Parallel Skyline Processing Using Space Pruning on GPU

Chuanwen Li
College of Computer Science and Engineering
Northeastern University, China
lichuanwen@mail.neu.edu.cn

Yu Gu
College of Computer Science and Engineering
Northeastern University, China
guyu@mail.neu.edu.cn

Jianzhong Qi
School of Computing and Information Systems
The University of Melbourne, Australia
jianzhong.qi@unimelb.edu.au

Ge Yu
College of Computer Science and Engineering
Northeastern University, China
yuge@mail.neu.edu.cn

ABSTRACT

Skyline computation is an essential database operation that has many applications in multi-criteria decision making scenarios such as recommender systems. Existing algorithms have focused on checking point domination, which lack efficiency over large datasets. We propose a grid-based structure that enables grid cell domination checks. We show that only a small constant number of cells need to be checked which is independent from the number of data points. Our structure also enables parallel processing. We thus obtain a highly efficient parallel skyline algorithm named SkyCell, taking advantage of the parallelization power of graphics processing units. Experimental results confirm the effectiveness and efficiency of SkyCell – it outperforms state-of-the-art algorithms consistently and by up to over two orders of magnitude in the computation time.

CCS CONCEPTS

• **Information systems → Location based services**; • **Computing methodologies →** *Parallel computing methodologies.*

KEYWORDS

skyline, parallel computing, spatial query, GPU

ACM Reference Format:
Chuanwen Li, Yu Gu, Jianzhong Qi, and Ge Yu. 2022. Parallel Skyline Processing Using Space Pruning on GPU. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

The *skyline query* is an essential query in multi-criteria decision making applications such as recommender systems and business management (e.g., to compute the *Pareto frontier*) [3, 6, 9, 13, 20, 32]. It retrieves data points that are not *dominated* by any other points in a dataset. Suppose each point has d attributes. A point p_i dominates another point p_j if p_i is better than p_j in at least one attribute and is as good as p_j in all other attributes. The "better than" relationship is often quantified as having a smaller attribute value. Consider recommending restaurants to a user. In Table 1, there are four restaurants each with three attributes: average cost per person, distance to the user, and rating rank (1 is the top rank). Restaurants r_1 and r_3 are dominated by r_2, as they are more expensive, farther away, and rated lower. Neither r_2 nor r_4 is dominated. They are the skyline points, which can be used for recommendation.

Table 1: A Restaurant Recommendation Example

Restaurant	Average Cost	Distance	Rating Rank
r_1	\$12	9 km	3
r_2	\$8	3 km	2
r_3	\$10	17 km	4
r_4	\$26	8 km	1

In recent years, the rapid growth of online services has accumulated a large volume of data and a large user base which brings new challenges to skyline queries. Computing skyline queries to make recommendations from such large item sets requires highly efficient query algorithms, especially when the computation needs to be done online based on users' different search requirements and contextual features such as location, financial capacity, and personal preferences. Existing algorithms suffer in efficiency for such application scenarios. The state-of-the-art sequential skyline algorithm [20] takes 5 seconds (amortized) to process a skyline query on just 2 million 5-dimensional points (cf. Section 6), failing to reach the two-second response time requirement [22]. More efficient algorithms are in need, which is addressed in this paper.

On-going efforts [3, 4, 12, 21, 30, 36] have been made to parallelize skyline computation. *Graphics processing units* (GPU) are used for their strong parallelization capability. Existing algorithms mostly fall into two groups: *sorting-based* and *partitioning-based* [38]. However, algorithms in both groups are facing difficulties when parallelized. Most GPU-powered sorting-based skyline algorithms [2, 8] are adaptations of their sequential counterparts. These algorithms also check all points to grow the skyline buffer, which hinders their efficiency. The state-of-the-art sorting-based algorithm [20] turns back to sequential processing. This algorithm, however, requires expensive pre-computations and may suffer when there are updates. Partitioning-based algorithms have recursive partitioning procedures [17, 18, 25, 34] or tree-like structures to reduce point domination checks [3]. They are intrinsically difficult for GPU processing. To avoid such issues, the state-of-the-art partitioning-based skyline algorithm uses GPU with a grid partition [3]. It partitions each dimension into 16 segments regardless of the dataset size, which cannot fully exploit the GPU throughput and may cause branch divergence of GPU warps. Branch divergence occurs when some threads in a warp (which often consists of 32 threads) idle and wait for the other threads in the same warp to execute different conditional branches. This negatively impacts the GPU utility and computation efficiency.

A key limitation in the existing algorithms is that they mostly check for *point* domination to identify the skyline points. Although some approaches claim to use region-based pruning, their regions are still defined by points (cf. Section 7). These approaches lack efficiency as the number of data points becomes large. For example, computing the skyline points from an OpenStreetMap dataset of 1.6 billion points takes some 10 seconds even with the state-of-the-art GPU-based parallel algorithm [3] (cf. Section 6). This hinders user experience for online skyline queries as discussed above. We aim to achieve sub-second query times even on such large datasets.

We observe that the data space can be partitioned into regions such that domination checks can be performed among the regions. This enables pruning by regions without examining the points in each region. We show that only a small constant number of (non-dominated) regions contain skyline points. We thus propose an efficient algorithm to compute such regions and hence the skyline points, which scales much better with the dataset size.

We adopt a space pruning idea, which partition the data space with a regular grid and check for domination between the grid cells based on their relative position. Intuitively, the cells with smaller coordinates dominate those with larger ones. We show that only cells that are *not* dominated contain skyline points. Such cells are named the *candidate cells*.

We prove that the number of candidate cells is bounded by the data dimensionality and the grid granularity, and it is *independent* of the dataset size. We further show that a candidate cell can be partitioned recursively to form smaller candidate cells (in grids of larger granularities). As the grid granularity increases, each candidate cell becomes smaller, and the portion of the space covered by candidate cells decreases monotonically. For an 8×8 grid in 2-dimensional Euclidean space, there are 15 candidate cells (i.e., 23% of the 64 cells). When the granularity reaches 32×32, there are only 63 candidate cells (i.e., 6.2% of the 1,024 cells).

Based on these key properties, we proposed a **cell**-based **sky**line algorithm named **SkyCell** that progressively computes the candidate cells in grids with increasing granularities, until each candidate cell contains only a small number of points. From the resultant cells, skyline points can be computed efficiently with existing point domination based algorithms (e.g., sort-first skyline [9]).

SkyCell processes each candidate cell independently. This offers an important opportunity to improve the algorithm efficiency with parallelization. We thus further propose a parallel SkyCell algorithm using GPU. To take full advantage of the parallelization power of GPU, we carefully design our algorithm to avoid warp divergence, and we arrange the data to promote coalesced memory access. We thus achieve a highly efficient algorithm that outperforms state-of-the-art parallel skyline algorithms by up to two orders of magnitude.

In summary, we make the following contributions:

- We propose a novel approach for skyline computation based on grid partitioning and candidate cells. By using cell domination checks, our approach significantly reduces the number of domination checks and has a much better scalability.
- We derive a theoretical bound on the number of candidate cells to be examined. We further show how such cells can be recursively partitioned to yield smaller cells without missing any skyline points. Based on these, we propose a skyline algorithm named SkyCell.
- Since candidate cells can be computed independently, we further propose a parallel SkyCell algorithm, taking full advantage of GPU parallelization. Our algorithms do not require any pre-computation and hence are robust to data updates.
- We perform cost analysis and extensive experiments. The results confirm the superiority of our algorithm over the state-of-the-art skyline algorithms, which reduces the query times by over an order of magnitude in many cases.

2 MULTI-LAYER GRID PARTITIONING

We start with a problem statement and our core data structure.

Problem statement. Given a set $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ of n points in d-dimensional ($d > 1$) Euclidean space, we aim to compute the subset $\mathcal{S} \subset \mathcal{P}$ of all *skyline points* in $\mathcal{P}$, i.e., the *skyline set* of $\mathcal{P}$. Below, we define skyline points and key concepts.

Skyline points are defined based on *point domination.* Let $p[k]$ be the coordinate of a point p in dimension k.

Definition 1. *(Point domination) We say that a point p_i* dominates *another point p_j, denoted by $p_i \prec p_j$, if $\forall k \in [0, d), p_i[k] \leq p_j[k]$ and $\exists l \in [0, d), p_i[l] < p_j[l]$.*

Definition 2. *(Skyline point) We call $p_i \in \mathcal{P}$ a* skyline point *of $\mathcal{P}$ if p_i is not dominated by any other point $p_j \in \mathcal{P}$, i.e., $\nexists p_j \in \mathcal{P}, p_j \prec p_i$.*

We check for domination between space partitions to enable fast search space pruning. If a partition is dominated, all points inside can be pruned. Next, we describe our structure to enable this partition-based pruning.

2.1 Proposed Grid Structure

We consider the space as a d-dimensional unit hyper-cube and partition it with a multi-layer grid. The top grid layer (Layer 0) has the coarsest granularity (i.e., the entire data space is a cell), while the bottom layer (Layer ρ, where ρ is a system parameter) has the finest granularity. Each layer is a regular grid, with $2^{i \cdot d}$ *cells* in Layer i. In Fig. 1, $d = 2$, and we have $2^{0\times 2} = 1$ to $2^{4\times 2} = 256$ cells for Layers 0 to 4. Each layer has the same unit size. Layer 4 has been zoomed in for better visibility.

Let the set of cells in Layer i be C_i. A cell $c = C_i[cl_{d-1}, \ldots, cl_0]$ is indexed by its column numbers, i.e., it is at columns $cl_{d-1}, \ldots, cl_0$ in dimensions $d-1, \ldots, 0$, respectively. We use $c[k]$ to denote the index (column number) of c in dimension k: $c[k] = cl_k$. In Fig. 1, cell $c = C_4[10, 1]$ in Layer 4 is at column 10 in dimension 1 (the vertical dimension) and column 1 in dimension 0 (the horizontal dimension), i.e., $c[1] = 10$ and $c[0] = 1$.

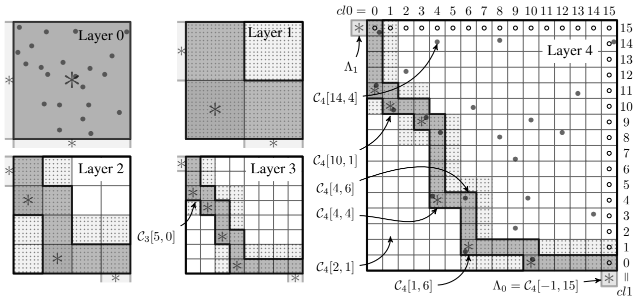


Figure 1: Example of multi-layered data space partitioning (the black points denote data points)

Since we consider points in a unit hyper-cube $[0,1)^d$, in Layer i, the cell c to which point p belongs is calculated by:

$$c = C_i[\lfloor p[d-1] \cdot 2^i \rfloor, \ldots, \lfloor p[0] \cdot 2^i \rfloor]. \tag{1}$$

For example, in Fig. 1, point $p = (0.63, 0.08)$ belongs to cell $C_3[\lfloor 0.63 \times 2^3 \rfloor, \lfloor 0.08 \times 2^3 \rfloor] = C_3[5,0]$ in Layer 3 and cell $C_4[\lfloor 0.63 \times 2^4 \rfloor, \lfloor 0.08 \times 2^4 \rfloor] = C_4[10,1]$ in Layer 4.

2.2 Candidate Cells

Our skyline algorithm only needs to consider a subset of the grid cells in each layer, which are named the *candidate cells* and are defined based on the *cell domination* relationship.

Cell domination. *In what follows, when multiple cells are used, they refer to cells of the same layer, unless otherwise stated.*

Definition 3. *(Cell domination) We say that cell c_i* dominates *cell c_j, denoted by $c_i \prec c_j$, if c_i is not empty (i.e., enclosing points in $\mathcal{P}$), and the index of c_i is less than that of c_j in each dimension:*

$$c \neq \emptyset \wedge \forall k \in [0,d), c_i[k] < c_j[k] \tag{2}$$

We say that c_i partially dominates *c_j, denoted by $c_i \preceq c_j$, if c_i is not empty, the index of c_i equals to that of c_j in at least one dimension, and the index of c_i is less than that of c_j in all other dimensions:*

$$c \neq \emptyset \wedge \forall k \in [0,d), c_i[k] \leq c_j[k] \wedge \exists k \in [0,d), c_i[k] = c_j[k] \tag{3}$$

We say $c_i \precsim c_j$ if c_i dominates or partially dominates c_j:

$$c_i \precsim c_j \iff c_i \prec c_j \vee c_i \preceq c_j \tag{4}$$

By definition, a cell partially dominates itself, i. e., $c \precsim c$, and the "$\precsim$" relationship is transitive:

Lemma 1. *If $c_i \precsim c_j$ and $c_j \precsim c_k$, then $c_i \precsim c_k$.*

Proof. Straightforward based on Definition 3. □

By domination, there are three types of cells in each layer.

(1) *Dominated cells* – cells that are dominated by some other cells, e.g., $C_4[14,4]$ in Fig. 1 is dominated by $C_4[10,1]$ which is non-empty (the dot in the cell represents a data point).

(2) *Irrelevant cells* – cells that are neither dominated nor partially dominated, and do not dominate other cells, e.g., $C_4[2,1]$ in Fig. 1. These are empty cells with small column numbers. Note that irrelevant cells are all empty, but empty cells are not all irrelevant (i.e., empty cells may also be dominated cells).

(3) *Candidate cells* – cells that do not belong to the two types above, e.g., $C_4[10,1]$ in Fig. 1.

No skyline points can be found from any cell c_j dominated by another cell c_i, since points in c_i must dominate those in c_j. Thus, *we can only find skyline points from the candidate cells.*

Key cells. Next, we define candidate cells formally, starting with a special subset of cells called the *key cells.*

Definition 4. *(Key cell) We call a non-empty cell that is neither dominated nor partially dominated by any other cell a* key cell.

We denote the set of all key cells in layer i as $\mathcal{KC}_i$. In Fig. 1, cells marked by a "$*$" are the key cells. Analogically, in a 2D scenario, think of each grid layer (e.g., layer 4) as a beach image where the empty cells at the bottom-left corner are pixels of the water. Then, the key cells are the pixels of the vertices of the beach line.

Any cell that is either empty or dominated by a key cell (cf. white cells in Fig. 1) cannot contain skyline points, and it is not a candidate cell. The remaining non-key cells each must be partially dominated by some key cell. We denote the set of cells partially dominated by a key cell c but not dominated by other key cells as $\mathcal{PDC}(c)$.

$$\mathcal{PDC}(c) = \{c' \mid c \preceq c' \wedge \nexists c'' \in \mathcal{KC}, c'' \prec c'\}. \tag{5}$$

We call a cell in $\mathcal{PDC}(c)$ a *partially dominated cell* of c. In Fig. 1, the gray cells in the same row or column of a key cell are those partially dominated by the key cell. Such a cell may be partially dominated by multiple key cells, but this will not impact discussions below.

Candidate cells. Continue with the beach image analogy. The set of all partially dominated cells, together with all the key cells, form the beach line, which are the *candidate cells.*

Definition 5. *(Candidate cell) The set of candidate cells of the i-th layer, denoted by CC_i, contains and only contains the key cells in $\mathcal{KC}_i$ and their partially dominated cells. Formally,*

$$CC_i = \mathcal{KC}_i \cup \bigcup_{c \in \mathcal{KC}_i} \mathcal{PDC}(c). \tag{6}$$

Next, we detail our skyline algorithm based on candidate cells.

3 THE SKYCELL ALGORITHM

Our skyline query algorithm named **SkyCell** is based on the following observation. All candidate cells of Layer $i+1$, i.e., CC_{i+1}, can be generated from the set of candidate cells of Layer i, i.e., CC_i. Intuitively, each cell in Layer i is partitioned evenly into 2^d cells in Layer $i+1$. As Fig. 1 shows, Layer 0 has only 1 cell, which is partitioned into $2^d = 2^2 = 4$ cells in Layer 1. If a cell c_j is dominated by another cell c_k in Layer i, any cell in Layer $i+1$ resulted from partitioning c_j must still be dominated by some cell resulted from partitioning c_k. Thus, only non-fully dominated cells (i.e., the candidate cells CC_i) in Layer i can yield candidate cells in Layer $i+1$. In Fig. 1, the dotted cells in any layer all come from partitioning of the gray cell in the previous layer, and all the candidate (gray) cells are dotted, e.g., the three gray cells at the top-left corner of Layer 2 come from the top-left gray cell of Layer 1.

Based the observation above, our SkyCell algorithm starts from CC_0 which is just C_0, i.e., a single cell containing the unit hyper-cube data space. It then iteratively computes CC_{i+1} from CC_i. When i increases, the area covered by candidate cells shrinks (cf.

the gray area from Layer 0 to Layer 3 in Fig. 1), since some of the cells in Layer $i + 1$ resulted from partitioning a candidate cell in Layer i may become dominated and do not qualify to be candidate cells any more. The number of points covered by the candidate cells, which are skyline candidates as discussed earlier, also decreases. In an extreme case, we can terminate SkyCell when every candidate cell contains at most one point, to minimize the number of skyline candidates. However, this may lead to too many grid layers and candidate cells to be computed which can be expensive. To balance the workload of candidate cell computation and final skyline point checking, we terminate when i reaches ρ which is a *partition ratio* parameter and will be chosen empirically.

```
Algorithm 1: SkyCell
  input  : Dataset $\mathcal{P}$
  output: Skyline set $\mathcal{S}$
1 Compute $C_\rho$ to $C_0$ from $\mathcal{P}$
2 $\mathcal{R}_0 \leftarrow C_0[0, \ldots, 0]$
3 for $i = 0$ to $\rho - 1$ do
4     $\mathcal{R}_{i+1} \leftarrow$ ShrinkKeyCells($\mathcal{P}$, $i$, $\mathcal{R}_i$, $C_{i+1}$)
5 return RefineSkyline($\mathcal{P}$, $\mathcal{R}_\rho$)
```

The algorithm. We summarize SkyCell in Algorithm 1. SkyCell first computes a ρ-layer grid partitioning over dataset $\mathcal{P}$ (Line 1). We store the points in an array and sort them according to the Layer-ρ cells to which they belong. Any cell ordering can be used, e.g., the Z-order [19]. We just require points from the same cell to occupy a consecutive segment of the array. Then, for each Layer-ρ cell, we record the starting and ending array indices of the points in the cell. An empty cell has the same starting and ending array indices. This constructs C_ρ of our grid structure.

We construct $C_{\rho-1}$ from C_ρ. For each cell $c \in C_{\rho-1}$, we record whether it is non-empty (i.e., encloses data points), which will be used for key cell testing later. This is done by a simple scan over the starting and ending array indices (which can be calculated easily) of the cells in C_ρ resulted from partitioning c. Similarly, we construct the other layers from $C_{\rho-2}$ back to C_0 (Line 1).

Then, we compute a set $\mathcal{R}_i$ of cells of interest for each Layer i based on the observation outlined above with a sub-procedure named `ShrinkKeyCells` (Lines 2 to 4, detailed later; we call it `ShrinkKeyCells` because we compute the key cells as the media to compute the candidate cells.). For our parallel algorithm, $\mathcal{R}_i$ contains key cells and candidate cells ($\mathcal{R}_i = \{\mathcal{KC}_i, CC_i\}$). Here, as mentioned above, C_0 has only one cell (i.e., the entire data space), which is used as $\mathcal{KC}_0$ and CC_0.

When $\mathcal{R}_\rho$ is computed, $\mathcal{KC}_\rho$ is also computed. We use $\mathcal{KC}_\rho$ to compute CC_ρ using a procedure similar to `ShrinkKeyCells` that computes candidate cells from key cells (details omitted for succinctness). We then compute and return the skyline points from points in CC_ρ as the result. As points from candidate cells partially dominated by different key cells do not dominate each other, the set of candidate cells partially dominated by different key cells can be processed in parallel. We use the *sort-first skyline* [9] algorithm to compute skyline points in each set (other algorithms may also apply). These steps are summarized in `RefineSkyline` (Line 5).

In the skyline point computation step, a cell may be partially dominated by more than one key cell. A point in such a cell c is a skyline point only if it is not dominated in any of the sets of partially dominated cells that contain c.

```
Algorithm 2: ShrinkKeyCells (Parallel)
   input  : Dataset $\mathcal{P}$, current layer number $i$, $CC_i$, $\mathcal{KC}_i$, $C_{\delta+1}$
   output: Candidate cells $CC_{i+1}$, key cells $\mathcal{KC}_{i+1}$
1  Assign $sub\_cell(CC_i)$ to pointers $o$ and $u$ for nodes in $\mathcal{T}[0]$
2  for $m = 0$ to $d - 2$ do
3      Reorder $\mathcal{T}[0]$ by rotation-$m$
4      if $m \neq 0$ then
5          for $j = 0$ to $|\text{sub\_cell}(CC_i)|$ par-do
6              $\mathcal{T}[0, j].u \leftarrow \mathcal{T}[0, j].l$
7      for $j = 1$ to $\mathcal{T}.h$ do
8          for $k = 0$ to $2^{\mathcal{T}.h-j} - 1$ par-do
9              $\mathcal{T}[j, k].u \leftarrow \mathsf{dom}_2(\mathcal{T}[j-1, 2k].u, \mathcal{T}[j-1, 2k+1].u)$
10     for $j = \mathcal{T}.h$ to 0 do
11         for $k = 0$ to $2^{\mathcal{T}.h-j} - 1$ par-do
12             if $k = 0$ then
13                 $\mathcal{T}[j, k].l \leftarrow \mathcal{T}[j, 0].u$
14             else if $k$ is odd then
15                 $\mathcal{T}[j, k].l \leftarrow \mathcal{T}[j+1, (k-1)/2].l$
16             else
17                 $\mathcal{T}[j, k].l \leftarrow \mathsf{dom}_2(\mathcal{T}[j+1, k/2-1].l, \mathcal{T}[j, k].u)$
18 $CC_{i+1} \leftarrow \{\mathcal{T}[0, j].o \,|\, \mathcal{T}[0, j].l \leq \mathcal{T}[0, j].o, 0 \leq j < |sub\_cell(CC_i)|\}$
19 $\mathcal{KC}_{i+1} \leftarrow \{\mathcal{T}[0, j].o \,|\, \mathcal{T}[0, j].l = \mathcal{T}[0, j].o, 0 \leq j < |sub\_cell(CC_i)|\}$
20 return $CC_{i+1}$, $\mathcal{KC}_{i+1}$
```

4 THE SHRINKKEYCELLS ALGORITHM

Next, we focus on our parallel `ShrinkKeyCells` algorithm (our sequential `ShrinkKeyCells` algorithm is omitted due to space limit and will be released as part of a technical report). The algorithm takes CC_i and $\mathcal{KC}_i$ as the input. It generates $sub\_cell(CC_i)$ and then compares the cells, to find those not dominated by other cells and those not partially dominated by other cells. This yields the candidate cells CC_{i+1} and the key cells $\mathcal{KC}_{i+1}$, respectively. We parallelize the cell comparison with a tournament-style procedure.

Cell preparation. To generate cells in $sub\_cell(CC_i)$, we partition each cell in CC_i into 2^d cells by an even partitioning in each dimension. Here, we do not prune the cells. We number the generated cells by their enumeration order (ascending). In Fig. 2a, the dotted cells denote the cells in $sub\_cell(CC_2)$ in Layer 3 (cf. Fig. 1). The number in cell denotes the *cell number.*

To compare the cells in $sub\_cell(CC_i)$ and identify those in CC_{i+1} and $\mathcal{KC}_{i+1}$, we construct an auxiliary binary tree $\mathcal{T}$. In this tree, each non-leaf node has two pointers u and l to point to the cells. Each leaf node has three pointers o, u, and l. We initialize both o and u of each leaf node (from left to right) to point to a cell in $sub\_cell(CC_i)$, in ascending order of the cell numbers. Fig. 2b shows such a tree for Fig. 2a. Every tree node (a circle) has two numbers. The upper (and lower) number represents the cell number of the cell pointed to by u (and l). At start, only the upper half (pointer u) of the leaf nodes are labeled with cell numbers from 0 to 27 (o points to the same cell as u does and is not plotted). The rest of the nodes and pointers are computed later. The tree levels are numbered bottom-up, i.e., the leaf level is Level 0, i.e., $\mathcal{T}[0]$.

Note that $\mathcal{T}$ is a complete binary tree of $|sub\_cell(CC_i)| = 2^d|CC_i|$ leaf nodes. As Corollary 1 (Section 5.1) will show, the number of candidate cells in each layer is bounded. Thus, the number of leaf nodes and the tree height $\mathcal{T}.h$ can be computed directly, and $\mathcal{T}$ is implemented as an array for fast parallel access.

Algorithm 2 summarizes parallel `ShrinkKeyCells`, where Line 1 corresponds to the cell preparation above. Note that we carefully parallelize the algorithm to fully utilize the numerous cores of GPU.

Cell domination. Next, we construct the upper levels of $\mathcal{T}$, during which cells in $sub\_cell(CC_i)$ are checked for domination. We first update pointer u for the tree nodes bottom up (Lines 7 to 9). At tree level j (j starts at 1, i.e., parent nodes of the leaf nodes), let the k-th node be $\mathcal{T}[j,k]$. Its pointer u, $\mathcal{T}[j,k].u$, will point to one of the two cells pointed to by the u pointers of its two child nodes:

$$\mathcal{T}[j,k].u = \mathsf{dom}_2(\mathcal{T}[j-1,2k].u, \mathcal{T}[j-1,2k+1].u)$$

Function $\mathsf{dom}_2(\cdot)$ checks for 2-*domination* ('$\precsim_2$') between the cells pointed to by $\mathcal{T}[j-1,2k].u$ and $\mathcal{T}[j-1,2k+1].u$.

Definition 6. *Given two cells c_1 and c_2, if c_1 k-dominates c_2, denoted by $c_1 \precsim_k c_2$, then $c_1 \precsim c_2$ and $\forall j \in [k, d-1], c_1[j] = c_2[j]$. Recall that '$\precsim$' denotes dominate or partially dominate.*

Intuitively, k-domination checks for domination (or partial domination, same below) in the lower k dimensions. We use $\mathsf{dom}_2(\cdot)$ to check for two dimensions each time, and we rotate the dimensions such that all dimensions will be checked (Lines 2 and 3). In each rotation, dimension k becomes dimension $k-1$ for $k > 0$, while dimension 0 becomes the new dimension $d-1$ (nodes in $\mathcal{T}[0]$ is also reordered by the new column indices of the cells). A total of $d-1$ rotations are needed for d dimensions. We only check for two dimensions each time to guarantee correctness.

Function $\mathsf{dom}_2(\mathcal{T}[j-1,2k].u, \mathcal{T}[j-1,2k+1].u)$ returns $\mathcal{T}[j-1,2k].u$ if it points to a cell that 2-dominates the cell pointed to by $\mathcal{T}[j-1,2k+1].u$. Otherwise, it returns $\mathcal{T}[j-1,2k+1].u$. In Fig. 2, $\mathcal{T}[1,0].u = \mathcal{T}[0,1].u$, i.e., pointer u of the left-most level-1 node should point to cell 1, because cell 0 is empty and it does not 2-dominate cell 1. As another example, $\mathcal{T}[2,1].u$ is 7 instead of 4 since $\mathcal{T}[1,2].u = 4$ does not 2-dominates $\mathcal{T}[1,3].u = 7$. The function is computed for every adjacent pair of nodes in parallel, as denoted by **par-do** in the algorithm.

Computing the u pointers bottom up pushes the cells that dominate more cells to higher levels (e.g., the root node points to cell 20 which partially dominates 7 cells, Lines 4 to 9). Such cells may dominate more than just the cells in adjacent nodes. Next, we run a top-down procedure to check for domination between such cells and the cells in non-adjacent nodes, with the help of the l pointers (Lines 10 to 17). At tree level j (j starts at $\mathcal{T}.h$), pointer l of the k-th node is updated according to whether k is 0, odd, or even (detailed by Lemma 5).

After the l pointers are updated, in the m-th rotation, for each node $N \in \mathcal{T}[0]$, $N.l$ points to a cell c that $(m+2)$-dominates the cell pointed to by $N.o$, while c is not dominated by others (detailed in Lemma 5). After $d-1$ rotations, for each node $N \in \mathcal{T}[0]$, $N.l$ points to the cell that dominates or partially dominates the cell pointed to by $N.o$ (if there exists such a cell). The cell pointed to by $N.o$ is a candidate cell if $N.l \neq N.o$ (Line 18), and a key cell otherwise (Line 19, e.g., cells 1, 12, 17, and 20 in Fig. 2).

GPU-based implementation. Line 3 of the algorithm reorders $\mathcal{T}[0]$ by rotation-m. Since the rotation procedure is predefined, it can be parallelized by assigning each entry in $\mathcal{T}[0]$ a thread. As the new position of each entry can be calculated directly, no locking is required. After each rotation, the reads and writes between Lines 4 and 17 all access adjacent memory addresses and hence follow the coalesced memory access pattern for optimal GPU utility.

The operations at Lines 18 and 19 need a special treatment. This is because, when we check $\mathcal{T}[0]$ in parallel, we cannot determine the exact position of each entry found. Therefore, we need to run Lines 18 and 19 twice. First, we check $\mathcal{T}[0]$ to decide the entries to be added into CC_{i+1} or $\mathcal{K}C_{i+1}$. We compute a parallel prefix-sum on these entries to assign each a proper position. Then, we write the results into CC_{i+1} and $\mathcal{K}C_{i+1}$ in parallel.

Correctness. Next, we prove the algorithm correctness. We use o, l, and u to refer to the cells pointed to by them. Our proof is built on the following four lemmas.

Lemma 2. *Given two cells c' and c, if $c' \precsim c$, then $c' \precsim \mu_i(c', c), \forall 0 < i < d$, where $\mu_i(c', c)$ is a cell with $\mu_i(c', c)[j] = c'[j], \forall 0 \leq j < i$ and $\mu_i(c', c)[j] = c[j], \forall i \leq j < d$.*

Proof. Straightforward based on Definition 3. □

We define sets $O(N)$ and $L(N)$ for node N: $O(N)$ contains o cells in the leaf nodes of the subtree rooted at N; $L(N)$ includes $O(N)$ and all o cells in the leaf nodes preceding the subtree rooted at N.

$$O(\mathcal{T}[j,k]) = \bigcup_{i=2^j k}^{2^j(k+1)-1} \{\mathcal{T}[0,i].o\}$$

$$L(\mathcal{T}[j,k]) = \bigcup_{i=0}^{2^j(k+1)-1} \{\mathcal{T}[0,i].o\}$$

For example, $O(\mathcal{T}[2,1]) = \{c_4, c_5, c_6, c_7\}$ and $L(\mathcal{T}[2,1]) = \{c_0, \ldots, c_7\}$.

Given a list of cells C, we define $\beta(C,k)$ as the cell that satisfies the following conditions:

(1) $\beta(C,k)$ belongs to C;
(2) $\beta(C,k)$ k-dominates *the last cell* of C; and
(3) $\beta(C,k)$ is not k-dominated by any other cell in C.

Here, "the last cell" is defined by sorting the cells in C by their column indices in ascending order with the current rotation of the dimensions. For example, if $C = \{c_0, c_1, c_2, c_3\}$ in Fig. 2, the last cell is c_3, and $\beta(C, 2)$ is c_1.

These definitions help show a property of the o cells for dimension rotation (iteration) $m = 0$.

Lemma 3. *In Algorithm 2, at the end of the iteration for $m = 0$, each node $\mathcal{T}[j,k]$ satisfies $\mathcal{T}[j,k].u = \beta(O(\mathcal{T}[j,k]), 2)$.*

Proof. We omit the proof due to space limit. This will be released as part of a technical report. □

We also show a property of the l cells of the leaf nodes for dimension rotation (iteration) $m = 0$.

Lemma 4. *In Algorithm 2, at the end of the iteration for $m = 0$, each node $\mathcal{T}[j,k]$ satisfies $\mathcal{T}[j,k].l = \beta(L(\mathcal{T}[j,k]), 2)$.*

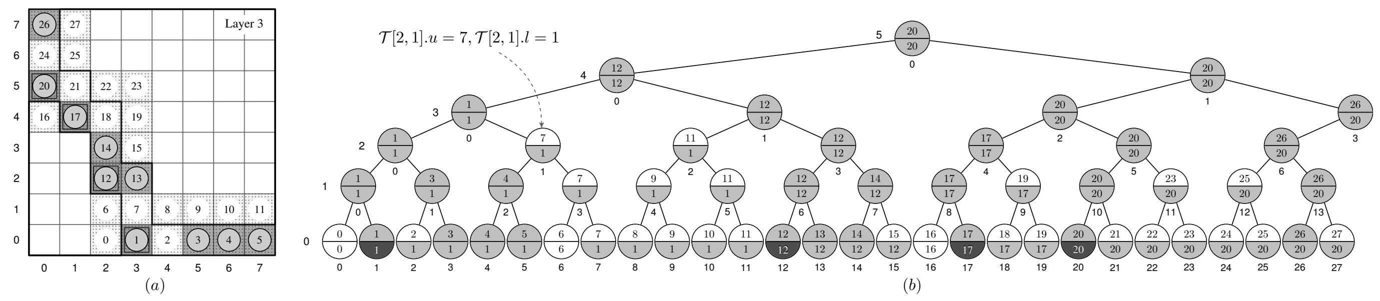


Figure 2: Example of parallel ShrinkKeyCells

Proof. At Line 10, for $j = \mathcal{T}.h$, there is only one node $\mathcal{T}[\mathcal{T}.h, 0]$, and $O(\mathcal{T}[\mathcal{T}.h, 0]) = L(\mathcal{T}[\mathcal{T}.h, 0])$. Let $\mathcal{T}[\mathcal{T}.h, k].l = \mathcal{T}[\mathcal{T}.h, k].u$ for the only value $k = 0$ at Line 13. Then, $\mathcal{T}[j, k].l = L(\mathcal{T}[j, k])$.

When nodes $\mathcal{T}[j, \cdot].l$ satisfy the lemma, nodes $\mathcal{T}[j-1, \cdot].l$ also satisfy the lemma, with three cases of k:

(1) $k = 0$: $O(\mathcal{T}[j-1, k]) = L(\mathcal{T}[j-1, k])$, we set $\mathcal{T}[j-1, k].l = \mathcal{T}[j-1, k].u$. Since $\mathcal{T}[j-1, k].u$ is $\beta(O(\mathcal{T}[j-1, k]), 2)$, $\mathcal{T}[j-1, k].l$ is $\beta(L(\mathcal{T}[j-1, k]), 2)$.

(2) k being odd: In this case, $\mathcal{T}[j-1, k]$ is the right child of $\mathcal{T}[j, (k-1)/2]$. Then, $L(\mathcal{T}[j-1, k]) = L(\mathcal{T}[j, (k-1)/2])$. Thus, we can simply set $\mathcal{T}[j-1, k].l = \mathcal{T}[j, (k-1)/2].l$.

(3) k being even: In this case, $L(\mathcal{T}[j-1, k]) = L(\mathcal{T}[j, k/2-1]) \cup O(\mathcal{T}[j-1, k])$. We set $\mathcal{T}[j-1, k].l = \mathtt{dom}_2(\mathcal{T}[j, k/2-1].l, \mathcal{T}[j-1, k].u)$. This satisfies the three conditions in a way similar to the two cases in Lemma 3. We omit the details for conciseness. □

We generalize the results to later iterations for $m \geq 0$.

Lemma 5. *In Algorithm 2, after the m-th iteration (Lines 3 to 17), for each leaf node N, $N.l = \beta(L(N), m+2)$.*

Proof. The lemma holds when $m = 0$. Suppose that the lemma holds when $m = \alpha$. We prove that it also holds when $m = \alpha + 1$.

Let $N(c)$ be the leaf node where $N.o = c$. After the α-th iteration, for any cell c that is $(\alpha+3)$-dominated, $N(c).l$ is the cell that $(\alpha+2)$-dominates c and is not dominated by any other cell. Further, $N(c).l[\alpha+2] = c[\alpha+2]$. Then, we know that, if $N(c).l$ is not dominated by other cells, then $N(c).l$ $(\alpha+3)$-dominates c. If $N(c).l$ is $(\alpha+3)$-dominated by another cell c', i.e., $c' \precsim_{\alpha+3} N(c).l$, after the α-th iteration, $N(c_\mu).l$ is now c', where $c_\mu = \mu_{\alpha+2}(c', N(c).l)$ (see Lemma 2 for definition of μ). This can be proven as c' satisfies the three conditions. Under rotation $\alpha+1$, c_μ is positioned before c. Then, after the $(\alpha+1)$-th iteration, $N(c).l$ will be replaced at $N(c_\mu).l$, which is c'. Thus, $N.l = \beta(L(N), \alpha+3) = \beta(L(N), m+2)$. □

Now we show the algorithm correctness with Theorem 1.

Theorem 1. *At the end of Algorithm 2, $\mathcal{T}[0, k].o$ is a candidate cell if and only if $\mathcal{T}[0, k].l \preceq \mathcal{T}[0, k].o$, and it is a key cell if and only if $\mathcal{T}[0, k].l = \mathcal{T}[0, k].o$.*

Proof. By Lemma 5, at the end of the algorithm, for each leaf node N, $N.l$ is $\beta(L(N), d)$, which is the cell that dominates or partially dominates cell $N.o$ or is $N.o$. When a cell is partially dominated but not dominated by others, it is a candidate cell. When a cell is neither dominated nor partially dominated, it is a key cell. □

5 COST ANALYSIS

A core step in our algorithm is to compute the candidate cells of a ρ-layer grid structure. We derive a bound on the number of candidate cells for each layer in Section 5.1, based on which we derive our algorithm costs in Section 5.2.

5.1 Bounding the Number of Candidate Cells

We first define two supporting concepts: *auxiliary key cells* and *auxiliary candidate cells*. We then show a one-on-one mapping between the auxiliary candidate cells and the candidate cells. Finally, we derive a bound on the number of candidate cells through counting the number of auxiliary candidate cells.

Auxiliary key cells. To ensure no false dismissals of skyline points, the candidate cells must cover the data space in each dimension. To derive the number of candidate cells, we use a set of auxiliary candidate cells that covers each dimension. The number of such cells can be derived easily, and we can establish an one-on-one mapping between them and the candidate cells. We use auxiliary key cells to simplify the description of auxiliary candidate cells.

To define auxiliary key cells, we first add d *auxiliary points* into the d-dimensional dataset $\mathcal{P}$. The i-th auxiliary point, λ_i, satisfies $\lambda_i[i] = \underline{1}$ and $\lambda_i[j] = \underline{0}$ for all $0 \leq j < d, j \neq i$. If $d = 3$, then $\lambda_0 = (\underline{0}, \underline{0}, \underline{1})$, $\lambda_1 = (\underline{0}, \underline{1}, \underline{0})$, and $\lambda_2 = (\underline{1}, \underline{0}, \underline{0})$. Here, $[i]$ counts from 0 and from right to left, and $\underline{1}$ (and $\underline{0}$) denotes a number infinitely close but *less than* 1 (and 0).

The auxiliary points are skyline points. However, they will not impact the skyline points of $\mathcal{P}$. This is because our data points fall in $[0, 1)^d$. The auxiliary points will not dominate or be dominated by any point in $[0, 1)^d$ (including the origin), as they have coordinate $\underline{1}$ in some dimension and coordinate $\underline{0}$ in all other dimensions.

The auxiliary points create d additional key cells outside each grid layer. They are the *auxiliary key cells*, e.g., the light-gray cells outside each layer in Fig. 1 (Λ_0 and Λ_1 for Layer 4).

Auxiliary candidate cells. The cells partially dominated by any auxiliary key cell are the *auxiliary candidate cells.*

Definition 7. *(Auxiliary candidate cell) The set of* auxiliary candidate cells *of the i-th layer, denoted by $\mathcal{AC}_i$, is formed by the cells partially dominated by some auxiliary key cell.*

In Fig. 1, the cells labeled by "○" are the auxiliary candidate cells of Layer 4. Such cells occupy the "top" column in each dimension.

Lemma 6. *In the layer-i grid of a d-dimensional space,*

$$\mathcal{AC}_i = \left\{ c \in C_i \,\middle|\, c[j] = 2^i - 1, j \in [0, d) \right\} \tag{7}$$

Proof. Let Λ_j be the auxiliary key cell corresponding to auxiliary point λ_j with coordinate $\underline{1}$ in dimension j and $\underline{0}$ in the other dimensions. By Equation 1, $\Lambda_j[j] = 2^i - 1$ and $\Lambda_j[k] = -1, \forall k \neq j, 0 \leq k < d$. In Fig. 1, the bottom-right light-gray cell of Layer 4 is $\Lambda_0 = C_4[-1, 15]$. By Definition 3, any cell $c \neq \Lambda_j$ and $c[j] = \Lambda_j[j]$ is partially dominated by Λ_j, i.e., $\mathcal{PDC}(\Lambda_j) = \left\{c \,\middle|\, c[j] = 2^i - 1\right\}$. In Fig. 1, the cells in the column of Λ_0 form $\mathcal{PDC}(\Lambda_j)$. Combining $\mathcal{PDC}(\Lambda_j)$ for all $j \in [0, d)$, we obtain Equation 7. □

Mapping between auxiliary candidate cells and candidate cells. We show a one-on-one mapping between the auxiliary candidate cells and the candidate cells in a grid layer. This help yield the number of candidate cells in each layer, as the number of auxiliary candidate cells can be derived from Equation 7.

Theorem 2. *In Layer i, there is a bijection between the set of candidate cells CC_i and the set of auxiliary candidate cells $\mathcal{AC}_i$.*

Proof. We omit the proof due to space limit. This will be released as part of a technical report. □

Number of candidate cells. Given Lemma 6 and Theorem 2, we compute the number of candidate cells, which is a function of i and d, and is independent of the dataset size.

Corollary 1. *In a d-dimension space, the number of candidate cells in Layer i, denoted by $|CC_i|$, satisfies:*

$$|CC_i| = \sum_{j=0}^{d-1} (2^i - 1)^j \cdot 2^{i(d-1-j)} \tag{8}$$

Proof. We omit the proof due to space limit. This will be released as part of a technical report. □

In Fig. 1, the numbers of candidate cells in Layers 0 to 4 ($d = 2$) are 1, 3, 7, 15 and 31, which conform to the corollary.

The candidate cells in different layers further satisfy the following corollary.

Corollary 2. *Given $i > j$, the volume (or area if $d = 2$) covered by the cells in CC_i must be smaller than that by the cells in CC_j.*

Proof. Intuitively, this is because candidate cells of a higher layer are all covered by those of a lower layer (cf. Fig. 1).

Recall that the number of candidate cells in Layer i is $\sum_{k=0}^{d-1}(2^i - 1)^k \cdot 2^{i(d-1-k)}$. This is the sum of a geometric sequence, which adds up to $2^{i \cdot d} - (2^i - 1)^d$. In this layer, the data space is partitioned into $2^{i \cdot d}$ cells, where each cell has volume (or area) $1/2^{i \cdot d}$. Thus, the candidate cells in CC_i cover a volume (or area) of $\mathcal{V}_i = \left(2^{i \cdot d} - (2^i - 1)^d\right)/2^{i \cdot d} = 1 - (2^i - 1)^d / 2^{i \cdot d}$. Similarly, we can write out the volume (or area) $\mathcal{V}_j$ covered by the cells in CC_j (by replacing every i with j). By basic arithmetic, we can show $\mathcal{V}_i - \mathcal{V}_j < 0$. Thus, the volume covered by the cells in CC_i is smaller than that by CC_j. We omit the detailed calculation for conciseness. □

5.2 Algorithm Costs

We show both the time and space costs of our algorithm.

Time costs. For SkyCell (Algorithm 1), sorting n points takes $O(n \log n)$ time. Constructing grid layers C_ρ to C_0 takes $O(2^{\rho \cdot d})$ time, where $2^{\rho \cdot d}$ is the number of cells in C_ρ.

In Algorithm 2, trees $\mathcal{T}_u$ and $\mathcal{T}_l$ are updated $d - 1$ times, each taking a logarithmic time to the number of candidate cells. Therefore, the algorithm time complexity is:

$$\begin{aligned} &O\left((d-1)\sum_{i=0}^{\rho-1} \log \sum_{j=0}^{d-1}(2^i - 1)^j 2^{i(d-1-j)}\right) \\ &= O\left(\rho \cdot d \cdot \log 2^{\rho \cdot d}\right) = O(\rho^2 \cdot d^2) \end{aligned} \tag{9}$$

Since there are only a few points (mostly skyline points) in each candidate cell in CC_ρ, and the cells can be processed in parallel, `RefineSkyline` has roughly a quadratic time to the number of points in each cell. Each cell is expected to contain $n/2^{\rho \cdot d}$ points, and `RefineSkyline` takes $O(n^2/2^{2\rho \cdot d})$ time.

Overall, when $\rho = (\log n)/d$ (i.e., the maximum ρ value given n uniformly distributed points), the time complexity of our algorithms are $O\left(\log^2 n\right)$.

Space costs. Our grid take $O(2^{\rho \cdot d})$ space. `ShrinkKeyCells` stores $\sum_{i=1}^{\log |CC_\rho|} |CC_\rho|/2^i$ cells for the tree, where

$$|CC_\rho| = \sum_{j=0}^{d-1} (2^\rho - 1)^j 2^{\rho(d-1-j)},$$

yielding an $O(\rho \cdot d \cdot 2^{\rho \cdot d})$ cost. When $\rho = (\log n)/d$, the space complexity of our parallel algorithm is $O(n \log n)$.

For comparison, the state-of-the-art skyline algorithm [20] takes $O(n^4 \log n)$ time and $O(n^{2d+1})$ space. Its computation is mostly spent on constructing an index to support dynamic skyline queries. In contrast, we can answer dynamic skyline queries by simply changing the origin point of the queries with little additional cost.

6 EXPERIMENTAL EVALUATION

We compare with three state-of-the-art algorithms, **Skyline Diagram (SD)** [20], **Hybrid** [7] and **SkyAlign** [3].

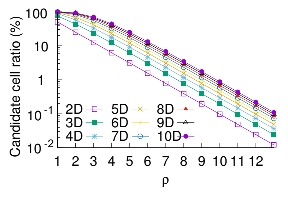
6.1 Settings

We implement all algorithms with C++ and CUDA 11.6 (source code will be released on GitHub). We use a 64-bit machine with 32 GB memory, a 2.1 GHz Intel Xeon Silver 4110 CPU (8 cores), and an Nvidia Quadro RTX6000 GPU (4,608 cores and 24 GB memory).

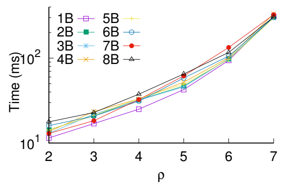
Datasets. We obtain 3.2 billion data points ($d = 2$) from OpenStreetMap [26] to form a real dataset, denoted by "**OSM**". We create subsets by random sampling for experiments on dataset cardinality. We further synthesize higher-dimensional real datasets by using randomly sampled coordinates from the first two dimensions as coordinates in the higher dimensions. We also generate synthetic data using a commonly used dataset generator [6] following previous studies [3, 7, 20]. The datasets generated include **Independent**, **Anti-correlated**, and **Correlated**, where the coordinates of a point in different dimensions are independent, anti-correlated, and correlated, respectively. We vary the data dimensionality $d \in [1, 10]$, and dataset cardinality $n \in \{1, 2, \ldots, 32\} \times 10^8$. By default, we set $d = 4$ and $n = 4 \times 10^8$. We run each experiment 10 times and report the average algorithm running times.

6.2 Results

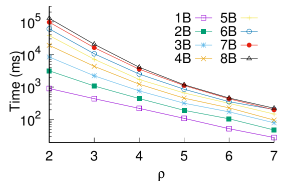
We first study the impact of the partition ratio ρ in Section 6.2.1, to help choose its value for the later experiments. Then, we study the performance of the parallel algorithms in Sections 6.2.2.



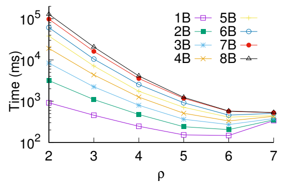
(a) The ratio of candidate cells vs. ρ



(b) Time for `ShrinkKeyCells`



(c) Time for `RefineSkyline`



(d) Overall running time

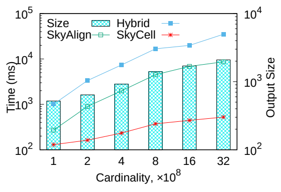
Figure 3: The impact of partition ratio ρ

6.2.1 Impact of Partition Ratio. Fig. 3a shows the ratio of Layer ρ (in our multi-layer grid) being covered by candidate cells, as computed by Corollary 1, for $\rho \in [1, 12]$ and $d \in [2, 10]$. Note that this ratio depends only on the layer number and d, and is independent from the dataset cardinality and distribution. We can see that the ratio of the space covered by the candidate cells decreases exponentially (note the logarithmic scale) with the increase of ρ. When $d = 2$, the candidate cells cover less than 1% of the space at $\rho = 7$, and this ratio further drops to 0.01% at $\rho = 12$. When $d = 10$, we still just need $\rho = 10$ so that the candidate cells only cover 1% of the data space. These results verify that our SkyCell algorithm can quickly prune a large portion of the data space (and hence the data points) from consideration with grids of only a few layers.

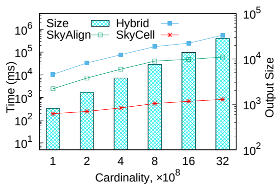
We further show in Fig. 3 the overall algorithm running time, the time for key cell shrinking, and the time for refinement (skyline point computation), as ρ varies from 2 to 7 over Independent data with 1 billion ("1B") to 8 billion ("8B") points (for parallel SkyCell and $d = 4$). As ρ increases, the time for key cell shrinking increases (Fig. 3b), while that for skyline point computation decreases (Fig. 3c), which are both expected. Their combined effect (Fig. 3d), is an optimal overall running time at $\rho = 6$. Also, as n increases, grids with a larger resolution (i.e., larger ρ) help prune more points from further checking. Thus, the curve of 8B drops faster than that of 1B with the increase of ρ. The algorithm performance on other settings shows a similar pattern. We thus use $\rho = 6$ as the default value.

6.2.2 Performance of Parallel SkyCell. We show the comparison results of the algorithms in Figs. 4 to 6. The lines show the running times and bars show the sizes of the output skyline points.

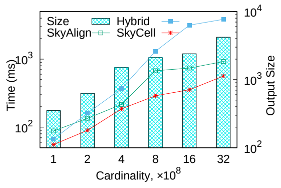
Impact of dataset cardinality n. We see that the algorithm running times increase with n (Figs. 4). Our SkyCell algorithm outperforms SD and Hybrid consistently on both synthetic and real data. Its running times are more stable (under 1,000 ms) across datasets of



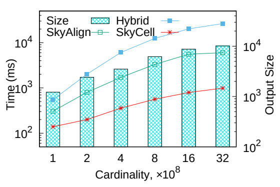
(a) Independent



(b) Anti-correlated



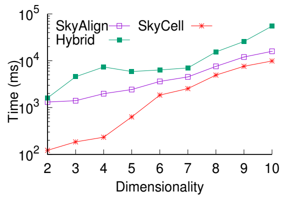
(c) Correlated



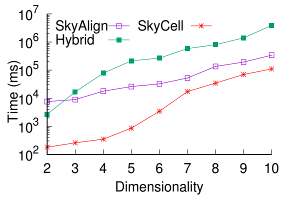
(d) OSM

Figure 4: Performance of SkyCell vs. n

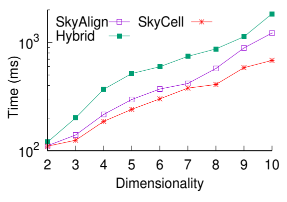
different distributions. This is because its cell-based pruning strategy is more robust against the data distribution. On Independent and Anti-correlated data, in general, SkyCell outperforms SD and Hybrid by one and two orders of magnitude (up to 60 and 700 times), respectively. On Correlated data, SD and Hybrid become closer to (but still worse than) SkyCell. There are fewer skyline points on such data (e.g., 4,203 skyline points among 32×10^8 data points), and many data points can be pruned by a skyline point, which benefit the point-based algorithms SD and Hybrid. Even in this extreme case, SkyCell runs the fastest. It computes the skyline points from 32×10^8 points in just about 0.6 seconds. On OSM, SkyCell outperforms SD and Hybrid by 6 and 27 times when $n = 32 \times 10^8$, and it again finishes in under a second. These confirm the scalability of our SkyCell algorithm.



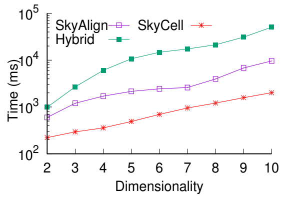
(a) Independent



(b) Anti-Correlated



(c) Correlated



(d) OSM

Figure 5: Performance of SkyCell vs. d

Impact of data dimensionality d. In Fig. 5, we vary d from 2 to 10. The algorithm running times increase with d in general, while

Hybrid has a fluctuation which is also observed in its original proposal [7]. SkyCell again is the fastest on all datasets. When $d = 10$, comparing with Hybrid, SkyCell reduces the running time by 82%, 97%, 62%, and 96% on the Independent, Anti-correlated, Correlated, and OSM data, respectively. When comparing with SkyAlign, these numbers become 38%, 67%, 44%, and 79%, respectively.

Note that we reduce the value of ρ to 5, 4, 4, 3, 3 for $d = 6, 7, 8, 9, 10$, respectively, for that the number of cells grows with d while many of the cells are empty. The decreasing ρ values slow down the growth in the number of cells and reduces the number of empty cells to be considered separately (although the number of non-empty cells still grows with d and so are the algorithm running times). The resultant grid at the lowest level has 2^{30} cells when $d = 10$. This contributes to the performance gains of SkyCell over the competitors as d grows larger.

Impact of number of threads. In Fig. 6, we test the capability of SkyCell to exploit the parallel power of GPU by running the algorithm on datasets of different cardinality and dimensionality while varying the number of threads used on the GPU from 1k to 4k. We use $\Delta(x\text{k}, y\text{k})$ to denote the time saved when running SkyCell on xk threads comparing with that on yk threads. We see that, given fixed dataset cardinality and dimensionality, the running time of SkyCell decreases significantly with the increase in the number of threads. This confirms the capability of SkyCell to take full advantage of the parallel processing power of GPU.

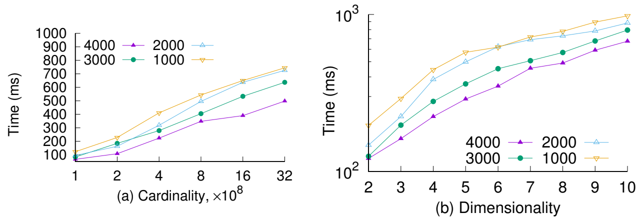


Figure 6: Performance of SkyCell vs. number of GPU threads

7 RELATED WORK

The skyline query was first studied in computational geometry and was called the *maxima* [16]. It was later introduced to the database community and was extensively studied [6, 10, 11, 14, 15, 29].

The state-of-the-art *Skyline diagram* (SD) [20] pre-computes a Voronoi-like diagram. Query points in the same cell of the diagram have the same skyline points, which are pre-computed. When processing a skyline query, SD only needs to locate the cell that encloses the query point to fetch the query answer. This algorithm is query efficient but may suffer in pre-computation and storage costs when there are many skyline points. We compare with it in our experiments.

The *balanced pivot selection* (BPS) [18] algorithm selects a pivot – the point with the smallest normalized attribute values – to split the data space into *incomparable regions*. Points in different incomparable regions do not dominate each other. Each region is further split recursively Points are assigned to regions by comparing against the pivots, and they are only checked for domination in their assigned regions. As each pivot is selected depending on previous pivot selections, one is hard to perform multiple pivot selection at the same time. Hence BPS does not fit in parallel scenarios.

We next focus on parallel skyline algorithms [1, 5, 13, 30, 35, 37] as they are more relevant. The *GPU-based Nested Loop* (GNL) [8] algorithm is a parallel extension of the block-nested-loop sequential algorithm [6]. It assigns a thread for each point and checks the point with all other points in parallel. *GPGPU Skyline* (GGS) [2] sorts the points by the Manhattan norm. It then runs domination checks in multiple iterations. In each iteration, GGS uses the top-ranked unchecked points as the skyline buffer and compares them against the other points in parallel.

SkyAlign [3] is a GPU-based algorithm that uses a global, static partitioning scheme. It uses controlled branching to exploit transitive relationships between points and can avoid some point domination checks. It does *not* use region-based domination checks, and it has a fixed number of partitions regardless of the dataset size, which cannot make full use of the GPU throughput and may cause branch divergence of GPU warps. *Spatial-GPU* [30] uses multi-level *independent regions* to filter candidate points. In different independent regions, skyline queries are evaluated in parallel. Thus, this method can be run in parallel with GPU or MapReduce. However, this method is designed for a special type of skyline queries called the *spatial skyline*. Its solution cannot be easily extended to general skyline queries.

New skyline variants are emerging. For example, Yang et al. [31] study G-Skyline, which returns a group of points based on *group dominance*. Mouratidis et al. [23] combine skyline with top-k queries, to provide personalized, output size controllable, and preference specification flexible results. It would be interesting future work to adapt our techniques for these problems. A few other studies use MapReduce [24, 28, 33]. They focus on workload balancing among the worker machines.

The main difference between the studies above and ours is that they focus on point domination checks, while we partition the space and check domination between the partitions, thus yielding significantly fewer domination checks and higher efficiency. Note that cell domination in our work checks the relationships between cells, i.e., we use cells to prune cells in each layer. In comparison, existing works [19, 27] use points to prune regions. Our cell domination is more efficient as the number of cells is much smaller than the number of points, and cell domination checks suit parallel processing as they do not overlap.

ACKNOWLEDGEMENT

This work is supported by the National Nature Science Foundation of China (61872071, 62072083), the Fundamental Research Funds of the Central Universities (N2216017, N2116010), and the CCF-Huawei Innovation Research Plan.

8 CONCLUSIONS

We studied skyline queries and proposed a grid structure that enables grid cell domination computation for search space pruning. We showed that only a small constant number of cells need to be examined, which is independent of the dataset cardinality, yielding highly efficient skyline computation. Our structure also enables parallel computation. We thus proposed a parallel skyline algorithm, taking advantage of the parallelization power of GPUs. Our cost analysis and experiments confirm the efficiency of the proposed algorithms. Our parallel algorithm outperforms state-of-the-art skyline algorithms consistently and by up to over two orders of magnitude in the algorithm response time.

REFERENCES

[1] Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. 2004. Efficient distributed skylining for web information systems. In *EDBT*. 256–273.

[2] Kenneth S. Bøgh, Ira Assent, and Matteo Magnani. 2013. Efficient GPU-based skyline computation. In *International Workshop on Data Management on New Hardware*. 1–6.

[3] Kenneth S. Bøgh, Sean Chester, and Ira Assent. 2015. Work-efficient parallel skyline computation for the GPU. *Proceedings of the VLDB Endowment* 8, 9 (2015), 962–973.

[4] Kenneth S. Bøgh, Sean Chester, and Ira Assent. 2016. SkyAlign: A portable, work-efficient skyline algorithm for multicore and GPU architectures. *The VLDB Journal* 25, 6 (2016), 817–841.

[5] Kenneth S. Bøgh, Sean Chester, Darius Šidlauskas, and Ira Assent. 2017. Template skycube algorithms for heterogeneous parallelism on multicore and GPU architectures. In *SIGMOD*. 447–462.

[6] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. 2001. The skyline operator. In *ICDE*. 421–430.

[7] Sean Chester, Darius Šidlauskas, Ira Assent, and Kenneth S. Bøgh. 2015. Scalable parallelization of skyline computation for multi-core processors. In *ICDE*. 1083–1094.

[8] Wonik Choi, Ling Liu, and Boseon Yu. 2012. Multi-criteria decision making with skyline computation. In *IEEE International Conference on Information Reuse & Integration*. 316–323.

[9] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. 2003. Skyline with presorting. In *ICDE*. 717–719.

[10] Katja Hose and Akrivi Vlachou. 2012. A survey of skyline processing in highly distributed environments. *The VLDB Journal* 21, 3 (2012), 359–384.

[11] Zhiyong Huang, Christian S. Jensen, Hua Lu, and Beng Chin Ooi. 2006. Skyline queries against mobile lightweight devices in MANETs. In *ICDE*. 66–66.

[12] Md. Saiful Islam, Wenny Rahayu, Chengfei Liu, Tarique Anwar, and Bela Stantic. 2017. Computing influence of a product through uncertain reverse skyline. In *SSDBM*. 1–12.

[13] Henning Köhler, Jing Yang, and Xiaofang Zhou. 2011. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*. 85–96.

[14] Donald Kossmann, Frank Ramsak, and Steffen Rost. 2002. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*. 275–286.

[15] R. D. Kulkarni and B. F. Momin. 2019. Skyline computation for big data. In *Data Science and Big Data Analytics*. 267–276.

[16] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P. Preparata. 1975. On finding the maxima of a set of vectors. *J. ACM* 22, 4 (1975), 469–476.

[17] Guanling Lee and Ying-Hao Lee. 2017. An efficient method of computing the k-dominant skyline efficiently by partition value. In *International Conference on Information Management*. 416–420.

[18] Jongwuk Lee and Seung-Won Hwang. 2014. Scalable skyline computation using a balanced pivot selection technique. *Information Systems* 39 (2014), 1–21.

[19] Ken C. K. Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. 2007. Approaching the skyline in Z order. In *VLDB*. 279–290.

[20] Jinfei Liu, Juncheng Yang, Li Xiong, Jian Pei, and Jun Luo. 2018. Skyline diagram: Finding the Voronoi counterpart for skyline queries. In *ICDE*. 653–664.

[21] Zekri Lougmiri. 2017. A new progressive method for computing skyline queries. *Journal of Information Technology Research* 10, 3 (2017), 1–21.

[22] Robert B. Miller. 1968. Response time in man-computer conversational transactions. In *The December 9-11, 1968, Fall Joint Computer Conference, Part I*. 267–277.

[23] Kyriakos Mouratidis, Keming Li, and Bo Tang. 2021. Marrying top-k with skyline queries: Relaxing the preference input while producing output of controllable size. In *SIGMOD*. 1317–1330.

[24] Kasper Mullesgaard, Jens Laurits Pederseny, Hua Lu, and Yongluan Zhou. 2014. Efficient skyline computation in MapReduce. In *EDBT*. 37–48.

[25] Aziz Nasridinov, Jong-Hyeok Choi, and Young-Ho Park. 2017. A two-phase data space partitioning for efficient skyline computation. *Cluster Computing* 20, 4 (2017), 3617–3628.

[26] OpenStreetMap. 2021. https://www.openstreetmap.org/

[27] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *ACM Transactions on Database Systems* 30, 1 (2005), 41–82.

[28] Yoonjae Park, Jun-Ki Min, and Kyuseok Shim. 2013. Parallel computation of skyline and reverse skyline queries using MapReduce. *Proceedings of the VLDB Endowment* 6, 14 (2013), 2002–2013.

[29] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi, et al. 2001. Efficient progressive skyline computation. In *VLDB*. 301–310.

[30] Wenlu Wang, Ji Zhang, Min-Te Sun, and Wei-Shinn Ku. 2019. A scalable spatial skyline evaluation system utilizing parallel independent region groups. *The VLDB Journal* 28, 1 (2019), 73–98.

[31] Zhibang Yang, Xu Zhou, Kenli Li, Yunjun Gao, and Keqin Li. 2021. Progressive approaches to flexible group skyline queries. *Knowledge and Information Systems* 63, 6 (2021), 1471–1496.

[32] Wenhui Yu, Jinfei Liu, Jian Pei, Li Xiong, Xu Chen, and Zheng Qin. 2020. Efficient contour computation of group-based skyline. *IEEE Transactions on Knowledge and Data Engineering* 32, 7 (2020), 1317–1332.

[33] Ji Zhang, Xunfei Jiang, Wei-Shinn Ku, and Xiao Qin. 2015. Efficient parallel skyline evaluation using MapReduce. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2015), 1996–2009.

[34] Shiming Zhang, Nikos Mamoulis, and David W. Cheung. 2009. Scalable skyline computation using object-based space partitioning. In *SIGMOD*. 483–494.

[35] Haoyang Zhu, Peidong Zhu, Xiaoyong Li, Qiang Liu, and Peng Xun. 2017. Parallelization of skyline probability computation over uncertain preferences. *Concurrency and Computation: Practice and Experience* 29, 18 (2017), e4201.

[36] Vasileios Zois. 2019. *Complex query operators on modern parallel architectures*. Ph. D. Dissertation. UC Riverside.

[37] Vasileios Zois, Divya Gupta, Vassilis J Tsotras, Walid A Najjar, and Jean-Francois Roy. 2018. Massively parallel skyline computation for processing-in-memory architectures. In *International Conference on Parallel Architectures and Compilation Techniques*. 1–12.

[38] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. 2008. A novel spectral coding in a large graph database. In *EDBT*. 181–192.