Composing Distributed Computations Through Task and Kernel Fusion

ANONYMOUS AUTHOR(S)

We introduce Diffuse, a system that dynamically performs task and kernel fusion in distributed, task-based runtime systems. The key component of Diffuse is an intermediate representation of distributed computation that enables the necessary analyses for the fusion of distributed tasks to be performed in a scalable manner. We pair task fusion with a JIT compiler to fuse together the kernels within fused tasks. We show empirically that Diffuse's intermediate representation is general enough to be a target for two real-world, task-based libraries (cuNumeric and Legate Sparse), letting Diffuse find optimization opportunities across function and library boundaries. Diffuse accelerates unmodified applications developed by composing task-based libraries by 1.86x on average (geo-mean), and by between 0.93x-10.7x on up to 128 GPUs. Diffuse also finds optimization opportunities missed by the original application developers, enabling high-level Python programs to match or exceed the performance of an explicitly parallel MPI library.

1 INTRODUCTION

Task-based runtime systems have emerged as an effective way to program distributed and heterogeneous machines [Augonnet et al. 2009; Barham et al. 2022; Bauer et al. 2012; Danalis et al. 2015; Moritz et al. 2018], and as frameworks for building distributed libraries that effectively compose [Bauer and Garland 2019; Yadav et al. 2023]. Applications and libraries developed in task-based systems decompose computations into a sequence of tasks issued to a runtime, and map data onto runtime-managed collections of distributed data. *Tasks* are user-defined functions, whose bodies we call *kernels*, that operate on subsets of these collections. Task-based runtimes are responsible for extracting parallelism from the input sequence of tasks and for computing the necessary synchronization and communication required between tasks. This automation greatly improves the productivity of programming distributed machines and enables independently written libraries using the same tasking runtime to share distributed data.

However, applications built through the composition of task-based libraries can leave performance on the table when compared to monolithic applications that are specialized to a problem. Individual operations in task-based libraries are implemented and executed as highly optimized, but isolated, tasks. Optimizations that cross task boundaries must be performed to more effectively compose task-based libraries, and the most important of which are the fusion of tasks and the fusion of kernels within fused tasks. Task and kernel fusion have several potential benefits: 1) reduction of task launch overhead by launching fewer tasks into the runtime system, 2) faster task execution, as fused kernels may reuse data from caches and registers, and 3) lower memory usage due to the removal of intermediate data structures. These performance benefits are especially available in programs that compose operations from multiple functions or libraries together, where the operations share the same distributed data structures.

To tackle this problem of efficient library composition, frameworks like Weld [Palkar et al. 2017] and Split Annotations [Palkar and Zaharia 2019] were developed to fuse computations across functions within and and across libraries. While not directly targeting task-based libraries, these systems have a similar goal, and yield some of the benefits listed earlier. These works show how to fuse computations across functions and libraries in shared-memory settings. However, the presence of distributed memory complicates program analyses, as distributed data structures may alias and require communication in order to be kept up-to-date. For example, the same sequence of

2024. 2475-1421/2024/6-ART1 $15.00
https://doi.org/

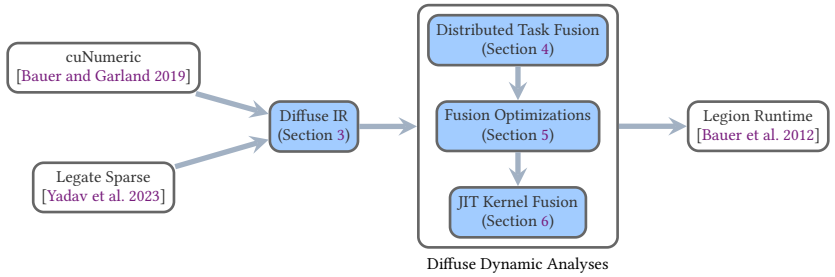


Fig. 1. System Overview of Diffuse. Highlighted components indicate our contributions.

element-wise operations on a pair of distributed arrays may or may not be sound to fuse depending on whether the arrays are aliasing views of the same distributed data. In this work, we show techniques to achieve efficient composition through fusion on distributed memory machines.

We present Diffuse, a system that automatically performs task and kernel fusion for distributed, task-based runtime systems, transparently achieving optimizations found in hand-tuned programs. An overview of Diffuse is in Figure 1. Diffuse reasons over a task-based intermediate representation (IR) of distributed computation, modeling computation as a sequence of tasks operating on partitioned data. Diffuse's IR is *scale-free*, meaning that the size of the IR and analyses on it are independent of the number of processors in the target machine. Diffuse pairs these analyses with an MLIR-based [Lattner et al. 2020] JIT compiler to fuse and optimize kernels within tasks fused by the prior analyses, enabling data reuse across independent tasks. By targeting analysis on a task-based IR, Diffuse's optimizations are not tied to the semantics of any particular library.

The contributions of this work are:

(1) A scale-free intermediate representation of task-based, distributed computation (Section 3),
(2) A dynamic analysis for task fusion in a distributed-memory setting (Section 4), and
(3) A JIT compilation pipeline for kernel fusion on fused task bodies (Section 6).

We implement Diffuse as a middle layer between high-level task-based libraries and the low-level Legion runtime system. We modify the implementations of the distributed libraries cuNumeric [Bauer and Garland 2019] and Legate Sparse [Yadav et al. 2023] to target Diffuse's IR, and we modify these libraries to expose their task implementations in MLIR for Diffuse's compiler to process. Diffuse then performs dynamic analyses to fuse the tasks and kernels issued by these libraries before forwarding the optimized tasks to Legion for execution. As a result, programmers developing cuNumeric and Legate Sparse programs benefit from Diffuse without modifying their applications.

To evaluate Diffuse, we apply it to micro-benchmarks and several full scientific computing applications developed in cuNumeric and Legate Sparse, including iterative Krylov subspace solvers and physical simulations. We compare against the standard implementations of cuNumeric and Legate Sparse and show that Diffuse achieves 1.86x speedup on average (geo-mean) over unmodified applications on up to 128 GPUs. We additionally compare against the high-performance PETSc [Balay et al. 2022] library for distributed sparse linear algebra, an explicitly parallel MPI library, and show that Diffuse enables naturally-written NumPy and SciPy Sparse programs to match or exceed the performance of PETSc (1.4x geo-mean speedup). Finally, we show that Diffuse is able to find fusion and optimization opportunities missed by the original application developers, achieving 1.23x speedup on average (geo-mean) over already hand-optimized code.

2 MOTIVATING EXAMPLE

Figure 2 is an example of Diffuse optimizing the cuNumeric [Bauer and Garland 2019] program in Figure 2a, which performs a 5-point stencil computation, a common pattern in scientific computation.

```
1  import cunumeric as np
2  grid   = np.random.rand((N+2, N+2))
3  # Create multiple aliasing views
4  # of the distributed grid array.
5  center = grid[1:-1, 1:-1]
6  north  = grid[0:-2, 1:-1]
7  east   = grid[1:-1, 2:  ]
8  west   = grid[1:-1, 0:-2]
9  south  = grid[2:  , 1:-1]
10 for i in range(niters):
11   avg = center + north + \
12         east + west + south
13   work = 0.2 * avg
14   center[:] = work
```

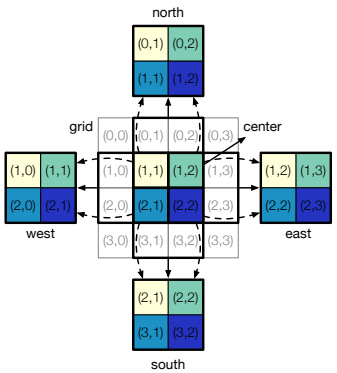
(a) cuNumeric source code.

```
1  # ADD, MULT and COPY are in
2  # cuNumeric's implementation.
3  ALLOC ARRAY t1
4  ADD(center, north, t1)
5  ALLOC ARRAY t2
6  ADD(t1, east, t2)
7  ALLOC ARRAY t3
8  ADD(t2, west, t3)
9  ALLOC ARRAY avg
10 ADD(t3, south, avg)
11 ALLOC ARRAY work
12 MULT(0.2, work, avg)
13 COPY(work, center)
```

(b) Stream of tasks and allocations issued by the main loop.

```
1  # FUSED_ADD_MULT is a new
2  # task generated by Diffuse.
3  ALLOC ARRAY work
4  FUSED_ADD_MULT(
5    center,
6    north,
7    east,
8    west,
9    south,
10   0.2,
11   work
12 )
13 COPY(work, center)
```

(c) Operation stream after Diffuse's optimization.



(d) 4-node distributed execution, where colors denotes the cells held by each node.

```
1  # ADD, MULT and COPY are
2  # elementwise operators.
3  def ADD(a, b, c):
4    for i, j in a:
5      c[i, j] = a[i, j] + b[i, j]
6  def MULT(s, a, b):
7    for i, j in a:
8      b[i, j] = s * a[i, j]
9  def COPY(a, b):
10   for i, j in a:
11     b[i, j] = a[i, j]
```

(e) Tasks invoked during standard execution.

```
1  # FUSED_ADD_MULT performs
2  # the scaled five-way add.
3  def FUSED_ADD_MULT(
4    a, b, c, d, e, s, out
5  ):
6    for i, j in a:
7      out[i, j] = s * (
8          a[i, j] + b[i, j]
9        + c[i, j] + d[i, j]
10       + e[i, j])
```

(f) Fused task generated by Diffuse.

Fig. 2. Execution example of Diffuse on a distributed, multi-GPU cuNumeric 5-point stencil application.

cuNumeric is a drop-in replacement for NumPy [Harris et al. 2020] that scales unmodified NumPy programs to distributed machines by targeting the Legion [Bauer et al. 2012] runtime system. cuNumeric maps NumPy arrays to Legion's regions, and maps individual NumPy operations to distributed task launches over regions that are partitioned across the machine. The program execution on a four-by-four grid with four nodes is visualized in Figure 2d, where each node owns an element of each aliasing view of `grid`, and the dotted arrows represent communication required to propagate updates to `center` to the other aliasing views of `grid`. Figure 2b is a simplified representation of the task stream that cuNumeric issues during execution of the inner loop (lines 10-14 of Figure 2a), and Figure 2e contains pseudocode for each of the task implementations. This stream of operations involves the creation of multiple temporary distributed arrays for the results of individual operations, and separate tasks for each corresponding addition and multiplication. Diffuse optimizes this stream of operations to create a new fused task that performs the computation of the `work` array (lines 11-13) into a single operation and removes the temporary arrays, including `avg`, resulting in the stream of operations in Figure 2c and generated fused task in Figure 2f. However, Diffuse does not also fuse the task that performs `center[:] = work` (line 14 of Figure 2a).

To understand these decisions, we must introduce the distributed aspect of the tasks and data collections in Figure 2b. Each task in Figure 2b represents a group of parallel tasks launched over arrays that are partitioned across the machine, where each parallel task operates on a subset of

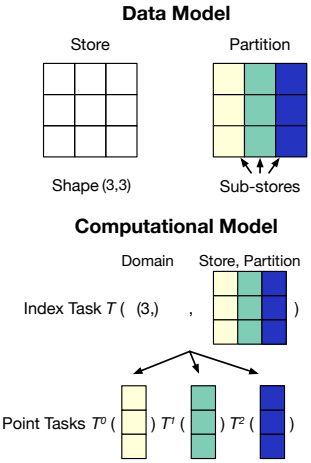
Syntax

Unique ID	id		
Point	p	::=	$(\mathbb{Z}, \ldots)$
Store	S	::=	Store(id, p)
Projection Function	F	::=	Projection$(id, \text{Point} \rightarrow \text{Point})$
Partition	P	::=	None | Tiling(s, s, F)
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) Diffuse's intermediate representation.



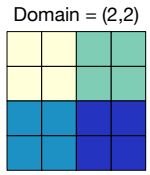
(b) Relationships between components of Diffuse's IR.

Fig. 3. Diffuse's intermediate representation. Diffuse exposes a data model for representing distributed data, a model for performing distributed computation on distributed data.

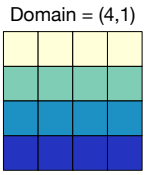
the partitioned data. Dependencies and communication that arise from parallel tasks operating on the same distributed data affect when fusion is possible. In our example, the arrays `center`, `north`, `east`, `west`, and `south` are aliasing views of the same array `grid`. Because these distributed arrays alias, Diffuse cannot fuse the task group that computes `center[:] = work` into the task group that reads from `north`, `east`, `west` and `south`, as the fusion would create a task group that concurrently reads and writes to aliasing data. Similarly, the `center[:] = work` task group issued at iteration i cannot be fused into the `avg` computation (line 11 of Figure 2a) at iteration $i+1$ because communication is required to propagate updates to `center` to arrays that alias with center. To reason about distributed computations over partitioned data, we develop a scale-free intermediate representation (Section 3) that models tasking runtime systems which support aliased views of distributed data. We then develop an algorithm for task fusion (Section 4) that reasons about communication in distributed computations to safely fuse groups of parallel tasks.

3 INTERMEDIATE REPRESENTATION

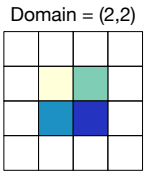
The first contribution of Diffuse is an IR and system organization that enables scalable fusion analyses through a *scale-free* representation of distributed programs, meaning that the size of the representation is not related to the total number of processors in the target system. Diffuse's IR is an abstraction over collections of concrete tasks and distributed data structures of a lower-level programming system like Legion, which usually have *scale-aware* representations. Instead of targeting Legion directly, cuNumeric and Legate Sparse target Diffuse's IR, which is then lowered to Legion after a series of optimizations. Diffuse's IR, presented in Figure 3, is designed to make the analyses required for fusion inexpensive, while still being able to express sophisticated computations. Diffuse's IR contains a data model to represent distributed data, and a computational model to define distributed computations over distributed data. The syntax of the IR is in Figure 3a, and a visualization of the IR's structure is shown in Figure 3b.



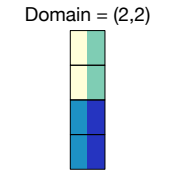
(a) 2x2 tiling of a 4x4 store.



(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.

Fig. 4. Examples of Tiling partitions. Each partition maps points in the denoted domain to sub-stores. Each color refers to the sub-store associated with a different point in the domain.

3.1 Data Model

Diffuse represents distributed data as *stores*, which are distributed arrays. Stores have a unique ID, as well as a rectangular shape defined by a tuple of non-negative integers, representing the upper bound of each dimension of the store. We refer to these rectangular shapes as *domains*, which are also used to describe the decomposition of data and compute across processors. Stores are partitioned across the target machine into *sub-stores*, which are subsets of a store.

Partitions of stores are represented as first-class objects in Diffuse. 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 by Diffuse in a structured manner, breaking different kinds of mappings into different syntactic groups. For simplicity of presentation, we consider two kinds of partitions, which is sufficient to explore the analyses used in Diffuse. As we discuss later, more partitions kinds are possible to represent with no additional technical insights. The first partition kind None represents the replication of a store, where all points in the 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 4a shows a tiling of a two-dimensional store using 2x2 tiles over a 2x2 domain, while Figure 4b shows a row-based tiling (i.e., tiles of size 1x4) of the same store over a 4x1 domain. Figure 4c shows a partitioning of a subset of the store beginning at the point $(1, 1)$. Tiling partitions also contain a *projection function* which 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 partially replicated data. For example, Figure 4d shows a tiling of a vector over a two-dimensional domain that uses a projection function to discard 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 4e. 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.

The full IR in our implementation has more partition kinds, such as *images* [Treichler et al. 2016], which necessary for partitioning sparse data by relating the partitions of multiple stores. We consider this restricted case of two partition kinds, as it is sufficient to study Diffuse's analyses. The main requirement on partition kinds 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 next section.

To reason about each sub-store referenced by each point of a partition, we include an explicit $\mathsf{SubStore}(S, P, p)$ construct in the IR, 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 when more complex partitioning operators are introduced. Our later definitions assume that it is possible to find the intersection ($\cap$) between two sub-stores, but our algorithm for fusion (Section 4) does not require computation of these intersections explicitly.

3.2 Computational Model

Diffuse models computation as a stream of *index tasks*, where the order of index tasks in the stream is the original program order. An $\mathsf{IndexTask}(d, A)$ represents a group of parallel tasks over points in a rectangular *launch domain* d that operate on the list A of stores, partitions and privileges, using the denoted privilege to access the requested partition of each store. As each index task represents a group of parallel tasks, each parallel task within the group correspondingly reads from, writes to, or reduces to the sub-stores referred to by the stores and partitions at each point. For the simplicity of presentation, we assume that the Reduce privilege refers to a single reduction function being applied (such as addition). The Read-Write privilege means that an index task will both read and write to a particular store through the chosen partition. Every store and partition pair may be given at most one privilege in A. 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 does not increase as the degree of parallelism increases, only the symbolic size of the launch domain increases.

Similar to the referencing of sub-stores, Diffuse's IR has a notion of a *point task*, 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 stores $[(S[P, p], pr) : \forall\ (S, P, pr) \in A]$. Instead of operating on stores and partitions, point tasks operate on the sub-stores corresponding to their index point.

We define the predicates $\mathsf{reads}(T, (S, P))$, $\mathsf{writes}(T, (S, P))$ and $\mathsf{reduces}(T, (S, P))$ 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 $\mathsf{reads}(T, (S, P))$ and $\mathsf{writes}(T, (S, P))$ are true. We also overload these predicates for point tasks and sub-stores, where $\mathsf{reads}(T^p, S)$ is true when point task T^p reads sub-store S, and similarly for writes and reduces.

The dynamic semantics of Diffuse's IR are defined by a translation to an underlying task-based runtime system. Every store in a Diffuse program is mapped to one of the underlying runtime system's distributed data structures, and Diffuse's first-class, structured partitions are mapped onto lower-level, unstructured partitions. In our implementation, we map stores and Diffuse's partitions to Legion's logical regions and partitions [Bauer et al. 2012]. Finally, index tasks are translated to tasks in the lower-level runtime system and issued for execution.

3.3 Legality of Task Streams

The main responsibility of a client library of Diffuse is to ensure that the issued stream of tasks is *legal*, and all analyses done by Diffuse ensure that the resulting stream is also legal. The legality

conditions arise from restrictions that exist in lower-level tasking runtime systems. The conditions are easily satisfied by tasks written by library developers, but care must be taken by an automatic fusion algorithm to maintain legality of fused tasks.

Definition 1. A stream of tasks $[T_1, \ldots, T_n]$ is *legal* if all index tasks $T_1, \ldots, T_n$ are *non-interfering*.

An index task is *non-interfering* if does not perform reads and writes to aliased data. This means that no two parallel tasks may either read and write or both write to the same piece of distributed data, and no point task may do the same for different aliasing sub-stores. Additionally, the index task must not read or write to data being reduced to; parallel reductions to aliasing data are permitted. Non-interference is similar to race-freedom between all parallel point tasks of an index task.

Definition 2. An index task T is *non-interfering* if the following three conditions hold, where S refers to a store, and P, P' refer to partitions:

write-diff-point: $\forall p \in \text{domain}(T), \nexists p' \in \text{domain}(T), S, P, P'$ such that $p \neq p'$ and $\text{writes}(T, (S, P)) \wedge (\text{reads}(T, (S, P')) \vee \text{writes}(T, (S, P'))) \wedge S[P, p] \cap S[P', p'] \neq \emptyset$

write-same-point: $\nexists p \in \text{domain}(T), S, P, P'$ such that $P \neq P'$ and $\text{writes}(T, (S, P)) \wedge (\text{reads}(T, (S, P')) \vee \text{writes}(T, (S, P'))) \wedge S[P, p] \cap S[P', p'] \neq \emptyset$

reduce-alias: $\forall p \in \text{domain}(T), \nexists p' \in \text{domain}(T), S, P, P'$ such that $\text{reduces}(T, (S, P)) \wedge (\text{reads}(T, (S, P')) \vee \text{writes}(T, (S, P'))) \wedge S[P, p] \cap S[P', p'] \neq \emptyset$

4 DISTRIBUTED TASK FUSION

In the previous section we described Diffuse's IR for distributed computation. We now discuss how Diffuse leverages properties of this IR to perform analyses and transformations to fuse distributed computations together. Successful task fusion then enables fusion of the kernels within fused tasks (Section 6), yielding end-to-end speedups through reduced runtime overheads and faster kernels.

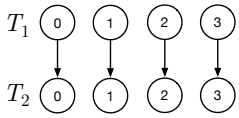
During program execution the stream of index tasks is represented by a current *window* of consecutive index tasks that have been issued by the program but not yet submitted for execution to the underlying runtime. An algorithm for distributed task fusion must find a prefix of index tasks within the window W that are possible to fuse, and to construct a fused index task from the prefix. Intuitively, the prefix of index tasks must be able to be executed in sequence without any communication between point tasks for fusion to be legal. Communication between point tasks blocks fusion, as point tasks running on different processors that must synchronize with each other cannot fused together. We define when communication may occur between index tasks and then describe when distributed task fusion is legal. We then give an algorithm for finding sequences of index tasks that satisfy the legality condition.

4.1 Dependencies

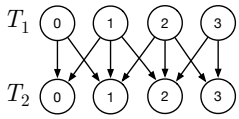
Communication is required between point tasks that have a dependence between them—the dependence implies synchronization and potentially data movement between the point tasks. Dependencies exist between any two point tasks that access the same data unless both task only read from or reduce to the data. Recall that for an index task T, we refer to the point task at point p as T^p. We define $\text{depends}(T_1^p, T_2^{p'})$ to be true if $T_2^{p'}$ depends on T_1^p. The definition of dependence is similar to non-interference, but instead relates point tasks from two different index tasks, rather than point tasks within a single index task.

Definition 3. Given point tasks T_1^p, $T_2^{p'}$ where index task T_1 is issued before index task T_2, $\text{depends}(T_1^p, T_2^{p'})$ if $\exists$ sub-stores S, S' with the same parent such that $S \cap S' \neq \emptyset$ and either

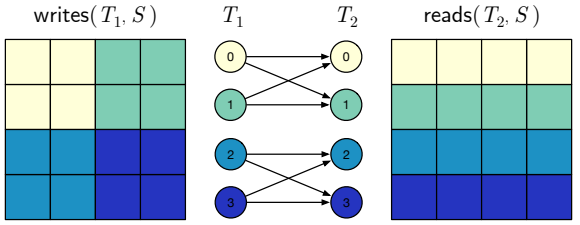
forward-dep: $\text{writes}(T_1^p, S) \wedge \Big(\text{reads}(T_2^{p'}, S') \vee \text{writes}(T_2^{p'}, S') \vee \text{reduces}(T_2^{p'}, S')\Big)$, or



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



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



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

Fig. 5. Visualization of different dependence maps $\mathcal{D}(T_1, T_2)$.

anti-dep: $\text{reads}(T_1^p, S) \wedge \left(\text{writes}(T_2^{p'}, S') \vee \text{reduces}(T_2^{p'}, S')\right)$, or

reduction-dep: $\text{reduces}(T_1^p, S) \wedge \left(\text{reads}(T_2^{p'}, S') \vee \text{writes}(T_2^{p'}, S')\right)$.

The dependencies between two index tasks T_1 and T_2 are defined by the pairwise dependencies between their component point tasks. We capture these dependencies through a map between the points of T_1 and T_2, representing all of the point tasks in T_2 that depend on point tasks in T_1. Figure 5 is an example of different dependence maps between index tasks over the launch domain $(4,)$.

Definition 4. $\mathcal{D}(T_1, T_2)$ is a *dependence map* between index tasks T_1 and T_2 where $\forall p \in \text{domain}(T_1)$, $\mathcal{D}(T_1, T_2)[p] = \{p' \in \text{domain}(T_2) : \text{depends}(T_1^p, T_2^{p'})\}$.

Having characterized the dependencies between two distributed index tasks T_1 and T_2, we can now define exactly 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 runs each point task does not need to communicate with any other processors. Precisely,

Definition 5. Index tasks T_1 and T_2 can be fused if $\forall p, \mathcal{D}(T_1, T_2)[p] \subseteq \{p\}$.

Fully computing and materializing $\mathcal{D}(T_1, T_2)$ to check this condition is an operation that scales with the number of processors. Even runtime 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 [Bauer et al. 2023]. However, a key insight in our work is that to perform distributed task fusion effectively, our analysis only needs to rule out cases where $\exists p, \mathcal{D}(T_1, T_2)[p] \not\subseteq \{p\}$. Diffuse's intermediate representation enables this analysis to be performed cheaply in a scale-free manner. Finally, the machinery developed here allows us to precisely define when fusion of index tasks is possible, which in turn allows us to formally show that our fusion algorithm is correct.

4.2 Algorithm

Our algorithm for performing distributed task fusion identifies when index tasks have point-wise dependencies through greedy application of a set of *fusion constraints* to identify a fusable 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 proof of correctness of our algorithm in the next section.

$$\text{launch-domain-equivalence}([T_1, \ldots, T_n]) = \forall i, \text{domain}(T_i) = \text{domain}(T_1)$$

$$\text{producer-consumer}([T_1, \ldots, T_n]) = \forall\, T_i \text{ s.t. } \text{writes}(T_i, (S, P)), \nexists T_j \text{ s.t. } \left(\text{reads}(T_j, (S, P')) \vee \text{writes}(T_j, (S, P'))\right) \wedge i < j \wedge P \neq P'$$

$$\text{anti-dependence}([T_1, \ldots, T_n]) = \forall T_i \text{ s.t. } \text{reads}(T_i, (S, P)), \nexists T_j \text{ s.t. } \text{writes}(T_j, (S, P')) \wedge i < j \wedge P \neq P'$$

$$\text{reduction}([T_1, \ldots, T_n]) = \forall T_i \text{ s.t. } \text{reduces}(T_i, (S, P)), \nexists T_j \text{ s.t. } \left(\text{reads}(T_j, (S, P')) \vee \text{writes}(T_j, (S, P'))\right) \wedge i \neq j$$

Fig. 6. Fusion constraints employed by Diffuse to identify potential communication between index tasks.

4.2.1 Fusion Constraints. Diffuse employs four constraints to identify when potential communication may occur between distributed index tasks, i.e. when $\exists p, \mathcal{D}(T_1, T_2)[p] \not\subseteq \{p\}$. The launch-domain-equivalence and producer-consumer constraints have been described at a high level by prior work [Sundram et al. 2022]. We generalize these constraints from prior work, present formal definitions and prove the correctness of our fusion algorithm. Diffuse's fusion constraints are sound, but not complete—for example, leveraging application knowledge could result in fusion opportunities out of scope for Diffuse. Figure 6 presents each of the constraints used by Diffuse by defining when a provided sequence of tasks satisfy the constraint.

Launch Domain Equivalence Constraint. The first, and simplest, constraint checks that the launch domain of each potential task to be fused is the same. This constraint ensures that $\mathcal{D}$ only maps between points of the same dimensionality. Applications targeting Diffuse's IR may decompose their computations in different ways, resulting in issued index tasks over different launch domains. Different computational decompositions generally require data movement between the views of the data, and the launch domain equivalence constraint catches this situation early.

Producer-Consumer Dependence Constraint. The next constraint used by Diffuse utilizes the partitions of stores and privileges with which they are accessed by index tasks to identify potential sources of communication between index tasks. The producer-consumer constraints presented in Figure 6 check that stores that are written to in a particular distributed view of the data are not operated on in a different distributed view of the data. Operating on two different distributed views of the same underlying data requires communication to occur between processors to switch between the views. Precisely, the producer-consumer constraint checks that if a task T_i writes to a store S through partition P, then there cannot be any task T_j after T_i that reads or writes to a S with an aliasing partition P'. However, operating on the same partition P is permitted, as the operation preserves point-wise dependencies between T_i and T_j.

Our analysis relies on the ability to check whether two partitions alias, which Diffuse does through a constant-time equality check between partitions. The scale-free structure of Diffuse's IR and the syntactic grouping of partitions into structured kinds enables constant-time alias checking. Diffuse does not need to compute pair-wise 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 aliasing analysis does not depend on the structure of the partitions themselves, as the constraints are defined without knowing whether partitions are Tiling or not. Finally, this aliasing check is not too coarse, since partitions of different syntactic kinds are likely to alias.

Anti-Dependence Constraint. The anti-dependence constraint ensures that $\mathcal{D}$ does not contain write-after-read dependencies across different points in different distributed index tasks and that the resulting fused task maintains the non-interference property (Definition 2) necessary for task legality. The constraint defined in Figure 6 ensures that if a task T_i reads a store, then any write operations to that store must be to the same distributed view as the read. As a consequence, a fused task may read from multiple different distributed views of a store (like the multiple offset views in a stencil computation, as in Figure 2a), but then cannot write to any one of the views, as such an operation would require communication of the written data.

Reduction Constraint. The reduction constraint is the final fusion constraint, making sure that the resulting fused task is still legal around reduction operations. It does not permit a task that reads or writes a store to be fused with a task performing a reduction to any view of the same store.

4.2.2 Using The Constraints. Our algorithm for distributed task fusion greedily applies these fusion constraints on the input task window to find the longest fusable prefix of the task window. Verifying the launch domain equivalence and reduction constraints is straightforward. The producer-consumer and anti-dependence constraints are verified through a forwards dataflow analysis on a candidate prefix of index tasks. The analyses iterate through the candidate prefix of tasks, and track the effects that each task applies to its argument stores.

4.2.3 Fused Task Construction. Once a suitable prefix of the input task window has been identified by the fusion constraints, Diffuse builds a fused task out of all index tasks within the prefix. Building a fused task involves creating a new task that both has all of the same store arguments and executes the same computation as the original sequence 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. Diffuse constructs the body of the fused task by composing the bodies of each task in the prefix in program order—we discuss this process more in Section 6.

4.3 Proof of Correctness

We have described our algorithm to identify tasks that can be fused and the process to create a new fused task to replace the selected prefix of tasks. In this section we prove that these algorithms correctly fuse sequences of distributed index tasks. Concretely, 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 fusable, and
(2) F is a legal task, and
(3) 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 fusable, 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 $\text{depends}(T_i^p, T_j^{p'})$. Then one of the three dependencies in Definition 3 must exist. Suppose the first dependence is true, meaning that $\exists S, P, P'$ such that $S[P, p] \cap S[P', p']$, $\text{reads}(T_i, (S, P))$ and either $\text{writes}(T_j, (S, P'))$ or $\text{reduces}(T_j, (S, P'))$. $\text{reduces}(T_j, (S, P'))$ is a contradiction, as the reduction constraint would disallow fusion. If $\text{writes}(T_j, (S, P'))$, if $P = P'$ then no sub-stores of (S, P) may alias, as T_j is non-interfering (Definition 2). Thus, p must equal p', a contradiction. If $P \neq P'$, then the anti-dependence constraint disallows fusion, a contradiction. The reasoning for

the other two kinds of dependencies is analogous. Here, we show that our algorithm is sound by identifying cases where fusion is possible—we do not claim completeness by proving the converse.

F can be shown to be non-interfering through contradiction in a similar way as the proof that $[T_1, \ldots, T_f]$ are fusable by leveraging the fusion constraints and initial legality assumptions.

We have already 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.

4.4 Discussion

We argue that fusion at Diffuse's intermediate layer of abstraction is key for the definition of domain-agnostic analyses that apply to a broad set of programs, and for scalability of the analysis as the size of the target machine increases. We compare against fusion on high-level domain-specific libraries, and then against fusion at a lower-level, such as within a runtime system like Legion.

Fusion on individual distributed libraries has shown to be effective in improving performance through various domain-specific algorithms [Abadi et al. 2016; Bradbury et al. 2018; Unger et al. 2022; Yadav et al. 2022a,b]. Approaches that perform fusion on a specific library (or set of domain-specific computations), nearly always use algorithms and analyses that are tied to the domain of computations being optimized, especially around analyses related to distributed memory. As a result, these techniques are difficult to generalize across libraries. Instead, Diffuse targets fusion and optimization in the more general case after computations have already been decomposed into tasks in a domain-specific manner, enabling domain-agnostic analyses to find optimization opportunities across function and library boundaries. We expect that domain-specific techniques may be used in conjunction with the analyses performed by Diffuse, rather than having to pick one or the other.

While generality is lost when fusing operations within individual libraries, scalability becomes a concern when analyzing lower-level program representations. As discussed above, a key design decision in Diffuse's intermediate representation is that it is scale-free, as the size of the parallel task groups and the representation of partitions of distributed data are independent of the degree of parallelism used. This design enables Diffuse to understand aliasing relationships (at a coarse level) between distributed data structures through constant-time queries, which are heavily used when defining the fusion constraints in Figure 6. In contrast, lower-level systems like Legion represent partitioned data through representations that explicitly map points to arbitrary sets of indices into the distributed data. These mappings do not compress away the degree of parallelism, and the size of the data structures scale with the number of pieces data is partitioned into. These representations are more flexible than Diffuse's, but result in the aliasing relationship queries needed by a fusion algorithm to scale with the degree of available parallelism, rather than running in constant time.

5 TASK FUSION OPTIMIZATIONS

We have described our algorithm for task fusion, and proved that the algorithm only fused tasks with point-wise dependencies, and preserves the legality of the input stream of tasks. We now describe optimizations that are key to a practical implementation of fusion. We present an algorithm to eliminate temporary distributed data structures (Section 5.1) and technique to memoize the fusion analysis (Section 5.2). Temporary elimination and memoization of analyses are widely applied optimizations; we discuss how to perform these optimizations in a distributed, task-based setting.

5.1 Temporary Store Elimination

Once Diffuse has identified a prefix of tasks to fuse, there is opportunity to eliminate stores that fusion has made temporary by promoting temporary stores into task-local allocations. Conversion

```
1 import cunumeric 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) cuNumeric code fragment with temporaries.

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

(b) Emitted task stream.

Fig. 7. Example of distributed temporaries.

of distributed data into task-local allocations is critical for realizing the benefits of fusion, as task-local allocations can then be optimized away (Section 6) to maximize data reuse in caches and registers. Intuitively, a store is temporary in the fusion of a list of tasks if any effects performed on the store by the list of tasks are not visible after the execution of the list of tasks, either to the application or any pending tasks yet to execute. As with identification of fusion in distributed settings, Diffuse's IR enables efficient identification of temporary data.

To introduce the constraints describing when a store is temporary, consider the cuNumeric program in Figure 7a and resulting task stream in Figure 7b. This example introduces some new operations, specifically `flush_window`, which sends all pending tasks through Diffuse to the underlying runtime system, and the Python `del` operator, which drops references to stores. The program performs arithmetic computation and creates the stores `x`, `y`, `z`, `w`, and `v`. Consider the state of the program after line 9 is executed. After line 9, the tasks that initialize `x` and `y` have already been executed, as the first `flush_window` call already sent those tasks to Diffuse. The fusion algorithm determines that the tasks issued by lines 4–6 can be fused, while the final `norm` must be excluded, failing the producer-consumer constraint on the read of the slice of `w`. We now consider which stores are temporary. First, `v` is not temporary because the application still maintains a reference to it, meaning that it could request to read `v` with a future task. 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. Now consider the stores `x` and `y`. The fused task reads from but does not write to either store; since the data read from `x` and `y` in the fused task was produced by prior tasks, these stores are also not temporary. Only `z` is temporary because it is produced entirely within the fused task, is not visible to any pending tasks, and has no remaining application references. We formalize this intuition as constraints that must be satisfied to determine that a store is temporary.

Definition 6. Given tasks $[T_1, \ldots, T_f, \ldots, T_n]$ where tasks $[T_1, \ldots, T_f]$ are being fused, a store S is *temporary* in the fusion of $[T_1, \ldots, T_f]$ if

(1) If $\exists\, T_j, P$ such that $\text{reads}(T_j, (S, P))$, $\exists\, T_i$ such that $i < j$, $\text{writes}(T_i, (S, P))$, and $\text{covers}(S, P)$,
(2) $\nexists\, T_k, P$ such that $k > f$ and $\text{reads}(T_k, (S, P))$ or $\text{reduces}(T_k, (S, P))$[1], and
(3) S has no live application references.

The function $\text{covers}(S, P)$ is true when the partition P contains all points in the store S, and can be defined for each kind of partition in Diffuse's IR. 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

[1]This constraint can be relaxed to allow for a pending write of S before any a read or reduction, as the fused data is no longer visible to pending tasks.

```
T1([(S1, R), (S2, W)])
T2([(S2, R), (S1, W)])
T3([(S1, R), (S3, W)])
T4([(S3, R), (S1, W)])
```

```
T1([(S5, R), (S6, W)])
T2([(S6, R), (S5, W)])
T3([(S5, R), (S7, W)])
T4([(S7, R), (S5, W)])
```

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

(a) Isomorphic streams and canonical representation.

```
T1([(S5, R), (S6, W)])
T2([(S6, R), (S5, W)])
T3([(S7, R), (S7, W)])
T4([(S7, R), (S5, W)])
```

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

(b) Differing stream and canonical representation.

Fig. 8. Canonical representations of isomorphic and non-isomorphic task streams.

a tiered reference counting scheme in the implementation of Diffuse's IR. The tiered reference counting schemes separates references held by the application from references held by Diffuse's runtime. Once a store is deemed temporary, it is demoted from a distributed allocation into a task-local allocation, as described in Section 6.

5.2 Memoization of Analyses

The final component of our distributed task fusion pipeline is a technique to memoize the performed analyses and code generation (Section 6) when fusing a task. The key challenge in memoizing these analyses is developing a strategy for memoization so that analysis on streams of tasks may be replayed on *isomorphic* task streams, rather than just identical task streams. Consider the streams of tasks in upper portion of Figure 8, where partitions and launch domains are excluded. We let R and W represent the Read and Write privileges respectively.

Diffuse may reuse the analysis results from the stream in the left part of Figure 8a on the right stream in Figure 8a, as the pattern of stores among tasks is isomorphic. In contrast, the task stream in Figure 8b 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 Diffuse through a conversion to and comparison on a canonical, De-Brujin index-like representation. This representation is shown in the lower part of Figure 8a and Figure 8b. This memoization strategy allows Diffuse to reuse analysis results when the same sequences of tasks are executed repeatedly, as common in scientific applications. This technique has been previously used to avoid enumerating instruction sequences equivalent up to register renaming [Bansal and Aiken 2006].

6 KERNEL FUSION

We have shown how Diffuse identifies sequences of index tasks to fuse, and introduced optimizations to lower memory usage and cache analysis. The final component of Diffuse is a compilation stack to fuse and perform optimizations on the bodies of 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. Instead of defining a new intermediate representation for the computations within tasks, we leverage the existing 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 within Diffuse.

6.1 MLIR Background

We leverage the MLIR compiler infrastructure [Lattner et al. 2020] to build a JIT compiler for Diffuse. MLIR is an extension of LLVM [Lattner and Adve 2004] that aims to provide compiler infrastructure for program analyses on higher-level languages than assembly-like languages. The

```
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 task visible to Diffuse.

```
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.

Fig. 9. Diffuse generator API and generated MLIR fragment.

most relevant component of this infrastructure to this 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. Diffuse's MLIR-based compiler is no different, and leverages community-developed dialects and passes to optimize task bodies and lower those task bodies into executable multi-threaded CPU and GPU code.

6.2 Compilation Pipeline

To describe Diffuse's compiler, we walk through the pipeline stages that a fused task traverses before becoming executable code. The standard way for library developers to define tasks in cuNumeric and Legate Sparse is to implement variants that target CPUs or GPUs. Before a potential task for fusion enters the compilation pipeline, it must have a *generator* implementation registered with Diffuse, where the generator function returns an MLIR fragment describing the computation the task will perform. We found the integration effort of adding these generator functions to be minimal, requiring 50-100 lines of C++ code per operation. An example of this API, and a generated MLIR fragment by cuNumeric for an element-wise addition operation is shown in Figure 9.

The generated MLIR fragment in Figure 9b contains multiple dialects: 1) stores are mapped onto the `memref` dialect, which provides stronger aliasing guarantees than raw pointer accesses; 2) dense iteration is mapped onto the `affine` dialect, which is amenable to polyhedral compilation techniques [Bondhugula 2020]; 3) the computation itself is mapped onto the `arith` dialect, containing core arithmetic operations. Using the flexibility of MLIR's dialects, future work could explore using other dialects to express and fuse kernels, such as the Weld IR [Palkar et al. 2017].

When Diffuse identifies that a sequence of tasks may be fused together, it invokes the generator function for each task, and constructs an MLIR module containing the body of each task in the original program order. Figure 10a shows a fused task for the cuNumeric 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 9b) appear sequentially in the fused task. Before optimization of the task body, Diffuse first promotes distributed data into task-local allocations, resulting in Figure 10b.

After temporary stores have been eliminated, Diffuse applies a series of target-processor agnostic optimization passes. We leverage community-developed MLIR passes, applying passes that fuse nested loops together, remove task-local temporary allocations, and parallelize nested loops. The optimized code is shown in Figure 10c. The generated kernel is the ideal implementation for the

```
func.func @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] }
}
```

(a) Initial body of fused task.

```
func.func @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] }
}
```

(b) After temporary elimination.

```
func.func @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] }
}
```

(c) Fully optimized fused task.

Fig. 10. 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 9b.

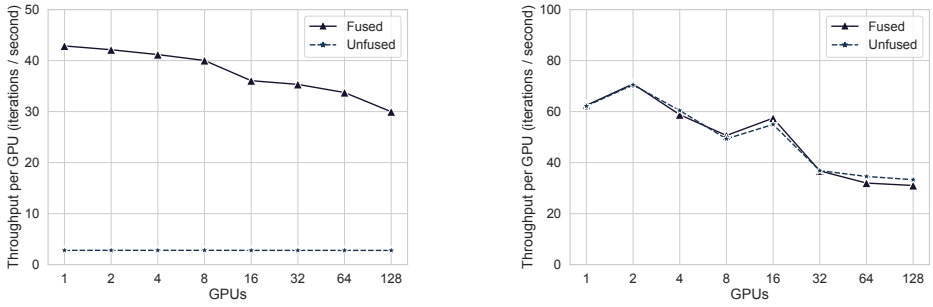
original program `c = a + b; e = c + d`: the two separate loops of the original task bodies have been combined into a single pass over the vectors, and the temporary `c` has been eliminated, storing the temporary result of each iteration in a register. The optimized kernel is then lowered through target-processor specific passes to executable code. Parallel loops are lowered to either GPU kernel launches or OpenMP parallel regions, depending on the target processor.

6.3 Qualitative Benefits

Diffuse factors reasoning about distributed computation into analyses on streams of tasks, and analyses on nested loops executing on a single processor. We note several qualitative benefits of our system architecture in contrast to approaches that attempt to perform analysis of distributed programs entirely through analysis of imperative code. A key design decision of Diffuse is to leverage a distributed data model in a scale-free IR of computation that enables cheap dependence analysis between distributed computations. Separating the reasoning about distributed computation and nested loop transformations enables each component of Diffuse to have an independent implementation, without intertwining loop optimizations with distributed communication analyses. This separation also allows for information gained during the distributed analysis phase to be used in code generation: inferred properties such as array non-aliasing are provided to and leveraged by the MLIR optimization passes to generate better code. Finally, the separation of distributed computation into tasks means that Diffuse can focus on optimizing the provided tasks instead of having first identify optimizable program fragments.

7 EVALUATION

Experimental Setup. We evaluate the performance of Diffuse on a cluster of NVIDIA A100 DGX SuperPOD nodes. Each 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, one for each socket. For each experiment, we perform a weak-scaling study, where the problem size per processor stays constant as the total number of processors increases and we report the throughput achieved per processor. Each reported value is the result of performing 12 runs, dropping the fastest and slowest



(a) Black-Scholes Option Pricing (b) Jacobi Iteration

Fig. 11. Microbenchmarks.

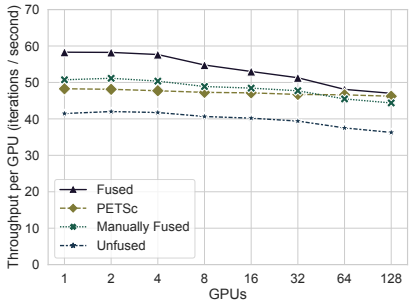
runs, and then computing the average of the remaining 10 runs. In each run of the weak-scaling experiments (Section 7.1), we exclude a set of warmup iterations from timing to measure the steady-state performance with and without Diffuse. We separately evaluate the overhead that Diffuse imposes due to compilation in Section 7.2.

Overview. We evaluate Diffuse on unmodified cuNumeric and Legate Sparse applications, from micro-benchmarks to full applications. These benchmarks come from different areas of scientific computing, including iterative solvers and physical simulations. We compare the performance of these applications when run without fusion and when run with Diffuse — no changes to the application are needed to enable Diffuse. For some applications, we compare against manually optimized implementations by the original authors, and against the industry-standard PETSc [Balay et al. 2022] library for distributed sparse linear algebra, which is an explicitly parallel MPI library. We show that when fusion opportunities are available, Diffuse can exploit these opportunities to find speedups in unmodified, distributed applications. Diffuse enables high-level programs to equal, and in many cases improve on, the performance of hand-optimized code. We do not perform an ablation study on the optimizations discussed in Section 5, as elimination of temporaries is a mandatory optimization for achieving speedup with kernel fusion and memoization is a requirement for a practical implementation, otherwise compilation times would dominate execution.

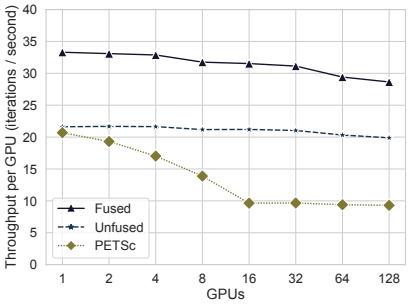
7.1 Weak Scaling Experiments

Black-Scholes. The Black-Scholes option pricing benchmark is a trivially-parallel micro-benchmark that contains a sequence of 67 data-parallel, and thus fusable, operations. It is a micro-benchmark that provides a reference point on potential improvement when the entire application is amenable to fusion. Figure 11a shows that Diffuse achieves a 10.7x speedup over the unfused program on 128 GPUs, as the fused program evaluates to a single task launch of a single GPU kernel that makes one pass over the data, greatly increasing the arithmetic intensity of the computation.

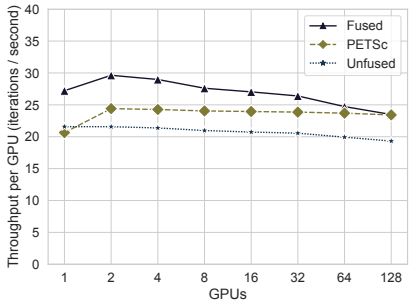
Dense Jacobi Iteration. In contrast to Black-Scholes, dense Jacobi iteration has negligible potential benefit from fusion. The inner loop of Jacobi iteration consists of three tasks: a dense matrix-vector multiplication that dominates the runtime, and two fusable vector operations that are negligible in runtime. Therefore, the goal of this benchmark is to show that our analyses do not have a significant negative impact on performance when there is no fusion. Diffuse achieves 0.93-1.08x of the performance of the unfused Jacobi iteration in Figure 11b, where we believe the cases of slight improvement are due to experimental variability.



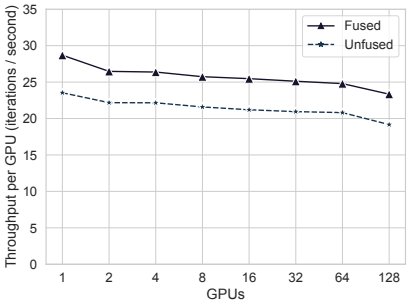
(a) Conjugate Gradient (CG)



(b) Conjugate Gradient Squared (CGS)



(c) BiConjugate Gradient Stabilized (BiCGSTAB)



(d) Geometric Multi-Grid Solver (GMG)

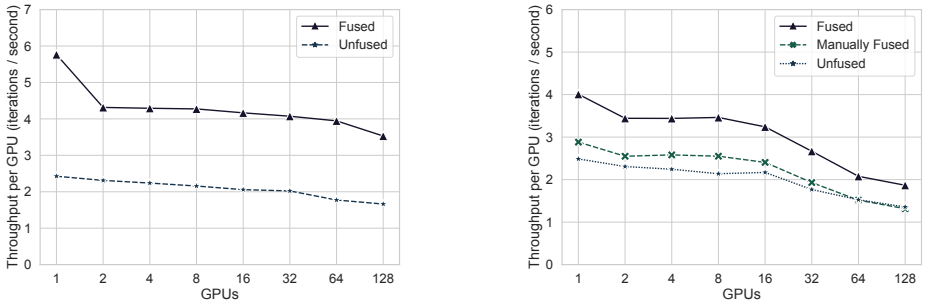
Fig. 12. Iterative linear solver benchmarks.

Krylov Subspace Solvers. We evaluate several sparse Krylov subspace algorithms using a combination of cuNumeric and Legate Sparse, namely Conjugate Gradient (CG), Conjugate Gradient Squared (CGS) and Bi-Conjugate Gradient Stabilized (BiCGSTAB). We first discuss CG before moving onto CGS and BiCGSTAB. To perform a controlled comparison against PETSc, we modify the implementation of 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.[2]

The original implementation of CG in Legate Sparse had been optimized manually to perform many of the optimizations that Diffuse does automatically. These optimizations improved performance, but the implementation no longer closely resembled the high-level, mathematical description of CG. In our experiments, we compare against this manually fused CG implementation as well as a naturally written CG implementation using cuNumeric and Legate Sparse. We also compare against PETSc, which utilizes fused operators. Figure 12a shows that Diffuse is able to automatically optimize the naturally written CG so that it runs faster than both the manually optimized version and PETSc. Through dynamic analysis of the task stream, Diffuse finds additional fusion opportunities across iteration boundaries by fusing together AXPY and dot-products from different iterations, and (specifically to Legate Sparse) eliminating copies performed by the identity pre-conditioner.

For the CGS and BiCGSTAB algorithms, we implemented an unfused version of each algorithm in cuNumeric and Legate Sparse and compared against PETSc. Figures 12b and 12c show that Diffuse can accelerate the high-level implementations of CGS and BiCGSTAB to outperform the unfused

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



(a) Computational Fluid Dynamics Simulation (CFD) (b) Shallow-Water Equation Solver (TorchSWE)

Fig. 13. Computational physics applications.

versions by 1.49x and 1.31x on average (geo-mean) and PETSc by 2.28x and 1.15x on average (geo-mean). We note that PETSc exposes several fused kernels to users for use in building iterative solvers, but these kernels can quickly become complicated and esoteric[3]. In contrast, Diffuse enables users to write high-level programs in cuNumeric and Legate Sparse and then automatically derives optimized kernels for efficient execution.

Geometric Multi-Grid Solver. Moving from smaller benchmarks to full applications, we apply Diffuse to the Geometric Multi-Grid (GMG) solver developed in Legate Sparse. The GMG solver is a CG-based iterative solver with a V-cycling pre-conditioning algorithm, the injection restriction operator, and a weighted Jacobi smoother. As with the previous benchmarks, using Diffuse 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 12d.

Computational Fluid Dynamics. We now move onto physics applications that only utilize cuNumeric. We apply Diffuse to the cuNumeric port of the final step of the Python CFD course [Barba and Forsyth 2019], a stencil code that solves the Navier-Stokes equations for 2D channel flow. The application performs a large number of element-wise operations on aliasing slices of distributed arrays to model the stencil-like interactions between simulated elements, exposing plenty of opportunities for fusion. Diffuse finds between 1.8x-2.3x speedup over the original cuNumeric implementation without Diffuse, as shown in Figure 13a. Note that Diffuse achieves higher speedup on a single GPU than on multiple GPUs. On a single GPU, data is not partitioned across multiple devices, enabling longer sequences of tasks to satisfy all fusion constraints, leading to more optimizations applied to the code. On multiple GPUs, stores are partitioned, causing the dependencies between aliasing views of the data to reduce the opportunities for fusion.

Shallow Water Equation Solver. Our final benchmark application is also our most complex: the cuNumeric port of the TorchSWE shallow-water equation solver [Chuang 2021]. We compare against the original cuNumeric port, as well as a version that the cuNumeric developers manually optimized through the use of `numpy.vectorize`. The `vectorize` utility JIT-compiles a user-defined element-wise operator for use on arrays, doing some of the optimizations that Diffuse performs automatically. Figure 13b shows the performance of TorchSWE with Diffuse compared to these baselines. We see that Diffuse achieves a 1.61x speedup on average (geo-mean) over the unfused version of TorchSWE, and a 1.35x speedup on average (geo-mean) over the manually vectorized

[3]Such as `VecAXPBYPCZ` in BiCGSTAB (https://petsc.org/main/manualpages/Vec/VecAXPBYPCZ/), performing a three way fused scale and multiply of vectors.

Benchmark	Standard (s)	Compiled (s)	Breakeven Iterations
Black-Scholes	0.38	0.06	N/A
Jacobi Iteration	0.53	0.43	N/A
Conjugate Gradient	0.67	1.30	99.44
Conjugate Gradient Squared	1.23	1.83	37.97
BiConjugate Gradient Stabilized	1.26	2.19	80.43
Geometric Multi-Grid Solver	0.49	1.38	118.75
Computational Fluid Dynamics	5.10	10.89	25.21
TorchSWE	0.97	8.82	43.88

Fig. 14. Warmup times of each application, with and without compilation.

version. Since Diffuse is analyzing the full stream of tasks issued by the application, it can find fusion opportunities missed by developers optimizing the program by hand.

7.2 Compilation Time

For each of the experiments, we measure that overhead that Diffuse's compilation imposes on the overall runtime of the program. When evaluating our benchmark applications, we run each for a set of warmup iterations, and then compute the achieved throughput of the application after the warmup iterations have concluded. To measure the effect of compilation, we measure this warmup time with and without compilation enabled. We then compute the number of iterations required for the fused version of the application to be faster than the normal version of the application including the warmup compilation time. The results are shown in Figure 14, where we see that Diffuse's compilation times are modest, requiring 25–119 iterations to amortize the cost of compilation. These costs are especially reasonable as the evaluated scientific applications would be run in production for thousands to millions of iterations.

8 RELATED WORK

Task Fusion. Task fusion is a widely applied technique in parallel and distributed systems to reduce the overheads of parallelism [Dask Authors 2023; Dyer 2013]. Most prior work considers the fusion of individual tasks—in this work, we consider a more complex variant of task fusion, the fusion of groups of distributed tasks, which is challenging due to the dependencies that may exist between distributed index tasks. The most related work to ours is that of Sundram et al. [Sundram et al. 2022], which identifies the problem and provides an initial solution for detecting when fusion of distributed task groups is possible. We improve on this prior work by developing a formal model for reasoning about distributed tasks, identifying new constraints on when fusion is possible and proving that the set is sufficient. Additionally, we pair task fusion with a JIT compiler to additionally fuse the task bodies, enabling our system to achieve significantly larger speedups than possible with just task fusion, as more potential benefits than runtime overhead removal are possible.

Kernel Fusion. Fusion of computation to improve locality and reduce redundancy has a long history. The fusion of nested loops within imperative, array-based programs is a well-studied problem, with various proposed solutions [Allen and Cocke 1971; Bondhugula 2008; Darte 1999; Kennedy and McKinley 1993]. Our work combines loop fusion approaches with the data and computational models of a tasking runtime to enable safe fusion of kernels in a distributed environment.

Kernel fusion has been explored heavily in different domains. Deforestation approaches aim to remove temporary lists and trees in functional programs [Wadler 1990]. Fusion of operations in collection-oriented, data-parallel languages has been explored to combine operations like map, reduce and filter into single passes over data structures [Brown et al. 2016; Chatterjee et al. 1991; Gill et al. 1993; Grust 2004; Westrick et al. 2022]. Various compilers have been developed to generate

fused code for operations over dense and sparse tensors [Chen et al. 2018; Kjolstad et al. 2017; Ragan-Kelley et al. 2013]. In machine learning, many frameworks perform operator fusion within neural networks [Bradbury et al. 2018; Jia et al. 2019; Niu et al. 2021; Sabne 2020]. These systems leverage domain knowledge to guide the desired fusion optimizations. In contrast, our work provides a domain-agnostic framework for identifying fusion in streams of distributed tasks, and could leverage techniques such as these for fusing the computations within tasks.

Efficient Composition of Parallel Software. A goal of Diffuse is provide a framework for efficiently composing operations within and across distributed libraries. Some recent projects have tackled this problem; we discuss each in turn. Weld [Palkar et al. 2017] provides a loop-based IR for users to define library computations in, and a runtime system that optimizes the IR to enable cross-function and cross-library optimizations. Split Annotations [Palkar and Zaharia 2019] provides annotations that users can attach to library functions to partition the computation done by the function. Then, Split Annotations uses the annotations to run cache-sized batches of user functions to maximize data reuse across function call boundaries. Both Weld and Split Annotations target a similar problem as Diffuse, but do not address the challenges of fusion in a distributed system. These systems would require a model of distributed data to safely perform optimizations in a distributed setting. DaCe [Ben-Nun et al. 2019] is a compiler that leverages an IR called Stateful Dataflow MultiGraphs to perform optimizations on Python/NumPy programs. Distributed programs in DaCe are explicitly parallel, including manual communication with libraries like MPI, which requires different kinds of analyses. Jax [Bradbury et al. 2018] is a runtime system and compiler for NumPy-like operations that leverages the semantics of NumPy operations to statically perform transformations like fusion and automatic differentiation. In contrast to Jax, Diffuse is not tied to the semantics of any particular library, and leverages dynamic analyses to gracefully handle data-dependent control flow.

Distributed Runtime Systems. Diffuse uses a scale-free representation of distributed computation and data to efficiently perform dependence and aliasing analyses needed to fuse distributed tasks. The representation is similar to Index Launches [Soi et al. 2021], a representation of distributed tasks that compresses the degree of parallelism. Diffuse's model of distributed data supports *content-based coherence*, meaning that the same data may be referred to in multiple different ways. Legion [Bauer et al. 2012], which Diffuse builds upon, is another system that supports content-based coherence of distributed data. Legion exposes a more general interface for partitioning data, allowing a partition to contain an arbitrary set of subsets. Legion then uses sophisticated algorithms for computing dependencies between tasks and maintaining coherence of distributed data [Bauer et al. 2023]. In contrast, Diffuse's restricted representation and relaxed requirements for fusion enable compact analyses for the dependence and coherence problems. In systems without content-based coherence, simpler approaches than ours may suffice, as aliasing distributed data is no longer a concern.

9 CONCLUSION

We introduce Diffuse, a system that performs dynamic task and kernel fusion on streams of distributed tasks, enabling optimizations that improve data reuse and remove allocations of distributed data structures in end user programs. Diffuse leverages a scale-free intermediate representation of distributed computation and data to perform these analyses in a scalable manner. These techniques enable Diffuse to compose computations in and across high-level libraries like cuNumeric and Legate Sparse, matching and in many cases exceeding the performance of hand-tuned code. We believe the techniques developed in Diffuse are the first steps towards a full suite of optimizations to efficiently compose distributed and accelerated software.

REFERENCES

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. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

Frances E Allen and John Cocke. 1971. A Catalogue of Optimizing Transformations. (1971).

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*. Springer, 863–874.

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

Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. *SIGOPS Oper. Syst. Rev.* 40, 5 (oct 2006), 394–403. https://doi.org/10.1145/1168917.1168906

Lorena Barba and Gilbert Forsyth. 2019. CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education* 2, 16 (2019), 21. https://doi.org/10.21105/jose.00021

Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. 2022. Pathways: Asynchronous distributed dataflow for ML. *Proceedings of Machine Learning and Systems* 4 (2022), 430–449.

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

Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. 2023. 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* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 218–231. https://doi.org/10.1145/3572848.3577515

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11. https://doi.org/10.1109/SC.2012.71

Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. https://doi.org/10.1145/3295500.3356173

Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* abs/2003.00532 (2020). arXiv:2003.00532 https://arxiv.org/abs/2003.00532

Uday Kumar Reddy Bondhugula. 2008. *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. Ph. D. Dissertation. USA. Advisor(s) Sadayappan, P. AAI3325799.

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

Kevin J. Brown, HyoukJoong Lee, Tiark Romp, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 194–205.

Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. 1991. Size and Access Inference for Data-Parallel Programs. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 130–144. https://doi.org/10.1145/113445.113457

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

Pi-Yueh Chuang. 2021. *TorchSWE: GPU shallow-water equation solver.*

Anthony Danalis, Heike Jagode, George Bosilca, and Jack Dongarra. 2015. Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 304–313.

A. Darte. 1999. On the complexity of loop fusion. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. 149–157. https://doi.org/10.1109/PACT.1999.807510

Dask Authors. 2023. *Dask Optimization*. Accessed: 2023-10-08.

Robert Dyer. 2013. Task Fusion: Improving Utilization of Multi-User Clusters. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity* (Indianapolis, Indiana, USA) *(SPLASH '13)*. Association for Computing Machinery, New York, NY, USA, 117–118. https://doi.org/10.1145/2508075.2514878

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

Torsten Grust. 2004. *Monad Comprehensions: A Versatile Representation for Queries*. Springer Berlin Heidelberg, Berlin, Heidelberg, 288–311. https://doi.org/10.1007/978-3-662-05372-0_12

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

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 301–320.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

Chris Lattner and Vikram Adve. 2004. 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* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020). arXiv:2002.11054 https://arxiv.org/abs/2002.11054

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3453483.3454083

Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A common runtime for high performance data analytics. (2017).

Shoumik Palkar and Matei Zaharia. 2019. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 291–305. https://doi.org/10.1145/3341301.3359652

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

Rupanshu Soi, Michael Bauer, Sean Treichler, Manolis Papadakis, Wonchan Lee, Patrick McCormick, Alex Aiken, and Elliott Slaughter. 2021. Index Launches: Scalable, Flexible Representation of Parallel Task Groups. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 66, 18 pages. https://doi.org/10.1145/3458817.3476175

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

Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent Partitioning. *SIGPLAN Not.* 51, 10 (oct 2016), 344–358. https://doi.org/10.1145/3022671.2984016

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. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 267–284. https://www.usenix.org/conference/osdi22/presentation/unger

Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. 2022. Parallel Block-Delayed Sequences. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 61–75. https://doi.org/10.1145/3503221.3508434

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

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

Rohan Yadav, Wonchan Lee, Melih Elibol, Taylor Lee Patti, Manolis Papadakis, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. 2023. Legate Sparse: Distributed Sparse Computing in Python. (2023).