

# Parallel and Accelerated Dependent Partitioning

Rohan Chanani  
Stanford University  
rchanani@stanford.edu

Alex Aiken  
Stanford University  
aaiken@stanford.edu

Michael Bauer  
NVIDIA  
mbauer@nvidia.com

Rohan Yadav  
Stanford University  
rohany@cs.stanford.edu

**Abstract**—Dependent partitioning is a flexible framework for compactly describing and efficiently computing partitions of distributed data structures that contain indirections and interconnected references. These data structures are commonly found in scientific applications, which operate over meshes, graphs, and sparse tensors. However, existing implementations of dependent partitioning process the individual elements of their input data sequentially on CPUs. This limitation makes dependent partitioning an expensive component of application initialization, and hinders its use in dynamic or on-line contexts. In this work, we present parallel algorithms for computing dependent partitions, and show that these algorithms perform well on accelerators like GPUs. Across a series of microbenchmarks and existing applications that use dependent partitioning, we show that our work achieves speedups of  $9.4\text{--}9,807\times$  for computing dependent partitions. We then show that our algorithms enable dynamic and on-line partitioning during steady-state execution.

**Index Terms**—dependent partitioning, runtime systems, GPUs, parallel algorithms, sparse data structures

## I. INTRODUCTION

Partitioning data structures is a fundamental requirement for executing large-scale computations on distributed memory machines. Partitioning schemes must optimize for locality, load balance, and minimal communication while simultaneously respecting crucial application invariants. In particular, many applications consist of interconnected data structures coupled together by indirections that capture the relationships between them. For example, graph-based applications often contain sets of nodes and edges with indirection arrays that store which nodes are connected by each edge, while sparse tensors leverage similar indirection arrays to represent the non-zero elements in each tensor. Therefore, when partitioning multiple data structures for distributed execution, it is imperative that applications respect these indirection relationships such that logically related data is partitioned consistently.

Traditionally, it has been the responsibility of the programmer to manually manage the correctness of partitions based on their indirection relationships (as in most MPI applications). While some recent systems provide first-class support for capturing such relationships in the form of explicit “ghost” cells/zones/nodes [1]–[3], these models effectively hard-code the kinds of indirection relationships that are supported. *Dependent partitioning* [4] is an alternative framework that instead provides a general and flexible approach to computing partitions for interconnected data structures based on their indirection relationships. Dependent partitioning allows independent partitions to be computed on some data structures (e.g., using METIS [5]) and then for those independent partitions to

be projected onto other data structures through the indirection relationships between the data structures to create *dependent partitions*. In particular, dependent partitioning treats indirection relationships as enumerated mathematical functions and supports operators that can project partitions both forwards (e.g., image) and backwards (e.g., preimage) through these functions. Using these operators, an initial independent partition can be used to construct dependent partitions of any related data structures regardless of how many levels or kinds of indirections are required by an application. A significant body of work has leveraged dependent partitioning across scientific computing domains, including simulations [4], sparse tensor algebra [6], [7], and iterative linear solvers [8].

While dependent partitioning is a highly flexible way to create partitions across interconnected data structures, its generality comes at the cost of performance today. Computing a dependent partition can be expensive as the system must often iterate over all the data in an indirection data structure; this indirection data is regularly distributed and proportional to the scale of the machine, requiring considerable execution time to process. Additionally, since dependent partitioning reckons directly with the inherent geometry and sparsity of the data encoded in the indirection relationships between the data structures, it must manipulate sets of geometric objects (points and rectangles in  $N$ -dimensional spaces) and leverage a variety of compression data structures to store these objects. The computational geometry algorithms required to compute dependent partitions can further result in imbalanced work and unpredictable memory usage. Consequently, to date, implementations of dependent partitioning operators have relied on CPU algorithms that process each operator’s input data sequentially. The resulting performance of dependent partitioning operators regularly limits their use to application start-up phases when delays are more permissible and prohibits their use during performance-sensitive main loops.

In this work, we propose the first parallel algorithms for dependent partitioning operators, enabling their efficient execution on parallel accelerators like GPUs. We develop an algorithmic framework for building parallel dependent partitioning operators that reformulates them around the efficient production and consumption of geometric partition representations. The framework has three core components. First, the framework provides parallel iteration over sets of geometric objects and acceleration structures for querying them. Second, the framework supports the efficient parallel production of compact partition representations through a novel multidimen-

sional, parallel adaptation of the sweep-line algorithm [9]. Finally, the framework incorporates an adaptive tiling-and-retry strategy that allows the parallel dependent partitioning operators to execute within a fixed memory budget, making them amenable to the limited memory capacity of accelerators like GPUs. Altogether, our framework provides a single execution flow for parallel dependent partitioning operators.

We implement our algorithms within the Realm runtime [10] and show how this implementation can be leveraged by a higher-level system like Legion [11]. First, we show across a series of microbenchmarks that our parallel algorithms for dependent partitioning operators when implemented on the GPU provide significant speedups over the existing Realm CPU implementation (9.4–9,807 $\times$ ). Next, we show that on existing Legion applications that leverage dependent partitioning to construct initial data partitions, our work significantly reduces the partitioning cost (15.2–25.5 $\times$  speedup on four GPUs). Finally, we show that the acceleration offered by our work allows for new use cases of dependent partitioning operators within performance-sensitive application components. Compared with the existing CPU-based dependent partitioning implementation, our work permits a Legion-based iterative solver to perform dynamic repartitioning computations 6.7 $\times$  as often during steady-state execution.

## II. BACKGROUND

### A. Dependent Partitioning

Dependent partitioning [4] is a framework for compactly describing partitions of distributed data and the co-partitioning relationships among components of distributed data structures. Dependent partitioning distinguishes between *independent* partitions, which are computed using arbitrary logic, including partitioning libraries such as METIS [5], and *dependent* partitions, which are derived from independent or other dependent partitions through structured operators. Dependent partitioning operators have been used as the core partitioning primitives in a variety of applications, including sparse tensor algebra [6], sparse linear solvers [7], [8], and simulations over unstructured meshes [4], [12]. The dependent partitioning framework consists of a data model and a set of core partitioning operators.

The dependent partitioning data model is defined over *index spaces*, which are (dense or sparse) sets of multi-dimensional indices. *Regions* are multidimensional arrays defined over index spaces that map each point to one or more data values. Regions may contain *pointers* into other regions by storing individual *points* or contiguous *rectangles* of dense points that are contained in another region’s index space. Regions with pointers can be composed to represent a variety of data structures such as sparse tensors [6]–[8] and meshes [4]. The dependent partitioning (DP) data model is visualized in Figure 1.

A *partition* in the DP framework is a mapping from *colors* to index spaces, where each color (usually a multidimensional index) identifies one index space within the partition. Intuitively, colors are often associated with a machine resource, such as a processor or memory. Independent partitions are

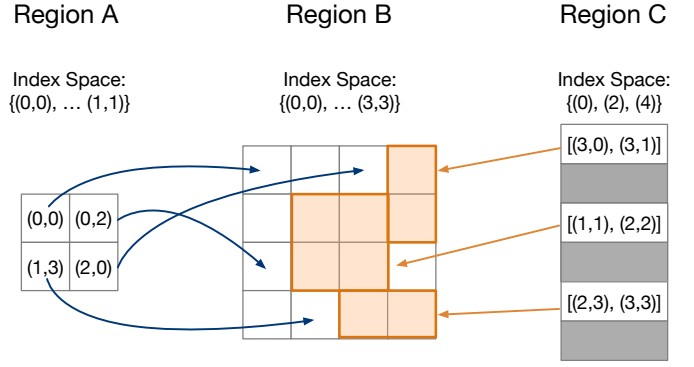


Fig. 1: An example of the dependent partitioning data model. The example contains a dense 2-D region *A* whose points reference a dense 2-D region *B*, and a sparse 1-D region *C* whose rectangles reference subsets of *B*.

described through the `byvalue(R)` operator, which accepts a region *R* in which each element  $R[i]$  stores a color *c*, and constructs a partition *P* such that  $P[c] = \{i \mid R[i] = c\}$ . Because arbitrary computation may populate *R*, its entries can encode application-specific partitioning heuristics and constraints. Dependent partitions are then derived from independent partitions (or other dependent partitions) through the `image`, `preimage`, `union`, `difference`, and `intersection` operators.

`image` and `preimage` relate partitions through data-dependent indirections between regions, such as for sparse matrices or meshes. `image(S, P, D)` projects a partition *P* through a source region *S* with pointer-values to construct a partition *P'* of *D* such that  $\forall c \in \text{COLORS}(P), \forall i \in P[c], S[i] \subseteq P'[c]$ . `preimage(D, P, S)` is the inverse of `image`, discovering a partition *P'* of *S* such that `image(S, P', D) = P`. `image` and `preimage` are visualized in Figure 2. `union`, `difference`, and `intersection` are standard set operators ( $\cup$ ,  $\setminus$ , and  $\cap$ ) between partitions. In this work, we focus on the `byvalue`, `image` and `preimage` operators, as these operators depend on data stored inside regions, while the set operators only manipulate the representation of partitions.

### B. Sequential Algorithms for Dependent Partitioning

Implementations of dependent partitioning require concrete data structures for representing index spaces and algorithms for computing dependent partitions. The state-of-the-art Realm [10] implementation exposes parallelism across independent index-space operations, but the work within each operation scales with the size of partition data and is performed sequentially. Our work addresses this shortcoming by developing a parallel framework for dependent partitioning; we first describe these existing sequential algorithms.

a) *Index Space Representations*: Any implementation of dependent partitioning must efficiently represent the index spaces manipulated by dependent partitioning operators, which may be sparse and irregular (as shown in Figures 1 and 2). The Realm implementation of dependent partitioning, on which we build, represents an index space with two components: a

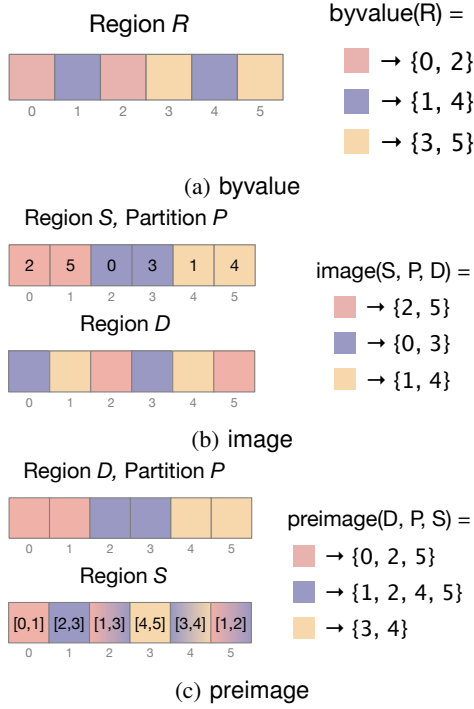


Fig. 2: Visualization of dependent partitioning operators.

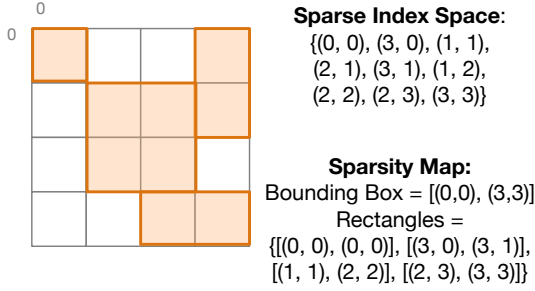


Fig. 3: Sparse index space and sparsity map representation.

bounding hyper-rectangle that contains all points in the index space, and a *sparsity map*. A sparsity map is a collection of disjoint hyper-rectangles whose union is exactly the set of points within the target index space. For dense index spaces, where the bounding hyper-rectangle is exactly the desired index space, the sparsity map is omitted. This representation is preferable to sparse-tensor formats such as CSR because it exploits dense substructures within irregular index spaces. An example index space and sparsity map representation is shown in Figure 3.

Algorithms for performing dependent partitioning operators must efficiently iterate over and construct these index space representations (in particular the sparsity maps). Manipulation of sparsity maps may produce unpredictable memory consumption and computational work, as the individual rectangles stored within sparsity maps may contain different numbers of points. Creation of new sparsity maps requires compacting and aggregating index-space points into a compact set of disjoint

### Algorithm 1 Sequential Point-Valued Image

```

1: function IMAGE( $S$ : Region,  $P$ : Partition,  $D$ : Region)
2:    $O \leftarrow \{c \mapsto \text{RECTLIST}() \mid c \in \text{COLORS}(P)\}$ 
3:   for each index  $i \in S$  do
4:      $d \leftarrow S[i]$ 
5:     if  $D.\text{CONTAINS}(d)$  then
6:        $\text{INSERTANDMAYBEMERGE}(O[\text{COLOR}(P, i)], d)$ 
7:   return  $\text{PARTITION}(\{c \mapsto \text{COMPACT}(O[c]) \mid c \in \text{COLORS}(P)\})$ 

```

rectangles to improve compression. Sparsity maps can enable high degrees of compression but require specialized algorithms to process and construct efficiently, especially in parallel.

*b) Algorithms:* The existing Realm implementation separates each dependent partitioning operator into two stages: determining the points in the output index spaces and assembling them into sparsity maps. Realm exposes task-level parallelism across independent partitioning tasks, but each task processes its partition data sequentially. Algorithm 1 shows the sequential algorithm executed by one such task for a point-valued image. For each source index  $i$  in the source region  $S$ , the algorithm reads the destination pointer  $d = S[i]$  and checks membership in the destination region  $D$ . If  $d \in D$ , the algorithm adds  $d$  to a local list of rectangles, opportunistically merging the point with existing rectangles that it touches. Once all pointer contributions have been included, the maintained rectangles are deduplicated and merged to construct a final sparsity map. The sequential algorithms for merging newly added points into an existing set of rectangles and for merging rectangle lists maintain trees of existing rectangles and perform queries and insertions one at a time. This approach is straightforward to implement, but does not expose data parallelism within each task, where the amount of work scales with the size of the partition data. This distinction is particularly important for accelerators: saturating the high memory bandwidth of modern GPUs requires substantially more memory-level parallelism than saturating CPU memory bandwidth.

The other dependent partitioning operators (*byvalue*, *preimage*) are implemented in a similar manner. *byvalue* uses the values stored in the region to add source points to the corresponding output index space, and *preimage* tests source pointers against the subspaces of the destination partition.

### III. PARALLEL ALGORITHMS

We now describe the key contributions of this work: parallel algorithms for dependent partitioning operators. Rather than defining separate algorithms for each dependent partitioning operator, we design a unified framework that can be instantiated differently for each operator (Section III-A). We describe the parallel algorithms that can be assembled within this framework (Section III-B) to instantiate the full implementations of dependent partitioning operators (Section III-C). Finally, due to limited accelerator memory, we propose a tiling-and-retry strategy that uses a single fixed intermediate

---

**Algorithm 2** Framework for Parallel Dependent Partitioning

---

```
1: type DPFUNCTOR:
2:   types:  $\tau_i, \tau_o$ 
3:   CREATECONTAINER()  $\rightarrow \tau_i$  set
4:   GETADDITIONS( $x$ )  $\rightarrow \tau_o$  set
5:   function DEPARTOP( $F$ : DPFunctor,  $CS$ : IndexSpace)
6:      $K \leftarrow \text{GETNUMCOLORS}(CS)$ 
7:      $C \leftarrow F.\text{CREATECONTAINER}()$ 
8:      $A \leftarrow \emptyset$ 
9:     for all  $x \in C$  in parallel do
10:        $\Delta \leftarrow F.\text{GETADDITIONS}(x)$ 
11:       accumulate  $\Delta$  into  $A$ 
12:      $\{S_0, \dots, S_{K-1}\} \leftarrow \text{CANONICALIZE}(A)$ 
13:     return  $\{S_0, \dots, S_{K-1}\}$ 
```

---

buffer instead of unpredictable intermediate allocations (Section III-D).

#### A. Algorithmic Framework

Although individual dependent partitioning operators differ in their semantics, implementations of each operator share the same high-level structure. Each operator consumes input index spaces or regions, produces a collection of point or rectangle contributions to some output index spaces, and then compacts each group of output contributions into compressed index space representations. The concrete strategy for each stage is then specific to a dependent partitioning operator.

This intuition is captured in Algorithm 2, which is parameterized over a user-defined DPFUNCTOR. The DPFUNCTOR contains the definitions of the functions CREATECONTAINER and GETADDITIONS. First, CREATECONTAINER defines the construction of a container data structure  $C$  amenable to parallel iteration. This data structure contains elements of the input type  $\tau_i$ , and defines which points or rectangles must be processed to construct the output partition. Second, GETADDITIONS maps each input element  $x \in C$  to a set of intermediate contributions of type  $\tau_o$ , representing the portion of the output partition induced by  $x$ . In our framework,  $\tau_i$  and  $\tau_o$  are always either multi-dimensional points or rectangles.

Using a DPFUNCTOR, Algorithm 2 initializes the input container data structure and accumulates all intermediate output contributions. Once the intermediate contributions have been generated, these contributions must be compressed into a compacted index space representation, as the intermediate contributions may contain duplicate points or overlapping rectangles and Realm’s sparsity map prohibits duplicate point representations. To perform this compaction, the framework applies a shared CANONICALIZE step, which converts the intermediate contributions  $A$  into a valid sparsity map representation: a set of disjoint, coalesced hyper-rectangles.

#### B. Algorithmic Components

We next describe the core algorithmic components used to instantiate the framework.

---

**Algorithm 3** Parallel Point Canonicalization

---

```
1: function POINTCANONICALIZE( $P$ : Points)
2:    $P \leftarrow \text{SORTBYKEY}(P, \lambda x. (x[N-1], \dots, x[0]))$ 
3:    $R \leftarrow \text{TO SINGLETON RECTS}(P)$ 
4:   for  $d = 0, 1, \dots, N-1$  do
5:      $R \leftarrow \text{PARALLELRLBYDIM}(R, d)$ 
6:   return  $R$ 
```

---

1) *Parallel Iteration:* Parallel dependent partitioning operators require the ability to perform parallel iteration over (potentially sparse) index space data structures. When an index space is dense, parallel iteration is trivial. For sparse index spaces, represented by a sparsity map containing a vector of disjoint rectangles, we perform a prefix sum over a vector constructed from the volume of each rectangle. The total volume  $V$  represents a flat set of indices, and a corresponding point within the sparse index space can be recovered from  $i \in [0, V)$  by binary searching over the computed prefix sum.

2) *Acceleration Structures:* Some dependent partitioning operators such as **preimage** require index space membership tests: checking whether a candidate point is a member of a target index space or a candidate rectangle intersects any index space points. To accelerate these membership tests over sparse index spaces, we build bounding volume hierarchies (BVHs) over the rectangles in a sparse index space. We leverage existing parallel BVH construction and query algorithms [13]. The BVH reduces the cost of membership and intersection queries from linear to logarithmic in the number of sparsity-map rectangles.

3) *Point Canonicalization:* When a dependent-partitioning operation produces point contributions, the intermediate point list may contain duplicates and must be compressed into the list of disjoint rectangles stored in a sparsity map. We canonicalize these intermediate contributions by removing duplicate points and coalescing adjacent points. Algorithm 3 first sorts the points lexicographically, with the output color as the most-significant key, and converts each point into a singleton rectangle. It then performs a parallel run-length-encoding pass along each dimension.

For each dimension  $d$ , PARALLELRLBYDIM identifies runs of consecutive rectangles that have identical bounds in every dimension other than  $d$ , and equal or adjacent intervals in  $d$ . Equal rectangles are filtered away, and adjacent rectangles are merged into a rectangle representing their union. These computations map cleanly to parallel sorts and prefix scans.

Figure 4 illustrates the algorithm in two dimensions. After sorting each  $(x, y)$  point by  $(y, x)$ , the first pass merges adjacent singleton rectangles across the  $x$ -dimension. The second pass then merges rectangles with identical  $x$ -ranges along the  $y$ -dimension. Each merge replaces a run of compatible, pairwise-disjoint rectangles with their union, preserving the represented point set and disjointness. Processing each dimension in order yields a dimension-wise coalesced rectangle cover, although the algorithm does not attempt to find a globally minimum-cardinality cover.

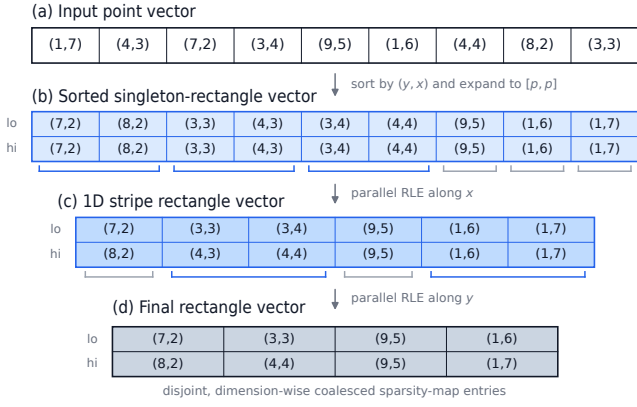


Fig. 4: Point canonicalization in two dimensions. Brackets indicate runs that will be merged into a single rectangle.

#### Algorithm 4 Parallel Rectangle Canonicalization

```

1: function RECTANGLECANONICALIZE( $R$ : Rectangles)
2:    $A \leftarrow \text{INITBOUNDARYPOINTS}(R, 0)$ 
3:    $A \leftarrow A + \text{SIGNEDCORNERS}(R)$ 
4:   for  $d = N - 1, N - 2, \dots, 0$  do
5:      $A \leftarrow \text{PREFIXSWEEP}(A, d)$ 
6:      $\mathcal{O} \leftarrow \text{EMITCOVEREDCELLS}(A)$ 
7:   return COALESCEBYDIMENSION( $\mathcal{O}$ )

```

4) *Rectangle Canonicalization*: Dependent-partitioning operators may also directly produce rectangles instead of points, such as a rectangle-valued image. Constructing a valid sparsity map from these rectangle contributions requires deduplication and removal of any overlapping components, as every rectangle in a sparsity map must be pairwise-disjoint. A naïve approach to rectangle canonicalization would be expanding each rectangle into its constituent points and applying the algorithm discussed in Section III-B3. However, this approach would make the time and space complexity of canonicalization proportional to the volume of all rectangles, eliminating any compression provided by the rectangle-based sparsity map representation. We instead directly canonicalize overlapping rectangles.

Algorithm 4 is our approach to parallel rectangle canonicalization, which generalizes existing sweep-line algorithms [9] to any number of dimensions. The key idea behind our algorithm is to discretize the volume covered by all input rectangles into a set of disjoint *boundary cells*, which overapproximate the output set of rectangles covered by the input rectangles. Then, by a series of dimension-aligned sweeps over these boundary cells, we can determine which cells are covered by input rectangles and which cells must be excluded. The remaining boundary cells already satisfy the disjointness invariant required for sparsity map construction; we further coalesce them into a compressed sparsity map of rectangles, yielding the desired result. Figure 5 visualizes the steps taken by our algorithm, and we now discuss the concrete details.

The first step of our algorithm is to convert the closed

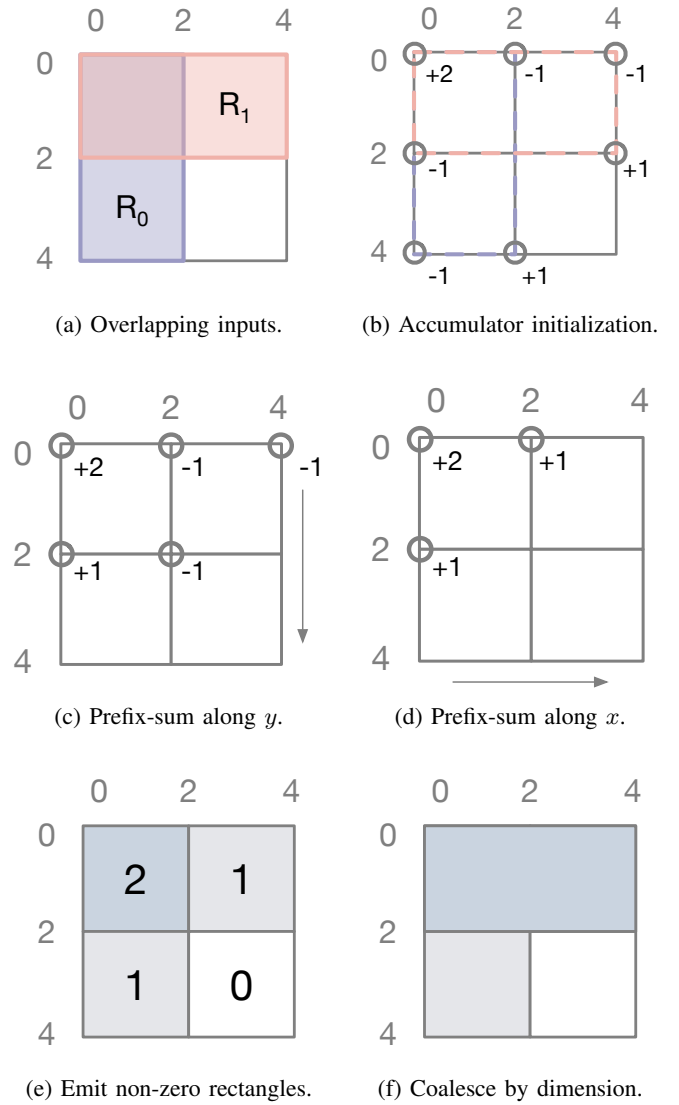


Fig. 5: Rectangle canonicalization in two dimensions.

rectangle representation to a half-open representation, where the upper-bound point of each rectangle is exclusive. Then, for each dimension  $d \in [0, N)$ , we define the *boundary coordinates*  $\mathcal{B}_d = \text{SORTUNIQUE}(\{\ell_d(r), u_d(r) \mid r \in \mathcal{R}\})$ .  $\mathcal{B}_d$  captures all coordinates along dimension  $d$  where an input rectangle starts or ends. We refer to consecutive values  $b, b' \in \mathcal{B}_d$  as a *boundary interval* over the range  $[b, b')$ . The Cartesian product over the boundary intervals of each dimension defines the *boundary cells*, and the Cartesian product over the boundary coordinates of each dimension defines the *boundary points*. By construction, no rectangle can start or end within the interior of a boundary cell; this property allows us to reason only about boundary points and boundary cells, rather than all points contained within the input rectangles. In Figure 5a, the boundary coordinates for each dimension are  $[0, 2, 4]$ , the boundary cells are  $[(0, 0), (2, 2))$ ,  $[(0, 2), (2, 4))$ ,  $[(2, 0), (4, 2))$  and  $[(2, 2), (4, 4))$ , and the boundary points are

$(0, 0), (0, 2), (0, 4), (2, 0), (2, 2), (2, 4), (4, 0), (4, 2), (4, 4)$ .

The goal of our algorithm is to determine the exact set of boundary cells covered by the input rectangles. Because we have the sorted unique boundary coordinates in each dimension, we can reconstruct a boundary cell from its bottom point; thus, we operate on boundary points and reconstruct the corresponding boundary cells at the end. Our algorithm attaches a signed value to each boundary point, which we refer to as a *coverage accumulator*. We initialize all coverage accumulators to 0. Then, we include initial contributions from the corners of every input rectangle as follows. For each rectangle corner, compute the parity of the number of dimensions in which it contains a “high” value of the rectangle, and contribute  $-1$  for an odd number and  $+1$  for an even number. Set the coverage accumulator at every point  $p$  to the sum of all initial corner contributions. For example, in Figure 5,  $(0, 0)$  is a corner for both  $R_0$  and  $R_1$ . For both  $R_0$  and  $R_1$ , it is high in neither dimension, so each contributes  $+1$ , yielding  $+2$ .  $(2, 0)$  is only a corner for  $R_0$  and high in 1 dimension ( $x$ ), so its sign is  $-1$  and the  $(2, 0)$  coverage accumulator is initialized to  $-1$ .  $(2, 2)$  has no corners so its coverage accumulator is initialized to 0.

Once the coverage accumulators over the boundary points have been created, we perform a series of dimension-aligned prefix sums over the events. For each dimension  $d$ , PREFIXSWEEP groups the current events that agree on all coordinates other than  $d$  into one-dimensional sweep-lines, and performs a segmented exclusive prefix-sum along each sweep-line. An example of the sweeps along each dimension can be seen in Figure 5c and Figure 5d. At the end of these sweeps, the coverage accumulator for a boundary point encodes how many of the input rectangles cover its corresponding boundary cell; thus, we can emit exactly the boundary cells with nonzero coverage accumulators as a pairwise-disjoint exact covering of the input rectangles, seen in Figure 5e.

Once the sweep-line procedure has produced disjoint rectangles from all boundary cells, they are coalesced into larger rectangles to better compress the representation. We apply the same strategy as discussed in Section III-B3, where we merge adjacent rectangles along each dimension in turn, seen in Figure 5f. As before, this approach maximally coalesces rectangles, but does not guarantee a globally optimal covering.

While our algorithm conceptually maintains a coverage accumulator for each boundary point, materializing all boundary points (the Cartesian product of all boundary coordinates) would be unnecessarily expensive. In practice, Algorithm 4 processes one dimension at a time and materializes only coverage accumulators with non-zero values: boundary points with zero-valued coverage accumulators do not affect the prefix sum, and cannot become emitted boundary cells. However, because coverage accumulators are signed, these sums are not monotonic, and a boundary point can go to zero and be eliminated in a sweep along one dimension, then become non-zero and require rematerialization in another sweep. The optimized version of our algorithm performs this compression and rematerialization using binary searches along each sweep-line

to determine which boundary points have non-zero coverage accumulators after each sweep.

The signed corners and prefix sweeps done by Algorithm 4 are the multi-dimensional analogue of a one-dimensional difference array. For a single interval, placing  $+1$  at its lower bound and  $-1$  at its exclusive upper bound causes a prefix sum to be one inside the interval and zero outside it. An  $N$ -dimensional rectangle applies this construction in every dimension: its  $2^N$  corners receive alternating signs, and each dimension-aligned sweep turns the contributions from those corners into coverage along the swept dimension. After all  $N$  sweeps, a single rectangle therefore contributes one to every boundary cell it covers and zero to every other cell. This construction is an instance of Möbius inversion, or multidimensional inclusion–exclusion [14]. Contributions from multiple rectangles are additive, so the final coverage accumulator is the number of input rectangles covering each cell. Emitting cells with positive accumulators thus recovers the input union, while the boundary-cell construction ensures that the emitted cells are pairwise disjoint.

The time and space requirements of this canonicalization procedure are data dependent. For  $M$  input rectangles in  $N$  dimensions, SIGNEDCORNERS produces  $2^N \cdot M$  events. Because  $N$  is usually a small constant, this initial step produces a linear number of non-zero boundary points to maintain. However, subsequent sweeps materialize a data-dependent number of boundary points, depending on the overlapping pattern of the input rectangles in each dimension. Our evaluation does not include a dedicated adversarial construction designed to realize the worst case, but this behavior can be characterized analytically. Pathological arrangements, such as checkerboard-like patterns, can eliminate the compression provided by the rectangle representation: an intermediate working set or uncoalesced output may approach the full Cartesian product of the boundary intervals, containing  $O(M^N)$  records in the worst case. For inputs whose boundary arrangements remain compressible, only a small fraction of this Cartesian product is materialized, retaining the benefits of the rectangle-based sparsity map representation. Despite the data-dependent runtime and space usage, the operations in Algorithm 4 map directly to standard parallel primitives (sorting, segmented scans and filters), allowing for efficient parallel execution on GPUs.

### C. Framework Instantiation

Each dependent partitioning operator instantiates the framework through its DPFUNCTOR, as described below.

*a) byvalue:* CREATECONTAINER returns the index space of the region being partitioned. For each point  $x \in \text{CREATECONTAINER}()$ , where  $R[x] = c$ , GETADDITIONS accumulates  $x$  into the output set for color  $c$ .

*b) image:* Like *byvalue*, the implementation of CREATECONTAINER for *image* returns the index space of the region through which the image is being computed. For each source point  $x$  of color  $c$ , GETADDITIONS reads  $y = S[x]$  and contributes the point or rectangle  $y$  to the output set for  $c$ .

c) *preimage*: For a  $\text{preimage}(D, P, S)$ , the implementation of `CREATECONTAINER` creates a BVH  $Q$  over the index spaces of the partition  $P$ , as described in Section III-B, and returns the index space of the source region  $S$ . For each point  $x$  in the index space of  $S$ , `GETADDITIONS` reads the point or rectangle  $y = S[x]$ , queries  $Q$  for the colors whose subspaces contain or intersect  $y$ , and accumulates  $x$  into each corresponding output set.

#### D. Tiling and Retrying

The overall framework and algorithmic components described in the previous sections require intermediate allocations of data-dependent sizes. For example, BVH construction and intermediate phases during point and rectangle canonicalization require buffer allocations whose sizes depend on the set of points contained by an index space and are thus unpredictable. While the sizes of these intermediate allocations can be given upper bounds, these upper bounds can be significant overestimates in the worst case, and allocating according to them may require more memory than is available on a GPU. These complications make a direct application of the previously discussed algorithms difficult within a larger asynchronous runtime system that manages its own memory.

To address the challenges of unpredictable intermediate allocations, we develop an extension to the framework in Section III-A that enables parallel dependent partitioning algorithms to operate within a fixed-size scratch allocation and tile the input when the entire operation does not fit. The key idea is to use rectangle canonicalization. The results of individual tiles of a dependent partitioning operation can be canonicalized into compact and disjoint lists of rectangles and combined when necessary. This approach also applies to point-valued dependent partitioning operators, as point canonicalization converts point sets into rectangle lists.

The complete tiling-and-retry algorithm is shown in Algorithm 5. The algorithm receives a fixed scratch budget  $B$ , and selects an initial tile that uses approximately half the available space, while reserving the remainder of the space for intermediate and output storage. `ATTEMPT` executes a tile and merges its output rectangles with the accumulated output. `ATTEMPT` returns successfully only if all steps of execution fit within the available scratch space.

If an `ATTEMPT` fails at a particular tile size, all temporary state is discarded and the same input prefix is retried with half the tile size. If the candidate tile fails when containing a single rectangle, then our implementation spills the accumulated rectangle outputs to host memory and performs the rectangle merging step on the CPU. After a successful attempt, the candidate tile size is re-estimated from the remaining space so that an earlier tile that required a large amount of memory does not permanently force subsequent tiles to be small.

Our tiling-and-retry approach bounds all GPU-side intermediate storage by  $B$ . If the operation’s peak scratch usage fits within  $B$ , the tiling-and-retry process adds no significant overhead. Under memory pressure, each backoff sequence

---

#### Algorithm 5 Tiling-and-retry algorithm.

---

```

1: Input: Input  $I$ , Scratch Budget  $B$ 
2:  $W \leftarrow I$ 
3:  $O \leftarrow \emptyset$ 
4:  $A \leftarrow B$ 
5:  $t \leftarrow \max(1, \text{ESTIMATE}(A/2))$ 
6: while  $W \neq \emptyset$  do
7:    $A \leftarrow B - \text{SIZE}(O)$ 
8:    $T \leftarrow \text{PREFIX}(W, t)$ 
9:    $(s, O') \leftarrow \text{ATTEMPT}(T, O, A)$ 
10:  if  $s$  then
11:     $O \leftarrow O'$ 
12:     $W \leftarrow W - T$ 
13:     $A \leftarrow B - \text{SIZE}(O)$ 
14:     $t \leftarrow \max(1, \text{ESTIMATE}(A/2))$ 
15:  else if  $|T| > 1$  then
16:     $t \leftarrow \lfloor |T|/2 \rfloor$ 
17:  else
18:     $(W, O) \leftarrow \text{RECOVERANDSPILL}(W, O)$ 
19:     $A \leftarrow B$ 
20:     $t \leftarrow \max(1, \text{ESTIMATE}(A/2))$ 
21: return FINALIZE( $O$ )

```

---

performs logarithmically many attempts, but each failed attempt discards nontrivial computation. In addition, merging each successful tile with the accumulated output may revisit previously produced rectangles. The total work is therefore data- and output-dependent and may be superlinear in the final output size, while peak device scratch usage remains fixed by the budget. We expose the buffer size  $B$  to let users trade memory usage for performance.

#### IV. EVALUATION

We evaluate our work through a set of microbenchmarks of individual dependent partitioning operators, weak scaling of application-derived dependent partitioning kernels, and a dynamic repartitioning experiment within a larger application. We show that our work achieves substantial speedups over the existing CPU-based dependent-partitioning strategy: 9.4–9,807 $\times$  on individual operators without tiling, 15.2–25.5 $\times$  on application-derived partitioning kernels at four GPUs, and up to 17.8 $\times$  higher throughput in dynamic repartitioning.

##### A. Experimental Setup

We perform all experiments on a single node of the Perlmutter supercomputer. Each Perlmutter node contains a 64-core AMD EPYC 7763 CPU and four NVIDIA A100 40GB GPUs. Our GPU implementation is written in CUDA C++ and realizes the algorithms through a combination of custom kernels and CUB bulk-parallel primitives. Because our goal is to enable efficient dependent partitioning on accelerators, we implement and evaluate our parallel algorithms on GPUs and compare against Realm’s existing task-parallel, data-sequential CPU implementation. Although our parallel algorithms are also amenable to multicore CPU execution, we do not evaluate

a multicore CPU implementation in this work. The reported speedups therefore capture the combined impact of our parallel algorithms and GPU execution, rather than isolating the algorithmic impact on equivalent hardware or the hardware impact with equivalent algorithms. For the microbenchmarks and application-derived benchmarks, input construction, host-to-device copies, and scratch allocation occur before timing. GPU runs are warmed up, and timings cover only the dependent-partitioning work through completion. Each plotted point is the median of three runs.

### B. Microbenchmarks

We first evaluate our work with a series of microbenchmarks that focus on individual dependent partitioning operators across different configurations of intermediate buffer utilization. We focus on point-valued **image** and **preimage** operators, and compare against the CPU-based strategy that currently exists in Realm [4]. Each benchmark performs a single-GPU dependent partitioning operation over a region containing  $10^6$  elements that reference points in the source or destination index space. The domain of the region is  $10^6$  in 1D,  $1000 \times 1000$  in 2D, and  $100 \times 100 \times 100$  in 3D. Dense subspaces are rectangles; sparse subspaces independently retain 5% of the points in the same bounding box. In each experiment, we vary the size of the available intermediate buffer with the variable  $p$ , which ranges from 1 to 100. If  $S_{\text{lower}}$  and  $S_{\text{upper}}$  are estimated lower and upper bounds for the target operation (Section III-D), then a value of  $p$  corresponds to a buffer of size  $S_{\text{lower}} + (p/100)(S_{\text{upper}} - S_{\text{lower}})$ . Each point is the median of three runs.

The results for **image** are shown in Figure 6, which contains results for the **image** operator through both the dense and sparse index spaces. At  $p = 100$ , every configuration completes in a single pass, eliminating tiling and fallback overhead. In these configurations, our system achieves speedups of 15.3–9,807 $\times$  over Realm’s CPU-based implementation. For the dense **image** operations, we see that the performance of the GPU-based dependent-partitioning operation corresponds to how much intermediate storage can be allocated (the value of  $p$ ). When the allocated intermediate buffer is sufficiently large to enable only a single pass over the data ( $p = 30$  for 1D,  $p = 25$  for 2D and 3D), the operation enters its single-pass, near-peak-performance regime. When smaller intermediate buffers are used, additional passes over the input are required, causing performance degradation. However, our system still achieves significant speedups over the CPU-based implementation in these cases. Even at the smallest buffer size ( $p = 1$ ), the dense **image** cases remain 1.2 $\times$ , 1.6 $\times$ , and 4.4 $\times$  faster than the CPU implementation in 1D, 2D, and 3D, respectively; across all tiled dense configurations, speedups range from 1.2–14.1 $\times$ . Due to the nature of the Legion runtime system, realized sparsity maps are not available when intermediate buffer requirements are estimated. As a result, the resulting bounds are disproportionately conservative for sparse index spaces. At runtime, sparsity-map compression and the memory efficiency of our algorithms then keep actual

intermediate storage much lower than these estimates: only two sparse configurations across both **image** and **preimage** trigger the tiling and retry mechanism. The low- $p$  dense configurations therefore serve primarily as a stress test of the tiling mechanism under constrained scratch space, rather than as examples of tiling that arises naturally in these workloads.

Next, we move on to **preimage** results, which are shown in Figure 7. The shape of these results is similar to the **image** results, where the largest speedups are observed when the buffers are large enough for a single pass over the input to be performed. At these buffer sizes, our implementation achieves speedups of 9.4–207 $\times$  over Realm’s CPU implementation. These results also show that the online construction of acceleration data structures performed by our algorithm does not negatively impact the end-to-end performance.

### C. Application-Derived Weak Scaling

In the next set of benchmarks, we evaluate the dependent partitioning operators used during the initialization of three scientific mini-applications: a circuit simulation [11], a solver for compressible Navier-Stokes equations (MiniAero [15]), and a Lagrangian hydrodynamics simulation (PENNANT [16]). Each application’s partitioning phase consists of a short sequence of Realm partitioning operators: eight measured stages for Circuit, six for PENNANT, and four for MiniAero, including by-field, **image**, **preimage**, and set-operation stages over the application’s mesh data structures. The partitioning performance of these applications with dependent partitioning has been evaluated in prior work [4]. We measure the dependent partitioning costs of each application, and exclude the partitioning-free steady-state execution. We weak-scale the problem size from one to four GPUs of a single Perlmutter node, and all experiments use  $p = 100$ , as enough memory is available for a full intermediate buffer.

Figure 8 contains the results for each of Circuit, PENNANT, and MiniAero. As the problem size weak-scales from one to four GPUs, our implementation achieves four-GPU weak-scaling efficiencies of 89.7%, 94.9%, and 78.0% for Circuit, PENNANT, and MiniAero, respectively. At four GPUs, our GPU-based dependent partitioning algorithms provide significant speedups over Realm’s CPU-based implementation, achieving speedups of 25.5 $\times$ , 23.1 $\times$ , and 15.2 $\times$  for Circuit, PENNANT, and MiniAero, respectively. These speedups directly follow from our microbenchmarks, where we report similar speedups for individual dependent-partitioning operators. These results show that our GPU-parallel algorithms greatly accelerate dependent partitioning in existing applications.

### D. Dynamic Repartitioning

The current cost of Realm’s implementation makes frequent in-loop dependent partitioning impractical. Our preceding results suggest that the reduced cost of our implementation may make such in-loop repartitioning practical. Because existing applications were designed around the substantially higher cost of dependent partitioning and therefore do not provide a natural dynamic-repartitioning benchmark, we evaluate this

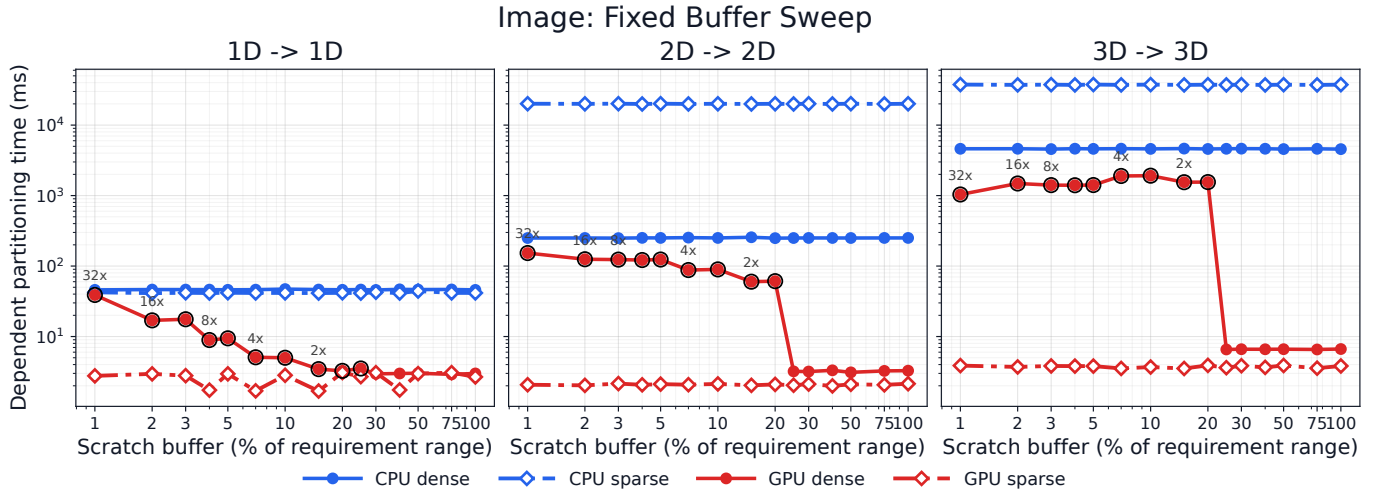


Fig. 6: **Image** latency under varying scratch-memory budgets for dense and sparse 1D, 2D, and 3D index spaces. Each point is the median of three runs. Lower values of  $p$  restrict temporary storage and may require tiled execution.

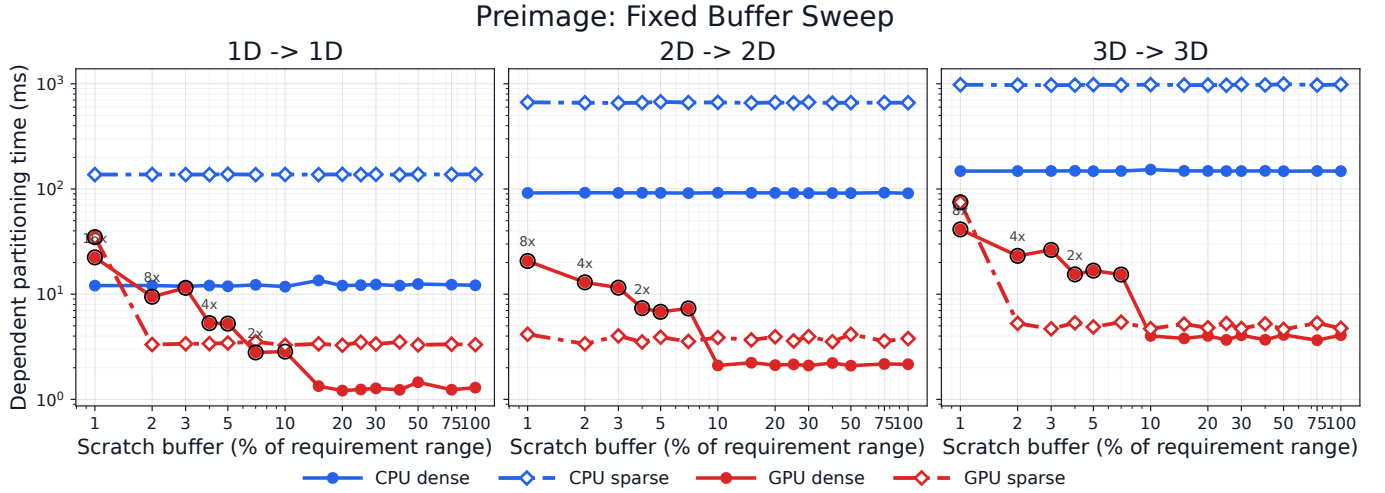


Fig. 7: **Preimage** latency under varying scratch-memory budgets for dense and sparse 1D, 2D, and 3D index spaces. Each point is the median of three runs. Lower values of  $p$  restrict temporary storage and may require tiled execution.

possibility synthetically. We modify the LegionSolvers [8] conjugate gradient solver to periodically rebuild the partitions of the sparse matrix and associated vectors during steady-state execution.

We run 2000 iterations of the conjugate gradient solver on 4 GPUs of Perlmutter, and sweep over a *repartitioning interval*, which is the number of iterations between two repartitioning operations. Each repartitioning operation rebuilds the necessary partitions of the sparse matrix and necessary vector components. Because the benchmark repeatedly reconstructs the same equal partitioning scheme, it measures the cost of in-loop partition reconstruction rather than the effects or migration costs of changing data placement. The reported timings include the computation costs of the solver loop as well as all dependent partitioning costs.

Figure 9 shows the results for this experiment, comparing

the throughput achieved at different repartitioning intervals for both CPU- and GPU-based dependent partitioning implementations. At the most aggressive repartitioning interval of one iteration, both implementations experience substantial throughput degradation, but the GPU-based implementation sustains  $17.8\times$  higher throughput than the CPU-based implementation. As the repartitioning interval increases, the GPU-based implementation reaches 90% of baseline throughput at an interval of 150 iterations, where the CPU implementation achieves only 37.8% of baseline throughput. The CPU implementation does not reach the same 90% threshold until a repartitioning interval of 1000 iterations, allowing the GPU implementation to repartition  $6.7\times$  as frequently while maintaining at least 90% of baseline throughput. At a repartitioning interval of 500 iterations, the GPU-based dependent partitioning implemen-

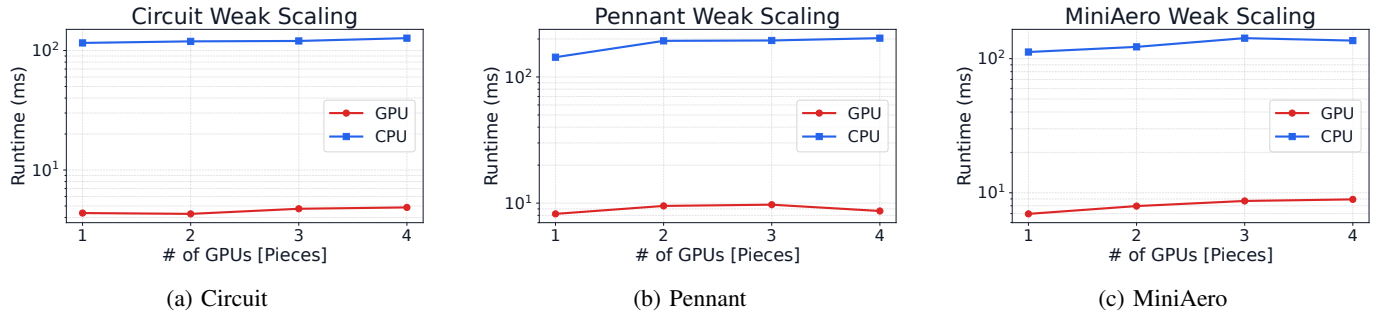


Fig. 8: Single-node weak scaling of the dependent-partitioning sequences derived from Circuit, Pennant, and MiniAero. The per-piece problem size is held fixed as the number of pieces, GPUs, and dependent-partitioning workers increases from one to four. Each point is the median of three runs.

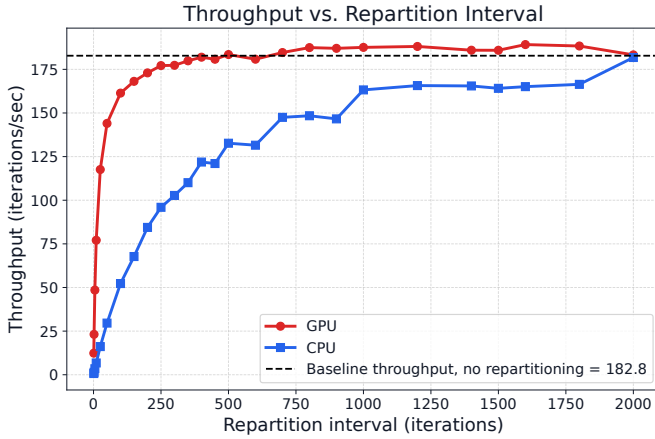


Fig. 9: Throughput of a sparse iterative solver on four GPUs with CPU and GPU repartitioning at different intervals.

tation is able to avoid any performance degradation relative to the no-repartition baseline, while the CPU implementation still achieves only 74.0% of baseline. These results show that our GPU-parallel algorithms substantially reduce the overhead of reconstructing dependent partitions within the hot loop of Legion applications, making much more frequent in-loop repartitioning feasible. This reduced cost could enable applications with dynamic data structures or adaptive load-balancing strategies that require steady-state repartitioning.

## V. RELATED WORK

### A. Dependent Partitioning

Dependent partitioning [4] is a logical framework for describing partitions of distributed data structures. Dependent partitioning distinguishes *independent* partitions, computed arbitrarily, from *dependent* partitions. Dependent partitions are derived from independent or other dependent partitions using structured operators that follow references among distributed data structures. The dependent partitioning framework has been implemented in the Legion [11] and Realm [10] runtime systems as well as the Regent [12] programming language. The dependent partitioning framework is especially powerful

for applications with dynamic data structures connected by indirections.

### B. Application-Specific Partitioning Algorithms

In terms of dependent partitioning, significant research has been done on algorithms for the construction of independent partitions for various applications and domains. For sparse matrix applications, the METIS [5] and PARMETIS [17] lines of work develop algorithms to partition graphs and sparse matrices in communication-minimizing ways. PT-Scotch [18] is a partitioning framework for graphs. Libraries like PETSc [1] and Trilinos [2] have built-in support for computing dependent partitions representing the “ghost zones” of structured meshes, or the “cones” or “supports” of unstructured meshes [19]. Stencil analyses in domain-specific language compilers [20], [21] map well onto dependent partitioning operators.

### C. Computational Geometry and Graphics Algorithms

Acceleration data structures, such as the bounding-volume hierarchies [13] used in this work, have been heavily studied in graphics computations such as rendering and ray-tracing [22]. Alternative data structures such as KD-Trees [23] and R-Trees [24] are also well-studied. These acceleration data structures have also been used within the Legion runtime to reason about data partitions to perform distributed dependence analysis [25]. This work also reformulates classical sweep-line techniques from computational geometry [9] for multidimensional parallel sparsity-map construction and compaction.

## VI. CONCLUSION

We have presented a parallel framework and algorithms for dependent partitioning on accelerators such as GPUs. We showed that these algorithms significantly accelerate dependent partitioning over existing CPU-based approaches while avoiding unbounded intermediate dynamic memory allocation. Our work accelerates partitioning computations of existing applications and enables a new class of applications to leverage high-performance repartitioning within performance-critical components.

## ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Award No. 2216964. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy User Facility using NERSC award ASCR-ERCAP 0037135.

OpenAI ChatGPT was used to assist with brainstorming, organization, restructuring, proofreading, and language editing in the Abstract and Sections I–VI. OpenAI ChatGPT and Codex were also used to assist with algorithm development, software implementation, testing, and experimental workflows underlying Sections III and IV. All AI-assisted technical contributions, code, experiments, and manuscript content were reviewed, revised, tested, and validated by the authors.

## REFERENCES

- [1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*. Boston, MA: Birkhäuser Boston, 1997, pp. 163–202. [Online]. Available: [https://doi.org/10.1007/978-1-4612-1986-6\\_8](https://doi.org/10.1007/978-1-4612-1986-6_8)
- [2] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Williams, and A. G. Kennon, “An overview of the Trilinos project,” *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, 2005.
- [3] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and B. Peterson, “Uintah: A scalable framework for hazard analysis,” in *Proceedings of the 2010 TeraGrid Conference on Gatewaying, Discovering and Geeking (TG ’10)*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 3:1–3:8.
- [4] S. Treichler, M. Bauer, R. Sharma, E. Slaughter, and A. Aiken, “Dependent partitioning,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 344–358. [Online]. Available: <https://doi.org/10.1145/2983990.2984016>
- [5] G. Karypis and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, Jan. 1998. [Online]. Available: <http://epubs.siam.org/doi/10.1137/S1064827595287997>
- [6] R. Yadav, A. Aiken, and F. Kjolstad, “SpDISTAL: compiling distributed sparse tensor computations,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’22. Dallas, Texas: IEEE Press, Nov. 2022, pp. 1–15. [Online]. Available: <https://dl.acm.org/doi/10.5555/3571885.3571963>
- [7] R. Yadav, W. Lee, M. Elibol, M. Papadakis, T. Lee-Patti, M. Garland, A. Aiken, F. Kjolstad, and M. Bauer, “Legate Sparse: Distributed Sparse Computing in Python,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1–13. [Online]. Available: <https://dl.acm.org/doi/10.1145/3581784.3607033>
- [8] D. K. Zhang, R. Yadav, A. Aiken, F. Kjolstad, and S. 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*. St. Louis, MO, USA: ACM, Nov. 2025, pp. 1351–1365. [Online]. Available: <https://dl.acm.org/doi/10.1145/3731599.3767501>
- [9] M. I. Shamos and D. Hoey, “Geometric intersection problems,” in *17th Annual Symposium on Foundations of Computer Science*, 1976, pp. 208–215.
- [10] S. Treichler, M. Bauer, and A. Aiken, “Realm: an event-based low-level runtime for distributed memory architectures,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 263–276. [Online]. Available: <https://doi.org/10.1145/2628071.2628084>
- [11] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.
- [12] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, “Regent: a high-productivity programming language for HPC with logical regions,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin, TX, USA: ACM, Nov. 2015, pp. 1–12. [Online]. Available: <https://dl.acm.org/doi/10.1145/2807591.2807629>
- [13] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 89–99. [Online]. Available: <https://doi.org/10.1145/2492045.2492055>
- [14] G.-C. Rota, “On the foundations of combinatorial theory I. theory of Möbius functions,” *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, vol. 2, no. 4, pp. 340–368, 1964.
- [15] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, “Improving performance via mini-applications,” Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.
- [16] C. R. Ferenbaugh, “PENNANT: an unstructured mesh mini-app for advanced architecture research,” *Concurr. Comput. : Pract. Exper.*, vol. 27, no. 17, pp. 4555–4572, Dec. 2015. [Online]. Available: <https://doi.org/10.1002/cpe.3422>
- [17] G. Karypis, K. Schloegel, and V. Kumar, “ParMETIS: Parallel graph partitioning and sparse matrix ordering library,” Department of Computer Science, University of Minnesota, Tech. Rep. TR 97-060, 1997. [Online]. Available: <https://hdl.handle.net/11299/215345>
- [18] C. Chevalier and F. Pellegrini, “PT-Scotch: A tool for efficient parallel graph ordering,” *Parallel Computing*, vol. 34, no. 6, pp. 318–331, 2008, Parallel Matrix Algorithms and Applications. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819107001342>
- [19] M. Lange, M. G. Knepley, and G. J. Gorman, “Flexible, scalable mesh and data management using PETSc DMplex,” *CoRR*, vol. abs/1505.04633, 2015. [Online]. Available: <http://arxiv.org/abs/1505.04633>
- [20] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, “Liszt: A domain specific language for building portable mesh-based PDE solvers,” in *SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.
- [21] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, “A stencil compiler for short-vector SIMD architectures,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 13–24. [Online]. Available: <https://doi.org/10.1145/2464996.2467268>
- [22] M. Pharr and G. Humphreys, *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.
- [23] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: <https://doi.org/10.1145/361002.361007>
- [24] A. Guttman, “R-trees: a dynamic index structure for spatial searching,” *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984. [Online]. Available: <https://doi.org/10.1145/971697.602266>
- [25] M. Bauer, E. Slaughter, S. Treichler, W. Lee, M. Garland, and A. 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*, ser. PPoPP ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 218–231. [Online]. Available: <https://doi.org/10.1145/3572848.3577515>