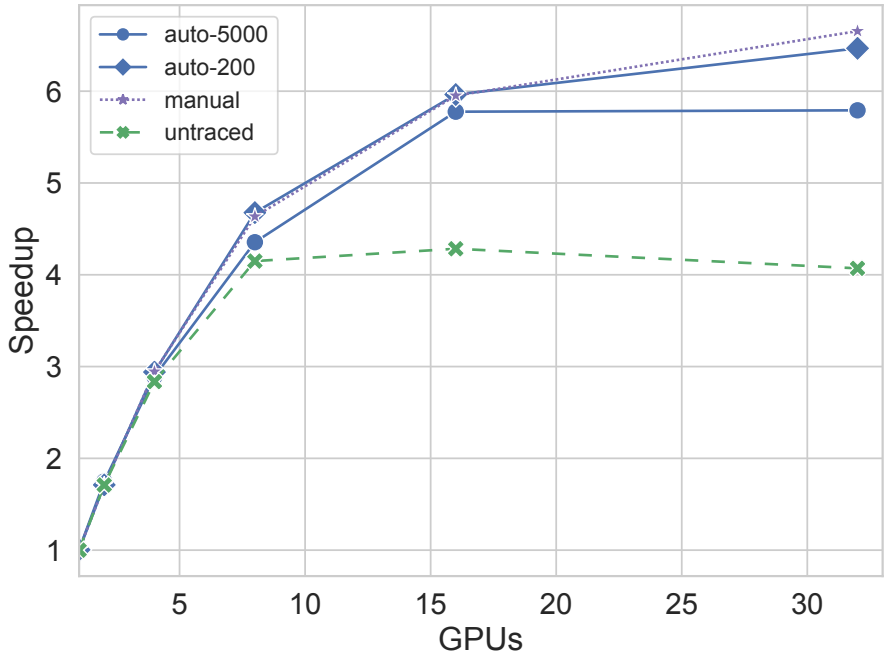


Figure 5.11: Strong scaling of FlexFlow on Eos.

as the execution time of the trace decreases, leading to shorter traces exposing less latency, and thus performing better[4]. On 32 GPUs, the configuration of Legion with a maximum trace length of 200 achieves between 0.97x the performance of the manually traced FlexFlow, and achieves a 1.5x speedup over the untraced FlexFlow.

5.5.4 Overheads of Automatic Tracing

We now discuss the overheads that automatic tracing imposes over standard execution with Legion. While we inherit the overheads of Legion's existing tracing infrastructure [35] (the cost of memoizing traces), automatic tracing imposes two new sources of overhead to measure: 1) the overhead on task launches and 2) the time taken until a steady state is reached.

As discussed in Section 5.3, the automatic tracing analysis intercepts the application's task launches and performs some analysis work before forwarding the task launches to Legion. This analysis work includes launching asynchronous token buffer processing jobs and manipulating traversals of the trie data structures used for online trace identification. To quantify this overhead, we ran a two node experiment on Perlmutter and measured the time it took to launch (not analyze or execute) Legion tasks with and without automatic tracing enabled. We ran a two node experiment to ensure that the coordination logic discussed in Section 5.4.1 was included in timing. We found that task launching took on average 7μs without automatic tracing, and on average 12μs with automatic tracing. While automatic tracing increases the task launch overhead, this overhead is still significantly lower than the amount of time it takes to replay a task as part of a trace, which is 100μs. As such, the task launching cost of automatic tracing can still be effectively hidden by the asynchronous runtime architecture. The asynchronous analysis jobs that Legion launches to process task histories do not affect the critical path, and utilize Legion's background worker threads. While in theory these jobs could compete for the resources necessary for Legion's dependence analysis, we have not yet encountered an application where they caused a detriment in performance.

[4]The Legion team is aware of this shortcoming and plans to address it in the future. Part of the work to reduce this overhead is discussed in Chapter 6.

Application	Iterations Until Steady State
S3D	50
HTR	50
CFD	300
TorchSWE	300
FlexFlow	30

Figure 5.12: Warmup iterations before Legion with automatic tracing reaches a replaying steady state.

To measure the time taken until Legion with automatic tracing reaches a steady state of replaying traces on our iterative applications, we report the number of iterations until a steady state is reached. Figure 5.12 contains the iteration counts needed for each application in Section 5.5.2 and Section 5.5.3, which range from 30 to 300. These simulation and machine learning workloads would be run in production for a significantly larger number of iterations, so speedup in the steady state corresponds closely to end-to-end speedup. We note that the cuPyNumeric applications have a larger number of required warmup iterations due to the dynamic behavior discussed in Section 5.1.1, where a single application-level iteration of the program does not necessarily correspond to a repeated sequence of tasks.

In terms of resource utilization, automatic requires a modest amount of CPU memory to store the history buffer of tasks for analysis. Legion runs the asynchronous string analysis (Section 5.3.2) on the existing pool of background worker threads. We have not found these resource requirements to impact application performance or memory utilization.

5.5.5 Trace Search

To give intuition about the search process that the automatic tracing analysis performs, we constructed a visualization of the amount of runtime overhead that Legion is removing over time. Figure 5.13 is a visualization of S3D over time (for 70 iterations), where each for task launched by S3D, we display how many of the previous 5000 tasks were traced. For iterative computations, this procedure yields the expected result, where Legion spends time during program startup discovering new traces, and

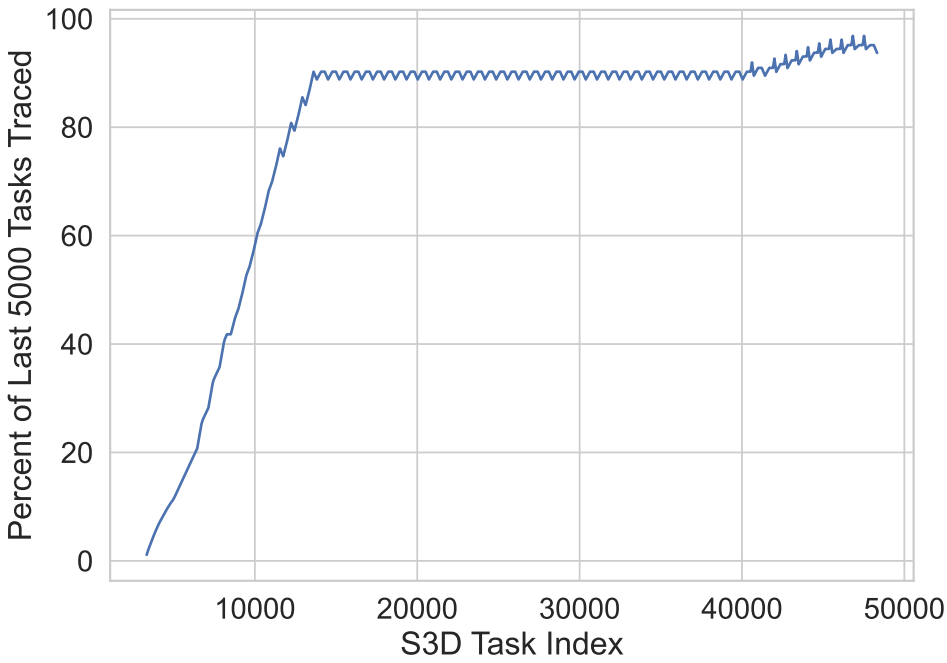


Figure 5.13: Visualization of Legion finding traces in S3D.

then settles into a steady state. The amount of traced operations increases slightly by the end of the execution, as Legion finds a better set of traces that lowers the number of untraced operations.

5.6 Conclusion

Some sort of synthesis of this section and connections into the larger thesis and the rest of the work.

Chapter 6

Controlling Overheads (Task-Based Execution)

1. Drop the actor-task paper
2. actor-task paper results

Chapter 7

Implementing a Legate Library

The previous chapters of this thesis described the architecture of the overall Legate runtime system, and described several key program representations and optimization strategies within Legate to enable the the composition of independent modules with high performance. In this chapter, we move from the internals of Legate to focus on the design and implementation of Legate Sparse [59], a prototypical Legate library. We describe how the architecture of Legate allows for the implementation of Legate Sparse to be independent of the context in which Legate Sparse is used; even individual operations within Legate Sparse are developed without consideration for execution strategies in other parts of the library. We do not include direct evaluation results for Legate Sparse in this section, as we used Legate Sparse to develop benchmark applications that were evaluated and discussed in Chapters 3 and 4.

7.1 SciPy Sparse

SciPy Sparse [3, 52] 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.

7.2 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 Legate stores to extend these single-node representations into distributed sparse matrix data structures by mapping each of the arrays used to represent sparse matrices to stores. For instance, the row, column and value arrays in the COO format are represented directly as stores 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 7.1, which allow for tighter interaction with Legate's partitioning constraints (discussed more in Section 7.2.1).

Our decision to represent sparse matrices as a set of stores 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 stores aligns with the Legate programming model, and

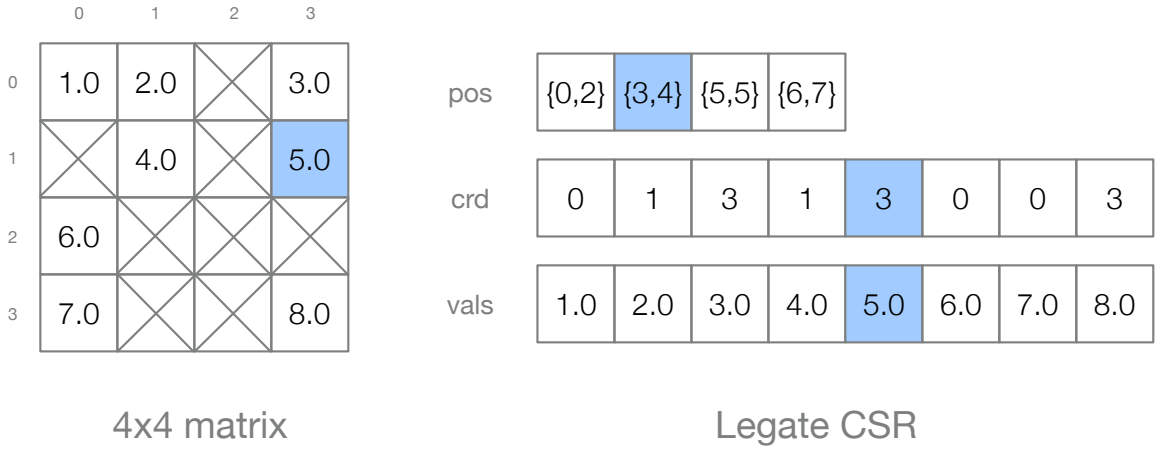


Figure 7.1: Legate Sparse's CSR sparse matrix encoding.

careful choices of partitioning enables description of non-trivial communication patterns. Additionally, this choice allows for inter-operation with other Legate libraries: since sparse matrices are constructed from stores, users of Legate Sparse can directly construct sparse matrices out of cuPyNumeric 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. Previously discussed experiments with Legate Sparse (Chapters 3 and 4) show that this choice of sparse matrix representation has low overhead while allowing for direct use with Legate and close alignment with SciPy Sparse's programming model.

7.2.1 Partitioning Constraint Interaction

The small variation from the standard representation of sparse matrices allows us to directly employ Legate's image constraints to relate partitions of the `pos` and `crd` stores with one another. We also use images to relate partitions of the `crd` store 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.

```
def spmv(self, A, x):
  # Compute y = A @ x and return y.
  y = cupynumeric.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 7.2: Python implementation of a row-based distributed CSR SpMV (adapted from DISTAL generated code).

We use an image from the partition of A's `crd` region to compute the referenced locations of x. 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.

A concrete example of a task implementation using the Legate constraint interface is shown in Figure 7.2, which implements a row-based distributed sparse matrix-vector multiplication (SpMV). Upon execution of the task launching code in Figure 7.2, the `task` object contains the following partitioning constraints: `equals(y, pos)`, `image(pos, crd)`, `image(pos, vals)`, and `image(crd, x)`. The constraints require `y` and `pos` to be partitioned in the same way, as each element of `y` and `pos` correspond to a row of the matrix `A`. Then, an image relates `pos` with `crd` and `vals`, as the range of coordinates and values stored for each row is represented by `pos`. Finally, the coordinates of `x` that each row of `A` will gather from are stored in `crd`; this means another image is required to link `crd` and `x`.

When Legate solves these constraints, it 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, Legate 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, Legate invokes Legion's image operation to construct partitions of `crd`, `vals` and `x` to satisfy the remaining constraints.

This implementation of SpMV in Legate Sparse has several attractive properties.

First, it does not require a specific data partitioning stored within Legate Sparse; an arbitrary partitioning of `A` may be used, perhaps created by a graph partitioning framework before the invocation of SpMV, and the implementation will adapt to the existing distributed layout. Next, the implementation does not contain explicit communication that assumes distributed data layouts for the input or output vectors, allowing for data movement required at the library boundary with cuPyNumeric to be implicitly discovered by Legate. Legate can discover the required communication, and dispatch to the correct APIs for inter-node and intra-node data movement, along with discovering the communication partners required for the sparse communication of `x`.

7.3 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.

Since the original development of Legate Sparse in 2022-2023, it has transitioned into a engineering-maintained project at NVIDIA. Subsets of the implementation have been adapted to support requested optimizations, removed to control the exposed library surface area, or new components have been added based on customer feedback. We discuss here the status of the original prototype of Legate Sparse.

7.3.1 Generating Kernels with DISTAL

We used the DISTAL [55, 56] compiler (work done in PhD that is not part of this thesis) 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 (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 allowed developer time to be spent elsewhere when optimizing the library. We first give an overview of DISTAL, and then describe how it was used to generate code for Legate Sparse.

DISTAL compiles a tensor algebra domain specific language (DSL) into C++ code that directly targets 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 Sparse. Since Legate's constraint solver already 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 (in the modern day with a LLM coding assistant), 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 7.3, the generated C++ task body is found in Figure 7.4, and the constraint based task launching code in Figure 7.2 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

```
// 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 7.3: Distributed, multi-threaded CSR SpMV in DISTAL.

```
void CSRSpMVTask(vector<Store> stores) {
  auto y = stores[0];
  auto pos = stores[1];
  auto crd = stores[2];
  auto vals = stores[3];
  auto x = stores[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 7.4: DISTAL-generated C++ task for row-based, multi-threaded CSR SpMV, with minor modifications.

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 [6].

7.3.2 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 leveraging the composability that the Legate ecosystem offers, we were able bootstrap our library with itself and cuPyNumeric 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 cuPyNumeric.

7.3.3 Hand-Written Implementations

7.3.4 Unimplemented Components

1. Give an overview for legate library implementation (decomposing computations etc). Unless this is already done earlier in the thesis.

2. Talk about the kinds of kernels in Legate Sparse

3. Talk about using DISTAL to generate kernels. (it's unclear how much detail i should go into DISTAL to begin with, as it seems to be diverging in a tangent from the rest of the thesis.

4. Talk about translating library functions from scipy and cupy as-is, which is a testatement to the rest of this software stack doing what it is supposed to.

Chapter 8

Related Work

Just thinking here, the order is not quite fixed.

1. Composition in the shared-memory world
    - Weld, Fern, Manya's latest paper
    - Jack Dennis's old work around composable dataflow machines.
    - This thread scheduling paper in multi-process/multi-runtimes thing in my stack
2. Distributed libraries for machine learning (jax, pytorch)
3. Similar libraries like DaCe that take more static approaches
4. Alternative recent "Python at scale" systems
    - Dask, Ray (focus on runtime system gaurantees)
    - Arkouda (chapel numpy), and maybe more.
5. Other kinds of old compilation work that tried to analyze code to plan distribution.
    - Thinking about work like Saman's.
6. Alternate task-based runtime systems, and how those might fit in

- StarPU, ParSEC, Ray etc. Can talk about why these systems wouldn't actually work for what we want

7. Actor programming models for contrast with the actor-task work.
8. New networking technology, how that might change how runtime systems work, like NVL72/NVL SHARP.
9. Just-In-Time compilers, and how our work relates to that.
10. String processing algorithms, as discussed in the apophenia work
11. Other components in the Legion world that are important for this work?
    - Static constraint-based parallelism
    - Regent optimizations, SCR, DCR
    - Tracing?

Chapter 9

Conclusion

Bibliography

[1] CANDLE — Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer — wordpress.cels.anl.gov. `https://wordpress.cels.anl.gov/candle/`. [Accessed 06-05-2024].

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

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

[4] 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. PETSc Web page. `https://petsc.org/`, 2022.

[5] Kihiro Bando, Steven Brill, Elliott Slaughter, Michael Sekachev, Alex Aiken, and Matthias Ihme. *Development of a discontinuous Galerkin solver using Legion for heterogeneous high-performance computing architectures*. 2021.

[6] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. Mosaic: An interoperable compiler for tensor algebra. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[7] Manya Bansal, Dillon Sharlet, Jonathan Ragan-Kelley, and Saman Amarasinghe. Lightweight and locality-aware composition of black-box subroutines. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[8] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGOPS Oper. Syst. Rev.*, 40(5):394–403, oct 2006.

[9] Lorena Barba and Gilbert Forsyth. Cfd python: the 12 steps to navier-stokes equations. *Journal of Open Source Education*, 2(16):21, 2019.

[10] Michael Bauer. *Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[11] Michael Bauer and Michael Garland. Legate numpy: accelerated and distributed array computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 105–118, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and

Alex Aiken. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 105–118, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. Visibility algorithms for dynamic dependence analysis and distributed coherence. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 218–231, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[16] Laura Susan Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. Scalapack: a portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, page 5–es, USA, 1996. IEEE Computer Society.

[17] Dan Bonachea and Paul H. Hargrove. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, volume 11882 of *Lecture Notes in Computer Science*. Springer Int'l Publishing, October 2018. https://doi.org/10.25344/S4QP4W.

[18] Uday Bondhugula. High performance code generation in MLIR: an early case study with GEMM. *CoRR*, abs/2003.00532, 2020.

[19] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, USA, 2008. AAI3325799.

[20] 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. JAX: composable transformations of Python+NumPy programs, 2018.

[21] Pi-Yueh Chuang. Torchswe: Gpu shallow-water equation solver. https://github.com/piyueh/TorchSWE, 2021.

[22] NVIDIA Corporation. Welcome to Legate.STL — NVIDIA legate.core — docs.nvidia.com. `https://docs.nvidia.com/legate/24.06/legate.stl/source/legate-stl.html`. [Accessed 04-03-2026].

[23] Mario Di Renzo, Lin Fu, and Javier Urzay. Htr solver: An open-source exascale-oriented task-based multi-gpu high-order code for hypersonic aerothermodynamics. *Computer Physics Communications*, 255:107262, 2020.

[24] 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. Quantum optimization of maximum independent set using rydberg atom arrays. *Science*, 376(6598):1209–1215, 2022.

[25] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery.

[26] 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. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[27] Michael Allen Heroux and Jack. Dongarra. Toward a new metric for ranking high performance computing systems. 6 2013.

[28] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, jul 1996.

[29] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[30] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[31] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[32] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[33] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[34] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. 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*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. Dynamic tracing: memoization of task graphs for dynamic task-based runtimes. In *Proceedings of the Int'l Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

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

[37] NVIDIA. legate-io. Accessed 2026-03-05.

[38] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

[39] Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[41] QuEraComputing. Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture., 2023.

[42] RAPIDS. legate-boost, 2023. Accessed 2026-03-04.

[43] RAPIDS. legate-dataframe, 2024. Accessed 2026-03-04.

[44] RAPIDS. legate-raft, 2024. Accessed 2026-03-04.

[45] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. Loop rerolling for hardware decompilation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[46] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, oct 1982.

[47] Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1):843–856, 2002.

[48] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: an event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 263–276, New York, NY, USA, 2014. Association for Computing Machinery.

[49] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick Mccormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, and Jacqueline H. Chen. S3d-legion: An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. 11 2017.

[50] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 344–358, New York, NY, USA, 2016. Association for Computing Machinery.

[51] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.

[52] 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. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[53] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[54] Wikipedia. Ruler function — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Ruler%20function&oldid=1193825609`, 2024. [Online; accessed 02-May-2024].

[55] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Distal: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 286–300, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Spdistal: Compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[57] Rohan Yadav, Michael Bauer, David Broman, Michael Garland, Alex Aiken, and Fredrik Kjolstad. Automatic tracing in task-based runtime systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 84–99, New York, NY, USA, 2025. Association for Computing Machinery.

[58] Rohan Yadav, Joseph Guman, Sean Treichler, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. On the duality of task and actor programming models, 2025.

[59] Rohan Yadav, Wonchan Lee, Melih Elibol, Manolis Papadakis, Taylor Lee-Patti, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. Legate sparse: Distributed sparse computing in python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[60] Rohan Yadav, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. Composing distributed computations through task and kernel fusion. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 182–197, New York, NY, USA, 2025. Association for Computing Machinery.

[61] David Kai Zhang, Rohan Yadav, Alex Aiken, Fredrik Kjolstad, and Sean Treichler. Kdrsolvers: Scalable, flexible, task-oriented krylov solvers. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, page 1351–1365, New York, NY, USA, 2025. Association for Computing Machinery.

[62] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[63] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.