Future Generation Computer Systems 141 (2023) 269–283

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

SkyFlow: Heterogeneous streaming for skyline computation using FlowGraph and SYCL

Jose Carlos Romero *, Angeles Navarro, Andrés Rodríguez, Rafael Asenjo

Department of Computer Architecture, University of Málaga, Spain

ARTICLE INFO

Article history:
Received 1 June 2022
Received in revised form 25 September 2022
Accepted 18 November 2022
Available online 24 November 2022

Keywords:
Skyline
Stream of queries
Heterogeneous computing
Integrated GPU
OneAPI
SYCL

ABSTRACT

The skyline is an optimization operator widely used for multi-criteria decision making. It allows minimizing an n-dimensional dataset into its smallest subset. In this work we present SkyFlow, the first heterogeneous CPU+GPU graph-based engine for skyline computation on a stream of data queries. Two data flow approaches, Coarse-grained and Fine-grained, have been proposed for different streaming scenarios. Coarse-grained aims to keep in parallel the computation of two queries using a hybrid solution with two state-of-the-art skyline algorithms: one optimized for CPU and another for GPU. We also propose a model to estimate at runtime the computation time of any arriving data query. This estimation is used by a heuristic to schedule the data query on the device queue in which it will finish earlier. On the other hand, Fine-grained splits one query computation between CPU and GPU. An experimental evaluation using as target architecture a heterogeneous system comprised of a multicore CPU and an integrated GPU for different streaming scenarios and datasets, reveals that our heterogeneous CPU+GPU approaches always outperform previous only-CPU and only-GPU state-of-the-art implementations up to 6.86×and 5.19×, respectively, and they fall below 6% of ideal peak performance at most. We also evaluate Coarse-grained vs Fine-Grained finding that each approach is better suited to different streaming scenarios.

1. Introduction

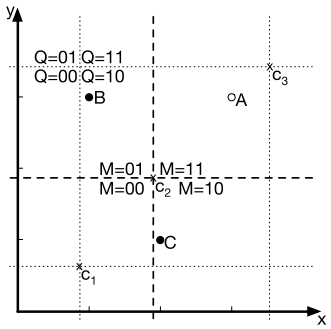
The skyline, initially introduced in [1], is an optimization operator widely used for multi-criteria decision making. It allows to minimize a n-dimensional dataset into the smallest subset, usually using as a reduction metric the *minimum* value for each dimension.

Fig. 1 shows a toy example of a dataset and its corresponding skyline. The skyline is the subset of points that are not eliminated (or not dominated) by any other point in the dataset. Point *C* has lower values in its two dimensions with respect to point *A*, thus point A is eliminated from the skyline by point *C*. Point *B* has lower value in *x* than point *C*, but higher in *y*, so neither can eliminate the other (they are incomparable), so both *B* and *C* end up in the skyline set. For large datasets with points of several dimensions, the computation of the skyline becomes a computationally expensive task.

In order to increase the skyline performance, it is key to avoid the all-to-all comparison between points. To that end, two approaches are usually adopted: (1) sorting-based or (2) partitioning-based. The main disadvantage of sorting-based algorithms is that for high dimensional skylines, the methodology generates a large candidate buffer, causing performance degradation due to brute-force quadratic search. State-of-the-art sequential algorithms use recursive, point-based partitioning approaches. The current state-of-the-art multicore algorithm, *Hybrid* [2], is a point-based method that dynamically constructs a quad-tree with the skyline points. One optimization of this algorithm is that it flattens the tree into an array structure for better access patterns, and also it processes points in blocks (tiles) to improve parallelism. However, point-based strategies are not well suited to heterogeneous architectures. For instance, in [2] the tree is constructed on the fly, incrementally, and sequentially, so frequent synchronization points are necessary to accommodate the sequential insert phase, a strategy that adversely affects performance on the GPU. Moreover, the uncontrolled branching in the tree traversal tends to serialize execution within each warp on account of branch divergence. On the other hand, the current state-of-the-art algorithm for GPU architectures, *SkyAlign* [3], initially constructs a statically-defined quad-tree, being the key algorithmic idea that points are physically sorted by grid cells and statically partitioned, and threads are mapped onto that sorted layout. Additionally, the actual computation is loosely ordered with *d* carefully placed synchronization points (being *d* the number of dimensions). This type of order simultaneously achieves good spatial locality, homogeneity within warps, and

* Corresponding author.
E-mail addresses: jromero@ac.uma.es (J.C. Romero), angeles@ac.uma.es (A. Navarro), andres@ac.uma.es (A. Rodríguez), asenjo@uma.es (R. Asenjo).

https://doi.org/10.1016/j.future.2022.11.021



Point	x	y
A	3	3
B	1	3
C	2	1

Fig. 1. Dataset and skyline example. Skyline points are B and C. Pivot points, c_i, and mask values, M and Q, are discussed in Section 3.

independence among threads, in particular when the number of dimensions (d) is high. This strategy creates more predictable tree traversals that minimize branch divergence. As a novelty in our paper, we design a new implementation of the *SkyAlign* algorithm based on SYCL [4]. SYCL is a programming standard that advocates for the single source code approach, which enables to target multiple devices using the same programming model in order to have cleaner, portable, and more easy to maintain applications. That way, SYCL allows using the same algorithm and source code both on the GPU and the CPU (contrary to CUDA), obtaining reasonable performance on both devices. This flexibility and portability of SYCL is evaluated in a recent work [5] where the Rodinia benchmarks were ported to SYCL using CUDA as backend and compared with the native CUDA implementations. Authors also compared SYCL on CPU with OpenMP versions of the benchmarks. In particular, in our implementations we leverage oneAPI [6] that is a promising framework that simplifies programming heterogeneous architectures by providing several libraries and the DPC++ (Data Parallel C++) [7] SYCL compiler.

The previously mentioned skyline algorithms are designed to compute skylines over static datasets rather than dynamic ones that occur in data streaming environments. In the context of analytic applications that process multi-source data streaming queries, or applications in data exploration and multi-criteria decision making that project the multi-dimensional data into different subset of the attributes (i.e., some subspaces of interest) [8,9], we find that the data stream is provided with diverse independent queries for each of which the computation of the skyline operator is required. Examples include the development of smart technologies in the context of the rapid advancement in the Internet of Things (IoT) ecosystem. In this paper we tackle the problem of computing the skyline operator over a stream of independent data queries using as target architecture a heterogeneous system comprised of a multicore CPU and an integrated GPU, which to the best of our knowledge it has not yet been addressed. In our work, we propose a heterogeneous graph-based engine, called SkyFlow based on the FlowGraph feature provided by the oneAPI Threading Building Blocks library [10], which enables to exploit graph parallelism in streaming scenarios by efficiently scheduling the data queries computations among the devices while ensuring near-optimal throughput. Our proposal adapts to different streaming scenarios using two heterogeneous approaches: Coarse-grained (SkyFlow-CG) and Fine-grained (SkyFlow-FG).

SkyFlow-CG computes concurrently one query per device. In this work, we experimentally validate the performance of our SYCL implementations, both on the GPU and the CPU finding that although the SYCL code is portable, it is not "performance portable" because it performs better on the GPU than on the CPU. In fact, as we will see in Section 3.4, for our platform the original OpenMP-based *Hybrid* implementation is faster than the SYCL-based *SkyAling* on the CPU. Also, during our research, we found that specific datasets perform better under *Hybrid* on the CPU than under *SkyAlign* on the integrated GPU, or vice versa, depending on the distribution of points in the dataset and its spatial structure, size, or number of dimensions. Moreover, we also compared the performance of our SYCL-based *SkyAlign* implementation on a discrete GPU against the OpenMP-based *Hybrid* running on the CPU obtaining similar conclusions. Thus, our SkyFlow-CG proposal adopts a hybrid strategy: each device will run the algorithm best suited to the specific features of the corresponding device, it is, *Hybrid* on the CPU (implemented with OpenMP) and *SkyAlign* on the GPU (implemented with SYCL). As we aim at optimizing system performance and resource utilization in the context of a stream of independent data queries, we must devise a scheduling strategy that at runtime is able to consider the arriving data query characteristics and the occupancy of the resources to dispatch the skyline computation to the appropriate device. In this paper we propose different scheduling strategies and evaluate and discuss their performance and optimality for different streaming scenarios.

On the other hand, SkyFlow-FG represents a heterogeneous CPU+GPU solution for the skyline computation in which each single dataset query is split between the CPU and the GPU devices. For it, we start from the SYCL-based *SkyAling* implementation that runs both on the CPU and GPU. The main challenge now is to find the optimal dataset partition for each arriving data query at runtime. We also evaluate different partitioning strategies for different streaming scenarios.

In our experimental evaluations we find that our heterogeneous approaches always outperform baselines implementations that only use one device. In fact, they outperform only-GPU and only-CPU baselines up to 5.19x and 6.86x, respectively. These results tell us that exploiting both devices with our heterogeneous solutions is usually more profitable than using just one device. We will also discuss under which streaming scenarios is advantageous to use SkyFlow-CG, and in which ones SkyFlow-FG.

Summarizing, the main contributions of this paper are:

- We contribute with a novel SYCL-based implementation of the *SkyAlign* algorithm and evaluate its performance both on an integrated and a discrete GPU, and on CPU.
- We design a graph-based engine, SkyFlow based on oneTBB (provided within oneAPI), and propose two heterogeneous approaches for skyline computation over a stream of data queries: SkyFlow-CG (Coarse-grained) and SkyFlow-FG (Fine-grained). Coarse-grained keeps two skyline computation in parallel, one per device, while in Fine-grained a single skyline computation is split between the CPU and GPU devices. We validate the suitability of each approach for different streaming scenarios.
- We present two policies for scheduling the skyline computation of arriving data queries between devices in the Coarse-grained approach, where each device has a queue. The first strategy (Work Conserving) keeps the devices busy by offloading queries to the shortest queue. The second approach (Heterogeneous Earliest Finish Time) estimates the execution time for the arriving query on each device. To such end, we develop a model that, taking small chunks of points at runtime, estimates the execution time of an arriving query with negligible overhead. That estimated time is used to enqueue the incoming query on the device queue in which it will finish earlier.

This paper is organized as follows: Section 2 briefly describes the related work regarding skyline computation. Section 3 introduces the required background and the skyline algorithms. Section 4 presents our heterogeneous approaches for computing the skyline over a stream of data queries. Section 5 describes the schedulers and partitioning strategies devised to optimize the heterogeneous solutions. Section 6 discusses the experimental results, ending with some conclusions and future work in Section 7.

2. Related work

The skyline operator is an optimization problem widely used for multi-criteria decision making. It has been applied in the context of privacy-preserving skyline computation framework across multiple domains [11], the processing of skyline query over encrypted data in Cloud-enabled databases [12], an optimization of Quality-of-Services-aware big service processes with discovery of skyline services [13,14], or a resilient drone service composition framework for delivery in dynamic weather conditions [15]. Also, generalizations of skyline computation have been proposed in [8], where they work with all the $2^d - 1$ skyline query subdomains combinations from a dataset of d dimensions.

Initial implementations of the skyline operator [1] were based on the divide-and-conquer approach. Early improvements to the algorithm came in the form of tree-based indexing methods, B-trees [16] and R-trees [17]. In general, the optimizations of this algorithm focus on reducing the number of operations that typically are comparisons to assess if a point is in the skyline or not. For it, two main strategies are adopted: sorting and space partitioning. Sorting-based algorithms [18–20] introduce a data precomputation phase before entering the main loop. This precomputation phase facilitates both the elimination of points that do not belong to the skyline and the reduction of comparisons between points. Researchers have used different sorting strategies, such as the Manhattan norm [18], z-order [19] or the minimum of each dimension as a sorting attribute [20]. On the other hand, partitioning-based algorithms [21–23] divide the data space into regions, so that they avoid point comparisons between entire regions, which reduces the number of operations. These strategies typically are based on recursive methods such as pivot point-based partitioning [21], being *BSkyTree* [22] the current state-of-the-art for sequential skyline computation. The first multicore CPU parallel approach [23] partitioned the data space into blocks. *Hybrid* [2], the state-of-the-art in multicore algorithms, proposes a hybrid strategy: it applies sorting followed by partitioning with no recursion. This algorithm builds dynamically a two-level quad-tree in tiled batches to support multi-threading.

Algorithms optimized for GPU avoid synchronizations to achieve very high compute throughput. The first algorithm for skyline computation on GPU, *GNL* [24], assigns a point in the dataset to each thread without further optimization. *GGS* [25] improves on this implementation by adding a preprocessing stage that sorts the points according to the Manhattan distance. *SkyAlign* [3], which represents the current state-of-the-art for GPU, outperforms previous works by building a statically-defined quad tree. It also uses medians and quartiles of each dimension to construct virtual pivot points, which are defined globally to create more predictable tree traversals that minimize branch divergence. By contrast, *Hybrid* only uses medians. Authors in [26] propose an alternative to *SkyAlign* for high dimensional datasets and skylines variations with more relaxed rules for pruning points. In any case, in our work we want to focus on a general approach for computing the skyline for arriving data queries with any number of dimensions. In addition, the experimental validation of *SkyAlign* in [3] demonstrates that the parallel scalability of this algorithm is preserved when increasing the number of computing units, what makes it a good candidate for heterogeneous implementations, one of the goals in this work, so we keep *SkyAlign* as our algorithm of reference on the GPU. However, as we will show in the next section, on our platform and for specific datasets, running *Hybrid* on the multicore CPU outperforms *SkyAlign* on the GPU. Thus, *Hybrid* is still considered for the CPU in some of our heterogeneous proposals.

In any case, the second goal of this paper has to do with providing support for the skyline computation over a stream of independent data queries. There has been previous research on incrementally computing the skyline for a data query for which data points arrive over time in streams [27–33]. Typically, the output is a sequence or incremental update of skyline computations. Sequential solutions for continuous data streams maintain a sliding window of the most recent points [27–29]. Researchers in [30] propose parallel implementations of a previous proposal in [28]. Authors in [31,32] present parallel solutions on distributed systems for the sliding window approach. Nevertheless, these approaches focus on considering a stream of dependent sets of points to process. They need to keep and update a global skyline over time preserving the processed historical data in order to compare them with the new arriving points. Our work, however, considers as input a stream of independent datasets of points, producing as an output a stream of independent skylines, one per input received.

Parallel implementations of skyline computation over a stream of data queries targeting heterogeneous architectures are still an open research issue. We focus on heterogeneous architectures comprised of a multicore CPU and an integrated GPU. In order to improve performance productivity for this type of architectures, new programming environments such as oneAPI [6] and programming standards as SYCL [4] have been proposed and we consider them for the first time to solve the problem at hand. However, these frameworks do not solve the problem of the automatic partition of the workload and scheduling of tasks among devices, issues that we address in this paper. Thus, to the best of our knowledge, the new heterogeneous proposals introduced here represent novel contributions in this context.

3. Theoretical background

3.1. Definitions

Let us compute the skyline corresponding to the dataset (or data query) S of n points, p_i, with d dimensions. The value of p_i in a given dimension δ is known as $p_i[\delta]$.

Definition 1 (*Dominance*)**.** A point p dominates another q, $p \prec q$, if the following condition is satisfied: $\forall i \in [0, d-1] : p[i] \leq q[i]$ and $\exists j \in [0, d-1] : p[j] < q[j]$; that is, for every dimension, p is less than or equal to q and there exists at least one dimension in which p is strictly less than q. This operation is called dominance test (*DT*).

Definition 2 (*Skyline*)**.** A skyline can thus be defined as the subset of points in a dataset S that are not dominated by any other point in the dataset, i.e., $SKY(S) = \{p \in S \mid \nexists q \in S : q \prec p\}$.

Definition 3 (*Incomparability*)**.** Two points $p, q \in S$, are incomparable, $p \sim q$, if $p \not\prec q$ and $q \not\prec p$, that is, if p and q do not dominate each other.

The smaller the number of *DT*s, the better the work-efficiency and the faster the skyline computation. In the worst case, for a dataset of size n, the algorithm will have quadratic cost with

$n{\cdot}(n-1)/2$ *DT*s. Avoiding *DT*s resulting in incomparability reduces the computational cost of the problem. Sorting and partitioning based algorithms work towards this goal. In sorting-based algorithms, traversing the sorted points reduces the number of *DT* operations. In partitioning-based, a partitioning of the space avoids *DT* between points that are known to be incomparable because their corresponding partitions are also incomparable. Fig. 1 shows a partitioning-based example, where a *DT* between points *A* and *C* is required because region $M = 11$ and region $M = 10$ are comparable. On the other hand, the *DT* between points *B* and *C* is unnecessary since *B* is in region $M = 01$ that dominates in *X*-axis, but *C* is in region $M = 10$ that dominates in *Y*-axis, resulting in the incomparability of all the points in both regions.

As introduced in Section 2, this work is based on two skyline algorithms: (1) *Hybrid* [2] initially implemented in OpenMP [34] is the state-of-the-art for multicore CPU architectures. We will refer to it as *OpenMP-CPU*. And (2) *SkyAlign* [3] initially developed in CUDA [35] is the state-of-the-art for GPU architectures, having considerable potential for scalability and a heterogeneous implementation. We have ported the implementation to SYCL to later exploit flexibility and portability of this programming language, so from now on, we will refer to it as *SYCL-GPU*. Next, we briefly introduce both algorithms and compare their performance.

3.2. OpenMP-CPU algorithm

Alg. 1 sketches the OpenMP-CPU algorithm, which combines two techniques to save *DT*s: sorting and partitioning.

Algorithm 1: OpenMP-CPU algorithm

```
   Input: S=Dataset of n points p and d dimensions.
   Output: Skyline of S: SKY(S)
1  SKY(S) ← ∅
2  Prefilter, partition and sort S
3  while S ≠ ∅ do
4  |  Q ← next α points of S
5  |  S ← S\Q
6  |  foreach i ∈ [0, Q.size) (in parallel) do
7  |  |  if ∃p ∈ SKY(S) : p ≺ Q[i] then
8  |  |  |  Mark Q[i] as dominated
9  |  Remove dominated Q[i] from Q
10 |  foreach i ∈ [0, Q.size) (in parallel) do
11 |  |  if ∃j ∈ [0, i) : Q[j] ≺ Q[i] then
12 |  |  |  Mark Q[i] as dominated
13 |  Remove dominated Q[i] from Q
14 |  Append Q to SKY(S)
15 return SKY(S)
```

The algorithm is divided into three blocks. The first one is represented by the line 2 that carries out a prefiltering, partitioning and sorting steps that we describe next. (1) **Prefiltering** performs a fast parallel comparison of points in chunks according to their Manhattan norm (L1), easily pruning dominated points; (2) **Partitioning** does the partition of the multi-dimensional space based on a pivot point, p_v. The p_v is the point with the median value of L1, which must necessarily be a skyline point. A mask, *m*, is assigned to each point for each dimension such that, $m[i] = (p[i] < p_v[i] ? 0 : 1)$.[1] p_v divides the dataset into 2^d regions so that the binary mask of each point identifies its corresponding region. Thus, binary mask comparisons can be used to reduce the number of the more expensive *DT* operations. (3) **Sorting** is carried out according to the binary mask and the L1 norm. This sorting maintains the property that $p \not\prec q$ if *p* precedes *q* in the sort order. The sorting stage ensures that: (1) once a point is appended to the $SKY(S)$ it will not leave this set since no subsequent point in *S* will dominate it; and (2) points that are likely pruning others are processed earlier (which remove the *DT* operations corresponding to these early pruned points). See more details in [2].

After this preprocessing, the resulting dataset, *S*, is traversed in blocks of α points[2] (line 4). Each block, *Q*, is processed by two consecutive parallel loops. A first parallel stage (lines 6–8) compares each point *p* of the block with the points of the global skyline known so far, to check if any of them dominates *p*. After the parallel stage, a synchronization stage sequentially eliminates dominated points from the iteration space (line 9). The second parallel stage (lines 10–12), compares the surviving points among them. The second synchronization stage sequentially eliminates dominated points in line 13. The points that pass through this two sieves are added to the global skyline, $SKY(S)$, in line 14. This process is repeated for all the blocks in *S* updating the global skyline after processing each block. Although not explicitly indicated in Alg. 1, vectorization is used for the implementation of the *DT* operations (lines 7 and 11).

3.3. SYCL-GPU algorithm

The *SYCL-GPU* follows a different approach in order to avoid the synchronization stages that keep a global skyline in the *OpenMP-CPU* algorithm. On the GPU these synchronization steps have a higher impact on performance. As in the *OpenMP-CPU* alternative, similar optimizations are also considered: partitioning and sorting.

Now, the partitioning divides each dimension of the dataset in quartiles and median. This is, three global pivot points (instead of just one as in *OpenMP-CPU*) are defined for each dimension: first quartile, c_1, median, c_2, and second quartile, c_3 (see Fig. 1). All GPU threads share the information of these three defined global pivot points. It should be noted that these points are probably virtual (i.e. c_1, c_2, and c_3 may not belong to the dataset as it happens with the pivot point in the *OpenMP-CPU* algorithm). This partitioning results in binary masks to classify the dataset points in two nested levels. At the first level, a binary mask *M* is assigned to each point, corresponding to its position relative to the median (equivalent to the pivot point in the *OpenMP-CPU*). On a second level, another binary mask *Q* is assigned, corresponding to its position relative to the first or second quartile (whichever is relevant for each point). The process is repeated for each dimension. These two binary masks per point ease locating each point's partition. Fig. 1 shows an example with a dataset of 3 points partitioned according to the quartiles and median points. For example, point *B* has median mask $M_B = 01$ because in its *x* dimension it is below the median value, c_{2x}, while in the *y* dimension is above the median, c_{2y}. Its quartile mask is $Q_B = 10$ because in the *x* dimension is above the c_{1x} quartile, but below the c_{3y} quartile for the *y* dimension.

Once the points have been classified according to their region in the space (identified by both masks *M* and *Q*) we can leverage the fact that points located in incomparable regions are also incomparable and avoid the corresponding *DT*. In essence, we trade *DT*s for *MT*s (mask tests). Note that *MT*s are computationally cheaper than *DT*s since they only perform one binary operation between two integers, while a *DT* requires $2{\cdot}d$ operations (see Definition 1). The sorting step has the same advantages that we mentioned in the previous section.

Alg. 2 presents a simplified description of the GPU algorithm. For a more precise explanation we refer the reader to [3]. The

[1] C language ternary operator: $m[i]$=0 if condition holds, and 1 otherwise.

[2] See [2] for more details.

algorithm is divided in two main stages: (1) preprocessing; and (2) the main loop.

The preprocessing includes prefiltering, partitioning and sorting. Prefiltering is carried out in lines 1–2. The idea is to first find a threshold, τ, that is calculated as the minimum of the maximum value in all the dimensions of each point. For example, in the Table of Fig. 1, the maximum values are computed row wise, resulting in the vector {3, 3, 2}, from with the minimum is $\tau = 2$. In parallel, each point is compared with the threshold and if it has no value less than the threshold, that point is dominated and eliminated. For example, in Fig. 1, point *A* does not have any value smaller than 2, so it is pruned. Once this prefiltering of points finishes, the static partitioning is created (lines 3- 6), assigning the binary masks (median and quartile) to each point. Finally, the dataset is sorted in line 7.

Algorithm 2: SYCL-GPU algorithm

```
   Input: S=Dataset of n points p and d dimensions.
   Output: Skyline of S: SKY(S)
1  τ ← min_{p∈S}(max_{i∈[0,d)}(p[i]))
2  S ← {p ∈ S|∃i ∈ [0, d) : p[i] ≤ τ}
3  foreach point p_i ∈ S (in parallel) do
4  |  foreach dimension δ ∈ [0, d) do
5  |  |  M_i[δ] ← (p_i[δ] > c_{2δ})
6  |  |  Q_i[δ] ← (p_i[δ] > (M_i[δ]? c_{3δ} : c_{1δ}))
7  Sort S according to M
8  foreach levels l ∈ [0, d)] do
9  |  foreach point p_i ∈ S : |M_i| ≥ l (in parallel) do
10 |  |  foreach p_j ∈ S do
11 |  |  |  if (!M_j ∼ M_i) then
12 |  |  |  |  if (!Q_j ∼ Q_i) then
13 |  |  |  |  |  if p_j ≺ p_i then
14 |  |  |  |  |  |  Mark p_i dominated; terminate thread
15 |  Remove dominated points from S
16 |  SKY(S) ← SKY(S) ∪{p_i ∈ S : |M_i| = l}
17 return SKY(S)
```

The algorithm's main loop goes from lines 8 to 16. The outer sequential loop with iterator l (level) has d iterations (line 8). In each iteration, a parallel loop compares every point, p_i, of order l with the rest of the points, being the order, $|M_i|$, the number of 1's in the mask M_i ($|M_i| == l$). First, the median masks, M, are compared (line 11). Success in the comparison means that the two points are incomparable (and the corresponding *DT* is avoided). If the comparison of the median mask fails, then the quartile masks, Q, are compared (line 12). Only if both *MT* fail, the corresponding *DT* is carried out (line 13). If the point is dominated (line 14), it is marked as dominated, and the thread terminated. At the end of each iteration of the outer loop, the dominated points are removed. The non-dominated points for that order are added to the skyline. The update of $SKY(S)$ requires a synchronization, but contrary to the *OpenMP-CPU*, the number of synchronization points is equal to the number of dimensions, which is smaller than the number of synchronization points in *OpenMP-CPU* for large datasets.

Parallel loops in our SYCL-GPU implementation are expressed with the `sycl::parallel_for` function and `sycl::nd_range <3>` (3-dimensional), so parallel work-items execute over the nd_range in work-groups of the specified size. Also the `sycl::group_barrier` is invoked to synchronize work-items in the same work-group. Additionally, we take advantage of the *Unified Shared Memory* (USM) mechanism [4], which allows reading and writing of data with conventional pointers. In particular, as we target heterogeneous executions in which the CPU and the GPU can work cooperatively, we implement the shared data using the shared allocations strategy, which allows the runtime to automatically move data, if necessary, from/to the GPU or the host CPU when referenced on each device.

3.4. Initial performance assessment

Up to now we have seen two different algorithms that solve the same problem: computing the skyline of a dataset (or data query). *OpenMP-CPU* is optimized for the CPU, whereas *SYCL-GPU* comes from a GPU optimized CUDA code. However, now that we have the GPU version written in SYCL, we can take advantage of the portability exhibited by any SYCL implementation. In SYCL the computation is enqueued to a device, which is configured using a `device_selector` object. Just by changing the `device_selector` from GPU to CPU and re-compiling, the SYCL code can run on the CPU. This means, that we actually have three versions: *OpenMP-CPU*, *SYCL-CPU* and *SYCL-GPU*, being the last two the same implementation but targeting different devices.

Remember that our overarching goal is to accelerate the skyline computation on a heterogeneous CPU+integrated GPU architecture. This requires first to assess the performance of the three versions. To this end, we have executed the versions on a octa-core Intel i9-9900K CPU @ 3.60 GHz that includes an integrated GPU (Intel UHD Graphics 630 with 24 Compute Units). See more details of the test-bed in Section 6.

Fig. 2 shows the execution times for *OpenMP-CPU*, *SYCL-CPU* and *SYCL-GPU* on an integrated GPU, and four datasets (described in Section 6). For each dataset, the subfigure on the left fixes $d = 8$ and changes n from $1 \cdot 10^6$ to $8 \cdot 10^6$. The right subfigure fixes $n = 8 \cdot 10^6$ and changes d from 4 to 10. In order to also study the performance of our SYCL implementation on a non-integrated GPU, we run the kernel on a discrete GPU (Intel Xe DG1 with 96 Execution Units), using the same datasets. In Table 1 we show a summary of the times for these executions (*SYCL-dGPU*) along with the previous times for *OpenMP-CPU*, *SYCL-CPU* and *SYCL-GPU* on the integrated GPU for comparison. As expected, the discrete GPU exhibits higher performance than the integrated one (up to 2x of improvement). From the plots and the table we can conclude two main takeaway messages. First, that there is no device (CPU or GPU) that always dominates the other. Depending on several factors (points distribution in the dataset, n and d) the CPU or the GPU can be the fastest device. Second, out of the two CPU implementations (OpenMP-CPU and SYCL-CPU), the OpenMP-CPU is usually faster (except for House and Covertype datasets and larger values of n and/or d). Although the SYCL implementation is portable, it is not "performance portable" on the CPU, and considering that it was developed with the GPU in mind (it derives from a CUDA implementation) we found that is not always optimal for the multicore architecture in this platform. Therefore, from now on, we will use the *OpenMP-CPU* version on the CPU and the *SYCL-GPU* version on the GPU, unless contrary stated.

A work-efficiency study conducted in [3] reports that, for higher dimension datasets, *SkyAlign* requires less *DT*s than *Hybrid*. Thus, the *SkyAlign* algorithm (i.e. *SYCL-GPU* and *SYCL-dGPU*) strives to offer both more parallelism and work-efficiency for higher dimension datasets. Therefore, except for Weather dataset, we corroborate that *SYCL-GPU* (and *SYCL-dGPU*) outperforms *OpenMP-CPU* for high dimensionality queries. A deeper study of the behavior of the GPU algorithm for the Weather dataset shows that: (1) the preprocessing stage is not able to prefilter any point before entering the main loop; and (2) in the main loop the number of *MT*s is significantly smaller than in the other datasets and, hence, higher the number of *DT*s. This degrades the work-efficiency of the GPU algorithm, causing higher execution times in comparison to *OpenMP-CPU*.

On the contrary, the *Hybrid* algorithm (i.e. *OpenMP-CPU*) is less affected by the spatial distribution of the points. Its two parallel loops exhibit more regularity than the GPU algorithm's main loop. The fact that *OpenMP-CPU* processes the dataset in blocks results

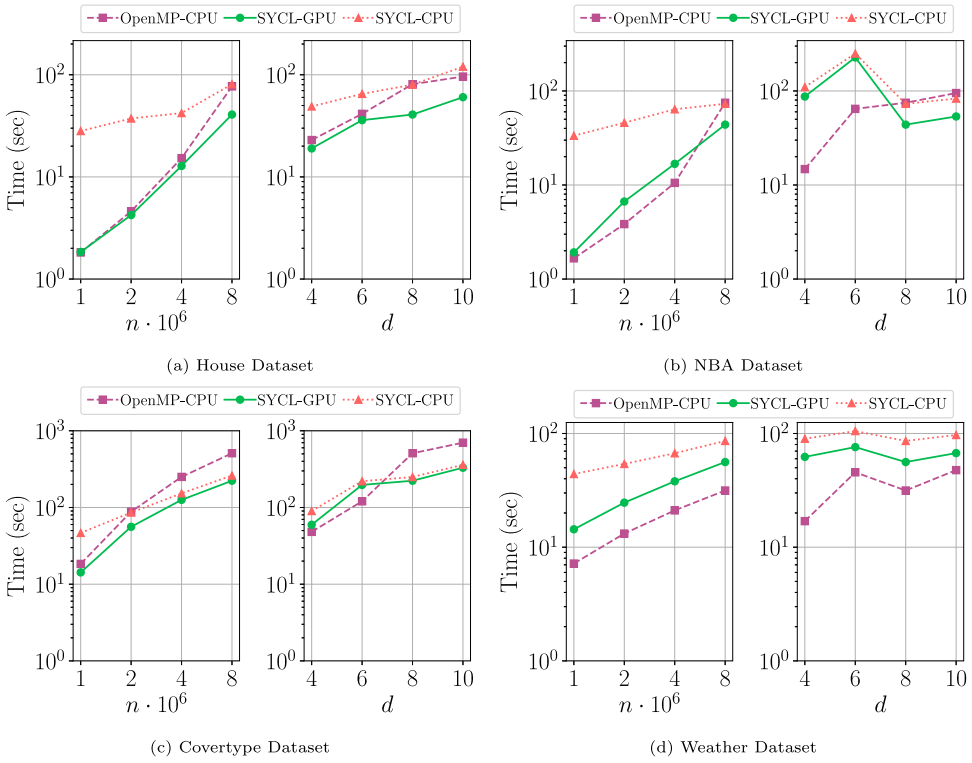


Fig. 2. Execution time (in seconds) for the three versions (*OpenMP-CPU*, *SYCL-CPU* and *SYCL-GPU* on a integrated GPU), and four datasets (the lower the better).

Table 1
Execution time (in seconds) for the *OpenMP-CPU* and *SYCL-CPU*, versus *SYCL-GPU* and *SYCL-dGPU* versions on an integrated and a discrete GPU, respectively, and two datasets. In bold we highlight the smallest times.

Implem.	House						Weather					
	$n \cdot 10^6$, $d = 8$			d, $n = 8 \cdot 10^6$			$n \cdot 10^6$, $d = 8$			d, $n = 8 \cdot 10^6$		
	1	2	4	4	6	8	1	2	4	4	6	8
OpenMP-CPU	1.84	4.60	15.26	22.95	41.45	80.84	**5.66**	**13.18**	**21.07**	**16.95**	**45.56**	**31.52**
SYCL-CPU	28.14	37.35	42.18	48.99	65.12	80.53	44.81	54.45	67.11	90.64	105.37	86.25
SYCL-GPU	1.82	4.23	12.81	18.99	35.95	40.80	14.37	24.65	37.97	62.26	75.90	56.08
SYCL-dGPU	**1.24**	**2.14**	**7.65**	**10.83**	**20.71**	**22.54**	9.42	18.31	30.66	34.51	55.72	38.23

in a better use of the cache hierarchy. Besides, the CPU is less affected by data and control divergence. On the other hand, the prefiltering step is less aggressive than in the GPU approach. To sum up, the skyline computation is highly irregular, heavily depending on the dataset configuration (distribution of points in the space, size, number of dimensions) and on the particular algorithm and target architecture.

In any case, for the rest of the paper and experimental validation, we focus on the more tightly connected CPU + integrated GPU architecture, where unified shared memory (USM) can be leveraged by advanced heterogeneous scheduling strategies. The study of the efficiency of our proposals on a CPU + discrete GPU is left for future work.

4. SkyFlow: Heterogeneous skyline over a stream of data queries

Now that we have an efficient implementation of the skyline algorithm for the CPU (*OpenMP-CPU*) and the GPU (*SYCL-GPU*), our goal is to devise an optimal graph-based engine to deal with a stream of data queries on a CPU+ GPU architecture, like the Intel i9-9900K described in Section 6. To this end, we rely on the FlowGraph classes provided by the Threading Building Block library [10] (part of the oneAPI [6] framework), that is cleverly designed to ease the optimization of data flow problems.

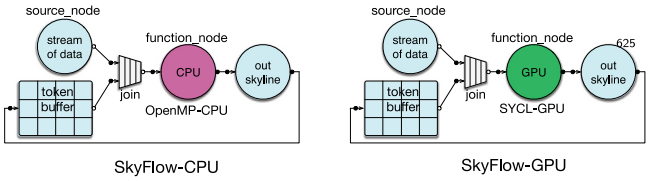


Fig. 3. Structure of the baseline SkyFlow graphs.

In the following sections we describe several data flow solutions, including the only-CPU and only-GPU baseline as well as two CPU+GPU heterogeneous approaches.

4.1. Baseline SkyFlow

As a baseline, we have developed a "single-device" version of our data flow approach, SkyFlow-CPU and SkyFlow-GPU, that only exploit the CPU or the GPU, respectively. Fig. 3 shows the FlowGraph that we describe next.

As we see in the figure, the graph is composed of a number of nodes connected by edges that we classify in three main sections: (1) *Source*, (2) *Execution*, and (3) *Output*.

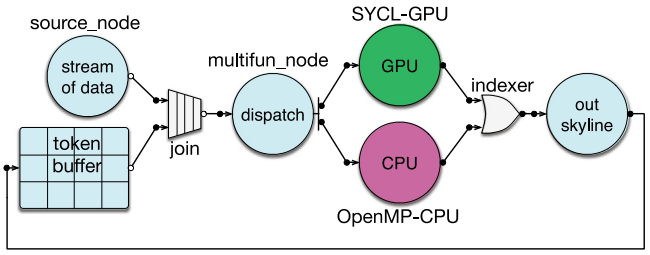


Fig. 4. Structure of Coarse-Grained SkyFlow graph, SkyFlow-CG.

- *Source*: It comprises a `source_node`, a `token buffer` and a `join_node`. The *source_node* generates the stream of data queries from which we require the skyline. However the stream is not fed directly into the graph to avoid oversubscribing the HW if the input data rate is much higher than the processing data rate. The idea is to rely on a token-based approach to limit the resource consumption. To that end, a buffer is pre-filled with a number of tokens and a `join_node` will forward a dataset (or data query) down the graph only if it can be paired with a token. That way, the maximum number of data queries in flight is limited to the number of tokens. At the output node, the token is recycled back into the buffer so that a new dataset can be injected into the *Execution* section of the graph.
- *Execution*: For the baseline solution, it contains a single `function_node` that computes the skyline. The SkyFlow-CPU configures this node to run the *OpenMP-CPU* algorithm whereas the SkyFlow-GPU runs the *SYCL-GPU* code.
- *Output*: Finally, the last node takes care of saving the resulting skyline, optionally checks that the computation is correct and recycles the token to enable the processing of a new dataset. It is also possible to use this node to merge several partial skylines in the case a dataset is partitioned into several blocks and processed separately by SkyFlow.

On our current platform with a multicore CPU and an integrated GPU we have validated that oversubscription is avoided by having just two tokens: one computing a skyline, and a second in the queue waiting to be dispatched as soon as the device becomes available. In SkyFlow-CPU the *OpenMP-CPU* parallel algorithm takes care of fully utilizing all the CPU cores.

4.2. Coarse-Grained heterogeneous SkyFlow

For our first heterogeneous approach we follow the easiest strategy that consists in combining in the *Execution* section the GPU and CPU algorithms concurrently, as depicted in Fig. 4.

We call this version Coarse-Grained SkyFlow, SkyFlow-CG, because it computes a whole dataset (or data query) either on the GPU or on the CPU. In the next section we describe a Fine-Grained approach in which a single dataset is split between the CPU and the GPU. Note that the *Execution* section now begins by a `dispatch` node that, for each dataset, decides whether it should be processed by the *SYCL-GPU* or the *OpenMP-CPU* algorithms. The number of tokens should be incremented in order to allow for at least two data queries in flight. Contrary to the `join` node, the `indexer` node (that comes after the GPU and CPU nodes) asynchronously passes incoming tokens to the next node without waiting for an input at each entry port, that way enabling that CPU tokens and GPU tokens traverse the flow graph at its own pace. This means that the resulting skylines can be computed out-of-order, but this is not a problem if we tag each result with the ID of the corresponding data query. If the output order is relevant it is always possible to insert before the output node a `sequencer` node [10] that would reorder the skyline results.

Assuming that the application bottleneck is in the *Execution* section (the one we are optimizing by exploiting both the CPU and the GPU at the same time), we implement two queues in the `dispatch` node (one per device). Now the problem is to feed these two queues so that the total execution time is minimized (throughput maximized). Two scheduling strategies have been considered and will be covered next: Work Conserving scheduling (WC) and Heterogeneous Earliest Finish Time (HEFT).

The goal of a Work Conserving scheduling is to keep all the scheduled devices busy. To that end, it strives to keep the queues of the devices with the same length (number of pending tasks). In our implementation we maintain two queues, GPU_q and CPU_q. An arriving data query will be enqueued in the shortest queue. If both queues have the same length, we have validated a tie-break heuristic that enqueues a data query in the CPU_q when its dimension, d, is smaller than 6 (since in our experiments it is probable that lower dimension datasets run faster on the CPU, as we can see in Fig. 2), and enqueued in GPU_q otherwise.

However, as we discussed in Section 3.4, this highly irregular problem is solved in very disparate execution times, sometimes smaller on the CPU or on the GPU, depending on many factors. If we want to optimize the execution time, keeping busy both devices is not enough because we could want to send the data query to the optimal device, so a more elaborated strategy is necessary to feed each device with the more suitable datasets. In this regard, the Heterogeneous Earliest Finish Time [36] is an interesting alternative. This scheduling policy also takes into account the "expected" execution time of a dataset in order to feed the queues. Now, it is not the length of the queue the relevant factor, but the expected accumulated execution time of all the data queries enqueued in each queue, GPU_t and CPU_t, and the expected execution time of the arriving data query both on the GPU and CPU, t_{gpu} and t_{cpu}. This is, an arriving data query will be enqueued in the queue in which it will finish earlier. More precisely, if $GPU_t + t_{gpu} < CPU_t + t_{cpu}$ the data query will be enqueued in GPU_q, and the other way around.

This HEFT policy poses two challenges, though. First we need an accurate enough estimation of the data query execution time. We propose a heuristic (detailed in Section 5.1) that can infer the total execution time after sampling the time required to compute a first chunk of points of the dataset, both on the GPU and on the CPU. Considering that the skyline computation for our datasets takes more than a second, we can afford to invest around 10 ms in precomputing the first chunk of a dataset in order to estimate the total execution time. Moreover, the result obtained after this precomputation is not wasted, but saved and never re-computed later on, so as we will see in the experimental evaluation, this strategy pays off well.

The second challenge is that estimating t_{gpu} and t_{cpu} is not possible if the GPU and the CPU are already busy processing data queries. This problem is tackled by batching the incoming data queries so that we conduct the precomputation and estimate total execution times for all the datasets in the batch. When the last data query of either the GPU or the CPU queue is launched, a new batch of data queries is sampled and HEFT is run to map them on the right queue. This not only avoids having to wait for the GPU or CPU to become idle to run HEFT, but also helps HEFT in having a farther view into the "future", which makes HEFT more profitable. For our experiments, we have found that a batching size of 5 data queries provides a good trade-off between the overhead due to the precomputation and the likelihood that one of the devices becomes idle when one of the queues runs out of queries.

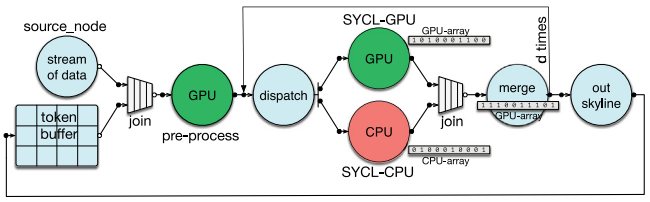


Fig. 5. Structure of Fine-Grained SkyFlow graph, SkyFlow-FG.

4.3. Fine-Grained heterogeneous SkyFlow

We also wanted to explore a different heterogeneous approach in which the GPU and the CPU are more tightly coupled. Instead of having two different data queries being processed simultaneously on the GPU and CPU with the SkyFlow-CG implementation, the idea now is to have a single dataset partitioned so that the GPU and the CPU collaborate in the skyline computation of this data query. We call this version Fine-Grained SkyFlow, SkyFlow-FG, and the corresponding flow graph is depicted in Fig. 5.

The starting point is the SYCL-GPU implementation that was described in Section 3.3. As we said, the code listed in Alg. 2 comprises two main stages: (1) preprocessing (lines 1–7); and (2) the main loop (lines 8- 16) that has d iterations (kernel invocations). The idea now is to efficiently distribute the computation between the GPU and the CPU, provided that the SYCL code is portable and the *SYCL-CPU* version runs on the CPU.

The preprocessing stage only accounts for around 10% of the execution time, and it is more than 10x faster on the GPU than on the CPU. Therefore, the slowest stage in the flow graph of Fig. 5 is not the preprocess one, which means that the preprocessing time can be hidden (one dataset is preprocessed and ready to be dispatched while the previous dataset is being computed). With all this, it is advisable to only exploit the GPU during this small fraction of the execution time.

However, the remaining 90% of the time is consumed in the main loop in which the same kernel in invoked d times (it depends on the number of dimensions). Provided that the kernel (lines 9–14 in Alg. 2) is executed over a range of (independent) points, it is possible to partition this range so that a number of sub-ranges (or chunks) are executed by *SYCL-GPU* and the rest by *SYCL-CPU*. This implies constructing two SYCL queues, one targeting the GPU and the other attached to the CPU device (see [7] for details on constructing device queues). It also requires merging the output of the GPU partial computation with the CPU one. Both devices write in private copies of the result array (GPU_array and CPU_array in Fig. 5) in which dominated points are marked with ones (line 14 in Alg . 2). Once both arrays have been fully written (note the `join` node after the CPU+GPU stage, instead of the `indexer` node used in Fig. 4), the `merge` node reduces them into the GPU_array. Marked points in this summarized array are pruned from the dataset (line 15 in Alg. 2).

Moreover, this Fine-Grained implementation also requires finding the right partition of the iteration space, so that load unbalance is avoided. Again, the `join` node after the CPU+GPU stage synchronizes both devices so the slowest one sets the stage time. The dispatch, CPU+GPU execution and merge, is repeated d times which reinforces the load balance requirement. The `dispatch` node takes care of computing the right partitioning. As we will see in the next section, a dynamic partitioning will be necessary to consider that the optimal partition depends on the relative speed of each device, which can change for each dimension (out of d) and during the traversal of each dimension. This dynamic partitioning splits the iteration space into several chunks of constant size. These chunks are processed on demand by the CPU and GPU devices (the device that becomes idle takes the next chunk until the iteration space is completed).

5. Coarse-Grained time estimation and Fine-Grained partition

5.1. Model for estimating Coarse-Grained execution times

As stated in Section 4.2, the HEFT strategy requires the estimation of the execution times on the GPU and the CPU, t_{gpu} and t_{cpu}, for each dataset in an incoming batch of data queries. In this section we provide the details of the model that computes those times, which are shown in Alg. 3. This model estimates the execution time of a dataset by profiling a small chunk of iterations both on the *OpenMP-CPU* and *SYCL-GPU* nodes.

Algorithm 3: Coarse-Grained HEFT model

Input: S=Dataset of n points and d dimensions;
Ch_{cpu}, Ch_{gpu}=CPU and GPU chunks.
Output: t_{cpu}, t_{gpu}= estimated CPU and GPU times for S.

```
1 (t_c, n_cpu)=launch_OpenMP-CPU (Ch_cpu)
2 ([t_g0 : t_gd-1], [n_g0 : n_gd-1], n_gpu)=launch_SYCL-GPU (Ch_gpu)
3 λ_c ← Eq. (1)
4 t_cpu ← Eq. (2)
5 foreach iteration i ∈ [0, d) do
6 |   F_i ← Eq. (4);   mg_i ← Eq. (5)
7 |   λ_gi ← Eq. (3)
8 t_gpu ← Eq. (6)
9 return (t_gpu, t_cpu)
```

In Alg. 3 we see that the heuristic to compute our model starts launching two chunks: Ch_{cpu} on the *OpenMP-CPU* node and Ch_{gpu} on the *SYCL-GPU* one (lines 1–2). The size of the chunk to perform the profiling is tuned at runtime. In our approach, by default it is set to 1% of the dataset size. We have found that if the reported time is around 10 ms, then the sample will typically help to provide an accurate estimation in our model. In the case that the chunk runtime is below 10 ms, then a new chunk twice the size of the previous one is launched. This process repeats until the reported processing time is above 10 ms. It is important to note that the work computed in this profiling stage is not wasted because the points computed in these chunks are recorded and counted for the complete execution later.

For the Ch_{cpu} chunk we record the execution time, t_c, and points explored, n_{cpu}. These results are used to calculate the throughput of the CPU chunk as we see in Eq. (1). For it, we assume a worst-case scenario in which all points of the chunk are compared when computing the OpenMP-CPU algorithm.

$$\lambda_c = \frac{(n_{cpu} \cdot (n_{cpu} - 1))/2}{t_c} \tag{1}$$

The estimated execution time on the CPU for dataset S, t_{cpu}, can be computed with Eq. (2), where again we assume a worst-case scenario where all points of the dataset (n) are compared. In Section 6.2 we analyze the accuracy of our assumption and provide results that indicate the estimated vs measured execution times for the *OpenMP-CPU* algorithm are always within the range $\pm 10\%$ in our datasets.

$$t_{cpu} = \frac{(n \cdot (n - 1))/2}{\lambda_c} \tag{2}$$

For the Ch_{gpu} chunk we now record the execution time, t_{g_i}, and points explored, n_{g_i}, in each iteration i of the main loop that traverses the d dimensions of the dataset (lines 8–16 in the *SYCL-GPU* algorithm). We also record the size of the GPU chunk, n_{gpu}. In the *SYCL-GPU* algorithm, at the end of each iteration of the d loop, the dominated points are removed, so fewer points enter into the next iteration. We use this information (and the time per iteration) to compute the throughput of the GPU chunk as we see in lines 5–7 in Alg. 3. In particular, we compute the GPU throughput for each dimension i, λ_{g_i} as we see in Eq. (3), where

we assume a worst-case scenario in which all the recorded points in the corresponding dimension (n_{g_i}) are compared.

$$\lambda_{g_i} = \frac{(n_{g_i} \cdot (n_{g_i} - 1))/2}{t_{g_i}} \tag{3}$$

From the information collected from the GPU chunk profiling, we can compute the ratio of points filtered when going from dimension $i-1$ to dimension i. This ratio, F_i, is shown in Eq. (4). Again, n represents all points of the dataset.

$$F_i = \begin{cases} \frac{n_{g_0}}{n_{gpu}} & \text{if } i == 0 \\ \frac{n_{g_i}}{n_{g_{i-1}}} & \text{otherwise} \end{cases} \tag{4}$$

From this ratio of filtered points, F_i, assuming an uniform distribution of the pruning of points for each dimension, we can extrapolate the number of points that will enter into each iteration of the d loop. This number of estimated points per iteration, mg_i is computed by Eq. (5).

$$mg_i = \begin{cases} n \cdot F_0 & \text{if } i == 0 \\ mg_{i-1} \cdot F_i & \text{otherwise} \end{cases} \tag{5}$$

Both the GPU throughput and estimated number of points for each dimension i, λ_{g_i} and mg_i respectively, are used to compute the estimated execution time on the GPU for dataset S, t_{gpu}, as we shown in Eq. (6). Again, we assume a worst-case scenario where all the estimated points in each dimension (mg_i) are compared. In Section 6.2 we analyze the accuracy of our assumptions and provide results that indicate that the estimated vs actual execution times for the *SYCL-GPU* algorithm are within ±2% in our datasets.

$$t_{gpu} = \sum_{i=0}^{d-1} \frac{(mg_i \cdot (mg_i - 1))/2}{\lambda_{g_i}} \tag{6}$$

5.2. Strategy for the Fine-Grained partitioning

As stated in Section 4.3, the main kernel of the SkyFlow-FG approach, which is launched d times to process the points of the dataset, can be partitioned into sub-ranges (or chunks) of independent points, which can be computed concurrently: while one chunk is computed by the *SYCL-CPU* node, another one can be computed by the *SYCL-GPU* node. We conducted a preliminary study in which we did not partition the datasets among the devices. Instead, we launched the main kernel d times on the same node (either on the CPU or on the GPU), without partitioning the points explored in each dimension, and measured the throughput per dimension d. We conducted this analysis for different configurations of our datasets (with different dataset sizes and dimensions) on our platform of reference (octa-core Intel i9-9900K CPU with integrated GPU, more details in Section 6.1). For instance, in Table 2 we show the GPU and CPU throughputs (`ThGPU`, `ThCPU`) for the NBA dataset with a configuration of 2 Million points and 7 dimensions. In addition to the throughput per dimension, we compute the total throughput and the relative speed or ratio ThGPU/ThCPU per dimension (column `Ratio`). As we see, the relative speed fluctuates for each dimension. A similar result was reported for other datasets and configurations. Thus, the partitioning strategy must be carefully designed to provide load balance for each d iteration. In this context, a dynamic partitioning strategy seems suitable.

In fact, in Fig. 6 we explore the behavior of a dynamic partitioning strategy that feeds the *SYCL-CPU* (or *SYCL-GPU*) node with chunks of points and see how the chunk size affects performance. In particular, we show the throughput on the CPU (or the GPU) device, for the first dimension traversal ($d = 1$) of the main kernel when a dynamic strategy partitions the iteration space in 10, 20, 30 and 40 chunks.

Table 2
Throughput per dimension for the main kernel on the *SYCL-GPU* and *SYCL-CPU* nodes (the higher the better). NBA dataset with $n = 2M$ and $d = 7$.

d	ThGPU	ThCPU	Ratio
1	7.08E+11	5.88E+11	1.20
2	5.96E+10	4.25E+10	1.40
3	4.13E+10	2.80E+10	1.47
4	3.98E+10	2.34E+10	1.70
5	3.24E+10	1.77E+10	1.83
6	1.53E+10	7.24E+09	2.11
7	2.76E+09	1.58E+09	1.75
Total	**6.65E+11**	**5.35E+11**	**1.24**

As we see in the figure, the throughput throughout the iteration space is highly variable in both devices. The same behavior is observed for the rest of dimensions of the main kernel, other datasets and configurations. This result points out that adaptive or predictive strategies based on the profiling of previous chunk samples can be misleading at this level of work granularity. So, it confirms that a dynamic partition strategy with a carefully selected chunk size must be adopted. From the figure we note that when the number of chunks is smaller, i.e. the chunk size is bigger, the measured GPU throughput tends to be higher. In fact, the GPU throughput degrades around 2% when the number of chunks doubles (i.e. the chunk size is halved). However, the CPU throughput tends to be independent of the chunk size. This result confirms that the *SkyAlign* algorithm at the core of the *SYCL-GPU* and *SYCL-CPU* implementations better exploits GPU architecture features such as coalesced memory accesses -thanks to the padding and re-packing of the main data structures-, as well as divergence minimization -thanks to the static mapping of the threads that in any given warp ensures that they work on a small set of aligned data blocks-.

In our SkyFlow-FG approach, the *dispatch* node (see Fig. 5) keeps track of the sub-range of points (chunks of Chunk_size iterations) assigned to each device for each dimension d that is traversed. This node is responsible for sending chunks to the *SYCL-GPU* and *SYCL-CPU* nodes once the previous chunk computation has finished on the corresponding device. The last sub-range of the iteration space (containing at most Chunk_size iterations) is partitioned again to keep balance among devices in the final stage of the computation. In any case, when designing our heterogeneous dynamic strategy we have to seek a trade-off chunk size: big enough to fully exploit the GPU micro-architecture features and to enable a near-optimal GPU throughput, but small enough to provide a sufficient number of chunks able to feed both devices while balancing the workload at the end of each dimension computation. Let us note again in Fig. 5 that although the chunks of iterations can be assigned asynchronously to the *SYCL-GPU*/*SYCL-CPU* nodes, after the computation of all chunks by those nodes there is a *join* node that synchronizes both devices.

In Section 6.3 we experimentally explore the performance of our heterogeneous dynamic strategy when selecting different chunk sizes for our datasets.

6. Experimental results

6.1. Experimental setting

All experiments shown in this section have been performed on an Intel(R) Core(TM) i9-9900K CPU @ 3.60 GHz with 8 cores, an integrated GPU (Intel UHD Graphics 630 with 24 Compute Units) and 32 GB DDR4 RAM. OS version of the system is Ubuntu 20.04.3 LTS.

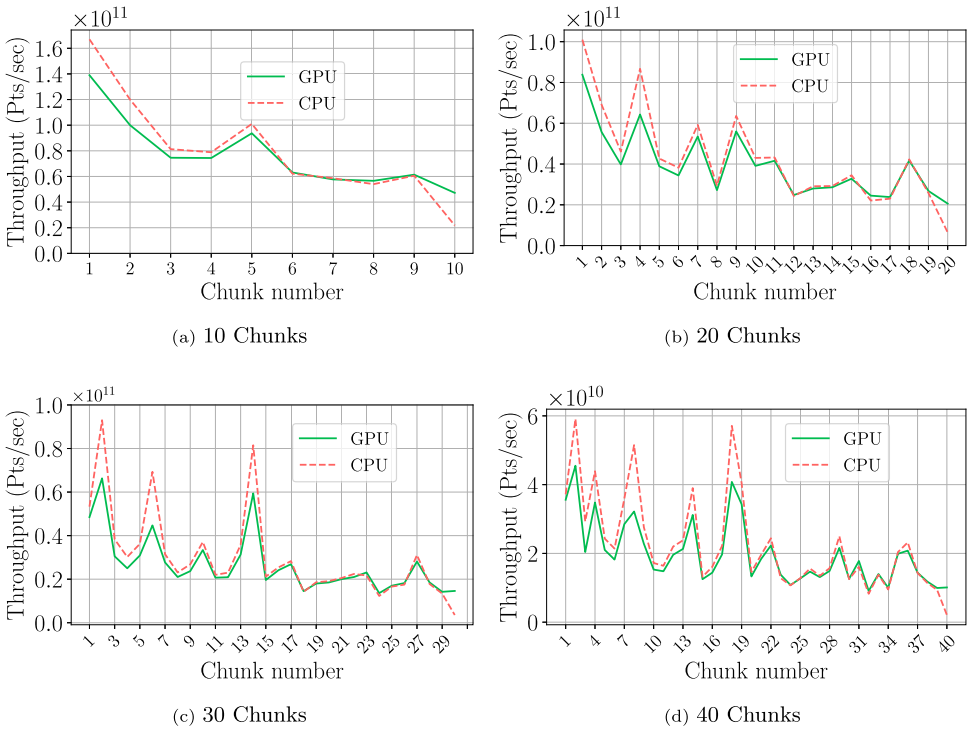


Fig. 6. Throughput per chunk when using a dynamic partition with 10, 20, 30 and 40 chunks for the first dimension traversal of the main kernel on the *SYCL-CPU* or the *SYCL-GPU* node (the higher the better). NBA dataset with $n = 2M$ and $d = 7$.

The *SYCL-GPU* and *SYCL-CPU* kernels have been generated using *Intel(R) oneAPI DPC++/C++ 2021.4* compiler and OpenCL 3.0 backend, while the *OpenMP-CPU* algorithm has been compiled using *g++ 9.3*, enabling OpenMP and AVX2. Also, the SkyFlow implementations (SkyFlow-GPU, SkyFlow-CPU, SkyFlow-CG and SkyFlow-FG) have been compiled with the mentioned *Intel(R) oneAPI DPC++/C++ Compiler*. CPU executions for OpenMP or SYCL algorithms use 8 threads. Times are measured using the *chrono* library [37]. The results shown in this section correspond to the median of 11 runs.

We conduct the experimental evaluation of our proposals using four real datasets, widely used in the skyline research literature: House [38], NBA [39], Covertype [40] and Weather [41]. For each dataset we can define different configurations changing the number of dimensions and points. In particular, the number of dimensions can go from 4 to 10 and the datasize from 1 Million to 8 Million points. This makes a total of 224 dataset configurations, from which we will illustrate the most relevant findings in the next sections.

6.2. Evaluation of CG-HEFT model accuracy

This section evaluates the accuracy of the time estimation model discussed in the Coarse-Grained HEFT Heuristic detailed in Section 5.1 and Alg. 3. Let us remember that the HEFT scheduling heuristic is used by the SkyFlow-CG approach to allocate any arriving data query on the queue that guarantees to finish earlier. For that, it must be estimated the execution times for computing the skyline of the incoming dataset, both under the *SYCL-GPU*/*OpenMP-CPU* algorithms (on the GPU and CPU device, respectively). For the evaluation of our model, we run the 224 dataset configurations first on the *SYCL-GPU* node (assuming that the *OpenMP-CPU* node is not available), and then on the *OpenMP-CPU* node (assuming now that the *SYCL-GPU* one is not available). For each experiment and configuration, we record the time estimated by our model (`Est-XXX`) and the actual execution time after the skyline computation (`Meas-XXX`), both on the GPU and CPU devices. In Fig. 7 we show a subset of these times. For each dataset, the subfigure on the left fixes $d = 8$ and changes n from $1 \cdot 10^6$ to $8 \cdot 10^6$. The right subfigure changes d from 4 to 10 for a fixed $n = 8 \cdot 10^6$.

From Fig. 7 we see that the estimated and actual measured times are very close, both on the GPU and CPU (*SYCL-GPU* and *OpenMP-CPU* algorithms, respectively). In particular for the *SYCL-GPU* results, the difference goes from [−1.2%,0.23%] for House, [−1.6%,0.61%] for NBA, [−1.8%,0.83%] for Covertype and [−1.9%,2%] for Weather. Now, for the *OpenMP-CPU* experiments the range goes from [−6.93%,7.2%] for House, [−8.28%,9.72%] for NBA, [−10.18%,10.45%] for Covertype and [−10.3%,10.5%] for Weather. A negative value means the model overestimates the actual measured time, while a positive one indicates that the model underestimates it. Although the accuracy is slightly worse for *OpenMP-CPU* compared to *SYCL-GPU*, our model is still accurate enough for the CG-HEFT scheduling heuristic. We base this claim in the fact that, for any given arriving data query the difference between the actual execution times on each device is much higher (from 1.5x to 4x) than the ± 10% of inaccuracy incurred by the model when making the decision to enqueue on one device. In other words, our model always selects the appropriate queue.

6.3. Evaluation of the partition strategy in Fine-Grained heterogeneous SkyFlow

In this section we analyze the performance of the heterogeneous dynamic partition strategy proposed in Section 5.2 whose goal is to find the near-optimal workload assigned to the *SYCL-GPU* and *SYCL-CPU* nodes in order to optimize the throughput in the SkyFlow-FG approach. As discussed in the mentioned section, this strategy asynchronously assigns chunks of iterations to the GPU and CPU devices for each traversal d of the main loop. The critical design issue is to find a chunk size big enough to ensure

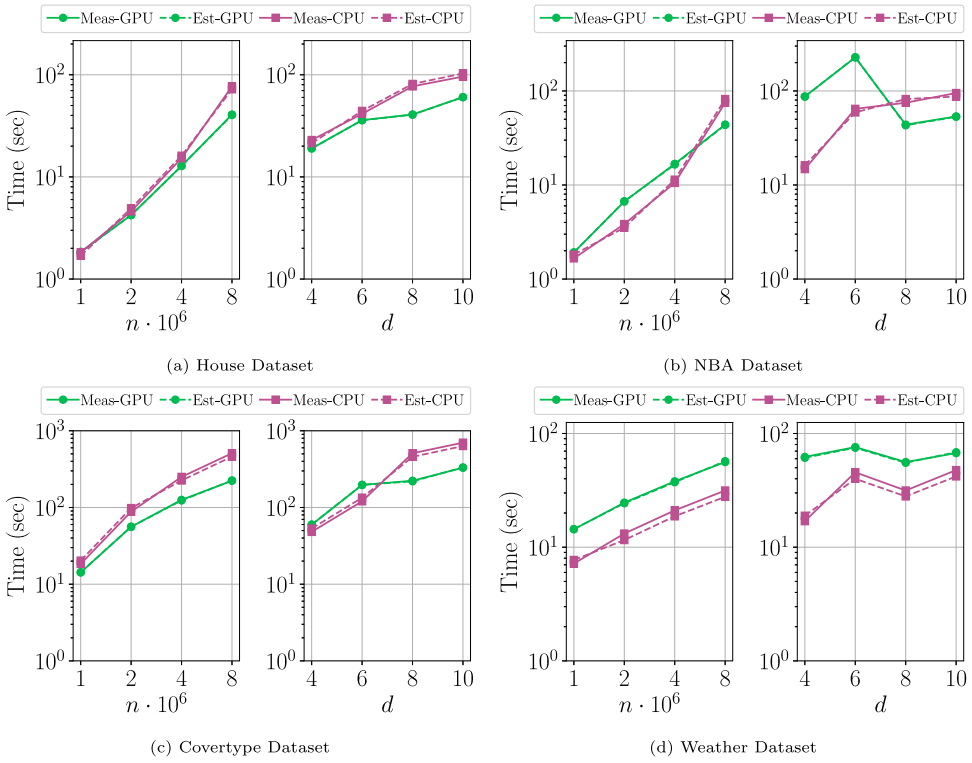


Fig. 7. Estimated vs actual measured times for the *SYCL-GPU* and *OpenMP-CPU* algorithms and four datasets (the lower the better).

a near-optimal GPU throughput, but small enough to provide a sufficient number of chunks able to feed both devices while balancing the workload for each traversal of the d loop. In Fig. 6 we showed the evolution of the throughput on each device when setting different number of chunks (and therefore different chunk sizes) in a dynamic partitioning on a configuration of the NBA dataset. Now in Fig. 8, for the same dataset configuration, we show the average throughput when running the dynamic partitioning only on the *SYCL-CPU* node (`ThCPU`), only on the *SYCL-GPU* node (`ThGPU`), and compare them with the average throughput when running the heterogeneous dynamic partitioning on the *SYLC-CPU+SYCL-GPU* nodes (`ThCPU+GPU`). Also, we depict an ideal throughput (`Ideal`) computed as the aggregation of the throughputs on the CPU and GPU without partitioning, so the partitioning overhead and load unbalance between devices is factored out.

As explained in Section 5.2, whereas the `ThCPU` is constant regardless the number of chunks, the `ThGPU` tends to slightly degrade when increasing the number of chunks (i.e., when reducing the chunk size). The GPU throughput degrades slightly: around 2% when the number of chunks is doubled. Interestingly, the heterogeneous `ThCPU+GPU` increases from 10 to 20 chunks, and then it degrades slightly from 30 onward (less than 1%). The values we see in the figure (the percentage of performance degradation of the heterogeneous execution with respect to the ideal throughput) helps to quantify, in part, the impact of load unbalance on the heterogeneous performance, because smaller chunks tend to minimize the unbalance. As we see, when the number of chunks is small (10, i.e. bigger chunks), load unbalance is the main factor that explains the 5.7% of performance degradation. Increasing the number of chunks to 20 (decreasing chunk size) reduces load unbalance to 1.95% (the sweet spot for this dataset configuration). However, from 30 chunks onward we see again degradation of the heterogeneous throughput compared to the ideal: now the minimization of the load balance is not compensated by the degradation of the GPU throughput due to smaller chunks. We conducted an exhaustive exploration of the optimal number of chunks (chunk size) for each dataset and configuration, and the optimal values that were found are used for the results of the SkyFlow-FG approach that we present in the next section.

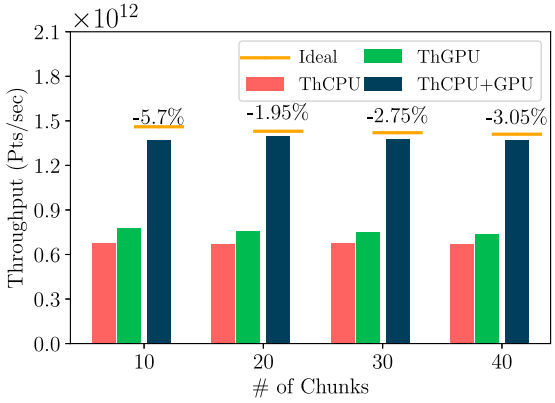


Fig. 8. Average throughput for the dynamic partitioning strategy in the SkyFlow-FG approach and different number of chunks (the higher the better). Device-only executions (`ThCPU`, `ThGPU`) are compared with heterogeneous execution (`ThCPU+GPU`) and ideal throughput (`Ideal`). The values represent the percentage of performance degradation of the heterogeneous CPU+GPU execution with respect to the ideal throughput (the lower the better). NBA dataset with 2M points and 7 dimensions.

6.4. Evaluation of heterogeneous SkyFlow approaches

In this section we present the performance results of the SkyFlow approaches that we introduce in Section 4. We measure the performance when streaming 100 data queries and record the median of 11 runs. One stream of data queries consist of mixed configurations (dataset size and dimensions) of the same dataset. As explained in Section 3.4, the performance of the

Table 3
Data queries mix for scenario R; Mean times (sec.) for baselines SkyFlow-GPU and SkyFlow-CPU (the lower the better) for scenarios R and U in our four datasets.

Dataset	R: GPU-CPU queries	R: SkyFlow-GPU time	R: SkyFlow-CPU time	U: SkyFlow-GPU time	U: SkyFlow-CPU time
NBA	44–56	2596.70	2464.87	4659.91	8719.60
House	38–62	894.04	890.81	701.56	4841.28
Covertype	13–87	3041.16	4032.88	4495.52	24311.40
Weather	22–78	1927.74	3040.70	2100.33	7438.54

skyline computation of a data query is highly irregular, heavily depending on the dataset configuration, algorithm and device. Thus, to thoroughly study the efficiency of our proposals, we evaluate two streaming scenarios: `Random (R)` and `Unbalanced (U)`. Whereas the `Random` scenario contains a random distribution of data queries, the `Unbalanced` scenario contains only data queries that run faster using *SYCL-GPU* than *OpenMP-CPU*. The second column of Table 3 provides details of the number of data queries that are faster with *SYCL-GPU* and with *OpenMP-CPU* under the `R` scenario for each dataset. The table also reports the mean times (in seconds) that the baselines SkyFlow-GPU and SkyFlow-CPU archive for the `R` and `U` stream scenarios.

The performance improvement of our heterogeneous proposals: SkyFlow-CG under WC scheduling (`SkyFlow-CG WC`), SkyFlow-CG under HEFT scheduling (`SkyFlow-CG WC`) and SkyFlow-FG under optimal dynamic partitioning (`SkyFlow-FG`), are presented in Fig. 9 for the two streaming scenarios (`R` -patterned bars- and `U` -solid bars-) in our four datasets. The performance improvement is presented as the speedup of each heterogeneous proposal vs. the baseline SkyFlow-GPU and the baseline SkyFlow-CPU (see Fig. 3), for both streaming scenarios (named `R-GPU`, `U-GPU` and `R-CPU`, `U-CPU` respectively). Evaluating the performance improvement of the heterogeneous implementations against the homogeneous baselines helps us to quantify the gain of heterogeneous implementations compared to single-devices ones in complex streaming scenarios.

As we see in Fig. 9, our heterogeneous approaches always outperform SkyFlow-GPU and SkyFlow-CPU baselines in the two streaming scenarios and four datasets. In fact, they outperform GPU and CPU baselines up to 5.19x and 6.86x, respectively. This result tells us that exploiting both devices with our heterogeneous solutions is usually more profitable than using just one device. Even if the device selected for the arriving data query is not the optimal one (CG approaches), or even if we partition the data points among devices (FG approach). For the `U` scenario, all the data queries are faster on the *SYCL-GPU* node, so the times for the baseline SkyFlow-CPU always take longer than for the SkyFlow-GPU (see the last two columns in Table 3). Thus, any heterogeneous approach that considers the GPU for this stream of data queries will show an important speedup when compared to SkyFlow-CPU (`U-CPU`) vs the speedup that we obtain when compared to SkyFlow-GPU (`U-GPU`) (see yellow solid bars vs blue solid bars). Regarding the `R` scenario (the patterned bars), the speedups against baseline SkyFlow-CPU (`R-CPU`) and SkyFlow-GPU (`R-GPU`) tend to be similar for NBA and House datasets, because the stream of data queries takes similar time in both datasets, while for Covertype and Weather datasets the speedup against SkyFlow-CPU is higher than the speedup against SkyFlow-GPU, because the times for SkyFlow-CPU take longer than for the SkyFlow-GPU in these cases (see third and fourth columns in Table 3). In any case, in the next subsections we discuss the main findings for our heterogeneous approaches.

6.4.1. Analysis of Coarse-Grained heterogeneous scheduling strategies

The SkyFlow-CG is a hybrid approach that considers two skyline algorithmic implementations that are optimal for each one of the two devices we target in this work: *SYCL-GPU* and *OpenMP-CPU*. As illustrated in Section 3.4, in our platform the optimal algorithm-device depends on characteristics of the arriving data query. The scheduling strategies proposed in Section 4.2, WC and HEFT, allocate any arriving data query either on the *SYCL-GPU* or the *OpenMP-CPU* queue following two different goals. The WC policy tries to keep the length of the queues equalized (same number of pending tasks) independently of which algorithm-device is best suited for the arriving query. By contrast, the HEFT strategy enqueues the incoming query in the queue in which it will finish earlier. While WC has minimum scheduling overhead and ensures that the devices are not idle if there are data queries ready, the HEFT strategy tries to optimize the system throughput.

From Fig. 9 we see that HEFT strategy always outperforms WC in all scenarios and datasets. Although WC scheduling minimizes the idle time on each device, it introduces a scheduling inefficiency by enqueueing queries in the non-optimal device. This inefficiency has a larger impact in the `R` scenario, where HEFT outperforms WC by 2.73x, 1.67x, 3.32x and 2.36 for NBA, House, Covertype and Weather, respectively. However, in the `U` scenario, HEFT outperforms WC by 1.02x, 1.04x, 1.08x and 1.07x, respectively, what demonstrates that even for non favorable situations HEFT still makes better scheduling decisions than WC. Another remarkable finding is that in the HEFT experiments we measure a time standard-deviation that goes from 0.17% to 0.47%, while in the WC runs it goes from 0.85% to 6.5%. These results point to the fact that HEFT also produces more stable executions.

Interestingly, in the most unfavorable scenario, that is `U` streaming, WC achieves improvements between 1.08x and 1.46x when compared to the optimal baseline SkyFlow-GPU. This corroborates the fact that from the point of view of the whole system performance, trying to keep both devices busy is still more beneficial that leaving the CPU idle, even if the CPU runs queries for which it is not the best device.

To measure the efficiency of our HEFT scheduling heuristic, we compare its performance with an Oracle approach that first computes offline the optimal device for each data query and uses this information to enqueue the query. Oracle also avoids the overhead of precomputing the expected execution time on any device for any incoming data query. Thus, Oracle represents the ideal peak performance for any CG scheduling policy. The results show that the performance of HEFT is below Oracle in 6.08%, 4.84%, 5.14%, 5.14% for NBA, House, Covertype and Weather, respectively. As we see, our scheduling heuristic introduces a low overhead while ensuring a near-optimal scheduling of data queries.

6.4.2. Study of Coarse-Grained vs Fine-Grained approaches

In this section we compare the performance of the SkyFlow-FG approach with the SkyFlow-CG HEFT, since the last one always outperforms SkyFlow-CG WC. Let us recall that in SkyFlow-FG, the workload of each data query is dynamically partitioned between the *SYCL-GPU* and *SYCL-CPU* nodes, and that the size (and number) of data chunks must be carefully selected, as we discussed in Section 6.3. From Fig. 9 we see that SkyFlow-CG HEFT always outperforms SkyFlow-FG in the `R` scenario, because SkyFlow-CG HEFT takes advantage of the better adaptation of

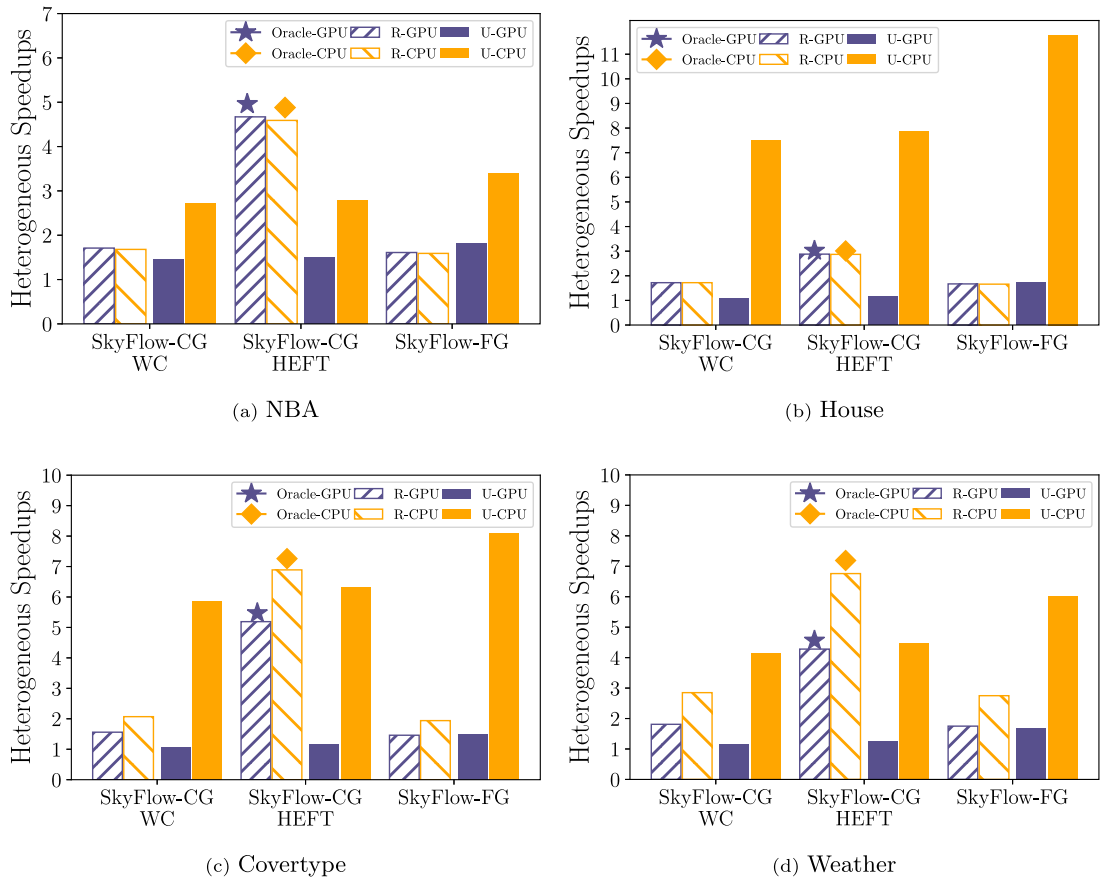


Fig. 9. Performance improvement of SkyFlow proposals for the four datasets and two streaming scenarios: `R` for random and `U` for unbalanced. The improvement is computed as a speedup vs the baseline SkyFlow-GPU (`-GPU`) and the baseline SkyFlow-CPU (`-CPU`). Oracle represents the optimal scheduling of queries for the SkyFlow-CG approach, which has been evaluated offline. The higher the better.

each specific query to the most suitable algorithm-device. However, fine-grained work partition in SkyFlow-FG does not pay off for the execution of queries with suboptimal performance under the SYCL-CPU implementation.

On the other hand, in the `U` scenario we get some interesting results. In this case all the queries run faster under the SYCL-GPU implementation. SkyFlow-FG will achieve optimal performance for all the queries, while SkyFlow-CG HEFT will degrade performance when executing queries in the *OpenMP-CPU* node. In this scenario SkyFlow-FG outperforms SkyFlow-CG HEFT by 1.22x, 1.5x, 1.27x and 1.37x for NBA, House, Covertype and Weather, respectively. In other words, when the incoming data queries run faster on the GPU device (SYCL-GPU) it can be advantageous to exploit a dynamic fine grain partition of the workload of each query between the GPU and CPU devices.

7. Conclusions

In this work we tackle the problem of computing the skyline operator over a stream of independent data queries targeting a heterogeneous architecture comprised of a multicore CPU and an integrated GPU. For it, we propose a heterogeneous graph-based engine, called SkyFlow to efficiently schedule the data queries between the devices. We propose two approaches that adapt to different streaming scenarios: Coarse-grained (SkyFlow-CG) and Fine-grained (SkyFlow-FG).

SkyFlow-CG computes concurrently one query per device, using a hybrid approach: each device runs the algorithm best suited to the specific features of the corresponding device. For our platform this means that the CPU runs the state-of-the-art *Hybrid* algorithm - based on an OpenMP implementation-, while the GPU executes the state-of-the-art *SkyAlign* - based on a SYCL implementation, novel in this paper-. Although both algorithms exploit work efficiency by reducing the number of dominance tests required during the skyline computation, for the real datasets evaluated in this paper we have found that on our system, some of them performs better under *OpenMP-CPU*, while others under *SYCL-GPU*. For the SkyFlow-CG approach we consider two scheduling strategies: Work Conserving (SkyFlow-CG WC) and Heterogeneous Earliest Finish Time (SkyFlow-CG HEFT). While WC aims to keep all devices busy by enqueueing any arriving data query on the shortest device queue, the HEFT strategy tries to optimize the system throughput by enqueuing the incoming query on the device queue in which it will finish earlier. HEFT requires estimating at runtime the computation time of an arriving query on each device. For it, in this paper we introduce a novel model that is based on an initial sampling of some points of the dataset, executed under *SYCL-GPU* and *OpenMP-CPU*. Through exhaustive evaluation we have found that the inaccuracy incurred by our model is within $\pm$ 10% of actual skyline computation times, which is always smaller than the time difference between algorithm-devices. As a result, our model always selects the optimal device. In any case, in our evaluation of the scheduling strategies, HEFT always outperforms WC in all streaming scenarios and datasets. In particular, the HEFT strategy outperforms WC up to 3.32$\times$ in random scenarios that contain a random mix of queries well suited for each algorithm-device. Moreover, when we compare the performance of our HEFT heuristic against an Oracle strategy that computes offline the optimal device for each query, we find that HEFT only degrades Oracle peak performance by 6% at most.

This corroborates that our model-based heuristic introduces a low overhead while ensures the optimality of the scheduling.

Secondly, the SkyFlow-FG approach dynamically partitions the workload of each arriving data query between the *SYCL-GPU* and *SYCL-CPU*. Through careful selection of the size (and number) of data chunks sent to each device, we have found that this fine-grained partition strategy is beneficial in a streaming scenario where the majority of data queries run faster under the SYCL algorithm. In this scenario SkyFlow-FG outperforms SkyFlow-CG HEFT up to 1.5×.

As a future work we plan on evaluate the efficiency of our proposals on alternative platforms that include a discrete GPU, or even an FPGA (another possible target of SYCL), which can be suitable devices to collaborate in the skyline computation. Also, it would be interesting to extend the evaluation on other platforms with different combinations of SYCL compilers and backends.

CRediT authorship contribution statement

Jose Carlos Romero: Investigation, Software, Validation, Writing – original draft, Visualization, Data curation. **Angeles Navarro:** Conceptualization, Methodology, Formal analysis, Writing – review & editing. **Andrés Rodríguez:** Software, Validation, Investigation, Visualization. **Rafael Asenjo:** Conceptualization, Data curation, Resources, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was partially supported by the Spanish projects PID2019-105396RB-I00, UMA18-FEDERJA-108 and P20-00395-R. Funding for open access charge: Universidad de Málaga / CBUA.

References

[1] S. Borzsony, D. Kossmann, K. Stocker, The skyline operator, in: Proceedings 17th International Conference on Data Engineering, 2001, pp. 421–430, http://dx.doi.org/10.1109/ICDE.2001.914855.
[2] S. Chester, D. Šidlauskas, I. Assent, K.S. Bøgh, Scalable parallelization of skyline computation for multi-core processors, in: 2015 IEEE 31st International Conference on Data Engineering, IEEE, 2015, pp. 1083–1094.
[3] K.S. Bøgh, S. Chester, I. Assent, SkyAlign: a portable, work-efficient skyline algorithm for multicore and GPU architectures, VLDB J. 25 (6) (2016) 817–841.
[4] The Khronos SYCL Working Group, SYCL 2020 specification (revision 3), 2021.
[5] G. Castaño, Y. Faqir-Rhazoui, C. García, M. Prieto-Matías, Evaluation of intel's DPC++ compatibility tool in heterogeneous computing, J. Parallel Distrib. Comput. 165 (2022) 120–129, http://dx.doi.org/10.1016/j.jpdc.2022.03.017.
[6] I. Corporation, oneAPI Specification 1.0 Rev. 2, 2021, https://spec.oneapi.io/versions/1.0-rev-2/ (Accessed: 05 Oct 2021).
[7] J. Reinders, B. Ashbaugh, J. Broadman, M. Kinsner, J. Pennycook, X. Tian, Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL, A Press, 2021.
[8] K.S. Bøgh, S. Chester, D. Šidlauskas, I. Assent, Template skycube algorithms for heterogeneous parallelism on multicore and GPU architectures, in: Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 447–462.
[9] K. Alami, N. Hanusse, P. Kamnang-Wanko, S. Maabout, The negative skycube, Inf. Syst. 88 (2020) 101443.
[10] M. Voss, R. Asenjo, J. Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, A Press, 2019.
[11] X. Liu, R. Lu, J. Ma, L. Chen, H. Bao, Efficient and privacy-preserving skyline computation framework across domains, Future Gener. Comput. Syst. 62 (2016) 161–174.
[12] A. Cuzzocrea, P. Karras, A. Vlachou, Effective and efficient skyline query processing over attribute-order-preserving-free encrypted data in cloud-enabled databases, Future Gener. Comput. Syst. 126 (2022) 237–251.
[13] H. Liang, B. Ding, Y. Du, F. Li, Parallel optimization of qos-aware big service processes with discovery of skyline services, Future Gener. Comput. Syst. 125 (2021) 496–514.
[14] S. Wang, L. Huang, L. Sun, C.-H. Hsu, F. Yang, Efficient and reliable service selection for heterogeneous distributed software systems, Future Gener. Comput. Syst. 74 (2017) 158–167.
[15] B. Shahzaad, A. Bouguettaya, S. Mistry, A.G. Neiat, Resilient composition of drone services for delivery, Future Gener. Comput. Syst. 115 (2021) 335–350.
[16] K.-L. Tan, P.-K. Eng, B.C. Ooi, et al., Efficient progressive skyline computation, in: VLDB, Vol. 1, 2001, pp. 301–310.
[17] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Trans. Database Syst. 30 (1) (2005) 41–82.
[18] J. Chomicki, P. Godfrey, J. Gryz, D. Liang, Skyline with presorting, in: ICDE, Vol. 3, 2003, pp. 717–719.
[19] K.C. Lee, B. Zheng, H. Li, W.-C. Lee, Approaching the skyline in z order, in: VLDB, Vol. 7, 2007, pp. 279–290.
[20] I. Bartolini, P. Ciaccia, M. Patella, Efficient sort-based skyline evaluation, ACM Trans. Database Syst. 33 (4) (2008) 1–49.
[21] S. Zhang, N. Mamoulis, D.W. Cheung, Scalable skyline computation using object-based space partitioning, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, 2009, pp. 483–494.
[22] J. Lee, S.-w. Hwang, Scalable skyline computation using a balanced pivot selection technique, Inf. Syst. 39 (2014) 1–21.
[23] S. Park, T. Kim, J. Park, J. Kim, H. Im, Parallel skyline computation on multicore architectures, in: 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 760–771.
[24] W. Choi, L. Liu, B. Yu, Multi-criteria decision making with skyline computation, in: 2012 IEEE 13th International Conference on Information Reuse & Integration, IRI, IEEE, 2012, pp. 316–323.
[25] K.S. Bøgh, I. Assent, M. Magnani, Efficient GPU-based skyline computation, in: Proceedings of the Ninth International Workshop on Data Management on New Hardware, 2013, pp. 1–6.
[26] Y.-W. Peng, W.-M. Chen, Parallel k-dominant skyline queries in high-dimensional datasets, Inform. Sci. 496 (2019) 538–552, http://dx.doi.org/10.1016/j.ins.2019.01.039, URL https://www.sciencedirect.com/science/article/pii/S0020025519300490.
[27] X. Lin, Y. Yuan, W. Wang, H. Lu, Stabbing the sky: Efficient skyline computation over sliding windows, in: 21st International Conference on Data Engineering, ICDE'05, IEEE, 2005, pp. 502–513.
[28] Y. Tao, D. Papadias, Maintaining sliding window skylines on data streams, IEEE Trans. Knowl. Data Eng. 18 (3) (2006) 377–391.
[29] M. Morse, J.M. Patel, W.I. Grosky, Efficient continuous skyline computation, Inform. Sci. 177 (17) (2007) 3411–3437.
[30] T. De Matteis, S. Di Girolamo, G. Mencagli, Continuous skyline queries on multicore architectures, Concurr. Comput.: Pract. Exper. 28 (12) (2016) 3503–3522.
[31] H. Lu, Y. Zhou, J. Haustad, Efficient and scalable continuous skyline monitoring in two-tier streaming settings, Inf. Syst. 38 (1) (2013) 68–81.
[32] S. Sun, Z. Huang, H. Zhong, D. Dai, H. Liu, J. Li, Efficient monitoring of skyline queries over distributed data streams, Knowl. Inf. Syst. 25 (3) (2010) 575–606.
[33] G. Mencagli, M. Torquati, M. Danelutto, Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams, Future Gener. Comput. Syst. 79 (2018) 862–877.
[34] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel programming in OpenMP, Morgan kaufmann, 2001.
[35] R. Farber, CUDA Application Design and Development, Elsevier, 2011.
[36] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274, http://dx.doi.org/10.1109/71.993206.
[37] Chrono library, 2022, https://en.cppreference.com/w/cpp/chrono (Accessed: 05 April 2022).
[38] Dataset house, 2021, http://usa.ipums.org/usa/ (Accessed: 20 Sept 2021).
[39] Dataset NBA, 2021, http://databasebasketball.com (Accessed: 20 Sept 2021).
[40] Dataset covertype, 2021, http://archive.ics.uci.edu/ml/datasets/Covertype (Accessed: 20 Sept 2021).
[41] Dataset weather, 2021, http://cru.uea.ac.uk/cru/data/hrg/tmc/ (Accessed: 20 Sept 2021).

Jose Carlos Romero received the engineering degree in industrial engineering in 2016 and the Master degree in mechatronics engineering in 2017, both in the University of Malaga. He obtained the Ph.D. degree in Computer Science in the Department of computer Architecture, University of Malaga in 2022. His research interests include heterogeneous architectures and parallel programming.

Angeles Navarro obtained a Ph.D. in Computer Science from the Universidad de Málaga, Spain, in 2000. She is a Full Professor in the Department of Computer Architecture at Universidad de Málaga. She has been a Research Visiting Scholar in the University of Illinois at Urbana-Champaign (UIUC), the Technical University of Munich (TUM), the EPCC at the University of Edinburgh, the University of Bristol, and a Research Visitor in IBM T.J. Watson Research Center at New York and in Cray Inc at Seattle. She has served as a program committee member for several High Performance Computing related conferences as PPoPP, SC, ICS, PACT, IPDPS, ICPP, EuroPar, ISPA and ISC. She is the co-lider of the Parallel Programming Models and Compilers group at the Universidad de Málaga. Her research interests are in programming models for heterogeneous systems, analytical modeling, compiler and runtime optimizations.

Andrés Rodríguez obtained a Ph.D. in Computer Science Engineering from the Universidad de Málaga, Spain, in 2000. From 1996 to 2002, he was an Assistant Professor in the Computer Architecture Department at Universidad de Málaga, being an Associate Professor since 2003. He lectures on operating system design, mobile devices architectures and IoT. His research interests are in parallel programming models, tools for heterogeneous architectures and edge computing.

Rafael Asenjo is Professor of Computer Architecture at the University of Málaga. He obtained a Ph.D. in Telecommunication Engineering in 1997. He has been using TBB since 2008 and over the last five years, he has focused on productively exploiting heterogeneous chips leveraging TBB as the orchestrating framework. In 2013 and 2014 he visited UIUC to work on CPU+GPU chips. In 2015 and 2016 he also started to research into CPU+FPGA chips while visiting the University of Bristol. He served as General Chair for ACM PPoPP'16 and as an Organization Committee member as well as a Program Committee member for several HPC related conferences (PPoPP, SC, PACT, IPDPS, HPCA, EuroPar, and SBAC-PAD). His research interests include heterogeneous programming models and architectures, parallelization of irregular codes and energy consumption. He co-authored the latest book (open access) on Threading Building Blocks (Pro TBB), is oneAPI Innovator, SYCL Advisory Panel member and ACM member.