BLAEQ: A Multigrid Index for Spatial Query on Geometry Data

Song Wang
wangsong@tsinghua-eiri.org
EIRI, Tsinghua University
School of Computer Science, Wuhan University
Chengdu, China

Chen Wang*
wang_chen@tsinghua.edu.cn
NERCBDS, EIRI, Tsinghua University
Beijing, China

Jianchun Wang
wangjianchun@cssrc.com.cn
China Ship Scientific Research Center
Wuxi, China

Shengguo Li
nudtlsg@nudt.edu.cn
National University of Defense Technology
Changsha, China

Rui Li
lirui@tsinghua-eiri.org
EIRI, Tsinghua University
Chengdu, China

Zhiyong Peng
peng@whu.edu.cn
School of Computer Science, Wuhan University
Wuhan, China

ABSTRACT

The efficiency of spatial queries is pivotal for the analysis of geometry data in the fields such as computational simulation, point cloud processing and digital engineering. Utilizing the computational capabilities of modern hardware, such as GPUs, offers a promising avenue for accelerating spatial query processing. However, conventional tree-based indexing methods are not optimized for maximal exploitation of GPU resources.

To address this problem, we introduce BLAEQ, a multigrid index designed to maximize the potential of GPUs. BLAEQ adopts a multigrid strategy, which represents an index tree with vectors as layers and matrices as connectors. Although BLAEQ shares conceptual similarities with traditional tree-based indexes, its innovative multigrid architecture facilitates effective parallelization on GPUs during the query phase. To optimize GPU utilization, BLAEQ is entirely constructed using BLAS (Basic Linear Algebra Subprograms), leveraging the efficiency of hardware-tuned BLAS libraries like CuBLAS. This design confers BLAEQ with enhanced performance over existing spatial query methods.

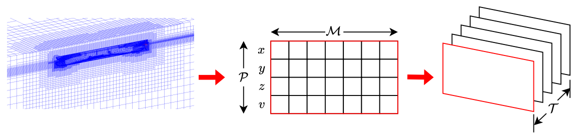
Our study assesses BLAEQ's performance against state-of-the-art spatial query techniques using a range of both real-world and synthetic datasets. The experimental outcomes demonstrate that BLAEQ outperforms the benchmark approaches in terms of query efficiency on geometry data.

PVLDB Reference Format:
Song Wang, Chen Wang, Jianchun Wang, Shengguo Li, Rui Li, and Zhiyong Peng. BLAEQ: A Multigrid Index for Spatial Query on Geometry Data. PVLDB, 18(8): 2533 - 2546, 2025.
doi:10.14778/3742728.3742746

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ft1573734/BLAEQ_CUDA.

*Chen Wang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742746



(a) Original Geometry Grid (b) Multi-dimensional Array (c) Post-processing Analysis

Figure 1: Demonstration of the data format of a CFD ship model.

1 INTRODUCTION

Geometry data processing has found applications in critical domains such as computational simulations [45, 49, 55], point cloud processing [16, 19, 25, 68], and digital engineering [30, 61]. These applications often demand flexible and efficient data access strategies, particularly in computationally intensive environments. To address this challenge, robust indexing methods for geometry data have become indispensable. To illustrate the necessity of such approaches, we examine a specific use case in computational simulation.

1.1 CFD Example

A computational simulation uses 3-D geometry models, called *meshes* or *grids*[1], composed of small cells to simulate real-world objects. To achieve high accuracy, these meshes are finely detailed, often containing millions of cells. Figure 1(a) illustrates a ship model and surrounding fluid body used in CFD (Computational Fluid Dynamics), a common simulation application.

The simulation calculates physical parameters (e.g., velocity, divergence) for each cell based on governing laws, assigning values that define the output data structure. Let $\mathcal{P}$ denote the set of physical parameters and $\mathcal{M}$ the 3-D model. The result is a multi-dimensional array of shape $|\mathcal{P}|\times|\mathcal{M}|$, representing a single timestep. Figure 1(b) shows an example with $\mathcal{P} = \{x, y, z, v\}$.

For dynamic simulations (e.g., a ship accelerating through water), a time dimension $\mathcal{T}$ is added, discretized into consistent intervals. The data structure then comprises $|\mathcal{T}|$ files, each of size $|\mathcal{P}| \times |\mathcal{M}|$,

[1]In computational simulation, mesh and grid are interchangeable terms.

known as post-processing files [18]. These files capture the temporal evolution of the simulation, as shown in Figure 1(c).

1.2 Workload Characteristics

Many analytical tasks necessitate the iterative examination of the mesh [3, 15, 35]. As an illustrative example, we employ a specialized analytical task known as *fluid flow analysis*. This task involves visualizing the path of fluid particles, allowing analysts to study the trajectories and assess the digital design of a ship.

During the analysis process, it is often more efficient for analysts to concentrate on a particular area, such as the region around the propeller, where fluid flow dynamics are particularly complex. The process of focusing on this area is a classic *spatial range query*. Specifically, if we define the post-processing file as **M**, then the retrieval of data for this task can be executed through a distinct query:

```
Q: SELECT v FROM M WHERE x in (x_min, x_max)
   & y in (y_min, y_max) & z in (z_min, z_max).
```

Where (x_{min}, x_{max}), (y_{min}, y_{max}) and (z_{min}, z_{max}) encloses the propeller area.

For a comprehensive analysis of the entire post-processing dataset, analysts must perform this query iteratively for each time step. Overall, the data access workload can be expressed as follows:

```
For t in T:
    execute Q on M_t.
```

Based on the application background of the workload, we briefly summarize their key characteristics:

- **Repetitive**. An analytical task often involves analyzing the whole post-processing data. Therefore, the queries need to be repetitively executed on all time steps.
- **Geometrical**. Filtering cells based on geometrical features $(x, y, z)$is a widely used query predicate when analyzing large meshes.
- **Immutable**. In many geometry data analysis use cases, such as in the simulation of a ship, or creating the 3-D model of a building, the geometry model is immutable, leading to no future update onto the mesh.

The repetitive nature of the workload underscores the criticality of the efficient execution of query Q. Even a slight decrease in the efficiency of Q can be magnified by the scale of time steps $\mathcal{T}$, resulting in a substantial decline in overall performance.

1.3 Limitations of Existing Solutions

While existing spatial query techniques can be adapted for use in computational simulation queries, both on-CPU and on-GPU solutions encounter significant limitations. We categorize these limitations as follows:

On-CPU spatial query solutions, despite their versatility, are constrained by the computational limitations of CPUs [13]. Techniques like the KD-Tree, R-Tree, and Octree [42] are highly effective for smaller datasets but suffer from performance degradation as the dataset size increases. For large-scale meshes, even SOTA (state-of-the-art) on-CPU solutions, such as the learned index [43], fall short, as we will demonstrate in experiments in Section 5.

The use of GPUs to enhance spatial query performance has gathered significant attention [42]. Nevertheless, the architectural differences between CPUs and GPUs mean that simply porting on-CPU algorithms to GPUs does not yield optimal performance. Unfortunately, to the best of our knowledge, the fundamental principle of SOTA on-GPU solutions still resembles traditional on-CPU, tree-based indexes [24, 40, 66]. Such architecture is ill-suited for the principle of GPU, therefore leading to the underutilization of the GPU's computational power.

1.4 Our Solution

We introduce BLAEQ (BLAS-based CAE Query), a novel approach optimized for executing spatial queries on GPUs. BLAEQ is designed to maximize GPU computational power through advantages in three key aspects: principle, structure, and implementation.

Principle. BLAEQ replaces conventional tree-based indexing with a **multigrid** architecture, ideal for parallel processing on GPUs. A multigrid consists of multiple layers of vectors connected via matrices, a technique widely used for solving linear equations [46, 60, 65]. Unlike CPU-based methods that use in-memory pointer navigation, BLAEQ converts index tree layers into vectors and their connections into matrices. This reformulation transforms tree navigation into sparse matrix-vector multiplication (SpMV) operations, which can be efficiently parallelized across GPU processors.

Structure. BLAEQ adapts index structures to better suit GPU architecture. Traditional CPU-optimized indexes use deep trees with few nodes per level, efficient for pruning [39] but insufficient for generating enough parallel tasks for GPUs. BLAEQ adopts a broad, shallow structure, ensuring each layer provides ample tasks for GPU processing. This design is theoretically justified in Section 3.5 and experimentally validated in Section 5.7.

Implementation. BLAEQ leverages BLAS (Basic Linear Algebra Subprograms) [23] to harness GPU computational power effectively. By utilizing pre-optimized BLAS libraries like CuBLAS [50], BLAEQ simplifies implementation and maximizes performance. This approach avoids the complexities of custom GPU programming while ensuring efficient task distribution and load balancing.

In summary, this paper makes the following contributions:

- We explore the novel application of spatial queries in computational simulation analytics.
- We propose BLAEQ, a multigrid-based spatial indexing method optimized for GPU execution.
- We demonstrate BLAEQ's effectiveness against state-of-the-art approaches through extensive experiments.
- To the best of our knowledge, BLAEQ is the first algorithm to utilize multigrid for spatial indexing, offering a design that could inspire future indexing research on emerging hardware.

1.5 Organization

The rest of the paper is organized as follows. In Section 2, we introduce the preliminaries of BLAEQ. The principle of BLAEQ is covered in Section 3. Additional content on implementation of BLAEQ is supplemented in Section 4. Experimental results are presented in Section 5, we introduce related work in Section 6 and conclude the paper in Section 7.

2 PRELIMINARIES

For the sake of clarity, the following conventions are used throughout the paper to distinguish between matrices, vectors, and scalars: Matrices are represented by bold, uppercase letters (e.g., **P**), vectors by bold, lowercase letters (e.g., $\boldsymbol{m}$), and scalars by regular lowercase letters (e.g., x).

2.1 Problem Formulation

Given a multi-dimensional grid $\mathbf{M}$ with D dimensions, a spatial query Q on $\mathbf{M}$ is defined as the intersection of sub-queries $q(d)$ on each dimension $d \in D$, i.e.

$$Q = \{r | r \in \cap_{d \in D} q(d)\},$$

Each sub-query $q(d)$ can be expressed as $q(d) = (q_d^-, q_d^+)$ where q_d^- and q_d^+ are the lower and upper bound of the query range in dimension d, respectively. If a sub-query on d-th dimension is not specified, we consider it a full-scale query $q_d = (-\infty, +\infty)$. The objective of BLAEQ is to efficiently process query Q on the multi-dimensional grid $\mathbf{M}$, ensuring optimal performance in the execution of queries.

2.2 BLAS Functions

Due to the extensive use of BLAS functions, math libraries [9, 65] have optimized these functions for performance using parallel computing and fine-grained tuning. This optimization is crucial for achieving high efficiency in numerical computations. Notable implementations include hardware-dependent packages for Intel processors [62], CUDA [50], FPGA [29], and supercomputers [64, 67].

The complete collection of BLAS [7] contains dozens of linear algebra functions. This section focuses on the subset of BLAS functions that are directly involved in the implementation of BLAEQ and the corresponding operators.

Matrix multiplication. Given two matrices $\mathbf{A}$ with shape $m \times k$ and $\mathbf{B}$ with shape $k \times n$ (a vector can be considered a matrix with only one row/column), matrix multiplication $\mathbf{A} \times \mathbf{B}$ produces $\mathbf{C}$ with shape $m \times n$, where m and n represent the number of rows and columns respectively. This operation is fundamental in linear algebra and forms the basis for many numerical computations.

Element-wise operations. Given two matrices $\mathbf{A}$ and $\mathbf{B}$, element-wise operation has four types: addition ($\mathbf{A} + \mathbf{B}$), subtraction ($\mathbf{A} - \mathbf{B}$), multiplication ($\mathbf{A} \odot \mathbf{B}$) and division ($\mathbf{A} \oslash \mathbf{B}$). These operations are performed element by element. For instance, element-wise multiplication $\mathbf{A} \odot \mathbf{B} = \mathbf{C}$ indicates $\mathbf{C}_{ij} = \mathbf{A}_{ij} \times \mathbf{B}_{ij}$. To perform such operations, $\mathbf{A}$ and $\mathbf{B}$ must be of identical shape. These operations are important in various numerical computations in BLAEQ, especially during queries.

Matrix-scalar operations. Let $\mathbf{M}$ be a matrix and s be a scalar, the matrix-scalar operation involves manipulating each element of the matrix with the scalar. The operators for matrix-scalar operations are identical to those for element-wise operations, e.g. $\mathbf{M} \oslash s = \mathbf{R}$, where $\mathbf{R}_{ij} = \mathbf{M}_{ij}/s$. These operations are relevant to BLAEQ as they allow for scaling and transforming matrices during prolongation.

Indicator function. Indicator function $where(C(v))$ evaluates a conditional expression C on each element of a vector v, returning an indicator vector r where $r_i = 1$ if $C(v_i)$ is true and $r_i = 0$ otherwise.

$$\mathbf{P} = \begin{bmatrix} \frac{x_0^0}{x_0^1} & \\ \frac{x_1^0}{x_0^1} & \\ & \frac{x_2^0}{x_1^1} \\ & \frac{x_3^0}{x_1^1} \\ & \frac{x_4^0}{x_1^1} \end{bmatrix}$$

(a) Original matrix

$$val = [\frac{x_0^0}{x_0^1}, \frac{x_1^0}{x_0^1}, \frac{x_2^0}{x_1^1}, \frac{x_3^0}{x_1^1}, \frac{x_4^0}{x_1^1}]$$

$$row = [0, 1, 2, 3, 4]$$

$$col = [0, 0, 1, 1, 1]$$

(b) COO format

$$val = [\frac{x_0^0}{x_0^1}, \frac{x_1^0}{x_0^1}, \frac{x_2^0}{x_1^1}, \frac{x_3^0}{x_1^1}, \frac{x_4^0}{x_1^1}]$$

$$row\_ptr = [0, 1, 2, 3, 4, 5]$$

$$col\_ptr = [0, 0, 1, 1, 1]$$

(c) CSR format

$$val = [\frac{x_0^0}{x_0^1}, \frac{x_1^0}{x_0^1}, \frac{x_2^0}{x_1^1}, \frac{x_3^0}{x_1^1}, \frac{x_4^0}{x_1^1}]$$

$$row\_ptr = [0, 1, 2, 3, 4]$$

$$col\_ptr = [0, 2, 5]$$

(d) CSC format

Figure 2: The COO, CSR, CSC storage formats of a matrix.

For instance, given a vector $v = [0, 1, 2]$, the indicator function $where(v > 1.5)$ would return $[0, 0, 1]$. This function is utilized for the pruning mechanism.

2.3 Sparse Matrix Formats

In BLAEQ, matrices and vectors are primarily sparse, which requires storing them using special formats. BLAEQ primarily employs three such formats: COO (Coordinate List), CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) [6]. Figure 2 illustrates the COO, CSR, and CSC formats for a matrix $\mathbf{P}$, which will be utilized in subsequent sections of the paper.

The COO format, as shown in subfigure (b), is based on a clear and straightforward principle. It utilizes three separate arrays to record the value and positional coordinates of each nnz (non-zero) element within a matrix. These arrays, denoted as `val`, `row` and `col` respectively, store the non-zero values and their corresponding row and column indices. The construction of this format is facilitated by its lack of strict sequential requirements for the nnzs, allowing for the appending of nnzs in any arbitrary order. However, the absence of data locality inherent in the COO format can result in suboptimal computational efficiency, an issue that we will demonstrate through the experimental results presented in Section 5.6.

The CSR and CSC formats, depicted in subfigures (c) and (d), are specifically designed for computational efficiency. In contrast to the COO format, these formats impose strict sequential constraints on the storage of non-zero elements. For instance, the CSR format arranges a sparse matrix in row-major order and consists of three arrays: `val`, `row_ptr` and `col_ptr`. The `val` and `col_ptr` arrays contain the values of the nnzs, and `col_ptr` holds their column indices.

The distinctive feature of the CSR format is the role of the `row_ptr` array. Each entry `row_ptr[i]` indicates the starting index in the `val` and `col_ptr` arrays for the nnzs of the i-th row. This structure is particularly conducive to sequential access of row elements. For example, to retrieve the values and column indices of non-zero elements in the i-th row, one need only extract the corresponding sub-arrays as follows:

```
values = val[row_ptr[i]:row_ptr[i+1]];
col_indices = col_ptr[row_ptr[i]:row_ptr[i+1]].
```

This efficient access pattern makes the CSR format ideal for operations that involve row-wise traversal of the sparse matrix.

Not all matrices benefit from row-major storage. Consider the **P** matrix, where the non-zero elements are more naturally organized in a column-major fashion. In such cases, the CSC format is more appropriate. The underlying principles of the CSC format are analogous to those of the CSR format, with a key distinction: the CSC format arranges non-zero elements in column-major order. Storing **P** matrix using the CSC format, as illustrated in Figure 2, exhibits a more compact representation compared to the CSR format.

3 BLAEQ

The principle of BLAEQ involves matrices and vectors of various dimensions and layers. For the sake of clarity, we summarize frequently used notations in Table 1.

Table 1: Frequently used notations

Notation	Description
$\mathbf{P}_d^{l+1\to l}$	Prolongation matrix from layer $l+1$ to l on dimension d
$\boldsymbol{m}_d^l$	The mesh of d-th dimension, l-th layer
b_d^l	The bandwidth of d-th dimension, l-th layer
q_d^-, q_d^+	The lower & upper bounds of a range query
N	The scale of the finest mesh, i.e. the original mesh
D	The dimensionality of the mesh
$\boldsymbol{a}$	An indicator vector
o	A specific point within a mesh
K	A configurable parameter

3.1 Intuition

The theoretical principle of BLAEQ is similar to a tree-based index, however, its design is tailored for on-GPU execution. Here, we explain the principle of BLAEQ with a straightforward example.

Figure 3 (a) illustrates three layers of meshes, with $\boldsymbol{m}_x^0$ being the finest mesh, which corresponds to the original data set. For clarity, we use subscripts to denote a specific dimension, e.g., x, and superscripts, e.g., 0, to denote the layer. An index built on $\boldsymbol{m}_x^0$ is a series of coarser meshes, which are denoted as $\boldsymbol{m}_x^1$ and $\boldsymbol{m}_x^2$, respectively. The data within each layer is structured as a vector, as shown in subfigure (b).

To effectively and efficiently accelerate a range query, it is essential to establish correspondences between mesh layers. In our example, the points x_0^0 and x_1^0 corresponds to x_0^1, while x_2^0 to x_4^0 corresponds to x_1^1. Consequently, a range query that excludes x_1^1 on layer $\boldsymbol{m}_x^1$ will also exclude the corresponding points on layer $\boldsymbol{m}_x^0$, thereby speeding up the query process. This correspondence mechanism is a cornerstone of spatial indexing.

Conventional indexes establish correspondences between points across layers using in-memory pointers. Such a mechanism is ill-suited for GPU, since a navigation process incurs data transfer between the GPU's processors and GPU's RAM (DRAM[2]). While similar data transfer processes occur on CPUs, the distinction lies in the GPU's larger number of processors. Each of these processors may initiate navigation requests, thereby amplifying the aggregate data transmission overhead.

BLAEQ addresses this issue by replacing pointer-based navigation with computational methods. Instead of following pointers to access data, BLAEQ uses multipliers to compute the desired values directly. For example, instead of using a pointer to access a value x_0^0 from a value x_0^1, BLAEQ uses a multiplier $\frac{x_0^0}{x_0^1}$ to compute x_0^0 directly from x_0^1. Generalizing the process above will produce a *prolongation matrix*, denoted as **P**. Specifically, consider the task of creating correspondences between $\boldsymbol{m}_x^0$ and $\boldsymbol{m}_x^1$. The goal is to create a matrix $\mathbf{P}_x^{1\to 0}$, such that $\boldsymbol{m}_x^0$ can be obtained by performing a SpMV as follows:

$$\mathbf{P}_x^{1\to 0} \times \boldsymbol{m}_x^1 = \boldsymbol{m}_x^0. \quad (1)$$

With $\boldsymbol{m}_x^0$ and $\boldsymbol{m}_x^1$ given, matrix $\mathbf{P}_x^{1\to 0}$ can be generated, as we illustrate in subfigure (c). Similarly, we can generate $\mathbf{P}_x^{2\to 1}$.

A prolongation matrix is a sparse matrix. Consider two adjacent layers $|\boldsymbol{m}_x^l|$ and $|\boldsymbol{m}_x^{l+1}|$ ($|\boldsymbol{m}_x^{l+1}| < |\boldsymbol{m}_x^l|$). The prolongation matrix $\mathbf{P}_x^{l+1\to l}$ will inherently be a matrix of shape $|\boldsymbol{m}_x^l| \times |\boldsymbol{m}_x^{l+1}|$, and it will have a sparsity pattern resulting in one nnz per row, giving it a sparsity ratio of $\frac{1}{|\boldsymbol{m}_x^{l+1}|}$.

The sparsity characteristic in a prolongation matrix is paramount, as it allows for effective pruning via SpMV. Consider subfigure (d) as an illustration. Imagine a scenario where a range query is conducted, and the pruning mechanism removes the element x_1^1 at layer $\boldsymbol{m}_x^1$. Logically, this should also remove the corresponding points of x_1^1 at the subsequent layer $\boldsymbol{m}_x^0$. To enable this pruning, BLAEQ sets the value of x_1^1 to 0, thereby creating the modified vector $\boldsymbol{m}_x^{1\prime}$. Consequently, when prolongation occurs, the operation $\mathbf{P}_x^{1\to 0} \times \boldsymbol{m}_x^{1\prime}$ generates a new vector $\boldsymbol{m}_x^{0\prime}$, where the entries corresponding to x_1^1 are also set to 0s.

In the realm of sparse linear algebra, the value '0' holds unique significance. To maximize computational efficiency, 0s within sparse matrices or vectors are neither stored nor processed. This practice significantly enhances the performance of SpMV. Consequently, when specific points are set to 0, BLAEQ inherently prunes these points, preventing their involvement in subsequent calculations. This pruning mechanism accelerates the overall query performance of BLAEQ.

3.2 Setup

With a clear understanding of BLAEQ's underlying principle, we now turn to its implementation. Without loss of generality, we specifically focus on dimension x. The algorithms can be generalized to other dimensions by treating each dimension individually.

3.2.1 Multigrid Construction. Similar to constructing a conventional tree-shaped index, the setup phase begins by generating

[2]https://docs.nvidia.com/deeplearning/performance

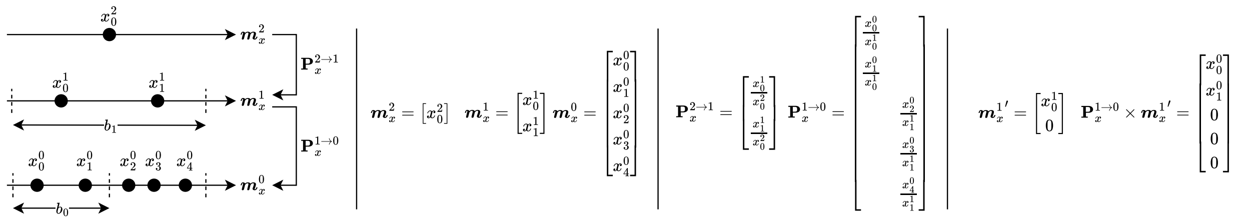


Figure 3: Demonstration of the principle of BLAEQ.

layers of vectors, where each point in a coarser layer corresponds to multiple points in the finer layer. Once the correspondence between points is established, a **P** matrix is constructed by populating the appropriate positions with the corresponding values.

For a mesh $\boldsymbol{m}_x^l$, BLAEQ partitions it into $|\boldsymbol{m}_x^l|/K$ disjoint, evenly sized bins with bandwidth:

$$b_x^l = \frac{\max(\boldsymbol{m}_x^l) - \min(\boldsymbol{m}_x^l)}{|\boldsymbol{m}_x^l|/K},$$

where K controls the multigrid hierarchy's structure (discussed in Section 3.5). The centroids of these bins form the coarser mesh $\boldsymbol{m}_x^{l+1}$, computed via $\boldsymbol{m}_x^{l+1} = Centroids(\boldsymbol{m}_x^l, b_x^l)$, as illustrated in Figure 3 for $K = 2$.

Each data point x_i^l maps to its coarser counterpart x_j^{l+1} through $j = \lfloor x_i^l / b_x^l \rfloor$, establishing correspondence $x_j^{l+1} \rightarrow x_i^l$. These relationships define the prolongation matrices where $\mathbf{P}_{ij}^{l+1\rightarrow l} = x_i^l / x_j^{l+1}$ for corresponding points.

The matrices are constructed by mapping all fine-mesh points to their coarse counterparts and populating **P** values accordingly. While a naive implementation uses for-loops, we optimize this via BLAS operations (Algorithm 1), returning CSC format for efficient SpMV.

Algorithm 1: BLAS-based Prolongation matrix generator

```
Input: Original mesh $M_d^l$, bandwidth $b_d^l$.
Output: Prolongation matrix $\mathbf{P}_d^{l+1\rightarrow l}$.
row = [0 to $|M_d^l| - 1$] //array [0, 1, 2,..., $|M_d^l| - 1$]
col = $\lfloor M_d^l \oslash b_d^l \rfloor$
val = $M_d^l \oslash ((col \odot b_d^l) + b_d^l/2)$
$P_d^{l+1\rightarrow l}$ = COO(row, col, val)
return $P_d^{l+1\rightarrow l}$.to_CSC
```

With the prolongation matrices constructed, the multigrid index is built. We assemble Algorithm 2 as the setup phase of BLAEQ. For multi-dimensional data, BLAEQ iterates through all its dimensions using a for loop.

3.3 Spatial Query

BLAEQ processes queries similarly to traditional tree-shaped indices, beginning at the coarsest layer $\boldsymbol{m}$. It prunes irrelevant elements, then performs a sparse matrix-vector multiplication (SpMV)

Algorithm 2: BLAEQ Setup

```
Input: Original mesh $\mathbf{M}^0$, Parameter $K$.
Output: Multigrid $G$.
// Phase 1: Initialization.
Prolongation matrices $\mathcal{P}$
Bandwidths $\mathcal{B}$
Coarsest Meshes $\mathcal{M}$
// Phase 2: Construction.
for $d \in D$ do
    Current mesh $\boldsymbol{m}_d^0 = \mathbf{M}^0[d]$
    $\boldsymbol{m}_d^l \coloneqq \boldsymbol{m}_d^0$
    $\mathcal{M}_d, \mathcal{P}_d, \mathcal{B}_d = \emptyset$
    while $|\boldsymbol{m}_d^l| > K$ do
        $b_d^l = \frac{max(\boldsymbol{m}_d^l) - min(\boldsymbol{m}_d^l)}{|\boldsymbol{m}_d^l|/K}$
        $\boldsymbol{m}_d^{l+1} = Centroids(\boldsymbol{m}_d^l, b_d^l)$
        $\mathbf{P}_d^{l+1\rightarrow l} = Prolongation\_Matrix(\boldsymbol{m}_d^l, b_d^l)$
        $\mathcal{B}_d.append(b_d^l)$
        $\mathcal{P}_d.append(\mathbf{P}_d^{l+1\rightarrow l})$
        $\boldsymbol{m}_d^l = \boldsymbol{m}_d^{l+1}$
    $\mathcal{M}_d.append(\boldsymbol{m}_d^l)$
$G = \mathcal{P} \cup \mathcal{B} \cup \mathcal{M}$
return $G$
```

to generate the finer layer via $\boldsymbol{m}' = \mathbf{P} \times \boldsymbol{m}$. This process iterates until reaching the finest mesh.

Figure 4 demonstrates the principle of performing a range query using BLAEQ. The query starts from the coarsest layer ($\boldsymbol{m}^3$) and iteratively prolongs the query results to the lower layers until reaching the original layer ($\boldsymbol{m}^0$).

3.3.1 Relaxation. When executing a query on the coarser meshes ($\boldsymbol{m}^1$ to $\boldsymbol{m}^3$), a special mechanism, *relaxation*, is required. Relaxation represents extending the query range on coarser meshes, which is crucial because a data point on a coarser mesh corresponds to a range on the finer meshes. Therefore, to ensure accuracy, the query range on the coarser meshes must be adjusted.

In BLAEQ, the extent of relaxation for each layer is determined during setup. Specifically, consider a lower-layer point o_x^l, its bandwidth b_x^l, and its corresponding point o_x^{l+1} on the coarser layer,

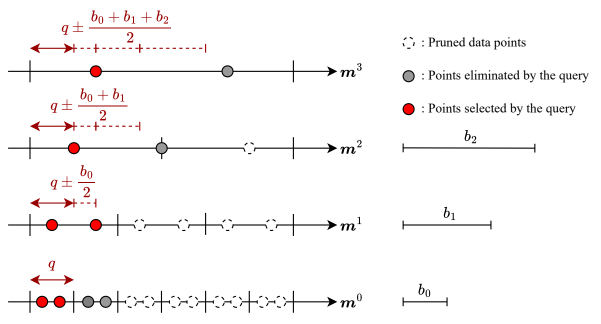

$q \pm \frac{b_0+b_1+b_2}{2}$
m^3
$q \pm \frac{b_0+b_1}{2}$
m^2
$q \pm \frac{b_0}{2}$
m^1
q
m^0
: Pruned data points
: Points eliminated by the query
: Points selected by the query
b_2
b_1
b_0

Figure 4: A demonstration of executing a range query using the multigrid.

which is the centroid of the bin that contains o_x^l. This relationship is formalized by the inequality

$$|o_x^{l+1} - o_x^l| \leq b_x^i/2. \tag{2}$$

This equation sets the upper bound for the distance between o_x^l and o_x^{l+1},

To ensure that the relaxation mechanism accurately captures the relevant data points from the finer meshes represented by the coarser meshes. We extend the query range by $b_x^i/2$ at coarser layer. Specifically, given a range query q_x, if $o_x^l \in (q_x^-, q_x^+)$, combining equation (2), we have

$$o_x^{l+1} \in (q_x^- - b_x^i/2, q_x^+ + b_x^i/2). \tag{3}$$

Since a range query q is supposed to be executed on M_d^0, on the $(l+1)$-th layer, to ensure the accuracy of relaxation, equation (3) should be applied to all the intermediate layers. In this case, the relaxation range on the $(l+1)$-th layer should be $\sum_{i=0}^{l} \frac{b_i}{2}$ (Figure 4). Therefore, the query range with relaxation on the $(l+1)$-th layer should be

$$o_x^{l+1} \in (q_x^- - \sum_{i=0}^{l} \frac{b_i}{2}, q_x^+ + \sum_{i=0}^{l} \frac{b_i}{2}). \tag{4}$$

For the sake of simplicity, we denote equation (4) as

$$o_x^{l+1} \in q_x \pm \sum_{i=0}^{l} \frac{b_i}{2}. \tag{5}$$

3.3.2 Query via SpMV. Given a coarser grid $\boldsymbol{m}_x^{l+1}$, a prolongation matrix $\mathbf{P}_x^{l+1\to l}$, a range query q_x is executed as follows:

First, we execute q_x with relaxation on $\boldsymbol{m}_x^{l+1}$ using equation (5), which returns an indicator vector $\boldsymbol{a}_x^{l+1}$, indicating which values within $\boldsymbol{m}_x^{l+1}$ falls within q:

$$\boldsymbol{a}_x^{l+1} = where(\boldsymbol{m}_x^{l+1} \in q_x \pm \sum_{i=0}^{l} \frac{b_i}{2}).$$

Then, we perform element-wise multiplication $\boldsymbol{m}_x^{l+1'} = \boldsymbol{m}_x^{l+1} \odot \boldsymbol{a}_x^{l+1}$, which preserves all data points within the relaxed query range and eliminates the rest by setting them to 0s. This process turns $\boldsymbol{m}_x^{l+1'}$ into a sparse vector that can be stored using CSC format.

Next, we perform SpMV

$$\boldsymbol{m}_x^{l\,'} = \mathbf{P}_x^{l+1\to l} \times \boldsymbol{m}_x^{l+1'}, \tag{6}$$

and generates the pruned, lower-layer mesh $\boldsymbol{m}_x^{l\,'}$. We repeat the previous steps until reaching $\boldsymbol{m}_x^{0\,'}$. Finally, where we perform a detailed scan to get the query result.

The process above is applicable to a single dimension. To execute a query with predicates on multiple dimensions, we perform

$$\boldsymbol{r} = \prod_{d\in D} \boldsymbol{r}_d^0,$$

where $\boldsymbol{r}_d^0$ represents the query result on dimension d, and $\boldsymbol{r}$ represents the final query result.

We summarize the query process of BLAEQ in Algorithm 3.

Algorithm 3: Multigrid Query

```
Input: Coarsest grids $\mathcal{M}$, bandwidths $\mathcal{B}$, prolongation
       matrices $\mathcal{P}$, range query $Q$.
Output: Query result $R$.
$I^0 = I$ //An identity vector
for $d \in D$ do
    $q = Q_d$
    Coarsest grid $\boldsymbol{m}_d^l = \mathcal{M}[d]$
    for $i \in [\mathcal{P}_d.size - 1, ..., 0]$ do
        //Iterates $i$ in descending order
        $\boldsymbol{a}_d^l = where(\boldsymbol{m}_d^l \in q_d \pm \sum_{i=0}^{l-1} \mathcal{B}_d[i]/2)$
        $\boldsymbol{m}_d^{l\,'} = \boldsymbol{m}_d^l \odot \boldsymbol{a}_d^l$
        $\mathbf{P}_d^{l\to l-1} = \mathcal{P}_d[i]$
        $\boldsymbol{m}_d^{l-1'} = \mathbf{P}_d^{l\to l-1} \times \boldsymbol{m}_d^{l\,'}$
        $\boldsymbol{m}_d^l = \boldsymbol{m}_d^{l-1'}$
    $\boldsymbol{a}^0 = \boldsymbol{a}^0 \odot \boldsymbol{a}_d^l$
$R = \bigcup_{d\in D} \boldsymbol{m}_d^l \odot \boldsymbol{a}^0$
return $R$
```

3.4 Complexity Analysis

3.4.1 Spatial Complexity. The data structure of BLAEQ is composed of three parts: the coarsest grids $\mathcal{M}$, prolongation matrices $\mathcal{P}$ and bandwidths $\mathcal{B}$. Notice that the intermediate levels $\boldsymbol{m}^i$ does not require storage since they can be computed via prolongation.

Due to the principle of BLAEQ, the scale of the coarsest grid is no larger than K. Therefore, the spatial complexity of $\mathcal{M}$ will be $O(K)$. Let n be the scale of the original grid. BLAEQ will have $log_K n$ layers of multigrid. Therefore, the spatial complexity of $\mathcal{B}$ will be $O(log_K n)$. If we store the prolongation matrices using sparse matrix formats, then the spatial complexity of $\mathcal{P}$ will be $O(n + \frac{n}{K} + \frac{n}{K^2} + ...)$. For large-scale CFD meshes, we generally have $n \gg K \gg log_K n$. As a result, the overall spatial complexity of BLAEQ is

$$O(K + log_K n + n + \frac{n}{K} + \frac{n}{K^2} ...) \approx O(n).$$

3.4.2 *Time Complexity.*

The time complexity of the setup process is primarily influenced by the computation of prolongation matrices. These matrices are constructed through element-wise operations, including multiplication and division. For a mesh with an original number of points n, the time complexity of these element-wise manipulations is $O(n)$. As a result, the overall time complexity of the setup process can be expressed as:

$$O(n + \frac{n}{K} + \frac{n}{K^2} + \ldots) \approx O(n).$$

Nevertheless, for a multigrid architecture with a large K , the terms with higher powers of K, e.g. $\frac{n}{K}, \frac{n}{K^2}\ldots$, become increasingly negligible. Therefore, the overall time complexity of the setup process is approximately $O(n)$, as the dominating term is $O(n)$.

The time complexity of query via BLAEQ is dominated by SpMVs. The time complexity of regular matrix-vector multiplication with a matrix of size $n \times n$ is $O(n^2)$. For SpMV, with the sparsity of matrix be $\frac{1}{n}$, the time complexity will be reduced to $O(n)$. Suppose the proportion between the requested points and the original grid is $p \in [0, 1]$. Theoretically, the time complexity of such a query on a mesh with a scale of n is

$$O(p \cdot n + \frac{p \cdot n}{K} + \frac{p \cdot n}{K^2} + \ldots) \approx O(p \cdot n).$$

3.5 Choosing K

The theoretical time complexity discussed above does not fully account for BLAEQ's performance under parallel execution. When using BLAS for parallel execution, the efficiency of sparse matrix-vector multiplication (SpMV) is heavily influenced by the shape of the prolongation matrix, emphasizing the importance of selecting parameter K appropriately.

A matrix multiplication $\boldsymbol{r} = \mathbf{P} \times \boldsymbol{m}$ (superscripts omitted) can be viewed as computing the linear combination of the **columns** of $\mathbf{P}$. Let $\mathbf{P} = [\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_n]$, where $\boldsymbol{c}_i$ is a column vector, and $\boldsymbol{m} = [m_1, m_2, \ldots, m_n]$, where m_i is a scalar. The result $\boldsymbol{r}$ is computed as:

$$\boldsymbol{r} = \boldsymbol{c}_1 \otimes m_1 + \boldsymbol{c}_2 \otimes m_2 + \cdots + \boldsymbol{c}_n \otimes m_n. \tag{7}$$

This formulation allows matrix-vector multiplication to be efficiently parallelized by performing scalar multiplication on each column concurrently and then accumulating the results, as summarized in Algorithm 4. This approach leverages the inherent parallelism of the operation, particularly beneficial for BLAEQ, where prolongation matrices are stored in column-major format.

Algorithm 4: Parallel CSC SpMV

```
Input: Matrix $\mathbf{P}$, vector $M$.
Output: Result vector $r$.
for $c_i \in \mathbf{P}, m_i \in M$ do
    $r_i = c_i \otimes m_i$ // In parallel
$r = \sum r_i$
return $r$
```

The time complexity of Algorithm 4 can be analyzed as follows. Let C be the number of columns in $\mathbf{P}$. The complexity of distributing columns among parallel processors is $O(C)$. The scalar multiplication step depends on the column size $|c_i|$, and the linear combination step also has complexity $O(C)$. Thus, the overall complexity is $O(2C + |c_i|)$, denoted as $f = 2C + |c_i|$. The goal is to minimize f.

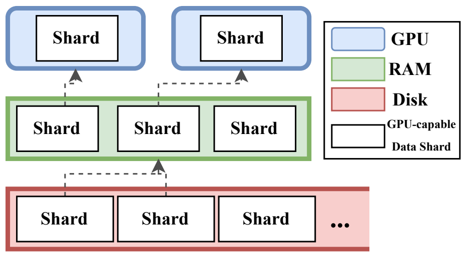


Figure 5: The principle of implementing BLAEQ in a hybrid architecture.

Assuming $\mathbf{P}^{1\to 0}$ has N non-zero elements, the column size can be estimated as $|c_i| = N/C$. Substituting this into f yields:

$$f = 2C + \frac{N}{C}.$$

Minimizing f leads to the conclusion[3]

$$K = \sqrt{2 \cdot N}. \tag{8}$$

Equation 8 reveals a structural feature of the multigrid architecture. Setting $K = \sqrt{2 \cdot N}$ limits the multigrid's height to three layers, as two coarser layers suffice to represent $K^2 = 2 \cdot N$ points, exceeding the original N points. This wide multigrid structure, unlike traditional deep tree indices, provides sufficient parallel tasks to fully utilize GPU computational power. Experimental validation is provided in Section 5.7.

4 IMPLMENTATION

In previous sections, we introduced the theoretical principles of BLAEQ. However, applying BLAEQ to real-life datasets requires overcoming implementation challenges, particularly in processing large-scale data that often exceed GPU capabilities. To address this, we developed variants of BLAEQ tailored for datasets of varying scales (see figure 5), all based on Algorithms 2 and 3, with differences primarily in data storage and allocation:

- **On DRAM:** Designed for smaller datasets, this variant preloads all data onto the GPU, enabling fully on-GPU execution.
- **On RAM:** Data are preloaded into RAM and transferred to the GPU as needed, requiring additional data transfer mechanisms.
- **On Disk:** Data are stored on disk and loaded into RAM and GPU incrementally. Using Grid Files [51], we partition datasets into blocks and prune data at the file level using metadata, ensuring tasks fit within GPU memory. This variant supports both **single-GPU** and **multi-GPU** execution, simulating real-world database environments.

[3]The theoretical deduction is omitted due to restriction of page length.

A key data structure in these implementations is the **shard**, which contains a complete set of multigrids with **P** matrices and coarsest-layer meshes $\boldsymbol{m}$, sized to fit a single GPU. Typically, $|S| < |DRAM|/2$, where $|S|$ is the shard size. Each shard includes metadata for data range identification, enabling shard-level pruning before GPU loading. While our implementation is intuitive, further performance tuning remains a future work, as the primary focus of this paper is introducing multigrid for geometry data indexing.

5 EXPERIMENTS

5.1 Experimental Setup

Environment. We conduct the experiments on an Aliyun cloud server equipped with a multi-GPU configuration. The server specifications include an Intel(R) Xeon(R) Platinum 8163 CPU (2.50 GHz) and 2*NVIDIA Tesla T4 GPUs.

Datasets. We employ data sets from three origins:

(1) Real-world CFD and FEA (Finite Element Analysis) datasets collected from CSSRC (China Ship Scientific Research Center). Experiments on these data sets represent the application potential of BLAEQ on computational simulation data management. These data sets range from 10^3 to 10^8, providing a range of experimental conditions on real-life engineering data sets.
(2) Point cloud Datasets [14, 56]. Point cloud data is a vital component of fields such as geospatial mapping[10], autonomous vehicles[16], and gaming engines[25], etc.
(3) Synthetic Datasets. The previous two scenarios are limited to geometrical data sets, with primarily 3-D data sets. We use synthetic experiments to evaluate the performance of BLAEQ on data sets of various dimensionalities and scales.

Table 2: The summary of data sets for overall evaluation. In this experiment, all datasets are three dimensional.

Use Cases	CFD		FEA		Point Cloud	
	Name	Scale	Name	Scale	Name	Scale
Data sets	FEA-1	8×10^3	CFD-1	3×10^5	CFD-4	1×10^8
	FEA-2	1×10^4	CFD-2	3×10^6	PCL-1	1×10^9
	FEA-3	5×10^4	CFD-3	2×10^7	PCL-2	5×10^9

In the context of geometrical spatial indexing, algorithms are commonly evaluated based on their performance in construction, query, and update. We specifically focus on the evaluation of the query. Since in many geometry data applications, such as CFD post-processing[18], smart manufacturing[47, 68], etc., the creation and update of the 3-D model is an offline process.

Workload. Range queries have a wide range of applications, in this paper, we enumerate three representative types of range query as our experimental workload:

- **Point-in-Range Query.** A Point-in-Range Query is used to determine whether a specific point lies within a defined spatial or numerical range. This type of query is particularly useful in applications such as scientific simulations, where verifying the presence of a data point within a specified range is critical.
- **Cross-Section Query.** A Cross-Section Query is designed to extract a two-dimensional slice or cross-section from a three-dimensional model. This is especially valuable in fields like CFD analysis and computer graphics, where analyzing internal structures or specific layers of a 3D model is necessary.
- **Region-of-Interest Query.** A Region-of-Interest (ROI) Query focuses on retrieving a specific subset of data from a larger dataset, typically by isolating a particular region or segment of a model. This is particularly beneficial in scenarios where processing the entire dataset is computationally expensive or unnecessary.

Algorithms. The experiment comprises eight competitors, representing the SOTA solutions for multi-dimensional data indexing problems. Given the critical role of implementation in performance, all the algorithms are evaluated using their publicly available code or implementations provided by trusted sources.

- **KD-Tree**[4]**.** KD-Tree is a binary space-partitioning structure, recursively dividing space along alternating axes, making it efficient for querying low-dimensional spaces.
- **R-Tree**[5] R-tree, designed for spatial databases, organizes objects using minimum bounding rectangles in a hierarchical manner, balancing query performance and storage overhead.
- **Octree**[6]**.** Octree partition space into eight equal regions, enabling efficient handling of low-dimensional, volumetric data.
- **LISA**[7]**.** LISA [43] is a learned index. Instead of indexing using a tree, it constructs a machine-learning model and predicts the query results. LISA represents the SOTA CPU-based solutions for spatial query.
- **LBVH-Tree**[8]**.** LBVH-Tree [36] is a GPU-accelerated, SOTA solution for point cloud indexing, a domain closely related to 3-D mesh query. It builds a balanced tree structure optimized for parallel execution on GPUs.
- **G-PICS**[9]**.** G-PICS [40] is a representative GPU-based solution. Since a tree-based search is difficult to parallelize for GPU, it reduces single-query consumption by executing a batch of queries simultaneously.
- **GTS**[10]**.** GTS [69] executes very similar to G-PICS. It refines G-PICS on pruning and memory management for improved parallel query, achieving better batch query performance.
- **TileDB**[11]**.**TileDB[53] is an array database designed for efficient multi-dimensional data management. While BLAEQ and other competitors are essential components of array databases, we implement a specialized variant of BLAEQ

[4]https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html
[5]https://pypi.org/project/Rtree/.
[6]https://github.com/PointCloudLibrary/pcl
[7]https://github.com/pfl-cs/LISA
[8]https://pypi.org/project/cupy-knn/
[9]The original code of G-PICS is inaccessible. Zhu et al. implemented G-PICS as a baseline: https://github.com/ZJU-DAILY/GTS/tree/master/Source%20Code/GPU-Tree
[10]https://github.com/ZJU-DAILY/GTS
[11]https://tiledb.com/

that retrieves data from disk. This experiment highlights BLAEQ's potential for integration into database systems.

We also implement multiple variants of BLAEQ for different experimental setups, including three GPU-dependent variants of BLAEQ: BLAEQ-DRAM, BLAEQ-RAM, and BLAEQ-Disk as we introduced in Section 4. We also introduce BLAEQ-CPU by replacing the BLAS library using CPU-dependent MKL [62] for a fully on-GPU implementation.

Metrics. To evaluate the details of algorithms' execution footprint, except for regular metrics such as memory consumption and time, we introduce NVIDIA Nsight Compute[12] to provide a detailed analysis for on-GPU algorithms. Specifically, we introduce the following metrics:

- **Occupancy.** Occupancy reflects the overall GPU occupancy during executing the algorithm on GPU. A low occupancy indicates an underutilization of the GPU.
- **Estimated Speedup.** Estimated Speedup (ES) is a comprehensive, function-level factor indicating how much a function can potentially be accelerated on GPU. A large ES indicates that the function *underutilizes* the GPU.
- **Duration**. Duration evaluates the function's on-GPU execution time.
- **Memory Throughput**. Memory throughput evaluates the function-level data I/O.

5.2 On RAM & DRAM Evaluation

Since many competitors do not support reading data from disk, to ensure fair competition, in this section, we compare the indexing algorithms, including Octree, LISA, GTS, etc., with BLAEQ-CPU, BLAEQ-DRAM, and BLAEQ-RAM. Meanwhile, we compare BLAEQ-Disk against ArrayDB in the next section.

In this section, we evaluate the overall query performance of the variations of BLAEQ and other competitors. We do not add RAM or DRAM constraints in the experiment, all algorithms can access all the available resources from the platform. The experimental results are demonstrated in Figure 6. We distinguish on-CPU and on-GPU algorithms using different line styles: The on-CPU algorithms are labeled using dotted lines, and on-GPU algorithms are labeled using dash-dotted lines. The coordinates of data sets are placed based on their scale (see Table 2).

Peak memory consumption (subfigure a)) measures the storage requirements for indices. Since memory usage remains stable across query types, we use the average peak memory consumption as the metric. GPU-based algorithms generally consume more memory than CPU-based ones due to inherent programming differences, where GPUs trade memory for parallel processing efficiency. For instance, BLAEQ-DRAM allocates redundant memory for parallel SpMV operations, while BLAEQ-CPU uses sequential scans for precise memory usage. This trade-off is common in high-performance computing [2, 17, 41].

As we examine the experimental outcome patterns, a significant contrast emerges between algorithms designed for CPU and those for GPU. In particular, for CPU-based algorithms such as KD-Tree and BLAEQ-CPU, there is a marked increase in computational time as the dataset size expands. Conversely, for GPU-based algorithms, including GTS and BLAEQ-DRAM, the increase in computational time is notably gentler. This discrepancy is rooted in the intrinsic difference in computational capabilities between CPUs and GPUs. GPUs possess greater computational power but at the expense of additional data transfer overhead between RAM and GPU. Consequently, the growth curves for GPU-based algorithms exhibit a smoother trend. However, due to this data transfer overhead, they are outperformed by CPU-based algorithms on smaller datasets.

An interesting exception among CPU-based algorithms is LISA, which exhibits a remarkably steady growth rate, divergent from other CPU-based methods. This is because LISA, as a learned index, operates on a principle distinct from conventional indexing. It leverages machine learning models to predict query outcomes rather than employing a layer-by-layer pruning approach. This mechanism enables LISA to handle large datasets effectively, albeit with reduced performance on smaller datasets.

When comparing algorithm performance across query patterns, CPU-based algorithms exhibit greater sensitivity to query result size, while GPU-based algorithms remain less affected. CPU algorithms rely heavily on pruning mechanisms, excelling in point queries with minimal result sizes but struggling with hyperplane queries requiring full scans. In contrast, GPU algorithms, designed with broader, shallower structures (Section 3.5), minimize reliance on pruning, ensuring stable performance across varying query result sizes.

In summary, for datasets exceeding $N > 10^6$, the on-GPU variants of BLAEQ exhibit superior query performance compared to the other competitors. This observation underscores the benefits of employing GPUs for query processing on large-scale datasets and the effectiveness of BLAEQ-DRAM in such contexts.

5.3 Comparison with ArrayDB.

In this experiment, we compare BLAEQ-Disk against TileDB, a representative array database. The purpose of this experiment is to evaluate the potential of applying BLAEQ to a database system. Notice that the multi-GPU capability of BLAEQ is also included in this experiment. The experimental results are demonstrated in Figure 7.

The experimental results show that BLAEQ outperforms TileDB for datasets of moderate size (e.g., $N \leq 10^8$), leveraging the superior computational power of GPUs despite slight I/O overhead. However, as the dataset size increases, the performance gap narrows, with TileDB slightly surpassing BLAEQ for larger datasets ($N \geq 10^9$).

This outcome stems from the fundamental differences between TileDB and BLAEQ On-Disk. TileDB is a highly optimized database system designed for managing extremely large-scale data. It performs fine-grained file-level pruning before loading data into RAM, minimizing I/O overhead. In contrast, BLAEQ On-Disk adopts a more intuitive approach, where data is split into shards primarily to avoid GPU out-of-memory (OOM) issues rather than optimizing file-level pruning. As a result, BLAEQ On-Disk incurs significant I/O overhead as dataset size grows, leading to performance degradation.

Additionally, we observe that using multiple GPUs does not compensate for the increased I/O overhead. Furthermore, for smaller

[12]https://developer.nvidia.com/nsight-compute

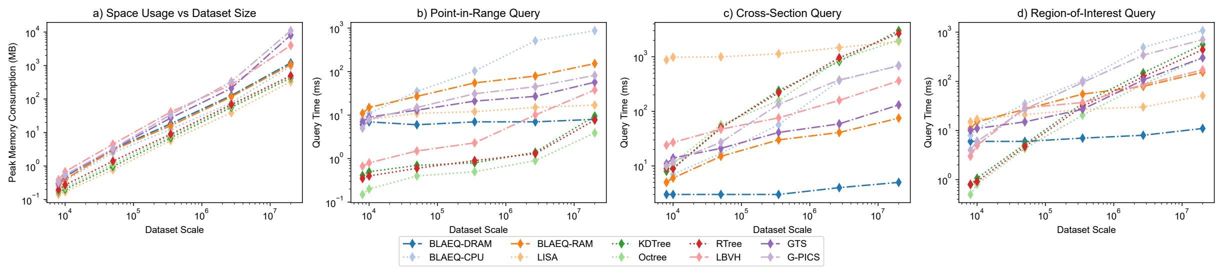


Figure 6: The query performance of different algorithms on the six smaller data sets.

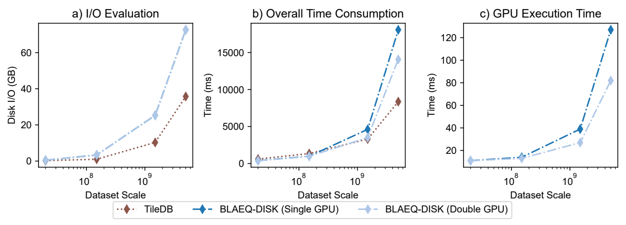


Figure 7: Comparison between BLAEQ-Disk with ArrayDB.

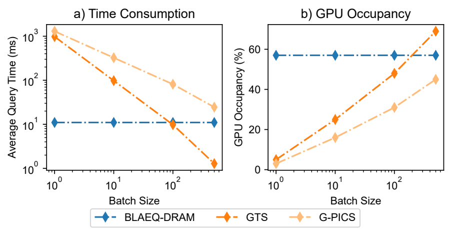


Figure 8: The multi-query performance between G-PICS, GTS, and BLAEQ-DRAM.

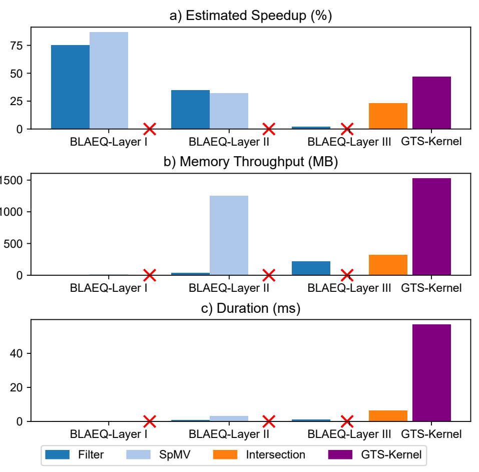


Figure 9: The function-level footprint analysis of BLAEQ, with GTS as baseline.

datasets that fit within a single GPU, the current shard design prevents BLAEQ from effectively utilizing a multi-GPU setup. These findings highlight the need for better tuning of BLAEQ's On-Disk implementation and more advanced techniques to leverage multiple GPUs efficiently.

5.4 Multi-query Evaluation

As G-PICS and GTS are specifically designed for batch query processing on GPU. This section assesses the capability of processing multiple queries against BLAEQ.

We present the evaluation of range queries on data set CFD-3 to ensure on-GPU execution with a large enough dataset. We evaluate both the average query time and GPU occupancy as metrics. The experimental results are visualized in Figure 8. From the results, we have the following observations.

The experimental results demonstrate both the advantages and disadvantages of BLAEQ against SOTA on-GPU algorithms. BLAEQ can already utilize GPU at a high occupancy when provided with a limited number of queries, leading to significant performance gains compared to the other algorithms. However, as more queries are executed concurrently, batch query methods prove to be efficient. For example, G-PICS exhibits a steady decline in average query time, overtaking BLAEQ-DRAM at a batch size of 10^3. GTS, an optimized variant of G-PICS, shows a near-linear reduction in average query time with respect to batch size, overtaking BLAEQ at batch size of 10^2, underscoring the efficacy of batch processing.

In conclusion, both the experiments demonstrate the high GPU utility of BLAEQ when processing single queries, and the potential of integrating batch query capability, which should be considered in the future work.

5.5 Detailed Evaluation

In this experiment, we conduct a function-level evaluation of BLAEQ, using the CUDA kernel performance of GTS as a baseline to compare against SOTA solutions. The experiments are performed on the CFD-3 dataset, with results shown in Figure 5.5. Based on BLAEQ's design, we decompose its execution into three key phases: **Filtering (F)**, **SpMV (M)**, and **Intersection (I)**. Each phase corresponds to one or more CUDA kernel functions, enabling us to pinpoint performance bottlenecks and underutilization. Since BLAEQ's multigrid structure consists of at most three layers, we further break down the execution footprint by layer to provide a clearer understanding of the algorithm's behavior.

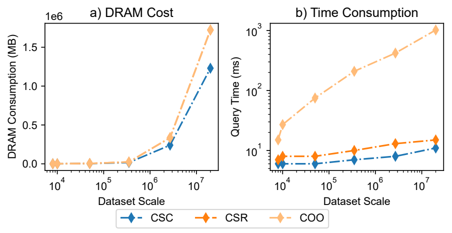


Figure 10: The query performance of BLAEQ with different prolongation matrix formats.

According to the ES metric, BLAEQ exhibits high ES values at the coarser layers and during the **I** phase. The high ES at coarser layers is expected, as the vectors and matrices at these layers are small, leading to unavoidable GPU underutilization. However, as shown in the *Duration* subfigure, the duration of these phases is negligible despite the high ES scores.

The experimental results highlight a significant bottleneck in BLAEQ: the **I** (Intersection) phase. This phase requires sorted input data for parallel intersections, but data in BLAEQ are sorted differently across dimensions. This necessitates additional sorting operations, which are inefficient on GPUs and result in substantial performance overhead. As seen in subfigure (c), the **I** phase consumes the majority of the execution time.

The effectiveness of SpMV is demonstrated during the SpMV phase at Layer II, where a prolongation operation is performed from the second layer to the finest layer. Subfigure (b) shows that the SpMV phase involves loading a large prolongation matrix, yet the operation is completed extremely quickly, as evidenced by subfigure (c). This underscores the efficiency of leveraging BLAS in BLAEQ for SpMV operations.

5.6 Matrix-Vector Multiplication

In Section 2.3, we highlight the importance of choosing the appropriate storage format for the prolongation matrices, as it significantly affects the efficiency of SpMV, which is a critical component of BLAEQ's query processing. This section demonstrates the experimental results by storing the prolongation matrices using the three most widely used formats: COO, CSR and CSC.

The experimental results are visualized in Figure 10. The time consumption gap between COO-SpMV and CSR/CSC-SpMV is substantial (by a factor of 10^1 on smaller data sets and 10^2 on kvlcc2_large), both in time consumption and spatial efficiency.

However, if we further consider the storage efficiency, we find that CSC is also more spatially effective than COO and CSR, where the CSR and COO consume almost identical space, significantly larger than CSC. Such a result is understandable, given that the **P** matrices are all column-dominant. Such a phenomenon can be observed in the exemplar matrices in Figure 2. The experimental results indicate that CSC is indeed the optimal storage format for **P** matrices of BLAEQ. This experiment highlights the importance of understanding the underlying principles of on-GPU implementation when utilizing a BLAS package to avoid suboptimal performance.

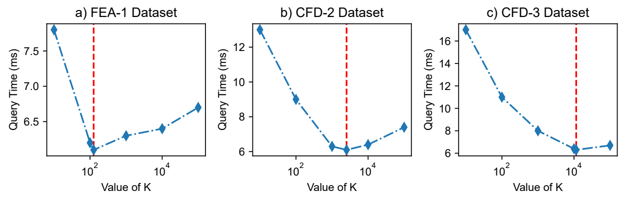


Figure 11: BLAEQ's query performance of choosing different values of K on different data sets.

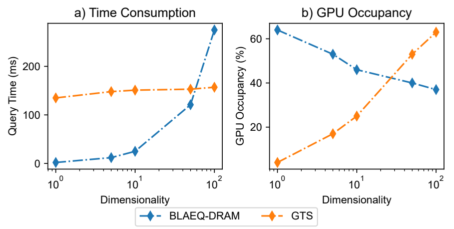


Figure 12: The performance of BLAEQ vs. GTS under various dimensions.

5.7 Parameter Selection

The choice of K significantly influences BLAEQ's performance. This section evaluates query performance across different K values, with results illustrated in Figure 11. The red dashed line represents the theoretical recommendation for K derived from Equation (8). Experimental results confirm that this recommendation identifies the optimal K across all datasets, validating the effectiveness of our approach. However, performance variations can arise due to environmental factors, such as background processes or hardware fluctuations. For instance, on the largest data set, query times vary by less than 3 ms for K values between 10^3 and 10^5, highlighting the robustness of our method in practical settings.

A sharp decline in query time occurs as K approaches the recommended value, driven by the multigrid's layer count. Small K values result in deep multigrids with numerous layers, increasing the number of SpMV operations (time complexity $O(N)$). This confirms that SpMV, rather than pruning, is the primary performance bottleneck. Conversely, performance degradation for large K values is gradual. Once K exceeds $\sqrt{2 \cdot N}$, the multigrid's layer count remains constant, but the middle layer's size increases, raising scan costs to $O(N/K)$. However, GPU parallelization mitigates this impact, leading to only minor performance degradation, as shown in Figure 11. These findings align with the theoretical model in Section 3.5 and the discussion on GPU-friendly wide multigrid structures in Section 1.

5.8 Scalability to Dimensionality

The previous experiments depend on 3-D data sets. In this part of the experiment, we evaluate the performance of BLAEQ-DRAM against GTS on a synthetic data set with configurable dimensionalities, the scale of the data set is always set to $N = 10^6$ to avoid OOM issues.

The experimental results are demonstrated in Figure 12. From the results, we observe that the performance of BLAEQ degrades significantly as the dimensionality of the dataset increases. This degradation is primarily due to the need to process each dimension individually, which introduces additional computational overhead. Furthermore, the high time consumption of the **I** (Intersection) phase, as analyzed in Section 5.5, exacerbates this issue, making BLAEQ less efficient for high-dimensional data.

In contrast, GTS is less affected by dimensionality and even benefits from higher-dimensional datasets. This is because higher dimensionality increases the number of computational tasks at each node, leading to better GPU utilization and improved performance.

These observations align with the underlying principles of BLAEQ, where the need to query each dimension separately and the inefficiencies in the **I** phase limit its scalability for high-dimensional datasets. However, BLAEQ's superior performance on low-dimensional datasets demonstrates that it remains highly efficient and effective for indexing and querying low-dimensional geometry data.

6 RELATED WORK

Spatial query has been widely studied in the past. Traditional spatial query solutions are primarily based on trees. These solutions include B-tree [5], R-tree [11], KD-tree [39], Octree [63], etc. Among these, the R-tree and Octree are often considered the most effective solutions for indexing geometry data. Consequently, many variants have been proposed for further improvement [1, 4, 48]. Balasubramanian et al. provide a detailed survey on these variants [4]. Nevertheless, these solutions are mainly dependent on sequential execution on a single CPU, thus limiting their overall performance, especially on large-scale data sets.

Other closely related works include solutions beyond indexing, such as Grid File [51] and array databases [32]. Grid Files are file-level solutions for storing and managing multi-dimensional data, enabling efficient pruning at the storage level. Array databases are proposed as specialized solutions for multi-dimensional data management, as the use case of this type of data raises new challenges that traditional database approaches find hard to conquer [12]. The representative array databases include SciDB [58], TileDB [53], etc.

Parallel computing is a promising direction for further improving the query efficiency of spatial queries. The MapReduce framework [22], a widely adopted parallel computing paradigm at the time, was a natural choice for this endeavor. Consequently, many distributed, parallel algorithms were developed, with Spatial Hadoop [27] emerging as a notable example in this domain. Singh et al. [57] provide a thorough survey on these solutions. Nevertheless, as a distributed environment, MapReduce has its natural drawbacks. It incurs heavy inter-node data transmission overhead [26] and data balancing issues [44], which can ultimately impact the overall performance of spatial query processing.

With the development of machine learning, learned index [38] has become a popular solution. These indices depend on techniques such as regression [43], neural networks [21], and reinforcement learning[33]. Compared with a tree-based structure, a learned index replaces the tree with a machine-learning model. The advantage of such architecture is that the scale of the data set will no longer dominate the scale of the index. As a result, a learned index shows great potential for indexing large-scale data. Sun et al. [59] conduct a comprehensive evaluation of the learned indices.

With advancements in hardware, GPU computing has become a cutting-edge approach for spatial indexing and query optimization. There has been a increasing interest in utilizing GPU for enhancing the performance of database systems[8, 52]. Algorithms like LBVH [36], G-PICS [40], and GTS [69] exemplify GPU-based solutions in this field. However, these methods face a significant challenge: traditional tree-based query execution models struggle to achieve effective parallelization on GPUs, limiting their ability to fully utilize GPU computational power. While G-PICS and GTS employ batch query processing to enhance parallelism, this approach introduces storage overhead, constraining performance due to limited GPU memory resources.

Linear algebra kernels, inherently suited for GPU computation, have been applied in database systems, particularly in graph databases [31, 34]. Graphs can be naturally represented as adjacency matrices, making BLAS a natural fit. Since Kepner et al. [37] established the theoretical foundation for converting graph operations into BLAS, BLAS-based graph query solutions have flourished [20, 28, 54]. This success suggests potential for extending BLAS-based approaches to other database queries. However, to the best of our knowledge, applying BLAS to spatial range queries on multi-dimensional data remains an open research area, offering opportunities for innovation.

7 CONCLUSION AND FUTURE WORK

This paper introduces BLAEQ, a multigrid index designed to accelerate spatial queries on geometric data, with a particular focus on optimization for GPU architectures. BLAEQ utilizes matrix operations to restructure traditional indexing into a multigrid architecture. Additionally, we present novel query execution strategies that harness the power of BLAS operations, while also taking advantage of the pruning capabilities inherent in indexing mechanisms. This approach facilitates the integration of GPU, thereby enhancing the overall performance.

Notwithstanding the advancements presented by BLAEQ, certain limitations remain in its current implementation. Our future work will prioritize the following areas for improvement:

Dimensionality Challenge. The current version of BLAEQ is limited to low-dimensional datasets, such as geometry data, due to performance bottlenecks with higher dimensions. Addressing this challenge is crucial for extending BLAEQ to broader applications.

Hybrid Architecture. While On-RAM and On-Disk implementations of BLAEQ have been developed, their current design is intuitive and leaves significant room for performance optimization. Further research is needed to refine these implementations for larger datasets.

Kernel Optimization. Although BLAEQ outperforms existing solutions on geometry data, it does not fully utilize GPU computational power, as some kernel functions face underutilization issues. Designing custom GPU kernels is a promising direction for improving performance.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (2023YFB3308000), National Natural Science Foundation of China (92267203, 62372337).

REFERENCES

[1] Daniar Achakeev and Bernhard Seeger. 2012. A class of R-tree histograms for spatial databases. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. ACM, New York, NY, USA.

[2] Ariful Azad and Aydin Buluc. 2017. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 688–697.

[3] Timothy J Baker. 2005. Mesh generation: Art or science? *Prog. Aerosp. Sci.* 41, 1 (Jan. 2005), 29–63.

[4] L Balasubramanian and M Sugumaran. 2012. A state-of-art in R-tree variants for spatial indexing. *International Journal of Computer Applications* 42, 20 (2012), 35–41.

[5] Rudolf Bayer. 1997. The universal B-tree for multidimensional indexing: General concepts. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 198–209.

[6] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

[7] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, and Andrew Lumsdaine. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151.

[8] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-data-path architecture for high-speed OLAP. *Proc. ACM Manag. Data* 2, 6 (Dec. 2024), 1–26.

[9] Eli Bressert. 2012. *SciPy and NumPy: An Overview for Developers*. "O'Reilly Media, Inc.".

[10] Martin Breunig, Patrick Erik Bradley, Markus Jahn, Paul Kuper, Nima Mazroob, Norbert Rösch, Mulhim Al-Doori, Emmanuel Stefanakis, and Mojgan Jadidi. 2020. Geospatial data management research: Progress and future directions. *ISPRS Int. J. Geoinf.* 9, 2 (Feb. 2020), 95.

[11] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient processing of spatial joins using R-trees. *SIGMOD Rec.* 22, 2 (June 1993), 237–246.

[12] Paul G Brown. 2010. Overview of sciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 963–968.

[13] Ebubekir Buber and Banu Diri. 2018. Performance analysis and CPU vs GPU comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. IEEE, 1–6.

[14] Howard Butler, Bradley Chambers, Preston Hartzell, and Craig Glennie. 2021. PDAL: An open source library for the processing and analysis of point clouds. *Comput. Geosci.* 148, 104680 (March 2021), 104680.

[15] Jianjun Chen, Zhoufang Xiao, Yao Zheng, Jianfeng Zou, Dawei Zhao, and Yufeng Yao. 2018. Scalable generation of large-scale unstructured meshes by a novel domain decomposition approach. *Adv. Eng. Softw.* 121 (July 2018), 131–146.

[16] Siheng Chen, Baoan Liu, Chen Feng, Carlos Vallespi-Gonzalez, and Carl Wellington. 2021. 3D point cloud processing and learning for autonomous driving: Impacting map creation, localization, and perception. *IEEE Signal Process. Mag.* 38, 1 (Jan. 2021), 68–86.

[17] Yuedan Chen, Guoqing Xiao, Kenli Li, Francesco Piccialli, and Albert Y Zomaya. 2022. FgSpMSpV: A fine-grained parallel SpMSpV framework on HPC platforms. *ACM Trans. Parallel Comput.* 9, 2 (June 2022), 1–29.

[18] T J Chung. 2002. *Computational Fluid Dynamics*. Cambridge University Press, Cambridge, England.

[19] L Comba, A Biglia, D Ricauda Aimonino, C Tortia, E Mania, S Guidoni, and P Gay. 2020. Leaf Area Index evaluation in vineyards using 3D point clouds from UAV imagery. *Precis. Agric.* 21, 4 (Aug. 2020), 881–896.

[20] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4 (Dec. 2019), 1–25.

[21] Angjela Davitkova, Evica Milchevski, and S Michel. 2020. The ML-Index: A Multidimensional, Learned Index for point, range, and nearest-neighbor queries. *Int Conf Extending Database Technol* (2020), 407–410.

[22] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[23] J J Dongarra, Jeremy Du Croz, Sven Hammarling, and I S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (March 1990), 1–17.

[24] Harish Doraiswamy and Juliana Freire. 2020. A GPU-friendly Geometric Data Model and Algebra for Spatial Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1875–1885.

[25] Mathias Eggert, Maximilian Schade, Florian Bröhl, and Alexander Moriz. 2024. Generating synthetic LiDAR point cloud data for object detection using the unreal game engine. In *Design Science Research for a Resilient Future*. Springer Nature Switzerland, Cham, 295–309.

[26] Ahmed Eldawy and Mohamed F Mokbel. 2013. A demonstration of SpatialHadoop: an efficient mapreduce framework for spatial data. *Proceedings VLDB Endowment* 6, 12 (Aug. 2013), 1230–1233.

[27] Ahmed Eldawy and Mohamed F Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1352–1363.

[28] Marton Elekes, Attila Nagy, David Sandor, Janos Benjamin Antal, Timothy A Davis, and Gabor Szarnyas. 2020. A GraphBLAS solution to the SIGMOD 2014 Programming Contest using multi-source BFS. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[29] Federico Favaro, Ernesto Dufrechou, Juan P Oliver, and Pablo Ezzatti. 2022. Time-Power-Energy Balance of blas Kernels in Modern fpgas. In *High Performance Computing*. Springer International Publishing, 78–89.

[30] Alberto Ferrari and Karen Willcox. 2024. Digital twins in mechanical and aerospace engineering. *Nat. Comput. Sci.* 4, 3 (March 2024), 178–183.

[31] Vijay Gadepally, Jake Bolewski, Dan Hook, Dylan Hutchison, Ben Miller, and Jeremy Kepner. 2015. Graphulo: Linear Algebra Graph Kernels for NoSQL Databases. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. ieeexplore.ieee.org, 822–830.

[32] V Gadepally, J Kepner, W Arcand, and others. 2015. D4M: Bringing associative arrays to database engines. *2015 IEEE High* (2015).

[33] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proc. ACM SIGMOD Int. Conf. Manag. Data* 1, 1 (May 2023), 1–26.

[34] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Adam Fuchs. 2015. Graphulo implementation of server-side sparse matrix multiply in the Accumulo database. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. ieeexplore.ieee.org, 1–7.

[35] John T Hwang and Joaquim R R A Martins. 2016. An unstructured quadrilateral mesh generation algorithm for aircraft structures. *Aerosp. Sci. Technol.* 59 (Dec. 2016), 172–182.

[36] J Jakob and M Guthe. 2021. Optimizing LBVH-Construction and Hierarchy-Traversal to accelerate k NN Queries on Point Clouds using the GPU. *Comput. Graph. Forum* 40, 1 (Feb. 2021), 124–137.

[37] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. ieeexplore.ieee.org, 1–9.

[38] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, New York, NY, USA, 489–504.

[39] J Kubica, J Masiero, R Jedicke, and others. 2005. Variable kd-tree algorithms for spatial pattern search and discovery. 18 (2005).

[40] Zhila-Nouri Lewis and Yi-Cheng Tu. 2022. G-PICS: A Framework for GPU-Based Spatial Indexing and Query Processing. *IEEE Trans. Knowl. Data Eng.* 34, 3 (March 2022), 1243–1257.

[41] Min Li, Yulong Ao, and Chao Yang. 2020. Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity. *arXiv [cs.DC]* (June 2020).

[42] Mingxin Li, Hancheng Wang, Haipeng Dai, Meng Li, Rong Gu, Feng Chen, Zhiyuan Chen, Shuaituan Li, Qizhi Liu, and Guihai Chen. 2024. A Survey of Multi-Dimensional Indexes: Past and Future Trends. *IEEE Trans. Knowl. Data Eng.* PP, 99 (2024), 1–20.

[43] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119–2133.

[44] Yi Liu, Ning Jing, Luo Chen, and Huizhong Chen. 2011. Parallel bulk-loading of spatial data with MapReduce: An R-tree case. *Wuhan Univ. J. Nat. Sci.* 16, 6 (Dec. 2011), 513–519.

[45] Luan Lyu, Wei Cao, Xiaohua Ren, Enhua Wu, and Zhi-Xin Yang. 2024. Efficient odd–even multigrid for pointwise incompressible fluid simulation on GPU. *Vis. Comput.* (Feb. 2024).

[46] Scott MacLachlan, Tom Manteuffel, and Steve McCormick. 2006. Adaptive reduction-based AMG. *Numer. Linear Algebra Appl.* 13, 8 (Oct. 2006), 599–620.

[47] Kaveh Mirzaei, Mehrdad Arashpour, Ehsan Asadi, Hossein Masoumi, Yu Bai, and Ali Behnood. 2022. 3D point cloud data processing with machine learning for construction and infrastructure applications: A comprehensive review. *Adv. Eng. Inform.* 51, 101501 (Jan. 2022), 101501.

[48] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. 2004. P2PR-tree: An R-tree-based spatial index for peer-to-peer environments. In *Current Trends in Database Technology - EDBT 2004 Workshops*. Springer Berlin Heidelberg, Berlin, Heidelberg, 516–525.

[49] F Moukalled, L Mangani, and M Darwish. 2016. The Finite Volume Method. In *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM® and Matlab*, F Moukalled, L Mangani, and M Darwish (Eds.).

Springer International Publishing, Cham, 103–135.

[50] A Myasishchev, S Lienkov, V M Dzhulii, and I V Muliar. 2019. USING GPU NVIDIA FOR LINEAR ALGEBRA PROLEMS. *Collection of scientific works of the Military Institute of Kyiv National Taras Shevchenko University* 64 (2019), 144–157.

[51] J Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. 1984. The grid file. *ACM Trans. Database Syst.* 9, 1 (March 1984), 38–71.

[52] Shweta Pandey and Arkaprava Basu. 2025. H-Rocks: CPU-GPU accelerated Heterogeneous RocksDB on Persistent Memory. *Proc. ACM Manag. Data* 3, 1 (Feb. 2025), 1–28.

[53] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB array data storage manager. *Proceedings VLDB Endowment* 10, 4 (Nov. 2016), 349–360.

[54] Jon Roose, Miheer Vaidya, Ponnuswamy Sadayappan, and Sivasankaran Rajamanickam. 2023. TenSQL: An SQL Database Built on GraphBLAS. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[55] Christopher J Roy. 2005. Review of code and solution verification procedures for computational simulation. *J. Comput. Phys.* 205, 1 (May 2005), 131–156.

[56] Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *2011 IEEE International Conference on Robotics and Automation*. IEEE, 1–4.

[57] Hari Singh and Seema Bawa. 2017. A survey of traditional and MapReduceBased spatial query processing approaches. *SIGMOD Rec.* 46, 2 (Sept. 2017), 18–29.

[58] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. 2013. SciDB: A Database Management System for Applications with Complex Analytics. *Comput. Sci. Eng.* 15, 3 (2013), 54–62.

[59] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *Proceedings VLDB Endowment* 16, 8 (April 2023), 1992–2004.

[60] E H van Brummelen, K G van der Zee, and R de Borst. 2008. Space/time multigrid for a fluid–structure-interaction problem. *Appl. Numer. Math.* 58, 12 (Dec. 2008), 1951–1971.

[61] Ricardo Vinuesa and Steven L Brunton. 2022. Enhancing computational fluid dynamics with machine learning. *Nat. Comput. Sci.* 2, 6 (June 2022), 358–366.

[62] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang (Eds.). Springer International Publishing, Cham, 167–188.

[63] Jane Wilhelms and Allen Van Gelder. 1992. Octrees for faster isosurface generation. *ACM Trans. Graph.* 11, 3 (July 1992), 201–227.

[64] Guoqing Xiao, Kenli Li, Yuedan Chen, Wangquan He, Albert Y Zomaya, and Tao Li. 2021. CASpMV: A Customized and Accelerative SpMV Framework for the Sunway TaihuLight. *IEEE Trans. Parallel Distrib. Syst.* 32, 1 (Jan. 2021), 131–146.

[65] Xiaowen Xu, Xiaoqiang Yue, Runzhang Mao, Yuntong Deng, Silu Huang, Haifeng Zou, Xiao Liu, Shaoliang Hu, Chunsheng Feng, Shi Shu, and Zeyao Mo. 2023. JXPAMG: a parallel algebraic multigrid solver for extreme-scale numerical simulations. *CCF Trans. High Perform. Comput.* 5, 1 (March 2023), 72–83.

[66] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on GPUs using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '13)*. Association for Computing Machinery, New York, NY, USA, 23–31.

[67] Fan Yuan, Xiaojian Yang, Shengguo Li, Dezun Dong, Chun Huang, and Zheng Wang. 2024. Optimizing Multi-Grid Preconditioned Conjugate Gradient Method on Multi-Cores. *IEEE Trans. Parallel Distrib. Syst.* 35, 5 (May 2024), 768–779.

[68] Wanyi Zhang, Xiuhua Fu, and Wei Li. 2022. Point cloud computing algorithm on object surface based on virtual reality technology. *Comput. Intell.* 38, 1 (Feb. 2022), 106–120.

[69] Yifan Zhu, Ruiyao Ma, Baihua Zheng, Xiangyu Ke, Lu Chen, and Yunjun Gao. 2024. GTS: GPU-based Tree Index for Fast Similarity Search. *arXiv [cs.DB]* (April 2024).