Disaggregated Design for GPU-Based Volumetric Data Structures

Massimiliano Meneghin[1][0000−0002−1295−9062] and
Ahmed H. Mahmoud✉[2][0000−0003−1857−913X]

[1] Autodesk Research, Milan, MI 20124, Italy
massimiliano.meneghin@autodesk.com
[2] MIT CSAIL, Cambridge, MA 02139, USA
ahdhn@mit.edu

Abstract. Volumetric data structures typically prioritize data locality, focusing on efficient memory access patterns. This singular focus can neglect other critical performance factors, such as occupancy, communication, and kernel fusion. We introduce a novel *disaggregated* design that rebalances trade-offs between locality and these objectives—reducing communication overhead on distributed memory architectures, mitigating register pressure in complex boundary conditions, and enabling kernel fusion. We provide a thorough analysis of its benefits on a single-node multi-GPU Lattice Boltzmann Method (LBM) solver. Our evaluation spans dense, block-sparse, and multi-resolution discretizations, demonstrating our design's flexibility and efficiency. Leveraging this approach, we achieve up to a 3× speedup over state-of-the-art solutions.

Keywords: Data layout · Parallel · Simulation · GPU · LBM.

1 Introduction

Since the 2000s, the *memory wall* [23, 1] has underscored the critical importance of data locality optimizations in computational tasks. This challenge is especially acute in memory-bound volumetric physics simulations, prompting research into strategies such as blocking [4], time-tiling [22], polyhedral optimizations [2], and cache-oblivious methods [5].

Although GPUs provide high memory bandwidth, achieving peak performance also requires addressing occupancy, load balancing, synchronization overhead, and data movement. Traditionally, volumetric data structures are designed to improve data locality first, with other optimizations (e.g., overlapping computation and communication [16], time skewing [22], kernel fusion [20], tiling [19]) introduced afterward. Because these methods are not considered during data structure design, significant performance opportunities are lost. We argue that additional objectives should be incorporated into volumetric data structure design. While data locality remains essential, selectively compromising it can improve end-to-end performance by addressing other goals.

In this paper, we propose *disaggregated design* for volumetric data structures, which balances multiple performance objectives by:

1. **Grouping voxels based on desired properties:** Rather than relying solely on spatial locality, we cluster voxels according to the traits most relevant for performance.
2. **Applying traditional data locality optimizations within each group:** Within these groups, we still exploit locality as appropriate while addressing other performance targets.

We evaluate our *disaggregated* approach on a LBM fluid solver running on single- and multi-GPU systems, achieving:

– A zero-copy multi-GPU implementation that overlaps computation and communication for dense discretizations, **minimizing transfer overhead** and delivering up to a 3× speedup over state-of-the-art solutions.
– A disaggregated interface and layout for block-sparse data structures that **reduce high register pressure** in complex boundary conditions (e.g., regularized LBM [10]), achieving up to a 2× speedup over naive implementations without extra boundary-data storage.
– A multi-resolution grid representation that **maximizes kernel fusion** in regions unaffected by neighboring cells of different sizes, yielding up to a 26% performance improvement on a single GPU.

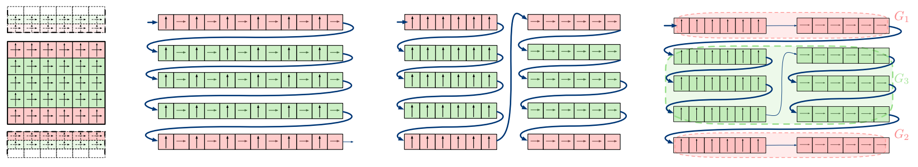
In Section 2, we formalize the disaggregated design methodology. Sections 3, 4, and 5 apply this approach to dense, sparse, and multi-resolution volumetric grids, respectively. Section 6 presents our evaluation using LBM solvers across multiple GPU architectures. We discuss related work in Section 7 and conclude in Section 8.

2 Disaggregated Design Method

Voxel-based representations, derived from Cartesian discretization, include **dense** (every voxel in a multidimensional interval is allocated), **sparse** (an irregular subset of the interval), and **multi-resolution** (voxels of different sizes in a single interval). Traditional design efforts have focused on data locality to reduce the growing gap between compute speed and memory latency. However, other performance-critical optimizations—e.g., minimizing communication overhead, reducing register pressure, and maximizing kernel fusion—are equally important. To address these, we introduce *disaggregated design*, a multi-objective method for volumetric data structures defined as follows:

Definition 1. *Given an optimization objective Φ to be considered alongside data locality, a disaggregated design maps data over a voxelized domain into a 1D memory space in four steps:*

1. ***Definition:*** *Identify properties $\mathcal{P}1, \ldots, \mathcal{P}n$ influencing Φ.*
2. ***Classification:*** *Group voxels $\mathcal{G}_1, \ldots, \mathcal{G}_n$ based on those properties.*
3. ***Mapping:*** *Within each group $\mathcal{G}_i$, map voxel data to memory using classical data-locality optimizations.*



(a) (b) AoS (c) SoA (d) Disaggregated

Fig. 1: Illustration of a five-point stencil on a two-component vector field in a 2D domain with partitioning along one axis (a) and three different memory layouts (b,c,d).

4. ***Operations:*** *Apply group-specific operations to maximize Φ.*

By incorporating objectives beyond locality (**definition**), grouping voxels accordingly (**classification**), and then applying classical optimizations locally (**mapping**), we enable targeted **operations** that yield higher end-to-end performance. Success depends on whether gains from optimizing Φ_i outweigh potential drawbacks, e.g., sub-optimal locality since locality may suffer if inter-group optimizations are underutilized or increased complexity since additional indexing is needed for separate groups.

We study disaggregated design within a **for-each** data-parallel pattern applying a side-effect-free function to each voxel. Under this model, we consider three compute patterns: (1) *Map Pattern* where each voxel depends only on local data, (2) *Uniform Stencil Pattern* where each voxel queries its neighbors (e.g., convolution), and (3) *Multi-resolution Stencil Pattern* where varying voxel sizes require neighbor access at different resolutions.

In the following, we apply disaggregated design to dense (Section 3), sparse (Section 4), and multi-resolution representations (Section 5). We then detail its performance impact on a fluid simulation solver in Section 6.

3 Disaggregation on a Dense Domain

In this setup, a dense grid is divided across multiple GPUs, each handling one partition. In stencil computations that require neighbor data, fetching data directly from neighboring GPUs each iteration is highly inefficient due to communication overhead. To mitigate this, each partition maintains a *halo region* (Figure 1a). Synchronizing these halos (the *halo update* [16]) can significantly add to execution time if performed before every stencil step. *Overlapping Computation and Communication* (OCC) addresses this by dividing the stencil update into two phases. First, **private** voxels (relying only on local data) are processed while the halo is updated in parallel. Then, **shared** voxels (requiring neighbor data) are computed. This hides communication costs and improves scalability on multi-GPU systems.

We adopt a communication model [3] with a constant setup time, t_{setup}, plus a term proportional to message size ($size$(msg)) and the interconnect throughput, b_{com}, such that $t_{send}(\text{msg}) = t_{setup} + \frac{size(\text{msg})}{b_{com}}$. If exchanged data resides in disjoint memory regions, multiple transfers are required.

We consider a 2D stencil on a vector field, where each point stores a 2D vector. Two common layouts are *Array-of-Structures* (AoS) and *Structure-of-Arrays* (SoA). SoA typically yields better coalesced GPU memory access [21]. For a grid of size $d_x \times d_y$, partitioned along one dimension (Figure 1a), the halo-update time for a generic partition can be approximated as:

$$t_{halo\_update} = \alpha t_{setup} + \beta \frac{size(T)}{b_{com}} \tag{1}$$

where α is the number of transfer operations, and β the total number of elements sent. With a 1D decomposition, $\alpha \geq 2$ (upper and lower neighbors), and $\beta = 2 \cdot d_y$ for shared boundary elements.

Using AoS (Figure 1b) keeps shared-voxel data contiguous, minimizing α to 2, but it breaks coalesced GPU access [21]. Conversely, SoA (Figure 1c) preserves coalesced accesses but splits data, increasing α to 4. This increase occurs because the 2D components are stored non-contiguously in memory, as illustrated by the four distinct regions in Figure 1c. To reduce the number of communication operations, the data would need to be copied into a contiguous buffer.

	α	β	**Coalesced**
AoS	2	$2 \cdot d_x$	No
SoA	4	$2 \cdot d_x$	Yes
Disag SoA	2	$2 \cdot d_x$	Yes

Table 1: Comparison of disaggregated, AoS, and SoA layouts for a five-point stencil using the model in Eq. 1. The 2D domain has dimensions $d_x \times d_y$, with a 1D partition along the y-axis.

To reduce α while retaining coalesced accesses, we apply *disaggregated design*. We define $\mathcal{P}_1$ and $\mathcal{P}_2$ to enforce contiguous mapping for voxels shared with the upper ($\mathcal{G}_1$) and lower ($\mathcal{G}_2$) neighbors, while remaining private voxels form $\mathcal{G}_3$ (Figure 1d). We then map each group in SoA format, preserving $\mathcal{P}_1$ and $\mathcal{P}_2$.

Table 1 compares these layouts, showing that the *disaggregated SoA* merges the benefits of AoS (minimal transfers) with SoA's coalesced memory access.

4 Disaggregation on a Sparse Domain

When the region of interest in a simulation domain is significantly smaller than the full domain, dense representations become inefficient. In these scenarios, *sparse* representations are preferred, allocating data only for actively used voxels and thus conserving memory and compute resources. A common use case in sparse domains involves handling boundary conditions in physics solvers, where computations on each voxel may vary based on its boundary type. For instance, in computational fluid dynamics, no-slip conditions are enforced on boundary voxels at walls, whereas interior (non-boundary) voxels typically follow the Navier–Stokes equations.

The ratio of boundary to total voxels is often small, as boundaries generally scale with surface area rather than volume. Here, we assume a block-sparse

	# Kernels	# Blocks	# Registers	Storage	Indexing
Naive	1	$n_b + n_{nb}$	r_b	$s_w n_{nb} b_{\text{size}}$	Direct
Disag - Bitmask	2	$n_b + n_{nb}$ $n_b + n_{nb}$	r_b r_{nb}	$s_i(n_b + n_{nb})b_{\text{size}}$	Indirect
Disag - Mem	2	n_b n_{nb}	r_b r_{nb}	0	Direct

Table 2: Comparison of the disaggregated design vs. a naive approach for a map pattern involving complex boundary conditions in a block-sparse representation. # Kernels is the number of kernels launched; # Blocks is the number of blocks per kernel; # Registers is the register usage; Storage quantifies additional space needed for boundary metadata (s_w is the memory size per boundary voxel, b_{size} is the number of voxels in a block, and s_i is the size of the indexing type).

representation (commonly managed by space-filling curves) though other layouts are possible. The computational load on boundary voxels varies based on their type, introducing a few challenges for efficient GPU implementations:

- **Register pressure:** Additional registers may be needed for boundary computations, decreasing kernel occupancy.
- **Memory overhead:** Managing boundary conditions often requires per-voxel metadata, increasing memory requirements.

Let r_{nb} be the resource needs for $\underline{\text{n}}$on-$\underline{\text{b}}$oundary computations and r_{b} for $\underline{\text{b}}$oundary computations. We focus on the practical case $r_{\text{b}} > r_{\text{nb}}$, which can degrade performance by reducing occupancy and increasing memory usage.

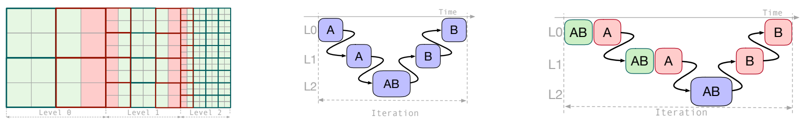
Naive Approach A naive solution launches a single kernel for all voxels. Because boundary logic is included, the kernel's resource demand is $r = \max(r_{\text{nb}}, r_{\text{b}}) = r_{\text{b}}$. Even though most voxels only need r_{nb}, the kernel is constrained by r_{b}. This often leads to suboptimal occupancy and high memory usage, i.e., allocating a full buffer for all voxels, even though only some are boundary voxels.

Disaggregated Approach Sparse domains typically exhibit heterogeneous workloads, with boundary voxels requiring significantly more resources. The *disaggregated design* alleviates this by separating the domain into two groups:

- **Boundary Group** ($\mathcal{G}_{\text{b}}$): Blocks containing at least one boundary voxel.
- **Non-Boundary Group** ($\mathcal{G}_{\text{nb}}$): Blocks with only non-boundary voxels.

This separation allows two specialized kernels, each optimized for its target group. Next, we consider two implementations of this concept:

Memory-Based Grouping: All boundary blocks are contiguous in memory, followed by non-boundary blocks. Each group is processed by a separate kernel, removing the need for runtime checks on block type and simplifying memory access.



(a) Multi-res domain (b) Comp. Graph (c) Disag. Comp. Graph

Fig. 2: A multi-resolution domain with three levels (a) where red blocks lie near resolution transitions, and green blocks are farther away. The computational graph (b) shows dependencies between two kernels: both run on red and green blocks, but kernel B waits for cross-level boundary data. Fusing kernels is traditionally feasible only at the finest resolution level. In the disaggregated approach (c), green blocks fuse computations at any level, while red blocks execute two kernels sequentially once boundary data is available, reducing iteration time.

Bitmask-Based Grouping: In this implementation, we use a bitmask at runtime to distinguish block types. Since the spans of the two groups are no longer contiguous, both kernels must execute over the entire domain. Memory for boundary-specific data is allocated using *indirect indexing*, where a unique identifier is assigned to each voxel. This identifier maps boundary voxels to their metadata, which is stored in a contiguous buffer.

Table 2 summarizes these approaches. We examine the performance results and trade-offs of these two implementations in Section 6.2.

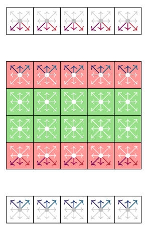
5 Disaggregation on a Multi-resolution Domain

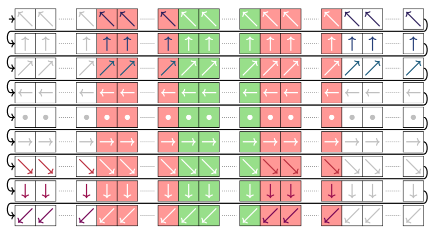
Multi-resolution data structures handle voxels of varying sizes in one domain (Figure 2), supporting both *intra-level* stencil operations (within a single resolution) and *cross-level* interactions (between adjacent resolutions).

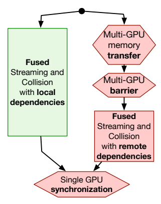
During each time step in multi-resolution solvers [9], only voxels near resolution boundaries require cross-level communication. Figure 2 distinguishes green (intra-level only) from red (cross-level) voxels. Due to producer/consumer dependencies, iterations are typically split into two steps and can be fused only at the finest level (Figure 2b).

Using the disaggregated design, we improve memory throughput by maximizing kernel fusion for intra-level computations. Voxels far from resolution jumps do not need cross-level data, so their iterations can fuse at any resolution level. To formalize this, we define a discrete distance property, $\mathcal{P}_d$, measuring how close a voxel is to a resolution jump (distance 0 indicates immediate proximity). Each resolution level is partitioned into: (1) $\mathcal{G}_{\mathrm{i}}$: Blocks where all voxels have distance ≥ 1, allowing fully fused operations, and (2) $\mathcal{G}_{\mathrm{c}}$: Blocks with at least one voxel at distance 0, requiring separate cross-level and intra-level steps.

We apply a standard memory locality layout to each group within each level. Under disaggregation, $\mathcal{G}_{\mathrm{i}}$ blocks use fused kernels across resolution levels, whereas $\mathcal{G}_{\mathrm{c}}$ blocks wait for boundary data (Figure 2c). This approach minimizes memory pressure for $\mathcal{G}_{\mathrm{i}}$ while preserving accurate cross-level operations for $\mathcal{G}_{\mathrm{c}}$.


(a) 1D Partition


(b) SoA Layout


Fused Streaming and Collision with local dependencies
Multi-GPU memory transfer
Multi-GPU barrier
Fused Streaming and Collision with remote dependencies
Single GPU synchronization
(c) Comp. Graph

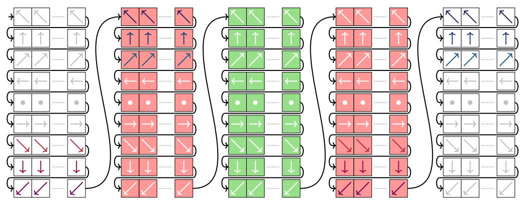

(d) Disaggregated

Fig. 3: (a) 1D-partitioned LBM grid: white arrows show local dependencies, and red/blue arrows show dependencies from upper/lower partitions. (c) Computation graph with OCC in the reference implementation. (b,d) SoA and disaggregated SoA layouts for a D2Q9 lattice in 2D, with black arrows indicating memory mapping.

6 Evaluation and Discussions

We evaluate the disaggregated design method using a fluid dynamics simulation based on LBM on dense, sparse, and multi-resolution grids. We selected LBM as a representative application since it could benefit from many of the objectives our design method targets. LBM models the time evolution of *velocity distribution functions* (f_i) along discrete lattice directions $e_i = (e_1, \ldots, e_q)$. In 3D, we use lattices with 19 (D3Q19) or 27 (D3Q27) directions. Each f_i value, or *population*, evolves through a *collide-and-stream* process. *Collision* is a nonlinear, local operation that modifies f_i at each lattice point. Here, we employ the BGK single-relaxation-time model for the collision [8]. *Streaming* is a non-local advection of f_i values along each of the Q discrete directions via a stencil.

In optimized GPU LBM implementations, collision, streaming, and boundary conditions are often fused into a single kernel [6]. We use the work of Meneghin et al. [16] as our baseline since they achieve state-of-the-art results on single- and multi-GPU.

6.1 Improving LBM Scalability

We evaluate our disaggregated design on a lid-driven cavity flow problem [12] within a cubic domain, using LBM and a dense voxel representation on single-node multi-GPU systems. We analyze both a theoretical communication model (Section 3) and runtime performance.

Reference Implementation In single-node multi-GPU systems, inter-GPU communication can occur via PCI or faster interconnects like NVLink. Here, we use native `cudaMemcpyPeer` for best performance [7]. Because these systems typically house up to 8–16 GPUs, we use a 1D partitioning scheme [7], giving each partition at most two neighbors (upper and lower) and enabling efficient zero-copy memory transfers.

Figure 3a shows LBM data dependencies with green *private* voxels (computed locally) and red *shared* voxels (requiring data from an adjacent partition). At the lattice granularity, some populations remain local (white or gray), while others must be exchanged (red or blue). Only certain populations of each shared voxel are transferred, which is a defining feature of the LBM streaming operation.

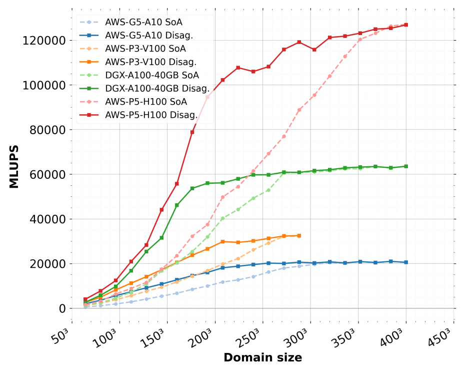
Efficient OCC is critical for achieving fine-grain scalability in LBM [16]. Figure 3c show the computation graph where private-voxel computation is overlapped with halo exchanges for shared voxels, hiding latency and improving performance. This OCC-based implementation is our baseline reference.

Modeling Communication Overhead The parameters α and β from Eq. 1 vary with LBM lattice and data layout (Table 3). Under AoS, all populations in a voxel are contiguous, yielding $\alpha = 2$ transfers (one for each neighbor), but forcing β to include unneeded populations. Conversely, in SoA (Figure 3b), populations are contiguous *per direction*, increasing α but minimizing β. Table 3 shows that AoS has a smaller α but higher β, while SoA has the opposite trade-off.

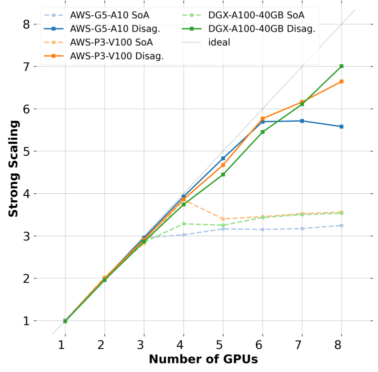
	D2Q9		**D3Q19**		**D3Q27**	
	α	β	α	β	α	β
AoS	2	18s	2	38s	2	54s
SoA	6	6s	10	10s	18	18s
Disag SoA	2	6s	2	10s	2	18s

Table 3: LBM communication parameters for Eq. 1; s is half the shared voxels: d_x in 2D and $d_x \cdot d_y$ in 3D.

Disaggregated Optimization We now extend the disaggregated layout introduced for stencil operations on a vector-valued field to fully support *zero-copy* communication by including halo regions. These halos enable direct data sharing without additional staging buffers. Concretely, we define distinct properties to ensure contiguous mappings for each critical region: upper halos, upper boundary voxels, lower boundary voxels, and lower halos. As with our original design, each of these groups is mapped using an SoA layout. Figure 3d shows both how the domain is split into groups and how these groups are placed in memory. In this arrangement, any data that needs to be transferred or received resides contiguously—the solid-colored (red or blue) populations in Figure 3d. This means each group's data is placed in a continuous block, allowing for a minimal number of bulk transfers. In the D2Q9 example, this disaggregated SoA configuration results in an α value of 2, so each partition



(a) D3Q19, speedup



(b) D3Q27, strong scaling

Fig. 4: (a) MLUPS performance of disaggregated vs. SoA on an 8-GPU lid-driven cavity flow (D3Q19). (b) Strong scaling for D3Q27 on a 192^3 domain.

sends just one message per neighbor. Meanwhile, β remains at 6, representing the exact amount of data required by the stencil. Table 3 shows that for D3Q19 and D3Q27, the disaggregated layout delivers similarly optimal α and β values, consistently outperforming basic AoS or SoA alone. Overall, the disaggregated SoA layout combines AoS-like benefits of minimal transfer operations with SoA's advantage of transferring only the necessary populations. By maintaining zero-copy efficiency, it reduces overhead in inter-partition data exchanges, making it theoretically the most communication-efficient layout for multi-GPU LBM.

Name	Arch	GPUs	Mem	Interc.
DGX-A100	A100-SXM4	8	40GB	NVLink-2
AWS p3	V100-SXM2	8	16GB	NVLink-1
AWS g5	A10	8	24GB	PCI

Table 4: Machines used in benchmarking.

Benchmarking We measure runtime on a lid-driven cavity flow with a cubic domain, using boundary conditions from Latt et al. [12]. Table 4 lists three single-node multi-GPU systems tested, spanning high-end (A100), midrange (A10), and previous-generation (V100) GPUs. A100 and V100 use NVLink; A10 relies on PCI. We exclude AoS due to poor coalesced performance in LBM. Figure 4a compares disaggregated and SoA layouts for 3D D3Q19 and D3Q27 lattices, using the Million Lattice Updates per Second (MLUPS) metric.

Disaggregated consistently matches or outperforms SoA, especially on smaller domains: up to 4× speedup below 150^3, about 2.5× from 150^3–250^3, and 1.5× for larger volumes. This reduction in performance improvement for larger domains is well explained by Eq. 1: with 1D partitioning, the impact of β (the amount of data transferred) increases with domain size while the number of private voxels grows cubically with the domain edge length L—significantly increasing the

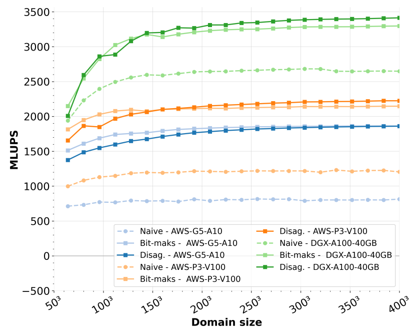


Fig. 5: Performance comparison (single GPU) between the naive implementation and the disaggregated approach (both bitmask and continuous block allocations). The domain uses a D3Q27 lattice on a block-sparse grid, with regularized inflow and outflow boundaries [11].

amount of computation that can be overlapped with communication. Finally, Figure 4b shows strong scaling for 192^3 domains. While traditional methods struggle to exceed $3\times$ scaling from a single GPU, the disaggregated layout consistently achieves $6\times$ or more, regardless of GPU architecture.

6.2 Improving LBM Register Allocation

We evaluate the effect of disaggregation on register usage by simulating fluid flow over an obstacle in a cubic domain. The simulation uses a block-sparse grid, with each block containing 4^3 voxels. This setup resembles a typical wind tunnel, where a *bounce-back* boundary condition [13] is applied to obstacle surfaces. Additionally, an *inflow* boundary condition is applied to one face, and an *outflow* boundary condition is applied to the opposite face.

The bounce-back boundary condition is *register-light*, requiring $2Q$ populations. By contrast, the inflow and outflow faces use a *regularized* boundary condition [11], which is *register-heavy*, needing $3Q$ populations. Combining bounce-back and regularized boundaries in one kernel forces resource requirements to accommodate $\max(2Q, 3Q) = 3Q$, potentially over-allocating registers for most voxels. Because the fraction of regularized voxels is $\mathcal{O}(\frac{1}{x})$ relative to domain size x, this wastes registers on the majority of the grid.

Disaggregated Solution. To address this, we classify voxels needing the regularized method as *boundary*, while bounce-back and other voxels are *non-boundary*. Any block containing at least one regularized voxel is marked a *boundary block*; the rest are *non-boundary blocks*. We then assign each category to its own specialized kernel, reducing register pressure for non-boundary blocks. This layout can be implemented by storing boundary blocks contiguously in memory or by using a bitmask to identify them. Both approaches let non-boundary voxels be processed using fewer registers, thus improving occupancy. Table 2 summarizes these advantages, showing how disaggregation avoids register-related bottlenecks.

Benchmarks We ran this scenario on a single GPU from each system in Table 4, letting CUDA compiler determine register allocation and spilling. Figure 5 shows the performance (MLUPS) for D3Q27 across different domain sizes. In every tested case, the disaggregated solutions (bitmask or memory-based) outperform the naive approach. Gains can reach 2× on V100 and A10 and 1.3× on A100. On V100, the naive kernel needs 55 registers—matching the boundary kernel in the disaggregated case—and suffers spills for the entire domain. Under disaggregation, only the smaller boundary block regions invoke the 55-register kernel, while the majority use a lower-resource kernel. For a domain of size 368, the naive approach has 2.2× more L2-DRAM traffic, explaining the 2× speedup.

On A100, all kernels (naive and disaggregated) also require 55 registers. Spilling occurs only in the naive approach, yet the A100's larger L2 cache reduces its overall penalty, limiting speedup to around 1.3×. Still, data traffic between L2 and DRAM is 1.3× higher for the naive kernel, matching the measured performance improvement. Finally, regarding memory overhead for boundary conditions, the regularized method stores a d-component velocity vector (where $d = 2$ in 2D and $d = 3$ in 3D). Let F be the floating-point type and I the indexing type. Then $s_w = \text{sizeOf}(F) \cdot d$ and $s_i = \text{sizeOf}(I)$. As noted in Table 2, only the disaggregated approach avoids additional storage for boundary voxels; other methods typically allocate boundary-related data throughout the entire domain.

6.3 Improving Multi-resolution LBM Kernel Fusion

We evaluated a single-GPU optimized multi-resolution LBM solver [14] enhanced with the disaggregated design. The data structure comprises multiple uniform block-sparse grids (one per resolution level), along with transition metadata to manage inter-level dependencies.

GPU	Size	Distribution	Ours	Baseline	**Gain**
A100	512^3	77, 4, 0.4	6072	4824	**25%**
A100	512^3	73, 3, 0.5, 0.003	6018	4769	**26%**
V100	320^3	15, 1, 0.1	4421	3770	**17%**
V100	320^3	15, 1, 0.1, 0.002	4422	3770	**17%**
V100	480^3	53, 4, 0.4	5006	4047	**23%**
V100	480^3	53, 4, 0.3, 0.008	5005	4050	**23%**
A10	320^3	15, 1, 0.1, 0.002	4083	3982	**2%**
A10	480^3	53, 4, 0.3, 0.008	4483	4306	**4%**
A10	512^3	77, 4, 0.4	4093	3901	**4%**
A10	512^3	73, 3, 0.5, 0.003	3890	3719	**4%**

Table 5: Performance comparison of our disaggregated multi-resolution LBM approach vs. a state-of-the-art baseline [14] on a lid-driven cavity flow. *Size* is the length of the virtual finest-level box; *Distribution* lists the fraction of active voxels per level (finest to coarsest); *Baseline* is baseline's MLUPS; *Ours* is our MLUPS; and *Gain* is computed as $(\mathit{Ours}-\mathit{Baseline})/\mathit{Baseline}\times 100\%$.

In multi-resolution LBM, each time step entails two key inter-level operations: (1) *Explosion* where collision results at one resolution feed into lower-resolution levels and (2) *Coalescence* where streaming data from higher resolution levels merges into lower-resolution blocks.

These create dependency edges in a graph similar to Figure 2b, with operator A for collision and B for streaming. Except at the finest resolution, explosion and coalescence prevent kernel fusion without additional processing.

We classify blocks as: (1) $\mathcal{P}_{\text{uniform}}$ which are blocks that operate uniformly and don't require inter-level synchronization and (2) $\mathcal{P}_{\text{jump}}$ which are blocks containing at least one voxel near a resolution jump, needing explicit explosion/-coalescence handling. Using the disaggregated interface, we eliminate unneeded synchronizations at the kernel level. For $\mathcal{P}_{\text{uniform}}$ blocks, we fuse collision and streaming into a single kernel. For $\mathcal{P}_{\text{jump}}$ blocks, the original multi-step execution is preserved to correctly process inter-level data.

Table 5 shows a lid-driven cavity flow benchmark at multiple resolutions on three different GPUs. In all configurations, the disaggregated approach outperforms the baseline. On the A100, a high-end architecture, improvements can reach 26% for domains of size 512^3. The V100 also achieves up to 23% on 480^3 grids. These speedups stem from merging collision and streaming on uniform blocks, thus reducing overhead where inter-level dependencies are not needed.

In contrast, the A10 exhibits smaller gains (2–4%), largely due to register spilling penalizing performance more acutely on midrange GPUs with smaller cache sizes and lower memory bandwidth. Nonetheless, the disaggregated method consistently demonstrates advantages for larger domains and higher active-voxel counts, confirming its suitability for diverse multi-resolution setups.

7 Related Work

While extensive research has been conducted on optimizing stencil computations, most efforts focus on improving data locality. To the best of our knowledge, this work is the first to propose a data structure design methodology aimed at multi-objective optimization, where data locality can be strategically traded off for other performance goals.

Optimizing communications via data layout: Zhao et al. [24] introduced a layout scheme for block representations designed to minimize communication overhead and enable zero-copy communication. Their approach incorporates virtual memory techniques to reduce the impact of indirect indexing which is effective for stencils with a radius that is a multiple of four. However, for stencils with a radius of one, users must resort to time tiling, which can increase message sizes or become infeasible if reductions are involved. While their method demonstrates significant speedups on distributed systems, it does not address the challenges of multi-cardinality fields.

Reducing Register Pressure and Spilling: Managing register pressure and minimizing spilling are critical challenges in GPU computing. Temporal blocking techniques, e.g., register blocking, serialize one domain dimension to improve data reuse [15]. Other strategies leverage GPU shared memory to mitigate the impact of spilling [17]. However, no prior work has explored using data structure design to address register pressure.

Kernel Fusion: Kernel fusion is a well-known optimization for memory-bound problems, as it reduces memory pressure by keeping shared data in registers between consecutive kernels. This technique is widely used in dense LBM implementations [12] and multi-resolution representations [18]. However, existing works do not explore leveraging data layouts to facilitate kernel fusion.

8 Conclusion and Future Work

Past advances in volumetric computation have helped data-structure designers create layouts that emphasize memory access efficiency. In this work, we introduced *disaggregated design*, a unified framework that broadens the optimization focus beyond data locality to include minimizing multi-GPU data transfers, reducing register pressure, and maximizing kernel fusion. Our analytical models and empirical results confirm the benefits of these objectives, while also clarifying the potential drawbacks of more intricate indexing schemes and slightly compromised locality. Ultimately, the overall gains depend on whether the performance improvements outweigh these costs.

This study represents the first in-depth analysis of disaggregated design and its practical scope. Several directions remain for future exploration: (1) extending the applicability of the disaggregated design to other spatial data structures, e.g., unstructured meshes and hash grids, and (2) exploring additional optimization objectives tailored to diverse computational workloads, e.g., load balancing for particle-based simulations.

Acknowledgments. The authors express their gratitude to Hesam Salehipour, Mehdi Ataei, and Oliver Hennigh for their invaluable insights into the intricacies of LBM and their support in providing data on H100 architecture. Ahmed Mahmoud acknowledges the generous support of National Science Foundation grant OAC-2403239.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Communications of the ACM **52**(10), 56–67 (Oct 2009). https://doi.org/10.1145/1562764.1562783
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 101–113. PLDI '08, Association for Computing Machinery, New York, NY, USA (Jun 2008). https://doi.org/10.1145/1375581.1375595
3. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: towards a realistic model of parallel computation. SIGPLAN Not. **28**(7), 1–12 (Jul 1993). https://doi.org/10.1145/173284.155333
4. Endo, T.: Applying recursive temporal blocking for stencil computations to deeper memory hierarchy. In: 2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA). pp. 19–24 (Nov 2018). https://doi.org/10.1109/NVMSA.2018.00016

5. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. ACM Trans. Algorithms **8**(1) (Jan 2012). https://doi.org/10.1145/2071379.2071383
6. Geier, M., Schönherr, M.: Esoteric Twist: An efficient in-place streaming algorithmus for the lattice boltzmann method on massively parallel hardware. Computation **5**(2), 19 (Mar 2017). https://doi.org/10.3390/computation5020019
7. Kraus, J.: Multi-GPU programming models, https://www.nvidia.com/en-us/on-demand/session/gtcfall21-a31140/
8. Krüger, T., Kusumaatmaja, H., Kuzmin, A., Shardt, O., Silva, G., Viggen, E.M.: The Lattice Boltzmann method. Springer Cham, 1st edn. (2016). https://doi.org/10.1007/978-3-319-44649-3
9. Lagrava, D., Malaspinas, O., Latt, J., Chopard, B.: Advances in multi-domain lattice Boltzmann grid refinement. Journal of Computational Physics **231**, 4808–4822 (may 2012). https://doi.org/10.1016/j.jcp.2012.03.015
10. Latt, J., Chopard, B., Malaspinas, O., Deville, M., Michler, A.: Straight velocity boundaries in the lattice Boltzmann method. Physical Review E **77**(5), 056703 (2008)
11. Latt, J., Chopard, B., Malaspinas, O., Deville, M., Michler, A.: Straight velocity boundaries in the lattice Boltzmann method. Physical Review E **77** (May 2008). https://doi.org/10.1103/PhysRevE.77.056703
12. Latt, J., Coreixas, C., Beny, J.: Cross-platform programming model for many-core lattice Boltzmann simulations. PLOS ONE **16**(4), 1–29 (04 2021). https://doi.org/10.1371/journal.pone.0250306
13. Lavallée, P., Boon, J.P., Noullez, A.: Boundaries in lattice gas flows. Physica D: Nonlinear Phenomena **47**(1), 233–240 (1991). https://doi.org/10.1016/0167-2789(91)90294-J
14. Mahmoud, A.H., Salehipour, H., Meneghin, M.: Optimized GPU implementation of grid refinement in lattice Boltzmann method. In: Proceedings of the 38th IEEE International Parallel and Distributed Processing Symposium. pp. 398–407 (Jul 2024). https://doi.org/10.1109/IPDPS57955.2024.00042
15. Matsumura, K., Zohouri, H.R., Wahib, M., Endo, T., Matsuoka, S.: AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. pp. 199–211. CGO '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368826.3377904
16. Meneghin, M., Mahmoud, A.H., Jayaraman, P.K., Morris, N.J.W.: Neon: A multi-GPU programming model for grid-based computations. In: Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium. pp. 817–827 (Jun 2022). https://doi.org/10.1109/IPDPS53621.2022.00084
17. Sakdhnagool, P., Sabne, A., Eigenmann, R.: Optimizing GPU programs by register demotion: poster. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 405–406. PPoPP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293883.3297859
18. Schornbaum, F., Rüde, U.: Massively parallel algorithms for the lattice boltzmann method on nonuniform grids. SIAM Journal on Scientific Computing **38**, C96–C126 (2016). https://doi.org/10.1137/15M1035240
19. Tran, N.P., Lee, M., Hong, S.: Performance optimization of 3d lattice Boltzmann flow solver on a GPU. Scientific Programming **2017**(1) (Jan 2017). https://doi.org/10.1155/2017/1205892

20. Wang, X., Qiu, Y., Slattery, S.R., Fang, Y., Li, M., Zhu, S.C., Zhu, Y., Tang, M., Manocha, D., Jiang, C.: A massively parallel and scalable multi-GPU material point method. ACM Trans. Graph. **39**(4) (Aug 2020). https://doi.org/10.1145/3386569.3392442
21. Wittmann, M., Zeiser, T., Hager, G., Wellein, G.: Comparison of different propagation steps for lattice Boltzmann methods. Comput. Math. Appl. **65**(6), 924–935 (Mar 2013). https://doi.org/10.1016/j.camwa.2012.05.002
22. Wonnacott, D.: Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In: Proceedings 14th International Parallel and Distributed Processing Symposium. pp. 171–180. IPDPS 2000 (May 2000). https://doi.org/10.1109/IPDPS.2000.845979
23. Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. ACM SIGARCH Computer Architecture News **23**(1), 20–24 (Mar 1995). https://doi.org/10.1145/216585.216588
24. Zhao, T., Hall, M., Johansen, H., Williams, S.: Improving communication by optimizing on-node data movement with data layout. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 304–317. PPoPP '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437801.3441598