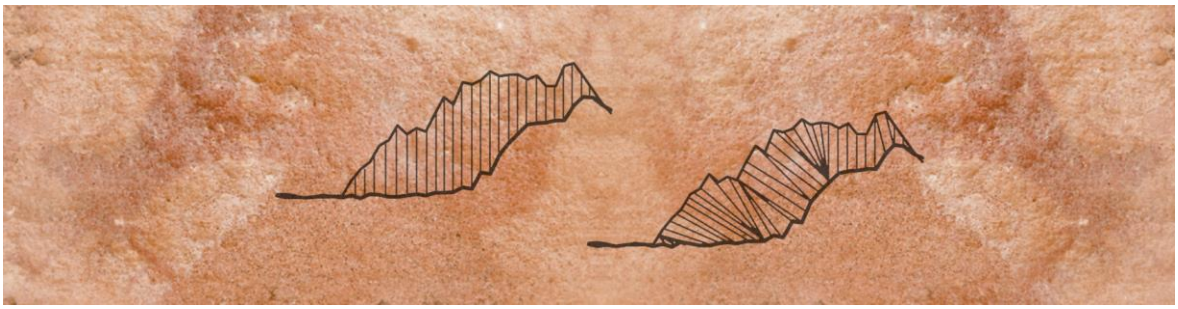
Marie Skłodowska-Curie Actions

Grant agreement ID 101031688

PLATON - Platform-aware LArge-scale Time-Series prOcessiNg



Deliverable D3.2

Report on the Accelerator Kernels for Data Series Processing

Edited by Panagiota Fatourou

List of Authors

Botao Peng
Panagiota Fatourou
Themis Palpanas

Code available at
https://github.com/PLATON-MARIE-CURIE-PROJECT/SING

September 2022

TABLE OF CONTENTS

LIST OF PAPERS

- Botao Peng, Panagiota Fatourou, and Themis Palpanas, `` SING: Sequence Indexing Using GPUs,” submitted for journal publication. LIPADE-TR-No 8, October 3, 2022. A preliminary version of this work has appeared in the Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE 2021), April 2021, IEEE, ISBN 978-1-7281-9184-3.

SUMMARY

PLATON aspires to build the necessary methods, algorithms and tools for highly-efficient, *accelerated* processing of big collections of data series. In particular, it aims at achieving enhanced performance by combining the power of general purpose CPUs with accelerators, such as Graphical Processing Units (GPUs). PLATON has achieved its goal of developing *a new index to ensure at least 5x better performance than the current state-of-the-art parallel solutions by utilizing GPU computing.*

This deliverable presents the tools and techniques for efficient *CPU and GPU co-processing* for highly-accelerated data series processing that have been developed in the context of the project. Specifically, we present SING, an in-memory index that uses the GPU's parallelization opportunities and combines them with the power of SIMD (Single Instruction Multiple Data) and multi-core processing, in order to accelerate similarity search. We conduct an experimental evaluation with several synthetic and real datasets, which shows that SING is up to 5.1x faster than the state-of-the-art parallel in-memory approach, and up to 62x faster than the parallel version of the state-of-the-art serial scan algorithm. SING achieves exact similarity search query times as low as 32msec on 100GB datasets, which enables interactive data exploration on very large data series collections.

1 Introduction

[Motivation] Several applications across many diverse domains, such as in finance, astrophysics, neuroscience, engineering, multimedia, and others [6, 17, 48, 51, 86], continuously produce big collections of data series[1], which need to be processed and analyzed [7, 8, 29, 30, 44, 63, 66, 75]. Often times, this is part of an exploratory process, where users ask a query, review the results, and then decide what their subsequent queries, or analysis steps should be [51]. The most common type of query that different analysis applications need to answer on these collections of data series is similarity search [17–19, 48]. The continued increase in the rate and volume of data series production, with collections that grow to several petabytes in size [6, 48, 51], renders traditional, serial-execution data series indexing technologies [5, 10, 60, 65, 68, 71, 83] inadequate [18, 19, 21, 22]. Thus, several efforts have focused on the development of parallel [54–56] and distributed [38, 39, 72–74] indexing techniques.

In this work, we present the first efficient parallel *exact query* answering scheme for *in-memory* data series processing which combines the full computational capacity of a single node (SIMD, multi-core, multi-socket) with the power of GPU accelerators. The necessity for fast in-memory and exact data series computations appears in real scenaria [16, 51], e.g., in Airbus [55]. Although Airbus stores petabytes of data series, reasoning about the behavior of aircraft components or pilots [27], requires experts to run analytics only on subsets of the data (e.g., on those relevant to landings from Air France pilots) that fit in memory. Often times, in such critical applications *exact answers* to queries are required (rather than approximate answers with no quality guarantees) [51].

Current state-of-the-art data series indexing schemes, such as ParIS+ [54, 56] (*disk-based*), and MESSI [55] (*in-memory*) exhibit advanced performance by exploiting the parallelism opportunities offered by the multi-socket, multi-core, and SIMD architectures. However, these indexes did not take advantage of the parallel computation power of Graphics Processing Units (GPUs). This is the research problem that we solve here. We present SING (Sequence Indexing Using GPUs), the first (in-memory) data series indexing scheme that combines the GPU's parallelization opportunities with those of SIMD, multi-core and multi-socket architectures, in order to accelerate exact similarity search.

[Challenges] The programming approach for GPU algorithms is distinct from that of CPU, and requires different design and techniques to achieve the desired performance improvement. Existing state-of-the-art iSAX-based[2] indexes [56, 57, 84] work as follows. They include a first phase, where they compute summaries for the raw data and they use them to build a tree index structure. In their second phase, they use this tree index structure to answer queries. We note that the query answering process also entails accesses to the raw data for some of the elements of the tree index, i.e. it accesses those data series that cannot be pruned using their summaries.

The following constraints of a GPU determine the phase in which GPU processors can be used to enhance performance, and the algorithmic choices we need to make. First, the on-board GPU memory is rather limited, and cannot hold the entire dataset. Note that the GPU we use in this work has 12GB of RAM, while our datasets are one order of magnitude larger, occupying 100GB; much larger data series collections are very common in practice [6, 18, 19, 51]. For this reason, storing the entire raw data set in the GPU memory is not possible in the common case (unless a more complicated and much more expensive GPU-farm solution is adopted).

Second, the solution of moving at query time all, or subsets of the raw data into the GPU (e.g., in batches, or streaming) for further processing entails also excessive costs because of the slow interconnect speeds: in our system (detailed in Section 169) that uses a PCI-Express 3.0 x16 bus, this speed was measured at 10GB/sec. Given that query answering times are in the order of 50-100msec, the above data transfer rate means that not only can we not afford to move the entire dataset to the GPU (that would need 10sec), but even moving small *ad hoc* subsets of data required by queries (i.e., those not pruned)

[1] A data series, or data sequence, is an ordered sequence of data points. If the ordering dimension is time then we talk about time series, though, series can be ordered over other measures. (e.g., angle in astronomical radial profiles, frequency in infrared spectroscopy, mass in mass spectroscopy, etc.).

[2] iSAX stands for indexable Symbolic Aggregate approXimation [67].

incurs a prohibitively high time cost (e.g., the raw data for an average 0.4% of a 100GB dataset would need >40msec). This disallows processing raw data (or even chunks of them) in the GPU memory.

Last, we note that the massive parallelism offered by GPUs comes at the cost of non-sophisticated Streaming Processors (i.e., GPU cores) that are geared towards simple computations, and cannot efficiently support complex computations. Therefore, GPUs are not readily suited to data structures and algorithms that frequently involve branching, such as in tree indexes. Moreover, GPU performance is heavily affected by the data access patterns and data locality, especially since the cache sizes in the GPU memory hierarchy are relatively small (and shared among a large number of GPU cores) [45]. The above considerations imply that modern multi-core architectures remain competitive to GPU solutions for the complex tasks we consider in this work. In fact, we point out that previous GPU solutions for data-series query answering and similarity search only compared to CPU baselines with up to 2 cores, and without the use of SIMD [40, 78, 79].

[Our Approach] SING provides a novel similarity search algorithm that runs on top of the tree structure created by iSAX-based indexes [56, 57, 84]. This algorithm ensures effective CPU+GPU co-processing (i.e., collaboration of both the CPU and GPU resources of the system) to produce the exact query answers. It is based on the following key ideas. First, it executes an in-order traversal of the root subtrees of the tree structure, and stores into a sorted array the iSAX summaries for all the data series in the index. It is this array of summaries (and not the entire raw data collection) that it needs to store in the GPU memory. Since the iSAX summaries are (almost) two orders of magnitude smaller than the raw data, SING can serve very large data series collections using the limited GPU memory, and performs mostly consecutive in-memory accesses. Second, it introduces a new pruning strategy that ignores entire root subtrees, in order to reduce the number of lower bound distance computations (i.e., distance computations between the summaries of the data series) that the GPU should execute. When datasets contain hundreds of millions of series, this leads to considerable time savings, even when executed in the GPU[3]. Third, unlike earlier iSAX based indexes [49] that compute lower bound distances using a dictionary (lookup table) with the values where two neighboring iSAX symbols meet (break points), SING employs a simple polynomial function that provides these values. As a result, the lower bound distance computations of each data series are executed in their entirety using only the registers of the GPU Streaming Processors (cores), avoiding expensive accesses to memory outside the Streaming Processors. Last, SING effectively divides the workload among the CPU and the GPU cores, and orchestrates their parallel execution. This means that the CPU workers start processing candidate answers without waiting for the GPU computation to complete. To achieve this overlap of computations between the CPU and the GPU, the work is split into chunks. The GPU streams the results of the work on each chunk to the CPU threads, and the CPU completes the similarity search computation for the data series that correspond to this chunk.

Overall, the above ideas reduce considerably the amount of work, as well as the execution time. Our experiments show that SING outperforms by a large margin the current state-of-the-art parallel (i.e., SIMD and multi-core) solutions in a variety of settings, even when the competitors use all 16 cores of our system.

[Contributions] Our contributions are summarized as follows[4].

- We propose SING, a data series index designed to take advantage of GPUs. At the same time, SING also exploits all available cores and sockets of the CPU.
- We describe a novel similarity search algorithm that effectively makes use of both the CPU and GPU resources of a system, ensuring that they operate in synergy and collaborate to produce the exact answers. This algorithm stores the summaries of all data series in the dataset directly in the GPU's memory.
- We design a new algorithm to calculate lower bound distances that is GPU friendly. In order to do

[3] We note that MESSI follows a different, more efficient, pruning strategy, which however entails branching computations that cannot be performed in the GPU in an efficient way.

[4] A preliminary version of this work has appeared elsewhere [58].

that, we also propose a new technique for representing the iSAX break points that is suitable for the GPU computations, leading (for this specific operation) to more than one order of magnitude better performance.

- Using several synthetic and real datasets, we experimentally show that SING achieves exact similarity search times as low as 32msec on 100GB datasets, demonstrating the efficiency of the proposed solution. The results show that SING is up to 5.1x faster at query answering time than MESSI [55, 57], the state-of-the-art parallel in-memory approach, and up to 62x faster than the state-of-the-art parallel serial scan algorithm, and remains the overall best performer even when all algorithms use 16 cores. Moreover, they demonstrate the performance benefits that are provided by the design choices of SING when compared to a naive GPU-based implementation of MESSI.

[Outline] In Section we provide the necessary background for the rest of this work. Section 69 describes the SING algorithms. In Section 169, we present the experimental evaluation of our approach. Finally, we discuss the related work in Section 169, and conclude in Section 169.

2 Background

[Data Series] A data series, $S = \{p_1, ..., p_n\}$, is defined as a sequence of points, where each point $p = (v, t)$, is associated to a real value v and a position t. The position corresponds to the order of this value in the sequence. We call n the *size*, or *length* of the data series. We note that all the discussions in this work are applicable to high-dimensional vectors, in general.

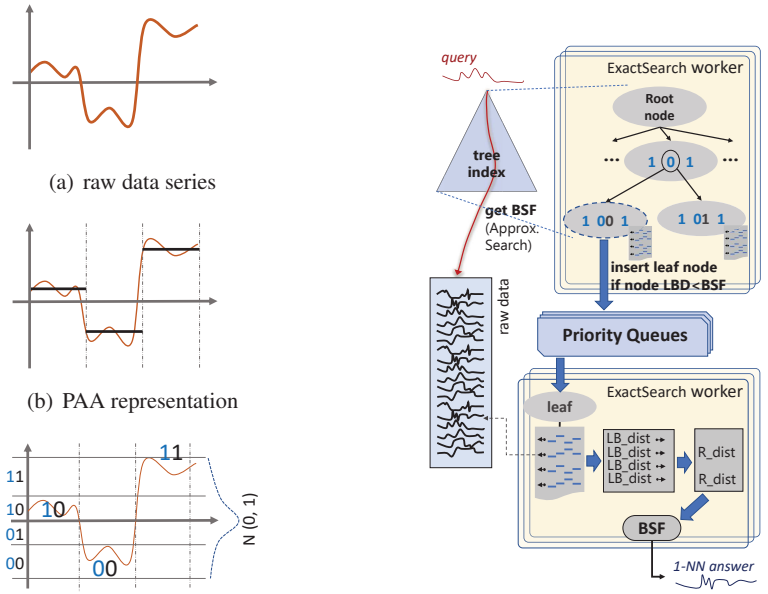
[Similarity Search] Analysts perform a wide range of data mining tasks on data series including clustering [62], classification and deviation detection [11, 69], and frequent pattern mining [46]. Existing algorithms for executing these tasks rely on performing fast similarity search across the different series. Thus, efficiently processing nearest neighbor (NN) queries is crucial for speeding up the above tasks.

NN queries are defined as follows: given a query series S_q of length n, and a data series collection $\mathcal{S}$ of sequences of the same length, n, we want to identify the series $S_c \in \mathcal{S}$ that has the smallest distance to S_q among all the series in the collection $\mathcal{S}$. (In the case of streaming series, we first create subsequences of length n using a sliding window, and then index those. In this work, we use Euclidean Distance (ED) [4]; though, Dynamic Time Warping (DTW) [61] can be supported, as well [55]. We define the *real distance* of a query series S_q to a data series S as the Euclidean distance between the raw values of S_q and the raw values of S.

Similar to previous work [55, 57], in this work, we focus on the case where the raw data fit in the CPU memory. At the same time, we assume that the summaries of the raw data fit in the GPU (global) memory; as we discuss later on, this is a valid assumption.

[iSAX] The iSAX representation (or summary) [67] is based on the Piecewise Aggregate Approximation (PAA) representation [31], which divides the data series in w segments of equal length, and uses the mean value of the points in each segment in order to summarize a data series. Figure 1(b) depicts an example of PAA representation with $w = 3$ segments (depicted with the black horizontal lines), for the data series depicted in Figure 1(a). Following previous work [49], we use $w = 16$.

Based on PAA, the indexable Symbolic Aggregate approXimation (iSAX) representation was proposed in [67] (and later used in several different data series indexes [36, 42, 50, 54, 69, 83]). This method first divides the (y-axis) space in different regions, and assigns a bit-wise symbol to each region. In practice, the number of symbols is small: iSAX achieves very good summarizations with as few as 256 symbols, the maximum alphabet cardinality, $|alphabet|$, which can be represented by eight bits [10]. It then represents each one of the w segments of the series with the symbol of the region the PAA falls into, forming the word $10_2 00_2 11_2$ shown in Figure 1(c) (subscripts denote the number of bits used to represent the symbol of each segment). We define the *lower bound distance* of a query series S_q and a data series S as the distance between the PAA of the query and the iSAX summary of S.



(a) raw data series

(b) PAA representation

(c) iSAX representation

(d) MESSI index

Figure 1: The iSAX representation, and the MESSI index

[MESSI Index] The MESSI index [55, 57] is the state-of-the-art in-memory data series index from the iSAX-based family of indexes (i.e., from those indexes [54–57, 84] whose design is based on the use of iSAX summaries). It was developed [49] to propose techniques and algorithms specifically designed for a concurrent multi-threaded environment and in-memory data. Many indexes in the iSAX-based family [54–57, 84] (including MESSI) make use of variable cardinalities for the iSAX summaries (i.e., variable number of bits for the symbol of each segment) in order to build a hierarchical tree index (see Figure 1(d)), consisting of three types of nodes: (i) the root node points to several children nodes, 2^w in the worst case, i.e., when the series in the collection cover all possible iSAX summaries; (ii) each inner node has two children and is assigned an iSAX summary which represents all the series stored in the nodes below it; and (iii) each leaf node stores the iSAX summaries of several data series, as well as pointers to these series. When the number of series in a leaf node becomes greater than the maximum leaf capacity, the leaf splits: it becomes an inner node and creates two new leaves, by increasing the cardinality (i.e., number of bits) of the iSAX summary of one of the segments (the one that will result in the most balanced split of the contents of the node to its two new children [10, 83]). The two refined iSAX summaries (new bit set to *0* and *1*, respectively) are assigned to the two new leaves. In our example, which illustrates the MESSI index tree, the series of Figure 1(c) will be placed in the outlined leaf node of the index (Figure 1(d)).

For query answering (see Figure 1(d)), these indexes first perform an *Approximate Search*: they traverse a path of the tree to find the best candidate series and compute an approximate answer by calculating the real distance, called BSF (Best-So-Far), between the query series S_q and this series. The best candidate series is in the leaf with the smallest lower bound distance to the query. Then, BSF is used to *prune* as many data series as possible from the collection.

In MESSI, this is done as follows. A number of *index workers* (i.e., distinct threads) start traversing the index subtrees (one after the other) using the BSF to decide which subtrees will be pruned. Each subtree is assigned to a single worker, so that the workers need to synchronize only on the choice of each subtree (this is achieved using a Fetch&Add object). The leaves of the subtrees that cannot be pruned are placed into (a fixed number of) minimum priority queues, ordered based on the lower bound distance between the PAA of the query series and the iSAX summary of the leaf node, in order to be further examined. (Threads are synchronized in accessing the priority queues by using locks.)

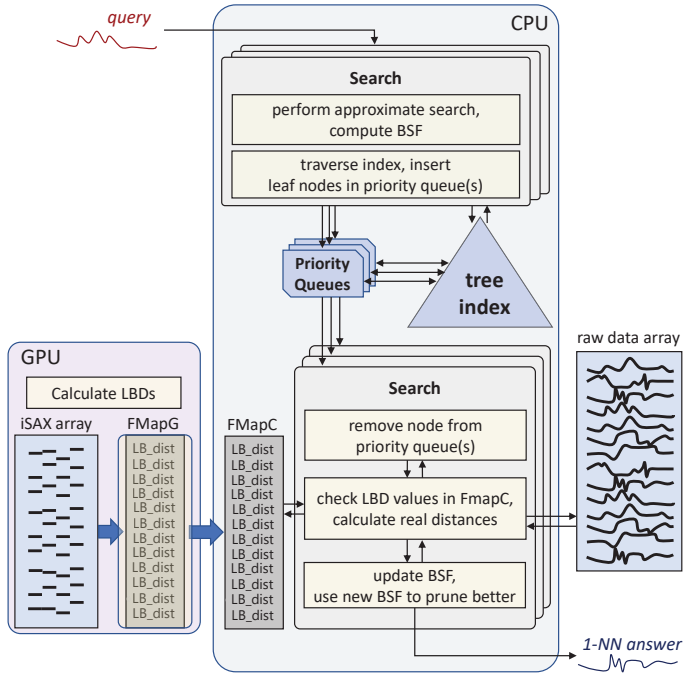


Figure 2: M+G flowchart for query answering.

As soon as the necessary elements have been placed in the priority queues, each index worker chooses a priority queue to work on, and repeatedly calls *DeleteMin()* on it to get a leaf node, on which it performs the following operations. It first checks whether the lower bound distance between the query and this leaf is larger than the current BSF: if it is then the leaf node does not contain any series that can be part of the answer, and can be pruned; otherwise, the worker needs to examine the series contained in the leaf node, by first computing the lower bound distance between the PAA of the query and the iSAX summaries of each of these series. If this lower bound distance is lower than the BSF, the worker also computes their real distance using the raw values. During this process, a series with a smaller distance to the query than the current value of the BSF can be discovered. Then, the BSF is updated accordingly. When a worker reaches a node whose distance is bigger than the BSF (and thus this node can be pruned), it gives up this priority queue and starts working on another, since all other elements in the abandoned queue have a higher distance to the query (and can be pruned). This process is repeated until all priority queues are processed, and the BSF is updated along the way. At the end, the value of the BSF is returned as the query answer. For details see [55, 57].

3 A Preliminary Solution

In this section, we focus on query answering. We describe *M+G*, a preliminary simpler solution than SING which could be seen as a first step in designing SING; we built upon M+G to get SING. Our experimental analysis reasons about the performance benefits of SING in comparison to M+G.

M+G stores the iSAX summaries of the data series (in the order they are stored in memory) in an array, called the *iSAX array*. At query time, the iSAX array has already been transferred in the GPU memory.

3.1 The M+G Baseline

[General Description] Like in all iSAX-based indexes, to answer a query in M+G, a CPU thread performs the approximate search and stores the first estimate of the answer it calculates in the BSF

Algorithm 1: M+G ExactSearch

```
1  Shared float BSF;                                              // best so far answer
2  Shared integer N_s = 0;                                        // root subtrees counter
   Input: QuerySeries QDS, Index index, Integer N_q, Integer N_c
                                                  // N_q: number of priority queues used
                                                  // N_c: number of CPU threads utilised
3  Float FMapC[], FMapG[];
                                                  // FMapG is copied in FMapC via streaming
4  QDS_PAA = calculate PAA for QDS;
5  BSF = ApproximateSearch(QDS_iSAX, index);
6  for i ← 0 to N_q − 1 do
7  |  index.queue[i] = Initialize the i-th priority queue;
8  end
9  ask GPU threads to perform LowerBoundDistanceCalculation(QDS_PAA, index, FMapG, BSF,
     index.datasize);
     // create a number of CPU threads to work conrurrently with the GPU threads
10 for i ← 0 to N_c − 1 do
11 |  create a thread to execute an instance of SearchWorkerM+G(QDS, index, i, N_q, FMapC);
12 end
13 if the GPU computation is done then
14 |  FMapC[]← FMapG[];
15 |  Barrier to synchronize with the search workers;
16 end
17 Wait for all search workers to finish;
18 return (BSF);
```

variable (refer to Figure 2).

Then, M+G transfers the PAA of the query and BSF in the GPU and instructs the GPU threads to calculate the distance between the iSAX summary of each entry of the iSAX array (which is already in the GPU) and the PAA of the query. The GPU outputs a float map (i.e., an array of float values) containing the lower bound distance for those iSAX summaries stored in the iSAX array. This float map is output to the CPU threads using streaming. Specifically, it is split into chunks and as soon as the data in a chunk becomes ready, the chunk is output to the CPU threads. The element of each row of the float map corresponds to the same-numbered row of the raw data array. This computation comprises the *lower bound distance calculation* phase and it is executed entirely in GPU.

At the same time (i.e. concurrently to the lower bound distance calculation phase), several CPU threads traverse the tree and create a number of priority queues, in the same way as in MESSI. Then, the CPU threads wait until the lower bound distance calculation has finished. Afterwards, the CPU threads start processing the priority queues. Specifically, each thread chooses a priority queue to work on and repeatedly deletes the node with the highest priority from it. If the node cannot be pruned, then for each data series stored in the node, the thread checks the lower bound distance stored in the float map for it. In this way, the CPU thread does not have to calculate the lower bound distance by itself. If the lower bound distance is larger than the current value of BSF, the data series is pruned. Otherwise, the real distance computation is performed. If any of these real distance computations results in a value smaller than the current value of BSF, then the BSF is updated to store the smaller value. This process continues until all nodes in the priority queues have either been processed or be pruned.

[Pseudocode] The pseudocode for M+G is provided in Algorithms 1-5. Recall that SAX, which stores the iSAX summaries for all data series, is already in the GPU memory. In all these algorithms, we assume that the index variable is a data structure containing a pointer to the root of the tree index, an array of N_q elements (which will eventually hold pointers to N_q priority queues), an integer $datasize$ determining the total number of data series in the raw data collection, and the iSAX array.

We start by discussing Algorithm 1. The shared variable BSF holds the current real distance value. N_s is a shared counter which indicates the root subtree to traverse next. The *exact search coordinator*

Algorithm 2: SearchWorkerM+G

19 **Shared integer** N_s; `// declared in Algorithm 1`

Input: QuerySeries QDS, **Index** $index$, **Integer** pid, **Integer** N_q, **Float** $FMapC$

20 **Integer** $q = pid \mod N_q$;

21 **while** *(TRUE)* **do**

22 $i \leftarrow$*Atomically* fetch and increment N_s;

23 **if** $(i \geq 2^w)$ **then** break;

24 *TraverseSubtree*(QDS, $index.rootnode[i]$, $index$, &q, N_q);

25 **end**

26 Barrier to synchronize with the search workers and the coordinator thread;
`// search workers synchronize with each other, with coordinator and GPU (through coordinator)`

27 $q = pid \mod N_q$;

28 **while** *(true)* **do**

29 *ProcessQueue*($QDS, index, index.queue[q], CMapC$);

30 **if** *index.queue[].finished=true, for all* $index.queue$ **then**

31 break;

32 **end**

33 $q \leftarrow$ index such that $index.queue[q]$ has not been processed yet;

34 **end**

Algorithm 3: TraverseSubtree

Input: QuerySeries QDS, **Node** $node$, **Index** $index$, **Integer** $*pq$, **Integer** N_q

35 $nodedist$ = *FindDist*(QDS, $node$);

36 **if** $nodedist > BSF$ **then**

37 break;

38 **else if** $node$ *is a leaf* **then**

39 acquire $index.queue[*pq]$ lock;

40 Put $node$ in $index.queue[*pq]$ with priority $nodedist$;

41 release $index.queue[*pq]$ lock;

`// next time, insert in subsequent queue`

42 $*pq \leftarrow (*pq + 1) \mod N_q$;

43 **else**

44 *TraverseSubtree*(QDS,$node.leftChild$, $index$, pq, N_q);

45 *TraverseSubtree*(QDS,$node.rightChild$, $index$, pq, N_q);

46 **end**

(i.e., the process executing Algorithm 1) first performs an approximate search (line 5) (using the tree index). This results in an initial upper bound on the actual distance between the query and the raw data series, which is stored in BSF. Then, it initializes N_q priority queues (lines 6-8). (We evaluate the effect of N_q in performance in Section 169).

Then, the exact search coordinator transfers the query PAA and the BSF to the GPU, and instructs the GPU threads to calculate the lower bound distances between the iSAX summaries stored in iSAX array and the query series PAA (line 9), by calling the $LowerBoundDistanceCalculation$ function.

As existing CPU solutions for lower bound distance camputation were leading to significant performance penalties when executed in the GPU, this algorithm was specifically designed for GPU computation in SING, and comprises one of its novelties. We discuss it in detail in Section 149.

The search coordinator, next, creates N_c CPU threads, called the *search workers* (lines 10-12), which will be executed concurrently with the GPU threads to create the N_q priority queues containing the nodes that require further examination. Each of these threads executes an instance the SearchWorker routine and terminates.

Periodically, the exact search coordinator synchronizes with the GPU to discover when the GPU finishes its computation. Then, it reaches a barrier to inform the search workers that the computation in the

Algorithm 4: ProcessQueue

```
    Input: QuerySeries QDS, Index index, Queue Q, FloatMap CMapC
47  while node = DeleteMin(Q) do
48      if node.dist < BSF then
49          realDist = CalculateRealDistance(QDS, index, node, CMapC);
50          acquire BSFLock;
51          if realDist < BSF then
52              BSF = realDist;
53          end
54          release BSFLock;
55      else
56          Q.finished = true;
57          break;
58      end
59  end
```

Algorithm 5: CalculateRealDistance

```
    Input: QuerySeries QDS, Index index, node node, FloatMap FMapC
60  Integer realDist = BSF;
61  for every (isax, pos) pair ∈ node do
62      if FMapC[pos] < BSF then
63          dist =
64          RealDist_SIMD(index.RawData[pos], QDS); if dist < BSF then
65              realDist = dist;
66          end
67      end
68  end
69  return (realDist)
```

GPU is completed and the computed lower bound distances have been stored in $FMapC$ (line 15). Note that the exact search coordinator uses two float maps, one ($FMapG$) that is written by the GPU threads and another ($FMapC$) that is read by the search workers. As soon as the exact search coordinator discovers that the GPU has completed its computation (and before reaching the barrier), it copies $FMapG$ to $FMapC$ to make its data accessible to the search workers. Finally, the exact search coordinator waits for all search workers to finish (line 17) their execution. Then, it returns the current value of BSF as the query response (line 18).

The pseudocode for SearchWorker is presented in Algorithm 2. Each search worker executes two phases, one called *priority queue preparation* phase (lines 21-25), and another called *priority queue processing* phase. Between the two phases, a search worker has to synchronize with the other search workers and the exact search coordinator by reaching a barrier (line 26). Thus, all search workers start executing the priority queue processing phase after all other search workers have completed the priority queue preparation phase and after the GPU is done and all the required data are stored into $FMapC$.

In the priority queue preparation phase, the search workers traverse the index tree and calculate the distance of the iSAX summary of each of the visited nodes to the the query PAA. If this distance is higher than BSF, then the entire subtree can be pruned. Otherwise, the leaves of the subtree with a distance to the query smaller than the BSF, are inserted in the priority queue. Roughly speaking, after a search worker identifies the queue where it will perform its first insertion (line 20), it repeatedly chooses a root subtree of the index tree to work on and process the nodes of this subtree by calling $TraverseSubtree$ (line 24). Synchronization among the different threads in accessing the root subtrees, is achieved using Fetch&Add, whereas in accessing the priority queues using locks (Algorithm 3, lines 39 and 41).

After all root subtrees have been processed and $FMapC$ is ready (line 26), a search worker starts executing the priority queue processing phase. It repeatedly chooses a priority queue (lines 28, 34) to work on and process its nodes by calling ProcessQueue (line 29).

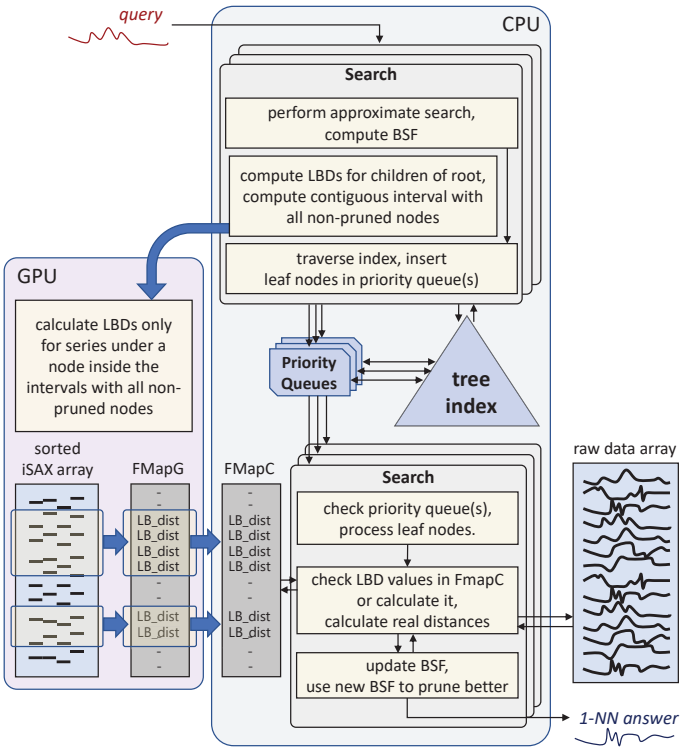


Figure 3: SING flowchart for query answering.

The pseudocode for $CalculateRealDistance$ is presented in Algorithm 5. It uses $FMapC$ whenever a lower bound distance is needed. Note that we perform the real distance calculations using SIMD.

4 SING

SING builts upon M+G. The GPU threads in SING compute a float map containing lower bound distances, which they output to CPU threads using streaming. However, in contrast to M+G, SING applies an initial pruning scheme to reduce the number of computations the GPU performs.

The CPU threads work concurrently to create the priority queues. However, in M+G, CPU threads wait for the GPU computation to complete before processing the priority queues, whereas in SING the CPU threads start this processing right after they finish the creation of the priority queues. This ensures that they do not stay idle during the time when the GPU performs computations, and results in additional performance benefits. Below, we provide details on the different design decisions made in SING.

4.1 Overview

We performed an experimental analysis to figure out the main overheads of M+G. The analysis revealed that the GPU computation time is often higher than the time the CPU threads require to create the priority queues. Thus, in M+G, the search workers (CPU threads) wait for the GPU to complete its computation which is wasteful in terms of computational resources. SING takes a number of measures to address this problem, which we discuss below.

The SING query answering approach, illustrated in Figure 3, adopts a new pruning strategy in order to efficiently support CPU+GPU co-processing. It applies an initial pruning technique which determines which part of the iSAX array is actually necessary to be processed by the GPU. This is done as follows. Before instructing the GPU to calculate lower bound distances, the CPU threads compute the lower bound distances between the query PAA and each of the tree root children. In this way, they identify a

collection of consecutive subtrees that cannot be pruned. Only these subtrees need to be processed.

If we maintained the iSAX array as in M+G, the iSAX summaries of the data series stored in the leaves of each sequence of consecutive subtrees of this collection would reside in scattered positions of the array. That would disallow their fast GPU processing. To address this problem, SING follows a different strategy for building the iSAX array than M+G. Specifically, as soon as the tree index is constructed, SING performs a recursive traversal of the tree, which visits the tree leaves from the leftmost to the rightmost (in this order)[5] and stores in the iSAX array, the iSAX summaries of all the data series stored in the leaves (in order). It is now chunks of consecutive elements of the iSAX array that need to be processed by the GPU, solving the problem above. In addition, this design decision allows the CPU threads to access consecutive elements of the float map computed by the GPU instead of performing random accesses in it. This reduces the number of cache misses caused during the priority queue processing phase; in this phase, the float map is accessed in order to check whether the elements contained in deleted nodes of the priority queues can be pruned. This is so since the elements of a node are examined in order, and therefore, it is beneficial to have their lower bound distances stored in the float map in this order.

The GPU is then instructed to calculate lower bound distances for each of the chunks of the iSAX array that need to be processed. Therefore, the GPU threads calculate lower bounds only for the series in the subtrees that cannot be pruned. Consequently, the number of lower bound distance calculations that are performed by the GPU in SING is much smaller than in M+G, and thus, the GPU execution time is also reduced.

Experiments show that, even after applying the techniques described above, the GPU computation time is often higher than the time the CPU threads require to create the priority queues. In M+G, the search workers (CPU threads) wait for the GPU to complete its computation which is wasteful in terms of computational resources. In constrast, in SING, the search workers start processing the elements in the priority queues they create without waiting for the GPU computation to complete. As soon as a search worker discovers that the priority creation phase has been completed by all CPU threads, it proceeds immediately to the priority queue processing phase. It chooses a priority queue to work on and calls *DeleteMin* to process the root element, i.e. the highest priority element in the queue. It first checks if the GPU computation has already performed the lower bound distance calculations for the elements of this node. If not, the search worker calculates these lower bounds itself, and then moves on to the next node to work on. If yes, the search worker uses the lower bound distances calculated by the GPU to prune if possible, decides whether further examination of the elements of the node is necessary (by calculating real distances), and moves on to the next node to work on.

To achieve this overlap of the queue processing phase with the GPU computation, the GPU float map is split into chunks, and the GPU outputs each chunk to the CPU threads, as soon as the lower bound distance calculations of the elements stored in the chunk have been performed. We denote by N_l the total number of chunks that the iSAX array (and the corresponding GPU float map) comprise.

4.2 Detailed Description

The pseudo code for SING is provided in Algorithms 6-11. We start by describing Algorithm 6. After executing an approximate search to get the BSF value (lines 75-76) (and after initializing the priority queues in lines 77-79), SING creates a number of N_c CPU threads, each of which executes an instance of $SINGWorker$ (Algorithm 7). SING employs the shared variables BSF and the counter N_s to count the processed root subtrees (lines 70 and 71), as well as the constant N_q to identify the number of priority queues. It also employs the float maps, $FMapG$ (in GPU space), where the GPU stores the lower bound distances it calculates, and $FMapC$ (in CPU space) where the GPU transfers these distances in order to be used by the CPU threads.

In $SINGWorker$ (Algorithm 7), a thread first executes $FindIntervals$ to identify the subtrees of the index tree that cannot be pruned. The sequence of root's children of the index tree is split into N_c equal parts and each part is assigned to a distinct thread. Thus, each thread examines the same number

[5] An inorder traversal of the root subtrees from the leftmost subtree to the rightmost subtree would accomplish this task.

Algorithm 6: $SING$

70 **Shared float** BSF; // best so far answer
71 **Shared Integer** $N_s = 0$; // root subtrees counter
72 **Shared Integer** N_f=0; // number of lbd computations completed by GPU
Input: QuerySeries QDS, **Index** $index$, **iSAX summarizations** SAX[], **Integer** $offset$[], **Integer** N_l, **Integer** N_q, **Integer** N_c
// N_l: number of chunks
// N_q: number of priority queues used
// N_c: number of CPU threads utilised
73 **Float** $FMapC$[], $FMapG$[];
74 **Integer** $w = index.w$; // number of segments
75 QDS_PAA = calculate PAA for QDS;
76 BSF = *ApproximateSearch*(QDS_PAA, $index$);
77 **for** $i \leftarrow 0$ **to** $N_q - 1$ **do**
78 | $queue[i]$ = Initialize the i-th priority queue;
79 **end**
80 **bool *** $activenodearray[], activechunk[]$;
// Create the CPU threads
81 **for** $i \leftarrow 0$ **to** $N_c - 1$ **do**
82 | create a thread to execute an instance of *SINGWorker*(QDS, $index$, $queue$[], i, N_q, $activenodearray$[], $activechunk$[]);
83 **end**
84 barrier to synchronize with CPU workers;
// ask GPU to compute lbd in batches
85 **for** $i \leftarrow 0$ **to** $N_l - 1$ **do**
86 | **if** *activechunk[i]* **then**
87 | | *LowerBoundDistanceCalculation*(QDS_PAA, $index$, $\&FMapG[index.datasize * i/N_l]$, BSF, $index.datasize/N_l$);
88 | | $FMapC[] \leftarrow FMapG[]$;
89 | | N_f=$(i+1)$*$datasize/N_l$;
90 | **end**
91 **end**
92 Wait for all SING workers to finish;
93 **return** (BSF);

of root children of the index tree. We denote by N_r the number of the non-null children of the root of the tree index. Let $s = N_r/N_c$ be the number of root subtrees that will be examined by each thread. Finally, a thread executing $SINGWorker$ calls $SearchWorker$ to act as a cpu worker, thus creating and processing priority queues.

During $FindIntervals$ (Algorithm 8), a thread identifies the subtrees that cannot be pruned from those assigned to it. This is done based on their lower bound distances (i.e., those for which the distance between the iSAX representation of the node and the query PAA is lower than the BSF). We call the roots of these non-pruned subtrees *active nodes*, and we call *active chunks* the chunks containing series that correspond to these active nodes (line 100); refer to Figure 4.

Once all threads finish the execution of $FindIntervals$ (line 84 of Algorithm 6 and line 95 of Algorithm 7), and the barrier of Algorithm 6 is met, then SING solicits the GPU to calculate the lower

Algorithm 7: SINGWorker

Input: QuerySeries QDS, **Index** $index$, **Queue** $queue$[], **Integer** pid, **Integer** N_q, **Bool*** $activenodearray$[], **Bool*** $activechunk$[]
94 *FindIntervals*(QDS, pid, $index.rootnode$[], $index.offset$[], $activenodearray$[], $activechunk$[]);
95 barrier to synchronize with other CPU workers and the thread executing Algorithm 6;
96 *SearchWorker*(QDS, $index$, $queue$[], pid, N_q, $activenodearray$[]);

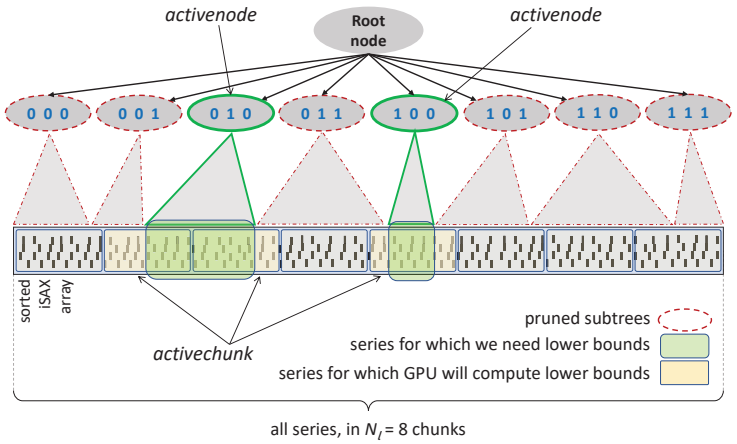


Figure 4: Illustration of the subtree pruning and identification of the iSAX array intervals for which the GPU needs to compute lower bound distances (Algorithm 8: $FindIntervals$), as well as the division of the iSAX array in chunks that will drive the above process (Algorithm 6: lines 85-91).

Algorithm 8: FindIntervals

Input: **QuerySeries** QDS, **Integer** pid, **node*** $rootnode[]$, **Integer*** $offset[]$, **Bool***$activenodearray[]$, **Bool*** $activechunk[]$

```
                                    // s: number of root subtrees a thread examines
97  for i ← 1 to s do
98      if (FindDist(QDS, rootnode[pid * s + i])< BSF) then
99          activenodearray[i]=True;
100         activechunkarray[offset[i]]=True;
101     else
102         activechunkarray[offset[i]]=False
103     end
104 end
```

bound distances for the series belonging to the chunks that must be examined (line 85 of Algorithm 6), transfer back the result (line 88), and update a shared variable, N_f, that keeps track of the GPU progress (line 89). In particular, N_f keeps track of the number of lower bound distance calculations in $FMapC$ that the GPU has completed. Note that we make several requests to the GPU to compute lower bound distances, each request involving a chunk of $DataSize/N_l$ lower bound distance calculations, where $DataSize$ is the total number of input data series. We adopt this strategy, because issuing a single request for the entire interval would mean that the CPU could only get updated on the progress of GPU lower bound distance calculations at the end, once these computations were over. On the other hand, issuing too many requests would diminish the data stream processing efficiency of the GPU. (We evaluate the effect of N_l in performance in Section 169).

The pseudocode for the SING search workers is shown in Algorithm 9. Note that these workers execute in parallel to the GPU threads performing the lower bound distance calculations. The CPU traverses the active nodes, i.e., the subtrees of the index root that were not pruned earlier (line 111), and populates the priority queues with the leaf nodes that cannot be pruned based on their lower bound distances (line 112). When all priority queues are ready, the workers start processing them by executing the $ProcessQueueHybrid$ function (line 118).

$ProcessQueueHybrid$ (Algorithm 10) first executes a *DeleteMin* operation on the priority queue (line 125), and then checks if the GPU has already finished computing the lower bound distances for the

Algorithm 9: SearchWorker

```
105 Shared integer N_s;                                          // declared in Algorithm 6
106 Shared integer N_f;                                          // declared in Algorithm 6
    Input: QuerySeries QDS, Index index, Queue queue[], Integer pid, Integer N_q,Bool* activenodearray[]
107 q = pid mod N_q;
108 while (TRUE) do
109     i ←Atomically fetch and increment N_s;
110     if (i ≥ 2^w) then break;
111     if activenodearray[i] then
112         TraverseSubtree(QDS, index.rootnode[i], queue[], &q, N_q);
113     end
114 end
115 Barrier to synchronize the search workers;
116 q = pid mod N_q;
117 while (true) do
118     ProcessQueueHybrid(QDS, index, index.queue[q]);
119     if all queue[].finished=true then
120         break;
121     end
122     q ← index such that queue[q] has not been processed yet;
123 end
```

Algorithm 10: ProcessQueueHybrid

```
124 Shared integer N_f;                                          // declared in Algorithm 6
    Input: QuerySeries QDS, Index index, Queue Q
125 while node = DeleteMin(Q) do
126     if node.dist < BSF then
127         CalculateRealDistanceSort(QDS, index, node,N_f);
128     else
129         Q.finished = true;
130         break;
131     end
132 end
```

data series of the deleted node (which has the highest priority). It then calculates the real distances for those series that cannot be pruned in this node (line 127; and Algorithm 11, line 140).

Pseudocode for *CalculateRealDistanceSort* is provided in Algorithm 11. This function calculates real distances between series in a leaf node and the query series, QDS. First, it checks if the part of $FMapC$ that corresponds to this node has been computed (i.e., the GPU has already computed the lower bound distances for the series in this leaf node (line 135). If this is so, the search worker directly uses the lower bound distances recorded in $FMapC$. Otherwise, the worker computes the lower bound distances by itself. Note that we only need to calculate the real distances for the series whose lower bound distances are less than BSF (line 140).

4.3 GPU-friendly Lower Bound Distance Computations

In this section, we present the new algorithm that SING employs to calculate lower bound distances using the GPU.

Like all iSAX based indices [49], ParIS+ [56] and MESSI [55,57] use a dictionary (lookup table) of break points in order to facilitate the computation of the lower bound distances. A break point is the value that separates two neighboring iSAX symbols, and the break point values for all iSAX symbols (depicted as horizontal lines in Figure 1(c)) are precomputed, and stored in a lookup table, or dictionary. We then

Algorithm 11: CalculateRealDistanceSort

```
133 Shared float BSF;
    Input: QuerySeries QDS, Index index, node node, Integer N_f
134 for i ← 0 to size of node do
135     if node.offset > N_f then
136         ldist=LoweroundDist_SIMD(QDS, node.isax[i]);
137     else
138         ldist=node.FMapC[node.offset + i]
139     end
140     if ldist < BSF then
141         dist =
142         RealDist_SIMD(index.RawData[pos[i]], QDS);
143         acquire BSFLock;
144         if dist < BSF then
145             BSF = dist;
146         end
147         release BSFLock;
148     end
149 end
```

access this dictionary every time we need these values, e.g., when computing lower bound distances [67]. We note that the iSAX break points were calculated according to the Gaussian distribution, so that all iSAX symbols of a Z-normalized collection of data series are equi-probable [67].

Nevertheless, the above dictionary-based solution incurs significant delays when implemented in the GPU, because searching for and reading the break point values in a dictionary is an expensive operation when executed in a Streaming Processor (GPU core). The reason is that the registers each thread has access to cannot be used for data dependent lookups [3]. Therefore, the dictionary needs to be stored in and accessed at the Streaming Multiprocessor shared memory, which is one order of magnitude slower than the Streaming Processor registers [14].

In contrast to all previous iSAX based indices [49], SING follows a different approach for performing each lower bound distance calculation. In the solution that we propose, illustrated in Figure 5, the GPU directly calculates the break point values of the iSAX symbols using a function, called *BreakPoly* (break point polynomial), that approximates the iSAX break point values. This approach avoids the high memory latency cost, thus requiring less time to perform the lower bound distance calculation phase. The *BreakPoly* function is a third degree polynomial, $ax^3 + bx$, with constants $a = 5.418 * 10^{-7}$ and $b = 7.797 * 10^{-4}$ (estimated using the Matlab *polyfit* function). Figure 6 depicts the iSAX and *BreakPoly* break points for an iSAX representation of cardinality 256. Our experimental evaluation shows that the use of *BreakPoly* leads to a significant speedup without a loss in accuracy (refer to Section 169).

The $LowerBoundDistanceCalculation$ algorithm that calculates the lower bound distances in the GPU is shown in Algorithm 12. For each segment of each series, the algorithm first computes the lower BP_l and upper BP_u break points (lines 155 and 161) using the *BreakPoly* function. It then checks if the current (intermediate) distance value is larger than BSF (line154), in which case it skips the rest of the calculations (early termination), writes the current lower bound distance in the appropriate position of $FMapG$ (line 168), and starts working on the next series. Otherwise, it finishes the computation of the

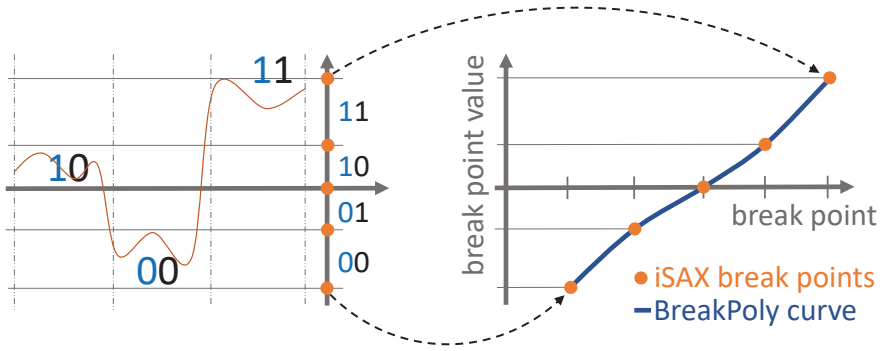


Figure 5: Functional representation of the iSAX break points.

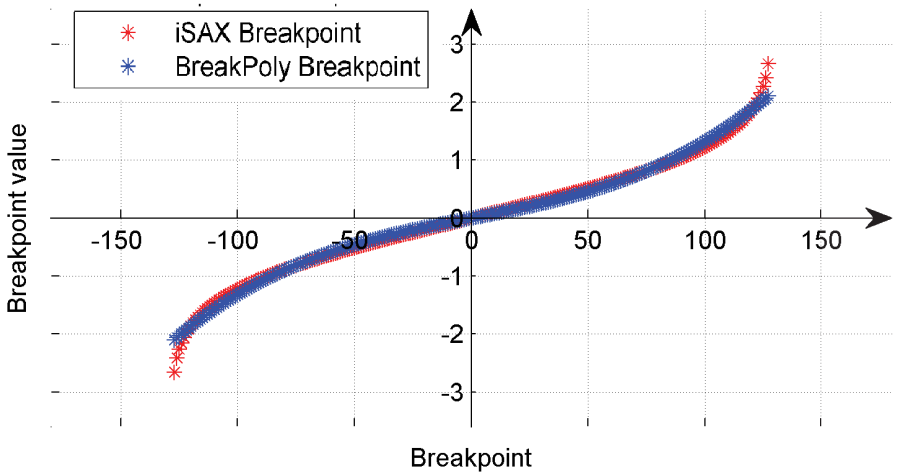


Figure 6: The iSAX (red/light) and *BreakPoly()* break points (blue/dark) for an iSAX representation with 256 symbols (the graph shows the values for the corresponding 255 break points).

lower bound distance for this series, and records the result in $FMapG$.

5 Experimental Evaluation

5.1 Setup

We used a server with 2x Intel Xeon Gold 6134 CPUs with 8 cores each, 320GB RAM and a Titan Xp GPU with 3840 NVIDIA CUDA Cores (12GB RAM), which represents a middle-range, commodity GPU.

Given that the iSAX summaries we need to store in the GPU memory are 64x smaller than the raw data[6], our GPU can accommodate the summaries of a series collection larger than 0.7TB. Similarly, modern GPUs with 48GB RAM can serve series collections larger than 3TB. We note that all raw data are represented using 32-bit floats.

All algorithms were implemented in C and C++, and compiled using GCC v7.4.0 and NVCC 10.1 on Ubuntu Linux v18.04. We use NVCC to compile the GPU part of the code as a lib file; we then link this lib to the main function of the program. The code for all algorithms is available online [2].

[Algorithms] We compared SING to the following algorithms: (i) MESSI [55], the state-of-the-art

[6]In this example, the 16 segments of the iSAX representation occupy 16 bytes, and summarize a series of 256 points that occupy 1KB (assuming 32-bit floats).

Algorithm 12: LowerBoundDistanceCalculation

```
Input: QuerySeries QDS_PAA, Index index, Float FMapG, Float BSF, Integer N_series
                                        // N_series: number of series to process
Integer w = index.w;                                        // number of segments
for i ← 0 to N_series − 1 do
    Float distance = 0;
    for j ← 0 to w − 1 do
        if distance<BSF then
            Float BP_l=BreakPoly(SAX[i * w + j]);
            if QDS_PAA[j] < BP_l then
                distance+ = (BP_l − (QDS_PAA[j])^2;
            else
                Float BP_u=BreakPoly(SAX[i * w + j] + 1);
                if QDS_PAA[j] > BP_u then
                    distance+ = (BP_u − (QDS_PAA[j])^2;
                end
            end
        else
            break;
        end
    end
    FMapG[i]=distance;
end
```

modern hardware data series index. (ii) UCR Suite-P, our parallel implementation of the state-of-the-art optimized serial scan technique, UCR Suite [61]. In UCR Suite-P, every thread is assigned a part of the in-memory data series array, and all threads concurrently and independently process their own parts, performing the real distance calculations in SIMD, and only synchronize at the end to produce the final result. (We do not consider the non-parallel UCR Suite version in our experiments, since it is almost 300x slower.)
(iii) UCR Suite GPU [64], a similarity search solution, where all computations take place in the GPU.
(iv) M+G, our baseline solution based on MESSI (cf. Section).
(v) Finally, we compare to P+G, our baseline solution that extends ParIS+ [56]. P+G performs an approximate search and then uses the GPU to calculate all the lower bound distances (just like M+G). The difference is that the GPU returns a *bitmap* representing the series that could not be pruned, for which the CPU calculates the real distances. The bitmap is streamed from the GPU to the CPU, so that both work in parallel. The CPU accesses the raw data in memory in a skip-sequential manner (there is no need for re-ordering the data, like in SING), and updates the BSF variable in both the CPU and the GPU.

SING, M+G, and P+G use both the CPU and GPU, while MESSI and UCR Suite-P only use the CPU. All operations are exclusively in memory: the index tree and raw data were already loaded in the main memory of the system, and the SAX array was already loaded in the GPU memory (for SING, M+G, and P+G).

[Datasets] In order to evaluate the performance of the proposed approach, we use several synthetic datasets for a fine grained analysis, and two real datasets from diverse domains. Unless otherwise noted, the series have a size of 256 points, which is a standard length used in the literature, and allows us to compare our results to previous work. All our datasets are Z-normalized[7].

[7]Z-normalization transforms a series so that it has a mean value of zero, and a standard deviation of one. This allows similarity search to be effective, irrespective of shifting (i.e., offset translation) and scaling [32]. Therefore, similarity search can return results with similar trends, but different absolute values. Moreover, minimizing the Euclidean distance on Z-

We used synthetic datasets of sizes 50GB-200GB (with a default size of 100GB), and a random walk data series generator (following previous studies [18, 19, 85]) that works as follows: a random number is first drawn from a Gaussian distribution N(0,1), and then at each time point a new number is drawn from this distribution and added to the value of the last number. This kind of data generation has been extensively used in the past (and has been shown to model real-world financial data) [10, 67, 71, 76, 83]. We used the same process to generate 100 query series.

For our first real dataset, *Seismic*, we used the IRIS Seismic Data Access repository [1] to gather 100M series representing seismic waves from various locations, for a total size of 100GB. The second real dataset, *Astro*, includes data from astronomy data series [70], for a total of 100M series of size 256, resulting in a total data size of 100 GB. Note that in all cases, the raw data and the index are stored in the CPU memory, while the iSAX representations for all series in the dataset are stored in the GPU memory[8].

In both cases, we used as queries 100 series that were not part of the datasets (produced using our synthetic series generator, since these datasets do not come with query workloads). Moreover, we report results with query workloads of increasing difficulty (similarly to earlier work [85]). For these workloads, we select series at random from the collection, add to each point Gaussian noise ($\mu = 0$, $\sigma = 0.01$-0.2), and use these as our queries; we label them with the σ value expressed as a percentage 1%-20% (remember that $\sigma = 1$ for our Z-normalized data).

In all cases, we repeated the experiments 10 times and we report the average values. We omit reporting the error bars, since all runs gave results that were very similar (less than 3% difference). Queries were always run in a sequential fashion, one after the other, in order to simulate an exploratory analysis scenario, where users formulate new queries after having seen the results of the previous one.

We note that in all cases, the answers produced by our algorithms are the exact, correct answers; the same is true for the competitors we compare against.

5.2 Parameter Tuning Evaluation

[Parameters N_q and N_l] We first present experiments, which evaluate the effect of the number of chunks, N_l, and number of priority queues, N_q, parameters on the time performance of the proposed algorithm.

Figure 7(a) shows the time performance results when we vary the N_l parameter. We observe that query answering time grows as N_l grows above 20. This is because a large number of chunks leads also to a large number of interruptions to the GPU streaming processing. On the contrary, when the number of chunks is very low, the GPU has less opportunities to communicate partial results to the CPU. Therefore, the CPU threads cannot start their work early. For example, having a single chunk leads to query answering time 46% larger than for $N_l = 20$.

Figure 7(b) depicts the results of varying the number of priority queues, N_q. Increasing N_q leads to lower query answering times, until the values converge. We observe that for $N_q > 32$ performance starts deteriorating, due to the contention in the priority queues.

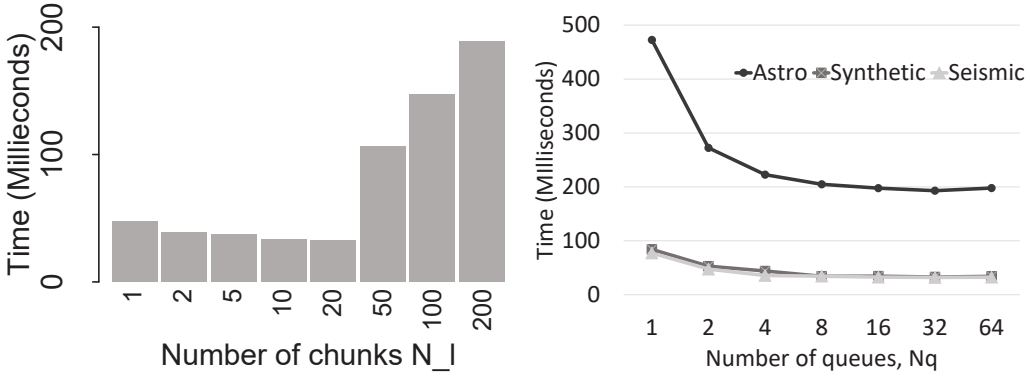
Given the above results, in the rest of the experimental evaluation, we use $N_l = 32$ and $N_q = 20$, which lead to the best overall performance. Note also that the observed performance remains relatively stable around these values.

[Function *BreakPoly()*] We now evaluate the performance of *BreakPoly()*, and compare it to the original solution of using a dictionary (lookup table). In both cases, the computations take place in the GPU.

Figure 8(a) shows the pruning effectiveness when using the *BreakPoly()* break points, when compared to the iSAX break points. We measure the number of non-pruned series that result from the application of the two break point approaches, for three different datasets. The results show that, overall, *BreakPoly()* does not alter the pruning characteristics of iSAX (it results in more pruning for the Seismic

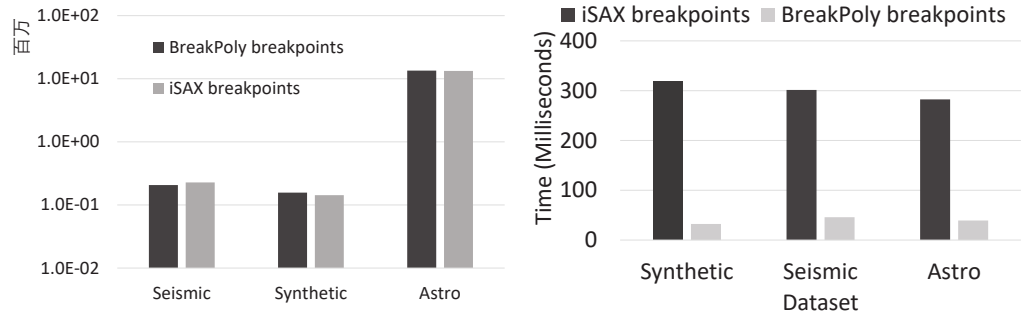
normalized data is equivalent to maximizing their Pearson's correlation coefficient [47]. For these reasons, Z-normalization is extensively used in both the literature [18, 19, 85] and in practice [6, 51].

[8]For a 100GB dataset, the iSAX summaries occupy less than 2GB of GPU memory. This means that our 12GB GPU memory could support query answering using SING for datasets as large as 700GB.



(a) Varying the number of chunks, N_l (b) Varying the number of priority queues, N_q

Figure 7: Query answering time (16 cores, 2 sockets; 100GB Synthetic dataset).



(a) Pruning effectiveness (number of non-pruned series) (b) Time to calculate 100 million lower bound distances

Figure 8: iSAX break-point values versus *BreakPoly()* break-point values (GPU).

dataset, and slightly less pruning for the Astro and Random datasets). Nevertheless, *BreakPoly()* is much faster, leading to 12x faster execution of the lower bound distances (see Figure 8(b)).

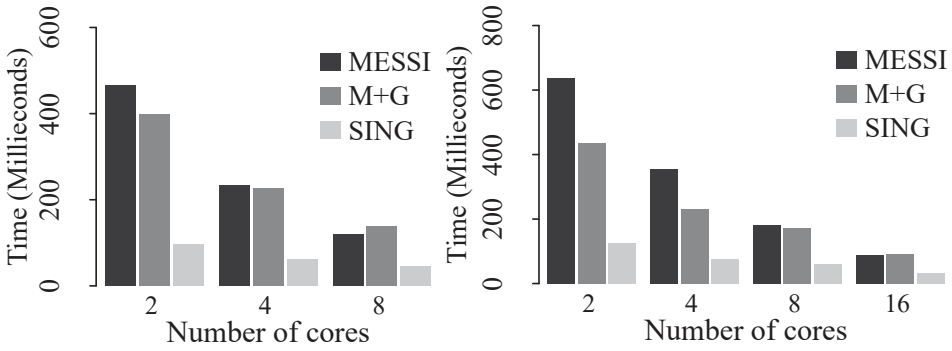
Figure 10 depicts the time performance breakdown for M+G and SING. The graphs shows that the choice of SING to spend time on preparing the lower bound distance calculation interval before the GPU starts computing the lower bounds is correct: these two times added together are less than the time it takes in M+G to execute *all* the lower bound distance calculations in the GPU. The graph also shows that SING achieves much better parallelism between the CPU and GPU, which also contributes to the time performance benefit of SING. SING is efficient between CPU and GPU. The workloads between CPU and GPU are generally balanced. Finally, we observe that the transfer of the results from the GPU to CPU is an operation that is for the most part overlapped with the GPU calculations, further contributing to the performance improvements of the proposed approach.

5.3 Comparison to Competitors

In the next set of experiments, we compare SING to our baseline solution and the current state-of-the-art approaches for exact data series similarity search.

Figures 9(a) and 9(b) report the execution times of MESSI, M+G and SING, as we vary the number of cores up to 8 in one socket, and up to 16 in two sockets. We observe that the performance of all algorithms improves when we use more cores. This improvement is more pronounced for MESSI, which starts from much higher execution times for small numbers of cores.

M+G can only beat the performance of MESSI when using a small number of cores (in either 1, or 2 sockets). In these cases, having the GPU calculate the lower bounds removes a heavy burden from the CPU, and translates to execution time savings. On the other hand, SING consistently outperforms the



(a) 1 socket; 100GB Synthetic dataset (b) 2 sockets; 100GB Synthetic dataset

Figure 9: Query answering time, varying the number of cores.

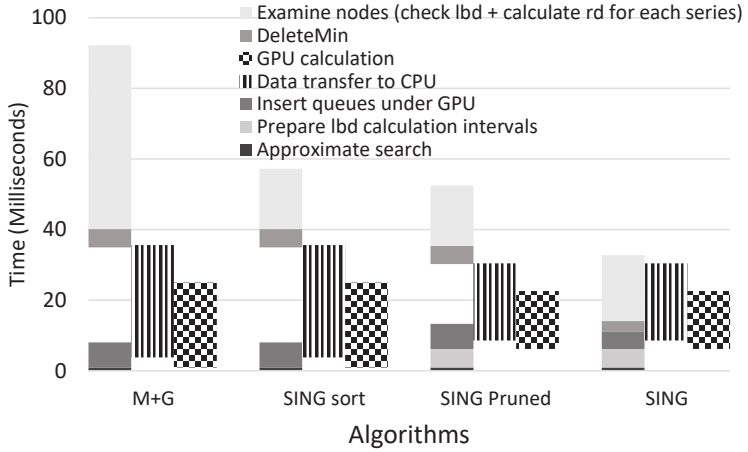


Figure 10: Query answering time performance breakdown for M+G and SING (16 cores, 2 sockets; 100GB Synthetic dataset). Bars depicted side-by-side denote operations that take place at the same time (in parallel).

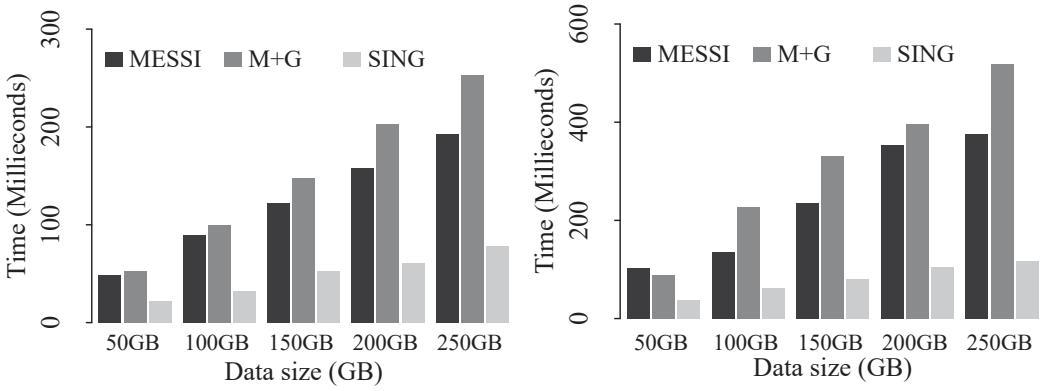
competitors across the board. SING is 5.1x faster than MESSI for 2 cores in 2 sockets. Even when we use all 16 cores of our system, SING is still 2.8x faster than MESSI. SING only needs 32ms to answer an exact similarity search query on a 100GB dataset.

In the next experiments, we evaluate the query answering time performance when varying the dataset size between 50GB-250GB. In Figure 11(a) we show results when all algorithms use the same number of cores (16 cores in 2 sockets), while in Figure 11(b) we show results when the algorithms use hardware with the same monetary value: MESSI uses 8 cores, while the SING and M+G use 4 cores and the GPU; this experiment helps us determine whether it is worth investing on a GPU, rather than on more CPU cores, for our problem. We observe that SING is once again faster than the competitors in all cases, and becomes increasingly faster as the dataset size grows.

Finally, we report the results of the comparison to the state-of-the-art parallel serial scan algorithm, UCR Suite-P[9]. Figure 13 (log-scale y-axis) reports the query answering time for three different datasets, with SING being up to 62x faster than UCR Suite-P. UCR Suite GPU is significantly slower, due to the cost of transferring the raw data in the GPU (remember that the size of the raw data is much bigger than the GPU memory). Thus, we omit this algorithm in the following.

[Performance Benefit Breakdown] In this part of the evaluation, we discuss the reasons behind the per-

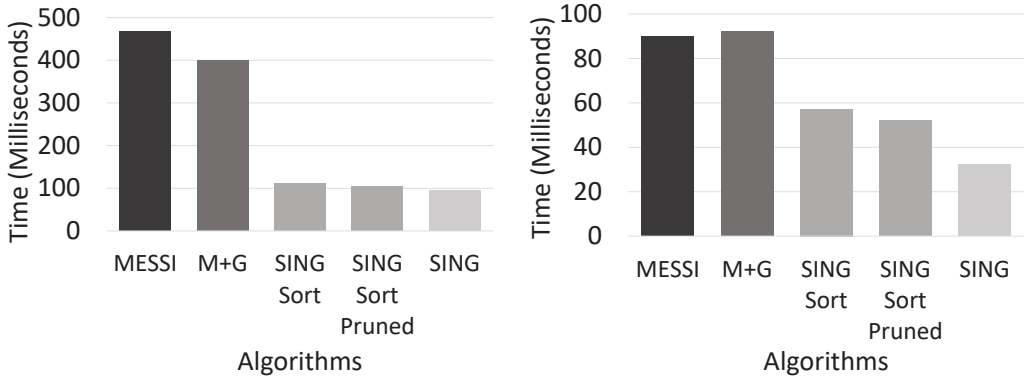
[9]Note that this algorithm was developed for subsequence matching, while in our case we are solving the problem of whole matching [18].



(a) 16 cores, 2 sockets; 100GB Synthetic dataset)

(b) hardware of same monetary value: MESSI uses 8 cores (1 socket); M+G and SING use 4 cores (1 socket) and the GPU (100GB Synthetic dataset)

Figure 11: Query answering time, varying the dataset size.



(a) 2 cores, 1 socket

(b) 16 cores, 2 sockets

Figure 12: Performance benefit breakdown.

formance benefits of SING, and we examine the impact of our design choices on the final performance outcome.

In Figure 12, we show the query answering times (for 2 cores in 1 socket, and 16 cores in 2 sockets) for several intermediate solutions leading to the final SING design. M+G is the first step after MESSI, where we move the lower bound distance calculations from the CPU to the GPU: M+G is 14% faster for 2 cores. SING Sort implements the sorted iSAX array idea, which leads the algorithm to perform mostly consecutive memory accesses. SING Sort is almost 3.6x faster for 2 cores, and 1.6x faster for 16 cores than M+G. SING Sort Pruned is an extension of the previous variation, where we prune the root subtrees based on their lower bound distances to the query, and only ask the GPU to calculate lower bounds for a subset of the iSAX array. This idea makes SING Sort Pruned 6% faster for 2 cores, and 10% faster for 16 cores. Finally, SING additionally divides the iSAX array (and the computation of the corresponding lower bounds) in chunks, which enables better parallelism between the CPU and the GPU. SING is 9% faster than SING Sort Pruned for 2 cores, and 21% faster for 16 cores.

In Table 1, we report a breakdown of the number of operation executions that the different algorithms perform. These numbers help explain the observations (and design choices) mentioned above. Note the large (and expected) difference in the lower bound distance calculation numbers. MESSI performs the fewest lower bound distance calculations (9% of the total), while M+G and SING Sort need to perform all lower bound distance calculations, which is a significant cost. SING Sort Pruned and SING reduce this cost by pruning a significant number of lower bound distance calculations: they perform 55% of the

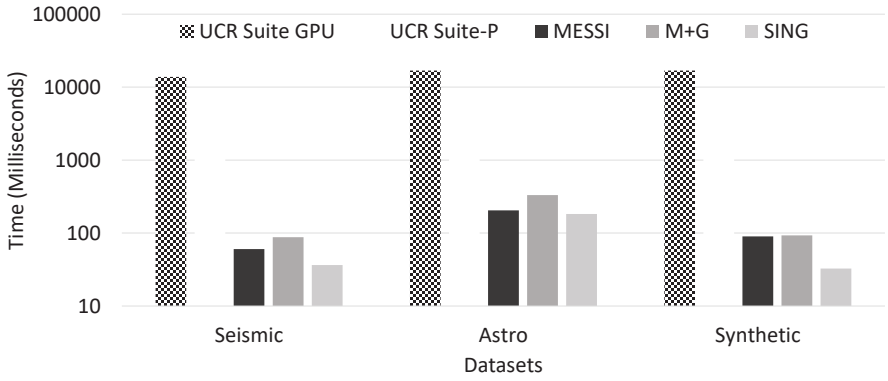


Figure 13: Query answering time, various algorithms (16 cores, 2 sockets; 100GB datasets).

Table 1: Query answering algorithms comparison: number of times an operation is executed (average over 100 queries; 100GB Synthetic dataset).

	MESSI	**M+G**	**SING Sort**	**SING Sort Pruned**	**SING**
PQ ins. node	15K	15K	15K	15K	15K
PQ del. node	11K	11K	11K	11K	11K
LBD calcul.	9M	100M	100M	55M	55M
RD calcul.	53K	52K	52K	52K	52K

total.

[Varying Query Workloads] In this set of experiments, we evaluate the different algorithms on the real datasets with a variety of query workloads, which range from relatively easy (can prune a lot) to relatively hard (cannot prune a lot).

Figures 14-17 show the experimental results for different number of cores. The shown results verify the superiority of SING over its competitors for different datasets and a wide range of settings, including small/large number of cores, and easy/hard queries. Figure 15 -17). In order to understand why this happens, we measure the number of lower bound (Figure 18) and real (Figure 19) distance calculations performed by each algorithm. We can see that, MESSI is the most effective in pruning unnecessary calculations, computing the least number of lower bounds. The reason is that MESSI uses the tree index to drive this operation, which allows the algorithm to visit the nodes in an *ad hoc* order, and prune both internal and leaf nodes. On the other hand, M+G has to perform all lower bound distance calculations, while SING manages to prune some of these calculations.

Figure 19 illustrates the number of real distance calculations, which are almost the same for all algorithms. Therefore, it is the number of lower bound distance calculations that play a crucial role in the overall performance of the algorithms. In this context, it is also interesting to examine the role of the priority queues: Figures 20-22 depict the number of (leaf) nodes inserted in and deleted from the priority queues by each algorithm. We observe that updating the BSF along the computation of the real distances is effective at pruning the nodes in the priority queues (the nodes that are *not* removed have been pruned, because their lower bound distance to the query is larger than the current BSF distance), and results in early termination of the execution.

In Figure 23, we present the query answering time results when the algorithms use hardware with the same monetary value with the respect to the hardware in our experimental setup: MESSI uses 8 cores, while the SING algorithms use 4 cores and the GPU. The graph shows that in this setting as well, SING is the best performer in most of the cases.

5.4 Complex Analytics Task: Classification

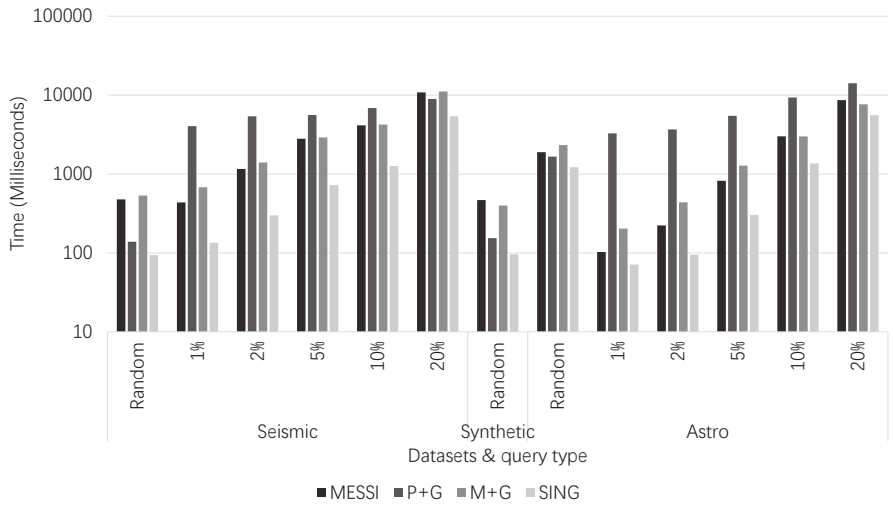


Figure 14: Query answering time for different datasets and query workloads (2 cores, 1 socket).

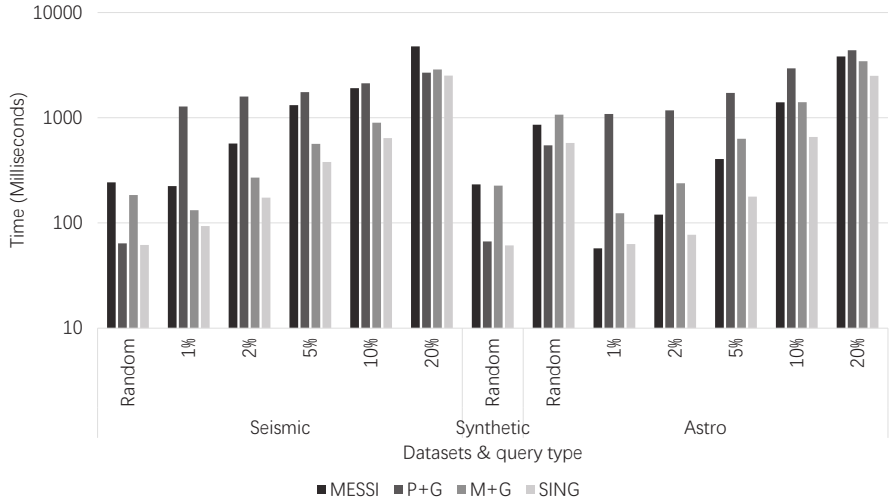


Figure 15: Query answering time for different datasets and query workloads (4 cores, 1 socket).

In the following experiment, we tested SING on a complex analytics task. In particular, we evaluated its performance in a classification task, and measured the benefit it would bring to a k-NN Classifier. This classifier assigns a new object to the majority class of the k nearest neighbors (NN) of that object (a data series, in our case).

The results, depicted in Figure 24, report the performance of SING and MESSI for different values of k on a 100GB dataset (100M series of size 256 values, generated with our synthetic data generator). The results show that a k-NN Classifier using SING can finish a classification task up to 2.4x faster than when using MESSI, which can reduce the total processing time for classifying 100K objects from 2.5 hours down to 62min. When we do the comparison with the same price hardware, SING completes the classification task up to 3.5x faster than MESSI on a 250GB dataset (Figure 25).

We note that the purpose of this experiment was to measure the time performance of executing a k-NN classification task. Even though we did not study a real classification problem, the results are useful in that they report the expected time performance of using SING in such a task with a large data series collection.

Finally, we evaluated the overhead of executing k-NN queries. SING implements k-NN by simply

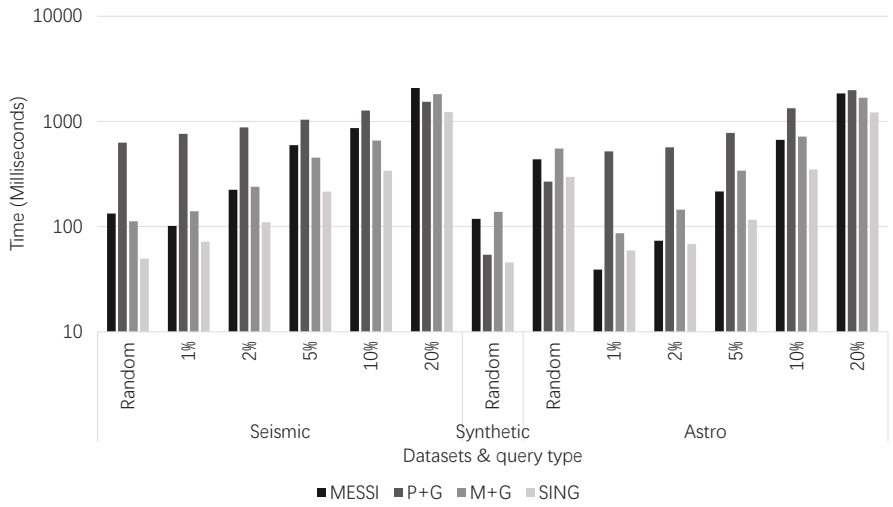


Figure 16: Query answering time for different datasets and query workloads (8 cores, 1 socket).

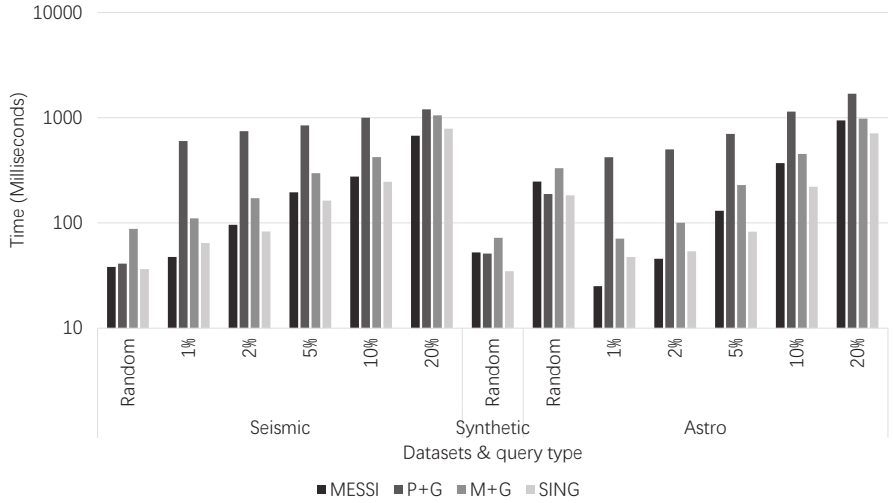


Figure 17: Query answering time for different datasets and query workloads (16 cores, 2 sockets).

maintaining a sorted array of the best k distances seen so far (i.e., BSF is now an array of k elements). The elements of the array are initialized by performing a single approximate search (like in 1-NN): we choose the k series with the smallest distances to the query and initialize the BSF array with their distances. Whenever a smaller distance than the biggest element of this array is calculated, the array is updated. This process does not result in more operations on the priority queues, or more tree Table 2 shows that the additional time for executing k-NN instead of 1-NN is negligible. The number of BSF updates per query remains small as k increases, thanks to the order in which we process the series, imposed by the priority queue. Consequently, the BSF update time cost remains small, as well: even for the 50-NN queries, the cumulative BSF update time accounts for a mere 0.4% of the total query answering time.

6 Related Work

There has been a flurry of activity, especially during the last years, related to the development of scalable data series similarity search techniques, especially for exact query answering [5, 9, 10, 12, 13,

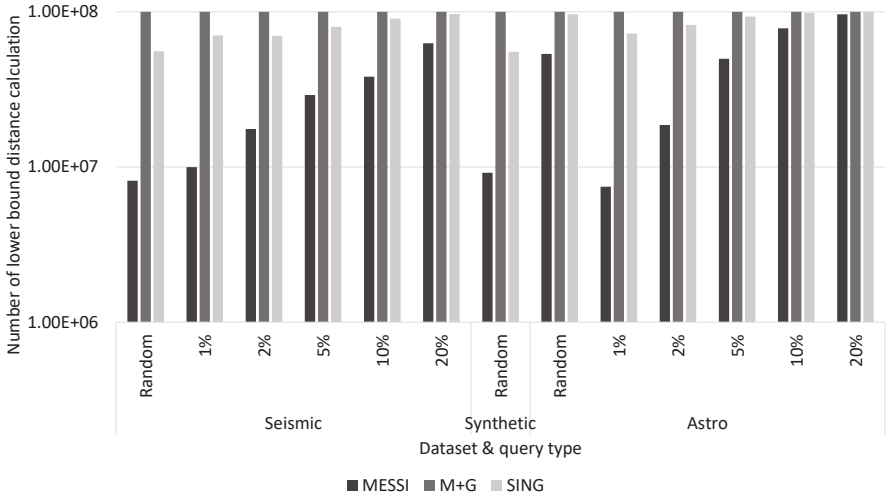


Figure 18: Number of lower bound distance calculations for different datasets and query workloads.

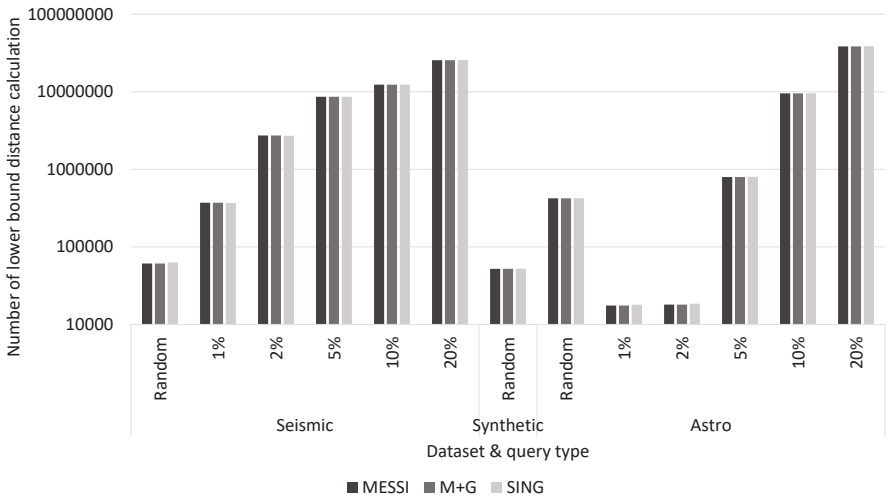


Figure 19: Number of real distance calculations for different datasets and query workloads.

36, 37, 42, 42, 43, 49, 54–56, 60, 65, 68, 71–74, 82, 83]. Nevertheless, none of these techniques considered the use of GPUs for performing part of the computations.

The problem of Nearest Neighbor (NN) queries in GPUs has been studied in the past (e.g., for the subgraph isomorphism problem [28, 77]), albeit, not for data series. We discuss some of these studies below. We note that in this work, we focus our attention to indexing structures specialized to data series, since other techniques cannot provide comparable performance in this high-dimensional context [18]. We also note the work of Sart et al. [64] that describes a GPU-based implementation of data series similarity search, albeit for subsequence matching (identify the matches of a short query series within a very long sequence). The focus of our work is on whole-matching, and as the experiments show, the subsequence matching solution is not suitable.

Gieseke et al. [20] propose the Buffer k-d Tree to process NN queries on a GPU. This approach reorganizes the querying process such that queries belonging to the same leaf of the index tree are processed together. The goal of this approach is to efficiently process together large batches of queries. In contrast, we focus on exploratory search, where queries arrive one by one: the results of an analyst's query determine what the next query will be. Nevertheless, we note that parallel query batch processing

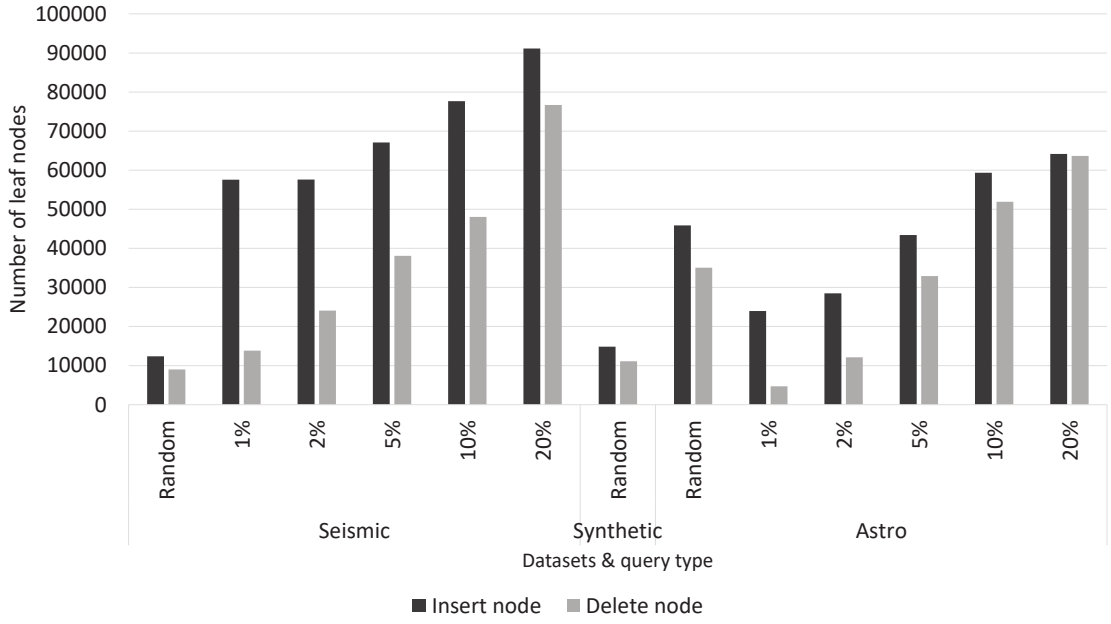


Figure 20: MESSI algorithm: number of leaf node insertions to/deletions from the priority queues for different datasets and query workloads.

Table 2: Update Frequency of the BSF array

	1-NN	**5-NN**	**10-NN**	**50-NN**
number of BSF updates/query	12.04	21.45	44.83	248.61
BSF update time μsec/query	0.63	8.68	23.03	224.86
BSF update time query time %	0.002 %	0.02 %	0.05 %	0.4 %

is an interesting future direction. Kim et al. [33] have proposed a tree-based index on GPUs for single-dimension integer data.

The use of GPUs in order to support spatio-temporal queries has been examined in the past [15, 40]. Doraiswamy et al. [15] describe STIG, a generalization of the Kd-Tree, that can quickly compute the result set of the queries. In this solution, the GPU is mainly used to accelerate the test of whether a (candidate) point lies within a (query) polygon. In a more recent work, Li et al. [40], design an update-efficient GPU accelerated grid index for k-NN queries for road networks. In this case, the index stores position in information about a large number of objects moving in a 2-dimensional space. The goal is to answer k-NN queries, while maintaining the correct positions of the objects as they get updated over time. They describe a lock free algorithm for the GPU, based on a special group of GPU functions named *warp shuffle*, which allow threads in the same thread group to exchange data at a very low cost. Their k-NN algorithm first uses the GPU to compute a candidate result set, and then moves the computation to the CPU to refine the candidate set and obtain the final answer. We also note that all the works addressing spatio-temporal queries propose and use indices designed for a 2-dimensional space, and there is no straight-forward way to apply it on data series with dimensionalities in the order of 100s. Moreover, earlier studies have shown that the indices used in these cases (such as grid-based, Rtrees, or Kd-Trees) do not perform well for the high-dimensional data series collections [18, 83].

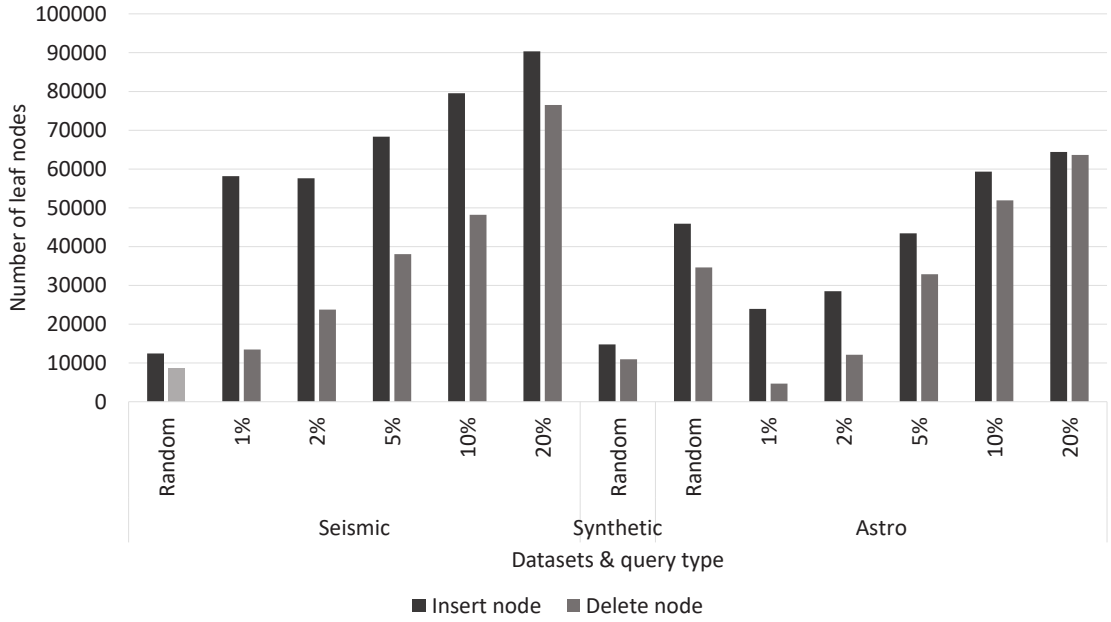


Figure 21: M+G algorithm: number of leaf node insertions to/deletions from the priority queues for different datasets and query workloads.

Previous work has considered the use of GPUs for speeding up similarity search using Locality Sensitive Hashing (LSH) [52, 53, 78]. Zhou et al. [78] in particular, propose a general framework based on an inverted index structure that operates in a GPU, which generalizes several types of similarity search queries. However, all these works only support approximate query answering. In our work, we focus on exact query answering that is required by several applications [51], and for which the full-dimensional raw data are needed.

Zhu et al. [79, 80], present a GPU implementation of *Matrix Profile*, an algorithm used to identify data series motifs (i.e., frequent subsequences). They present a base line solution, STAMP, followed by an optimized algorithm, STOMP, which avoids re-executing repeated calculations. Zimmerman et al. [81] extend the above work by describing a solution that operates on a cluster of distributed GPUs. We observe that Matrix Profile is used to reason about short subsequences within a long data series. The difference to our work is that we are interested in similarity search between a query series and a dataset containing a large number of data series. Matrix Profile is not applicable to our problem, for which a different set of techniques is needed.

Finally, earlier works have employed different notions of bit-wise decomposition for processing (single-dimensional) integer/real value data [41, 59], where the most significant bits of each value are used in a pruning step to reduce the required calculations on the fully detailed values. SING follows the same idea, with a much more elaborate mechanism, suitable for high-dimensional data series: it processes iSAX summaries in the GPU to prune the processing of the (much larger) raw data in the CPU.

7 Conclusions

Data series similarity search remains an important and challenging problem. In this work, we propose SING, the first data series index that answers similarity search queries by employing both CPUs and GPUs. SING takes into account the memory and data transfer rate limitations of GPUs, and adopts query processing strategies that effectively use all available computational resources. In our experi-

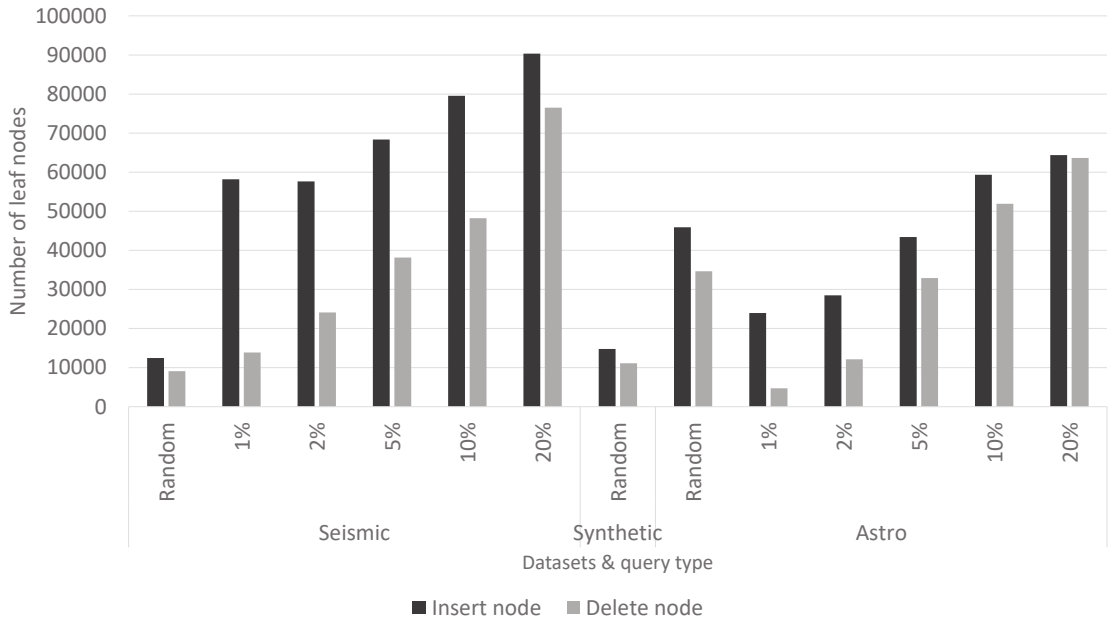


Figure 22: SING algorithm: number of leaf node insertions to/deletions from the priority queues for different datasets and query workloads.

ments with several synthetic and real datasets, SING extends considerably the scalability of similarity search, especially for low cost hardware (i.e., with a small number of cores). SING is up to 5.1x faster at query answering time than the state-of-the-art parallel in-memory approach, and up to 62x faster than the state-of-the-art parallel serial scan algorithm. The results demonstrate that SING achieves exact similarity search times at interactive speeds: as low as 32msec on 100GB datasets. In future work, we will study techniques that combine SING (parallelization in a single node) with distributed (across nodes) indexing [39, 72–74].

We note that recent advances in GPU hardware, most notably their increasing memory capacity and interconnect speeds, open up new research directions in this area. Even though, the raw data of large data series collections will still not be able to fit in the GPU memory, a smaller working set could fit in the collective memory of a few interconnected GPUs. In our future work, we plan to examine the similarity search problem in such settings.

Acknowledgments: This work was done while P. Fatourou was working at the LIPADE, Université Paris Cité, as an MSCA Individual Fellow in the context of the PLATON project (MSCA grant agreement #101031688).

References

[1] Incorporated Research Institutions for Seismology – Seismic Data Access. http://ds.iris.edu/data/access/, 2016.

[2] http://helios.mi.parisdescartes.fr/ themisp/sing/, 2020.

[3] CUDA Toolkit Documentation, NVIDIA Pascal Architecture. `https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html`, 2020.

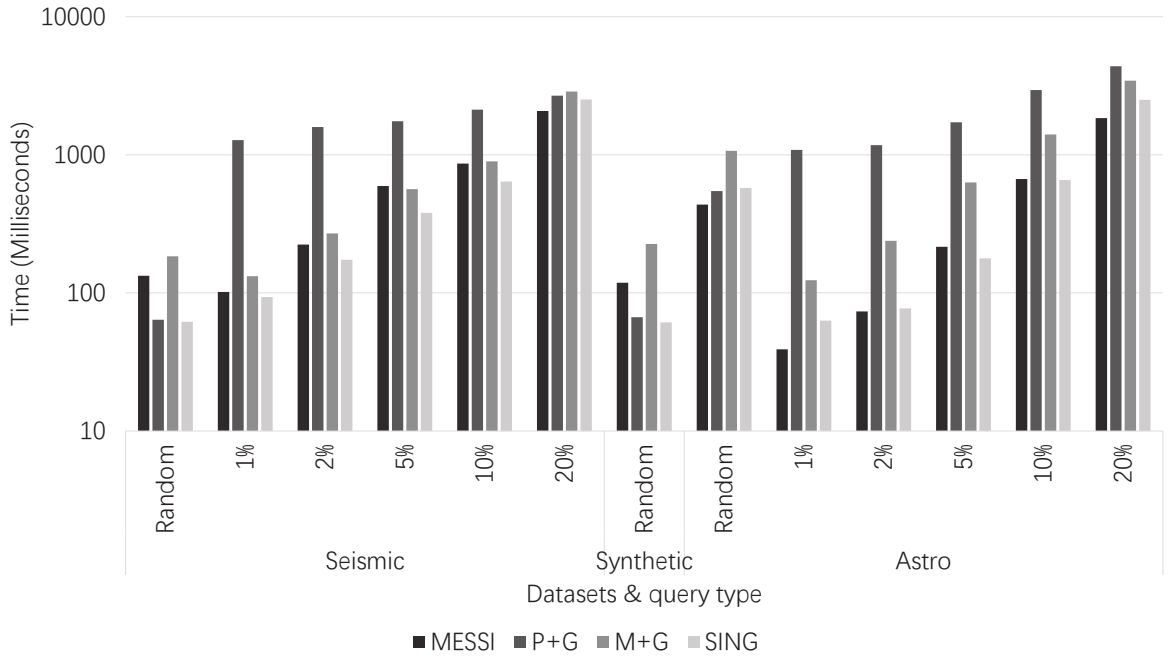


Figure 23: Query answering time for different datasets and query workloads, when all methods use hardware with the same monetary value (according to the experimental setup used in this study): MESSI uses 8 cores (1 socket); P+G, M+G and SING use 4 cores (1 socket) and the GPU.

[4] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, 1993.

[5] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: efficient time series search and retrieval. In *EDBT*, 2008.

[6] A. J. Bagnall, R. L. Cole, T. Palpanas, and K. Zoumpatianos. Data series management (dagstuhl seminar 19282). *Dagstuhl Reports*, 9(7), 2019.

[7] P. Boniol, M. Linardi, F. Roncallo, and T. Palpanas. Automated Anomaly Detection in Large Sequences. In *ICDE*, 2020.

[8] P. Boniol and T. Palpanas. Series2Graph: Graph-based Subsequence Anomaly Detection for Time Series. *PVLDB*, 2020.

[9] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. isax 2.0: Indexing and mining one billion time series. In *ICDM*, pages 58–67, 2010.

[10] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+. *KAIS*, 39(1), 2014.

[11] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *CSUR*, 2009.

[12] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, and S. Skiadopoulos. Local pair and bundle discovery over co-evolving time series. In *SSTD*, 2019.

[13] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, and S. Skiadopoulos. Local similarity search on geolocated time series using hybrid indexing. In *SIGSPATIAL*, 2019.

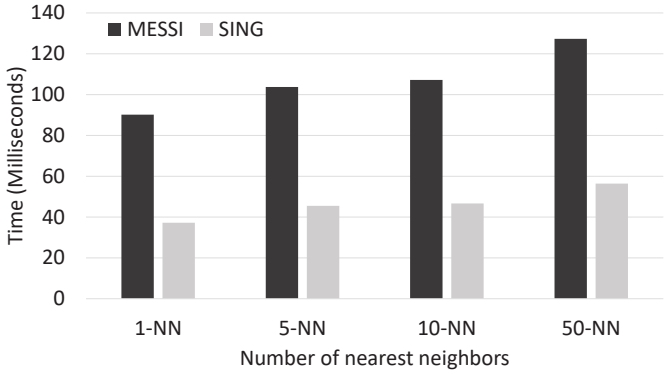


Figure 24: Time for a k-NN Classifier to classify one object (16 cores, 2 sockets; 100GB Synthetic dataset).

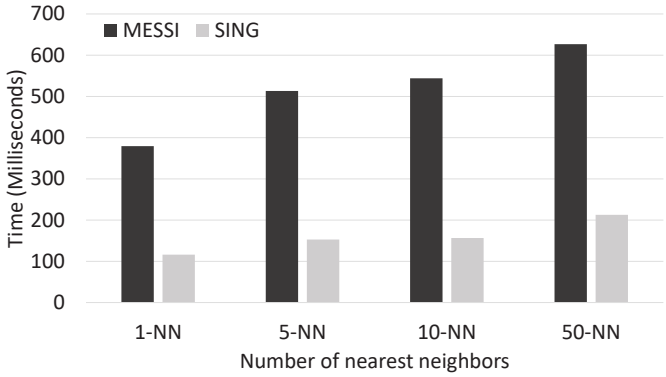


Figure 25: Time for a k-NN Classifier to classify one object, when all methods use hardware with the same monetary value: MESSI uses 8 cores (1 socket), while the SING algorithms use 4 cores (1 socket) and the GPU (250GB Synthetic dataset).

[14] M. A. Clark, P. C. L. Plante, and L. J. Greenhill. Accelerating radio astronomy cross-correlation with graphics processing units. *IJHPCA*, 2013.

[15] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire. A gpu-based index to support interactive spatio-temporal queries over historical data. In *ICDE*, 2016.

[16] K. Echihabi, K. Zoumpatianos, and T. Palpanas. Big Sequence Management: on Scalability (tutorial). In *IEEE BigData*, 2020.

[17] K. Echihabi, K. Zoumpatianos, and T. Palpanas. Scalable machine learning on high-dimensional vectors: From data series to deep network embeddings. In *WIMS*, 2020.

[18] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB*, 2018.

[19] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB*, 2019.

[20] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel. Buffer kd trees: processing massive nearest neighbor queries on gpus. In *ICML*, 2014.

[21] A. Gogolou, T. Tsandilas, K. Echihabi, A. Bezerianos, and T. Palpanas. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *SIGMOD*, 2020.

[22] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos. Progressive similarity search on time series data. In *EDBT*, 2019.

[23] M. Gowanlock and H. Casanova. Distance threshold similarity searches on spatiotemporal trajectories using gpgpu. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[24] M. Gowanlock and H. Casanova. Distance threshold similarity searches: Efficient trajectory indexing on the gpu. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2533–2545, 2015.

[25] M. Gowanlock and B. Karsin. Gpu accelerated similarity self-join for multi-dimensional data. *DaMoN*, 2018.

[26] M. Gowanlock, B. Karsin, Z. Fink, and J. Wright. Accelerating the unacceleratable: Hybrid cpu/gpu algorithms for memory-bound database primitives. In *DaMoN*. ACM, 2019.

[27] A. Guillaume. Head of Operational Intelligence Department Airbus. Personal communication., 2017.

[28] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*, 2020.

[29] P. Huijse, P. A. Estevez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *CIM*, 2014.

[30] K. Kashino, G. Smith, and H. Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.

[31] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *KAIS*, 2001.

[32] E. J. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *DAMI*, 2003.

[33] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[34] J. Kim, W.-K. Jeong, and B. Nam. Exploiting massive parallelism for indexingmulti-dimensional datasets on the gpu. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2258–2271, 2014.

[35] J. Kim and B. Nam. Co-processing heterogeneous parallel index for multi-dimensional datasets. *Journal of Parallel and Distributed Computing*, 113:195–203, 2018.

[36] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut: A scalable bottom-up approach for building data series indexes. *PVLDB*, 2018.

[37] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *VLDBJ*, 2019.

[38] O. Levchenko, B. Kolev, D. E. Yagoubi, R. Akbarinia, F. Masseglia, T. Palpanas, D. E. Shasha, and P. Valduriez. Bestneighbor: efficient evaluation of knn queries on large time series databases. *Knowl. Inf. Syst.*, 63(2):349–378, 2021.

[39] O. Levchenko, B. Kolev, D. E. Yagoubi, D. E. Shasha, T. Palpanas, P. Valduriez, R. Akbarinia, and F. Masseglia. Distributed algorithms to find similar time series. In *ECML/PKDD*, 2019.

[40] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu. A gpu accelerated update efficient index for knn queries in road networks. In *ICDE*. IEEE, 2018.

[41] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[42] M. Linardi and T. Palpanas. Scalable, variable-length similarity search in data series: The ulisse approach. *PVLDB*, 2019.

[43] M. Linardi and T. Palpanas. Scalable data series subsequence matching with ulisse. *VLDBJ*, 2020.

[44] M. Linardi, Y. Zhu, T. Palpanas, and E. J. Keogh. Matrix Profile Goes MAD: Variable-Length Motif And Discord Discovery in Data Series. In *DAMI*, 2020.

[45] S. Mittal. A survey of techniques for managing and leveraging caches in gpus. *JCSC*, 23(8), 2014.

[46] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, M. B. Westover, and N. B. Shamlo. A disk-aware algorithm for time series motif discovery. *DAMI*, 2011.

[47] A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *SIGMOD*, 2010.

[48] T. Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Record*, 2015.

[49] T. Palpanas. Evolution of a Data Series Index. *CCIS*, 1197, 2020.

[50] T. Palpanas. Evolution of a Data Series Index - The iSAX Family of Data Series Indexes. In *Communications in Computer and Information Science (CCIS)*, volume 1197, 2020.

[51] T. Palpanas and V. Beckmann. Report on the first and second interdisciplinary time series analysis workshop (ITISA). *SIGREC*, 48(3), 2019.

[52] J. Pan and D. Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *SIGSPATIAL*, pages 211–220, 2011.

[53] J. Pan and D. Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *IEEE ICDE*, pages 378–389, 2012.

[54] B. Peng, P. Fatourou, and T. Palpanas. Paris: The next destination for fast data series indexing and query answering. *IEEE BigData*, 2018.

[55] B. Peng, P. Fatourou, and T. Palpanas. Messi: In-memory data series indexing. In *ICDE*, 2020.

[56] B. Peng, P. Fatourou, and T. Palpanas. Paris+: Data series indexing on multi-core architectures. *TKDE*, 2020.

[57] B. Peng, P. Fatourou, and T. Palpanas. Fast data series indexing for in-memory data. *VLDB J.*, 30(6), 2021.

[58] B. Peng, P. Fatourou, and T. Palpanas. SING: Sequence Indexing Using GPUs. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2021.

[59] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *International Conference on Data Engineering (ICDE)*, 2014.

[60] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, 1997.

[61] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD*, 2012.

[62] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *ICDM*, pages 547–556, 2011.

[63] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical data prediction for real-world wireless sensor networks. *TKDE*, 2015.

[64] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *ICDM*, 2010.

[65] P. Schäfer and M. Högqvist. Sfa: a symbolic fourier approximation and index for similarity search in high dimensional datasets. In *EDBT*, pages 516–527, 2012.

[66] D. Shasha. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.*, 1999.

[67] J. Shieh and E. Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *SIGKDD*, 2008.

[68] J. Shieh and E. Keogh. isax: disk-aware mining and indexing of massive time series datasets. *DMKD*, 2009.

[69] J. Shieh and E. Keogh. iSAX: disk-aware mining and indexing of massive time series datasets. *DMKD*, (1), 2009.

[70] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[71] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *VLDB*, 2013.

[72] J. Wu, P. Wang, N. Pan, C. Wang, W. Wang, and J. Wang. Kv-match: A subsequence matching approach supporting normalization and time warping. In *ICDE*, 2019.

[73] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Dpisax: Massively distributed partitioned isax. In *ICDM*, 2017.

[74] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Massively distributed time series indexing and querying. *TKDE*, 32(1):108–120, 2020.

[75] L. Ye and E. Keogh. Time series shapelets: a new primitive for data mining. In *SIGKDD*. ACM, 2009.

[76] B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*. Citeseer, 2000.

[77] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. Gsi: Gpu-friendly subgraph isomorphism. In *ICDE*. IEEE, 2020.

[78] J. Zhou, Q. Guo, H. Jagadish, L. Krcal, S. Liu, W. Luan, A. K. Tung, Y. Yang, and Y. Zheng. A generic inverted index framework for similarity search on the gpu. In *International Conference on Data Engineering (ICDE)*. IEEE, 2018.

[79] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh. Matrix profile ii: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In *ICDM*. IEEE, 2016.

[80] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh. Exploiting a novel algorithm and gpus to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins. *KAIS*, 2018.

[81] Z. Zimmerman, K. Kamgar, N. S. Senobari, B. Crites, G. Funning, P. Brisk, and E. Keogh. Matrix profile xiv: Scaling time series motif discovery with gpus to break a quintillion pairwise comparisons a day and beyond. In *SoCC*, pages 74–86, 2019.

[82] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, 2014.

[83] K. Zoumpatianos, S. Idreos, and T. Palpanas. Ads: the adaptive data series index. *VLDB J.*, 2016.

[84] K. Zoumpatianos, S. Idreos, and T. Palpanas. ADS: the adaptive data series index. *VLDB J.*, 2016.

[85] K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and J. Gehrke. Generating data series query workloads. *VLDB J.*, 2018.

[86] K. Zoumpatianos and T. Palpanas. Data series management: Fulfilling the need for big sequence analytics. In *ICDE*, 2018.