Exploiting Reduced Precision for GPU-based Time Series Mining

Yi Ju*, Amir Raoofy†, Dai Yang‡, Erwin Laure*, Martin Schulz†
*Max Plank Computing and Data Facility {yi.ju, erwin.laure}@mpcdf.mpg.de
†Technical University of Munich {amir.raoofy, martin.w.j.schulz}@tum.de
‡NVIDIA GmbH {daiy}@nvidia.com

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00021

***Abstract*—The mining of *multi-dimensional time series* is a crucial step in gaining insights into data obtained from physical systems and from monitoring infrastructures. A widely accepted approach for this challenge is the matrix profile, which, however, is computationally very expensive. It relies on calculating large correlation matrices coupled with sort operations across all dimensions of the data, as well as on performing inclusive scans. All of these steps are inherently data parallel and can, therefore, benefit from execution on *GPUs*, and even more so from horizontal scaling on multiple GPUs. In addition, the nature of the matrix profile calculation allows the exploitation of *reduced precision* on GPUs. This offers further improvements to enable the analysis of ever growing data sets in real-world scenarios.**

Based on these motivations, we introduce the first parallel algorithm for multi-dimensional matrix profile on multiple GPUs exploiting reduced precision modes and provide a highly optimized implementation using novel optimization techniques. On one NVIDIA A100 GPU, our implementation achieves a 54x performance improvement in comparison to an optimized single-node execution on a state-of-the-art CPU-based implementation relying on double-precision computation and an additional factor of 1.4x when switching to reduced precision while maintaining sufficient accuracy. We study the accuracy and performance trade-offs for our proposed algorithm in detail and present synthetic and real-world case studies to demonstrate how the reduced precision improves the performance, while accomplishing sufficiently accurate results.

***Index Terms*—Multi-GPU algorithms, data mining, reduced precision, multi-dimensional time series, matrix profile.**

I. Introduction

Never in history have data, and hence data analytics played, a more significant role in human life and knowledge as today. Modern monitoring infrastructures provide scientists, analysts and developers with large amounts of data, which must be analyzed using big data and high performance data analytics (HPDA) techniques to better understand physical systems and phenomena. A collection of real-valued numbers with time-stamps, called *time series*, is one of the most common forms of data and is used in many domains. Specifically, *multi-dimensional* time series, which capture multiple sensor sources, have a significant importance, as they include collective information about many components at the same time and with that enable cross-sensor correlations. Therefore, efficient algorithms for the exploration of the complex similarity patterns in such multi-dimensional time series are crucial in extracting the needed insights.

The Matrix profile approach [3], [22] long serves as a fundamental data mining method to investigate time series, and provides a way to detect similar structures (patterns) when comparing two input time series. Matrix profile is widely accepted in the data science community and has been successfully applied to various application domains, including the investigation of earthquake foreshock [15], analysis of power system events in synchrophasor data [16], music information retrieval (MIR) [17], similarity searching of bacteria's DNA [21], and others.

Previous studies in the literature show that for large time series datasets, the matrix profile requires a high computational power to evaluate the large distance or correlation matrices [12], [13], [24]. The multi-dimensional case, which targets synchronously sampled time series, where each dimension corresponds to a separate data source, imposes more computational costs: this increase for one comes from the fact that the computational costs scales with dimensionality as each dimensions requires the calculation of a separate distance matrix. Additionally, in the multi-dimensional case we require extra repeated sort and inclusive scan operations to connect the dimensions, which increases the computational cost further.

While the state-of-the-art approach for computing multi-dimensional matrix profiles targets CPU-based systems [13], exploiting GPU systems for this workload is promising: previous studies show that this workload is memory-bound [12], suggesting that leveraging the High Bandwidth Memory (HBM) of GPUs promises performance improvements. Additionally, as this workload is not communication bound, the throughput is expected to scale with multiple GPUs. However, the use of GPU requires redesigning the parallelization scheme and data layout. While a GPU-based multi-dimensional matrix profile benefits from parallelization method used in the state-of-the-art GPU-based solution [27] for single-dimensional case, due to the extra computational costs and the resulting bottleneck shifts, development of a parallelization scheme for multi-dimensional case requires significant redesign.

On top of that, the problem of finding similar (and not necessarily exactly identical) patterns offers the door to reduced precision calculations, which has not been investigated before. Reduced-precision computations, aside from improving performance, can also reduce the memory footprint, resulting in an even more efficient usage of the GPU memory bandwidth and the ability to support larger problems. However, it does natu-

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00021

rally lead to more numerical errors and hence new challenges to preserve acceptable numerical accuracy.

In order to address both computational costs and accuracy aspects of computing multi-dimensional matrix profile on GPUs we, therefore, need 1) a careful design of kernels including the data layout, and data management design, efficient sorting and scalable tiling for efficient multi-GPU parallelization, and 2) suitable arithmetic approach for efficient and sufficiently-accurate reduced precision computation.

In this paper, we extend the state-of-the-art iterative approach for the multi-dimensional matrix profile, and introduce a new algorithm for GPUs that addresses the mentioned data management, kernel and arithmetic design challenges. Our algorithm targets multiple GPUs, and efficiently utilizes the GPU hardware features. We design novel reduced-precision computation modes with improved arithmetics to support a sufficiently accurate single and half-precision computation. We exploit a new tiling scheme in our algorithm, which not only enables parallelization of the workload on multiple GPUs, but also improves the accuracy of computation in the reduced precision modes by bounding the numerical error propagation.

In particular, we make the following contributions:

- we present the first algorithm and its implementation to compute matrix profile for multi-dimensional time series on a single and multiple GPU(s);
- we introduce several reduced-precision modes for computing multi-dimensional matrix profile;
- we develop a novel tiling scheme for multi-GPU parallelization, limiting the numerical errors in reduced-precision modes;
- we conduct a detailed performance evaluation using our implementation, including scaling experiments and demonstrate that multi-dimensional time series can be efficiently deployed on multi-GPU systems;
- we analyze the effect of reduced-precision computation on GPUs and use three real-world use cases to demonstrate the sensitivity of the reduced precision for pattern mining and classification.

Our approach offers a 54x performance improvement for double precision computation on one NVIDIA A100 GPU in comparison to the parallel execution of the state-of-the-art optimized matrix profile implementation on an Intel Skylake 16-core CPU. Our implementation with reduced precision introduces an additional advantage of a factor of 1.4x on the A100 compared to the double precision for common problem settings. Our tiling scheme allows the parallelization on multiple GPUs and improves the accuracy in reduced precision. Specifically, the tiling scheme on four A100 GPUs offers 3.8x faster performance than one, which equals a parallel efficiency of 95%, and enables over 80% numerical relative accuracy of reduced precision computation in comparison to reference FP64 CPU code in accuracy stress test, and reaches over 95% accuracy in real-world cases studies.

II. Related Work and Background

We first briefly review the related work and then provide a short theoretical background of the matrix profile.

A. Related work

History of matrix profile: Matrix profile was introduced by Yeh et al. [22] as a generic tool for the analysis of single-dimensional time series. Since then, various algorithms for efficient computation of matrix profiles were proposed, including STAMP [22], STOMP [26] and SCRIMP++ [25], which all introduced novel algorithmic and arithmetic manipulation of the kernels for efficient computation. However, none of these approaches directly targets multi-dimensional cases.

Multi-dimensional matrix profile analysis: In 2017, Yeh et al. [23] introduced matrix profile analysis for multi-dimensional time series and developed the mSTAMP algorithm. mSTAMP is built on top of previous algorithms and is an iterative method involving mean-centered streaming dot products (inherited from STOMP) to reduce the computational and memory requirements. mSTAMP is also included in the powerful and scalable python library, STUMPY [8]. To scale the computation of multi-dimensional matrix profile, Raoofy et al. [13] proposed Multi-dimensional Parallel Matrix Profile,$(MP)^N$, targeting large-scale CPU-based HPC systems. However, none of these approaches targets computation on GPUs and reduced precision computation.

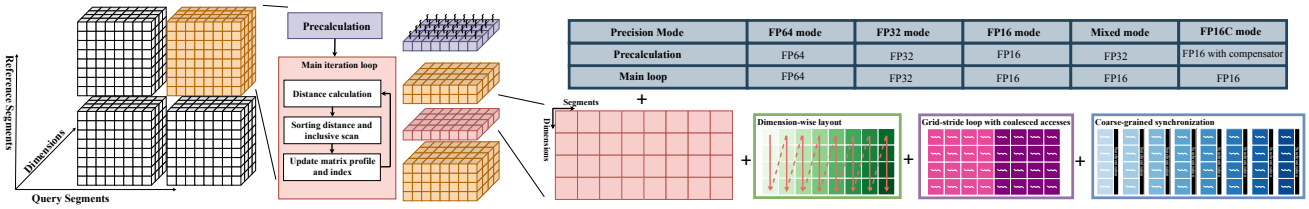
GPU deployment of matrix profile: Zhu et al. [26] introduced GPU-STOMP, which enabled the matrix profile computation on GPUs for the first time. Later, Zimmerman et al. introduced SCAMP [27], which improved and extended GPU-STOMP for Cloud deployment. SCAMP stands as the state-of-the-art GPU-based solution for single-dimensional matrix profile computation. Recently, Romero et al. [14] introduced a new approach called ScrimpCo for matrix profile computation on heterogeneous systems. However, none of these approaches addresses multi-dimensional time series.

Reduced-precision computation of matrix profile: Zimmerman et. al. [27] investigate single-precision computation for single-dimensional time series, and Fernandez [5] investigated matrix profile computation using FlexFloat [18]. However, there is no study on reduced-precision of matrix profile for multi-dimensional time series, especially for half-precision.

In summary, to the best of our knowledge, there are neither multi-GPU nor reduced-precision solutions targeting multi-dimensional matrix profile algorithms in the literature. No comprehensive studies on these aspects exist either. This paper overcomes this research gap.

B. Background

Matrix profile analyzes the similarities of an input time series, *query* time series, to a second time series, *reference* time series: the reference time series is often a well-known historical time series dataset and is used to characterize the motifs and patterns in an unknown query time series. During the matrix profile analysis, a *distance (or correlation) matrix* among all the segments (or subsequences)—which are local



Precision Mode	FP64 mode	FP32 mode	FP16 mode	Mixed mode	FP16C mode
Precalculation	FP64	FP32	FP16	FP32	FP16 with compensator
Main loop	FP64	FP32	FP16	FP16	FP16

Fig. 1. An overview of our design for deploying matrix profile computation to multiple GPUs with reduced precisions.

chunks of the input series consisting of consecutive samples—of the two input time series is computed. The segments with maximum correlations, corresponding to best matches of the query segments in the reference time series, are determined. In multi-dimensional time series, segments are multi-dimensional requiring the computation of multiple distance matrices, one for each dimension. Collection of all these matrices can be interpreted as a 3d matrix. This 3d matrix is calculated, and then sorted to enable a connection between the dimensions, and progressively averaged along the dimensions. This way, best matching 1- to d-dimensional sets of segments with the most similarities across all dimensions are determined.

To formally introduce matrix profile, we need to provide the following definitions: for the d-dimensional *reference* and *query* time series $T_r \in \mathbb{R}^{n_r \times d}$ and $T_q \in \mathbb{R}^{n_q \times d}$ with length n_r and n_q ($n = n_r = n_q$)[1] and segment (subsequence) length m, the *matrix profile*, $P \in \mathbb{R}^{(n_q-m+1)\times d}$, is a set of d real-valued distance vectors (equivalently correlation vectors), where i^{th} vector corresponds to the (Z-normalized Euclidian) distance of i-dimensional segments in T_q to their nearest neighbor segments in T_r. The *matrix profile index*, $I \in \mathbb{Z}^{(n_q-m+1)\times d}$, is a set of d indices vectors indicating the location of the aforementioned nearest neighbor of segments in T_q.

As the core part in computation of I and P, the 3d distance matrix, $\mathcal{D} \in \mathbb{R}^{(n_r-m+1)\times(n_q-m+1)\times d}$, represents the distances of all the segments of T_r and T_q in all the d dimensions. To evaluate it, state-of-the-art solutions use the *mean-centered streaming dot product*, Eq. (1), and indirectly compute the Euclidean distance $\mathcal{D}$ from Pearson correlations ($\overline{QT}$) of segments separately in each dimension. This formulation computes Pearson's correlation factors corresponding to a row of the distance matrix iteratively from the top-left element in the previous row Eq. (1). Although this update scheme introduces diagonal-wise dependencies, it only requires four floating-point operations (FLOPs) per dimension in each iteration. However, this small number of FLOPs, combined with the following sorting step, results in memory-bound performance.

$$\begin{aligned} \overline{QT}_{i,j,k} &= \overline{QT}_{i-1,j-1,k} + df^r_{i,k} \cdot dg^q_{j,k} + df^q_{j,k} \cdot dg^r_{i,k} \\ \mathcal{D}_{i,j,k} &= \left(2 \cdot m \cdot (1 - \overline{QT}_{i,j,k} \cdot {d^r_{i,k}}^{-1} \cdot {d^q_{j,k}}^{-1})\right)^{0.5} \end{aligned} \quad (1)$$

This iterative method relies on an earlier *precalculation* step to prepare the intermediate set of matrices, including $df^r, dg^r, {d^r}^{-1} \in \mathbb{R}^{(n_r-m+1)\times d}$ and $df^q, dg^q, {d^q}^{-1} \in \mathbb{R}^{(n_q-m+1)\times d}$ in a single pass over the two input datasets. As this computation can be implemented very efficiently in comparison to the rest, we do not repeat the equations here [13], but discuss its impacts on overall precision later.

After the computation of the 3d distance matrix, similar to (MP)^N, it is sorted along dimensions (2) and is followed by inclusive averaging based on inclusive scan of $\mathcal{D}^{'}$ along the dimensions [23] to consider *multi-dimensional segments*.

$$\mathcal{D}^{'}_{i,j,k} = \texttt{sort}(\mathcal{D}_{i,j,k});\ \mathcal{D}^{''} = \texttt{inclusive\_scan}(\mathcal{D}^{'}) \quad (2)$$

Then matrix profile is computed by applying a column-wise minimum on the results of inclusive scan $\mathcal{D}^{''}$ Eq. (3).

$$P_{j,k} = \texttt{min}(\mathcal{D}^{''}_{*,j,k});\ I_{j,k} = \texttt{argmin}(\mathcal{D}^{''}_{*,j,k}) \quad (3)$$

III. GPU-based Approach with Reduced Precision

We target high-end data analytics systems with multiple GPUs, which are common in HPDA. Our approach accelerates the computation with the efficient exploitation of GPU hardware, the multi-GPU parallelization and the reduced precision.

We use Eq. (1), (2) and (3) for GPU deployment to take the advantage of partial storage of distance matrix $\mathcal{D}$ in GPU memory and highly efficient arithmetic for the evaluation of distances between segments. However, this formulation introduces arithmetic challenges in accuracy for the reduced precision, which we address in our solution.

Fig. 1 provides an overview of our solution, including the illustration of our parallelization scheme on GPUs, the tiling scheme, and optimizations. We exploit an optimized data layout to enable coalesced memory accesses, and highly-optimized sort kernels. Additionally we efficiently exploit the hardware features in modern GPUs to to achieve high performance. Specifically, we tune the utilization of GPU Streaming Multiprocessors (SMs) and scratchpad memory to achieve high bandwidth and low latency in kernels. Our solution enables various reduced-precision configurations efficiently by exploiting hardware support for reduced-precision computation in modern GPUs. Additionally, we introduce a tiling scheme that extends the state-of-the-art to exploit multiple tiles and GPUs and to limit the propagation of numerical errors.

A. Single-Tile Algorithm

Pseudocode 1 describes our base algorithm that targets a single tile and single GPU. It starts with an asynchronous copy of input data from the CPU (host) to the GPU (device), followed by invocation of the compute kernels. `precalculation` kernel prepares the correlations $\overline{QT}$ associated with the first row of the distance matrix using a naive (non-streaming) dot product formulation. Additionally this kernel computes the variables df, dg, ..., used for the next n iterations using cumulative summations. In more details, each thread computes

[1]Without losing generality, we use the number of segments in the reference and query time series in k^{th} dimension, $n = n_r - m + 1 = n_q - m + 1$.

Pseudocode 1 Overview of Single-Tile Algorithm

Input: The reference and query time series T_r^{cpu} and T_q^{cpu}.
Configuration: s_{block} and s_{grid}.
Output: The matrix profile P^{cpu} and index I^{cpu}.
Note: All data resides on GPU unless marked otherwise.

1: T_r, T_q ← `input_async_cpy` (T_r^{cpu}, T_q^{cpu}, H2D)
2: QT_r, QT_q, df_r, dg_r, ... ← `precalculation` (T_r, T_q)
3: **for** $i \leftarrow 0$ to $(n-1)$ **do**
4: D ←`dist_calc`<<<s_{grid}, s_{block}>>>(QT_r, QT_q, df_r, dg_r, ...)
5: D'' ←`sort_&_incl_scan`<<<s_{grid}, s_{block}>>> (D)
6: P, I ←`update_mat_prof`<<<s_{grid}, s_{block}>>> (D'')
7: **end for**
8: P^{cpu}, I^{cpu} ← `output_async_cpy` (P, I, D2H)

one dot product ($\overline{QT}$) and the corresponding cumulative summations for each element, illustrated in Fig. 1 in purple.

In the i^{th} iteration, only one row (plane) of the distance matrices with size of $\mathcal{O}(n \cdot d)$ is computed (the highlighted red plane in Fig. 1). In more details, the matrix profile is computed as follows on the GPU:

1) `dist_calc` uses Eq. (1) to compute i^{th} row of the distance matrix $\mathcal{D}$. Each thread computes one element of the next row (plane) of the distance matrix using Eq. (1), which is parallelized in i (or j) and k (thread assignment is illustrated in Fig. 1 in magenta).
2) `sort_&_incl_scan` sorts the distances in ascending order along dimensions. For this kernel, multiple sorts (one for each dimension) are performed in parallel, and multiple threads cooperate on these sort operations. Therefore each individual sort operation is assigned to a group of threads. Additionally this kernel calculates multiple inclusive scans (Eq. (2)) in parallel along the dimensions. For this, threads in each group cooperatively perform an inclusive scan (thread and group assignment in this kernel is illustrated in Fig. 1 in blue).
3) `update_mat_prof` merges the computed distances in the i^{th} iteration to the results in previous iterations using Eq. (3). This kernel uses a similar thread assignment as in the `precalculation`, where all the threads update the elements of the resulting matrix profile (e.g., a plane) in an embarrassingly parallel fashion.

To achieve high efficiency on GPUs, we introduce the following optimizations in the above steps:

Data Layout: We use a dimension-wise data layout for storing the active rows (planes) in device memory, i.e., consecutive elements of each dimension reside next to each other in memory (shown in Fig. 1), for all the data involved in the computations of the different kernels.

Grid-Stride Loops: We structure the iterations in kernels to exploit grid-stride loops to ensure coalesced memory access. Moreover, the grid-stride loops enable more flexibility by supporting arbitrary kernel launch configurations, i.e., s_{block} and s_{grid}. Through this flexibility, our design promises a high performance by tuning kernel launch configurations (i.e., the configuration settings in Pseudocode 1) that match the GPU hardware architecture.

Coarse-Grained Synchronization: In *sort_&_incl_scan*, we use the $\mathcal{O}(\log^2 d)$ parallel sorting scheme (based on Bitonic sort) and $\mathcal{O}(\log d)$ parallel fan-in approach of inclusive scan, where many threads cooperatively sort and calculate the scans. Compared to the more intuitive batch-based parallelization, where only one thread performs a single sort and scan, our choice results in better utilization of the GPU resources, hence achieves a higher performance. However, our parallelization requires nested synchronization, which can potentially lead to large overheads. To minimize the overhead, we apply coarse-grained synchronizations among threads.

B. Multi-Tile Algorithm Targeting Multi-GPU Systems

To exploit parallelization on multiple GPUs and to bound the numerical error propagation, we introduce a novel tiling scheme that extends the state-of-the-art scheme in $(\text{MP})^N$. This tiling scheme (shown in the left in Fig. 1) is motivated by task-based programming models: It *decouples* the size of the distance matrix running on devices from the actual size of the input series and the corresponding distance matrix, therefore, despite the limited device memory, our algorithm can process arbitrary large (in sense of both the number of segments and the number of dimensions) problems on a single or multiple devices. Additionally, this design aims at limiting the excessive propagation of numerical errors of the iterative computation, as the precalculation step is repeated in each tile and breaks the error propagation in Eq. (1).

We describe this tiling scheme in Pseudocode 2. We first partition the distance matrix into smaller tiles (`compute_tile_list`), where each smaller tile is later executed on a GPU as a standalone matrix profile (task) with a smaller problem size (i.e., tile size). We statically assign these tiles (`assign_tile`) to n_{gpus} GPU(s) in a Round-robin fashion enabling maximum balance for parallel execution on multiple GPUs. The tiles are asynchronously computed ($\texttt{run\_tile}_{\texttt{gpu}}$) on GPUs using the same scheme in Pseudocode 1, and after the execution of each tile, its results are merged (`merge`) on the CPU using `min` and

Pseudocode 2 Multi-Tile Algorithm

Input: The reference and query time series T_r^{tile}, T_q^{tile} stored on CPU.
Configuration: s_{block}, s_{grid}, n_{tiles}, and n_{gpu}.
Output: The matrix profile P and its indexes I.
Note: All data resides on GPU unless marked otherwise.

1: *tile_list* ← `compute_tile_list` (n_{gpu}, n_{tiles})
2: **for each** tile ∈ *tile_list* **do in parallel with implicit synchronization**
3: *dev* ← `assign_tile` (*tile*)
4: P^{tile}, I^{tile} ← $\texttt{run\_tile}_{\texttt{gpu}}$ (Tr^{tile}, Tc^{tile}, s_{grid}, s_{block}, *dev*)
5: **end for**
6: **for each** tile ∈ *tile_list* **do with implicit synchronization**
7: P^{CPU}, I^{CPU} ← `merge` (P^{tile}, I^{tile})
8: **end for**

`argmin` operations. Data transfer and kernel execution for tiles benefit from implicit synchronization (CUDA Streams), to exploit maximal concurrency (i.e., through configuring number of streams) in kernel execution as well as to hide the communication latencies between CPU and GPU.

In addition to the kernel launch configuration itself, the number of GPUs (n_{tiles}) and tiles (n_{tiles}) are used as configuration parameters (i.e., the configuration settings in Pseudocode 2) for performance tuning. Additionally, this design simplifies tuning for accuracy through careful selection of the number of tiles n_{tiles}.

C. Reduced-Precision Computation

Our GPU-based approach for multi-dimensional matrix profile natively exploits double-precision (**FP64**) floating-point data format. To further improve the performance while ensuring the accuracy on a certain level, we introduce four additional reduced-precision modes including the *single-*, *half-*precision arithmetic, a *mixture* of both and *half*-precision with improved arithmetic in the precalculation step.

*Single precision (**FP32**):* In this mode, we use *FP32* for both storage and arithmetic, as it is widely used in ML and HPDA.

*Half precision (**FP16**):* For the half-precision mode, we store all the data and conduct all the computation in *FP16*. This mode promises the fastest computation, but the numerical errors in this mode are the most severe.

*Mixed precision (**Mixed**):* We also explore a mixed-precision mode that uses *FP16* for storage and computation similar to the *FP16* mode; however, it benefits from performing `precalculation` in higher precision using *FP32* arithmetic. This combination is promising for achieving results with higher accuracies, while enabling the performance benefits of half-precision computation, as the performance overhead of `precalculation` in a higher precision is a negligible portion of total runtime.

*Half precision with improved arithmetics (**FP16C**):* Finally, we explore another variation of half-precision computation, which again exploits a higher precision mode in `precalculation` with an improved variation of arithmetic that uses Kahan's compensated summation [7] in `precalculation`. The rest of the steps use *FP16* similar to mixed- and half-precision scenarios. With this compensated summation, we prevent the error propagation from severe cancellations that arises in the `precalculation` in *FP16* mode. Despite the additional computation, it does not result in any significant overhead as again the `precalculation` contributes very little to the overall runtime. This variation also promises similar accuracy and performance benefits to the Mixed mode.

IV. Implementation

We use `C++` for our implementation, and our code is freely available in Zenodo[2]. We are aiming for NVIDIA GPUs as our target devices, and use GCC compiler v.8.0, together with CUDA v.11.2, which supports all the required functionalities we need to realize our algorithm.

[2]DOI: 10.5281/zenodo.5827200

We choose our customized GPU implementation of Bitonic sort in the sorting step over the popular libraries, such as CUB [10] or Modern GPU [11], as it provides much higher performance for sorting operations. For the same reason, we customized our inclusive scan operations instead of using CUB. We exploit NVIDIA's *cooperative groups API* for coarse-grained synchronization in Bitonic sort and inclusive scan and exploit *shared memory* in thread block for storage.

For grid-stride loops, we use the kernel launch configuration that matches the hardware architecture: on `V100` we use 64 as grid size and 2560 as block size; on `A100` we use 64 as grid size and 3456 as block size. Our experiments validate that these configurations provide the best performance.

We rely on the Stream Management API in CUDA for implicit synchronization (especially in the multi-tile code): all the data transfers and kernel executions rely on CUDA streams. We use maximal 16 non-blocking streams on one GPU to avoid memory consumption limits while keeping a high concurrency.

For FP64 and FP32, we only adopt the data format and simply use the same native mathematical operators in `C++`. However, FP16, Mixed, and FP16C use `__half` data type and corresponding intrinsics from CUDA Math API, for there are no native half-precision data types and operators in `C++`.

V. Evaluations

Before the investigation of the *case study* on real world data (details in Section VI), we conduct stress tests and investigate *accuracy* and *performance* aspects and their tradeoff in various precision modes on synthetic datasets.

A. Experimental Setup

System Setup: We use two multi-GPU systems: a DGX-1 at Leibniz Super-computing Centre [1] and the Raven supercomputer from Max Planck Computing and Data Facility [2]. The DGX-1 system includes 8 NVIDIA Tesla V100 GPUs, each providing 7.8 TFLOP/s double-precision performance, 32 GB device memory capacity, 900 GB/s memory bandwidth and 80 Streaming Multiprocessors (SMs). Each node of Raven system includes 4 NVIDIA Tesla A100 GPUs, each providing 9.7 TFLOP/s double-precision performance, 40 GB device memory, 1,555 GB/s memory bandwidth and 108 SMs.

We tune and use optimal configurations to launch all the kernels, which match the corresponding GPU hardware architectures: we launch 163,840 threads on V100 (163,840 = 80 SMs · 64 warps · 32 threads), and 221,184 threads on A100 (221,184 = 108 SMs · 64 warps · 32 threads).

Dataset Description: We use a synthetic dataset of multi-dimensional time series with 80 groups of parameter settings (different n, m, and d) for performance and accuracy evaluation (stress tests). This dataset includes random noise combined with randomly-located injected repeating patterns, providing a reliable basis for pattern detection when we use practical accuracy analysis (we introduce practical accuracy in the following paragraph). We use eight different shapes (as illustrated in left side of Fig. 3) for the injected patterns

with different complexity, which helps to cover a sufficiently diverse set of patterns.

Accuracy Metrics: To better quantify and understand the level of accuracy that can be achieved with the different reduced precision settings, we introduce two accuracy metrics.

The first set of accuracy metrics represent the *numerical accuracy* of results, where we compare the numerical difference of results of our implementation (e.g., with reduced precision) to the CPU-based reference:

- Recall rate ($\mathscr{R}$) [4]: We consider the ratio of the number of matching matrix profile indices to the total number of indices as *recall rate*.
- Relative accuracy ($\mathscr{A}$) [25]: The relative discrepancy between the matrix profile computed with reduced precision and the reference FP64 calculation is considered as relative error, $\mathscr{E}$. We define $\mathscr{A} = 1 - \mathscr{E}$ as the relative accuracy measure and report it in percentage.

The second set of accuracy metrics provide the means for a *practical accuracy* evaluation. This accuracy evaluation scheme aims at presenting the practical use of our approach, despite possible numerical errors, where we can successfully detect patterns and develop accurate classifiers, as presented below. We specifically consider a specific use case of the matrix profile, e.g., motif discovery or nearest neighbor-based classification of time series data. In these cases, we focus on the actual accuracy in detecting the target patterns or classification accuracy.

- Recall for embedded motif detection ($\mathscr{R}_{embedded}$) [25]: For pattern detection cases, we quantify the number of successful matches of a specific pattern retrieved by the matrix profile, e.g., when computed using our implementations with reduced precision.
- Relaxed recall rate for for embedded motif detection ($\mathscr{R}^r_{embedded}$): In practice, the equivalency of matrix profile index in reduced precision is a strict requirement, and for some use cases, e.g., pattern detection, an approximation of the location of patterns is sufficient. Therefore, we also can consider a relaxed recall rate allowing a tolerance in computing the numerical accuracy: in case the resulting matrix profile index from the position determined by a reference calculation is within a predefined range, it is considered as a successful detection. We define the ratio of this extended range to the length of the segments as the relaxation factor (r) and consider it as a hyperparameter for tuning the accuracy in reduced precision.
- F-score for classification ($\mathscr{F}_{classification}$) [19]: Also, for classification, we look at the overall accuracy, F-score (F-score, i.e., the harmonic mean of precision and recall accuracy metrics), of the classification when we use matrix profile indices computed with reduced precision.

Performance Metrics: For performance evaluation of our implementations, we rely on the *total* execution time and *kernel* execution time. We also measure the kernel performances with NVIDIA Nsight Compute. For multi-GPU evaluation, the additional performance metric is the *parallel efficiency*, which is defined as the ratio of speedup (with respect to single GPU execution) and number of GPUs used.

Reproducibility and Stability of Results: To guarantee credibility of our experiments, we repeat each experiment five times and analyze the arithmetic average of the accuracy and performance metrics. Our experiments show that our implementation has a stable accuracy regardless of the GPU generation (Volta or Ampere) and the same execution time regardless of individual datasets. Therefore, we only report the averages of measured metrics in our evaluations.

B. Accuracy Evaluation

Numerical accuracy: We analyze the *numerical accuracy* of our implementation in Fig. 2.

The FP64 mode on the GPU can generate identical results as the CPU-based implementation. The FP32 mode also results in a high accuracy of roughly 100%, and the Mixed and FP16C modes result in almost the same accuracy and double the accuracy than FP16; the accuracy of FP16, Mixed and FP16C modes decreases and then stabilizes as the number of subsequences in the input time series increases; with an increasing dimensionality in the input time series, the numerical accuracy of FP16, Mixed and FP16C modes decreases and then increases; with increasing length of subsequences in the input time series, in Mixed and FP16C modes, the accuracy of the matrix profile increases, while the accuracy of the index decreases and stabilizes and in FP16 mode, the numerical accuracies stay unchanged at around 5%.

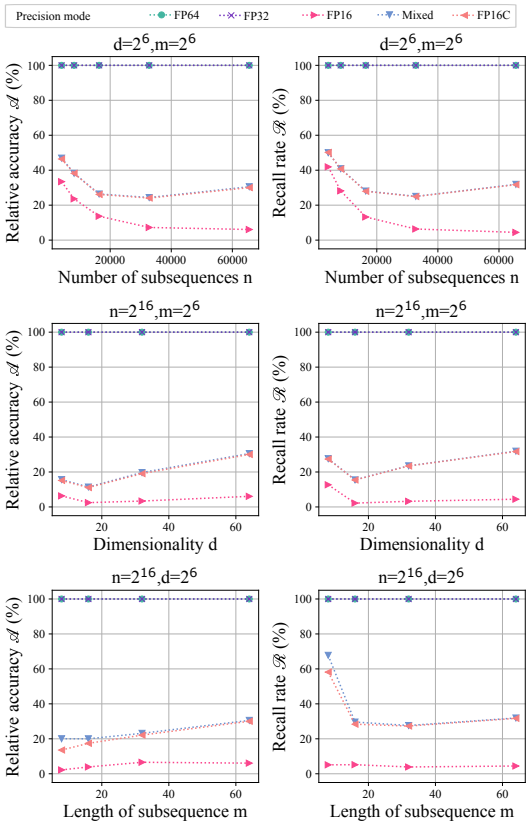


Fig. 2. Numerical accuracy ($\mathscr{A}$ and $\mathscr{R}$) of single-tile implementation in processing synthetic dataset compared to the CPU-based implementation.

As the FP16, Mixed and FP16C modes cannot generate identical results as the CPU-based code in sense of numerical accuracy, we trace the inaccuracies in results to limited numerical accuracy of the half precision for storage and arithmetic, and briefly discuss this in the following:

the condition number of the distance computation in Eq. (1) implies an ill-conditioned formulation for the flat regions in input time series. On the other hand, the regions with large deviations are prone to overflow as the variables used in Eq. (1), are limited by the maximal representable numbers, in particular in half precision.

We further have a look at the propagation of error in computing $\overline{QT}$. We can describe the iterative computation of $\overline{QT}$ in Eq. (1) as a large dot product and analyze its sensitivity to rounding errors, based on the analysis provided by Yang et. al. [20]. In this analysis, the error abounds for dot-product are proportional to the length of input vectors and machine precision ($e \propto (n \times \varepsilon)$). Therefore, we trace the numerical inaccuracies in matrix profile computation in reduced precision to *machine error* and *tile size*.

- *Machine error*: the single precision ($\varepsilon_{32} = 2^{-23}$) and the half precision ($\varepsilon_{16} = 2^{-10}$) provide less accurate computation compared to the double precision ($\varepsilon_{64} = 2^{-52}$). As similar segments have distances closer to zero, large machine errors can prevent finding accurate distances among similar segments with the repeated patterns.
- *Tile size*: using smaller tiles bounds the propagation of the numerical error for computing the matrix profile with reduced precision.

Practical Accuracy for Pattern Detection: We further focus on the practical accuracy of pattern detection, we rely the evaluation scheme used by Zhu et al. [25]. For this we embed patterns on predefined random locations and inspect the recall value ($\mathscr{R}_{embedded}$), when reduced precision is used for matrix profile computation. We use a diverse set of primitive patterns for embedding, which are marked with $P0-7$ in Fig. 3. In this case, with the except of the 98% accuracy achieved in the detection of Patterns 2 and 3 ($P2$ and $P3$) with FP16, Mixed and FP16C modes, all the reduced-precision modes can achieve exact 100% practical accuracy to detect *all* injected patterns as with FP64 mode. Despite the numerical inaccuracies, our implementations with reduced precision is delivering precise pattern detection.

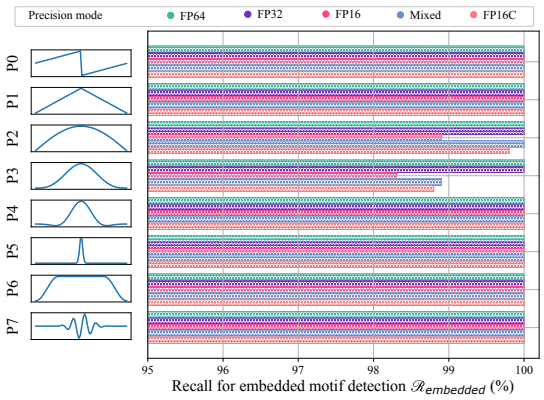


Fig. 3. Practical accuracy ($\mathscr{R}_{embedded}$) of single-tile implementation for pattern detection. We plot the patterns with time as x-axis ($x \in [0, m)$) and the normalized values as y-axis ($y \in [-1, 1]$)

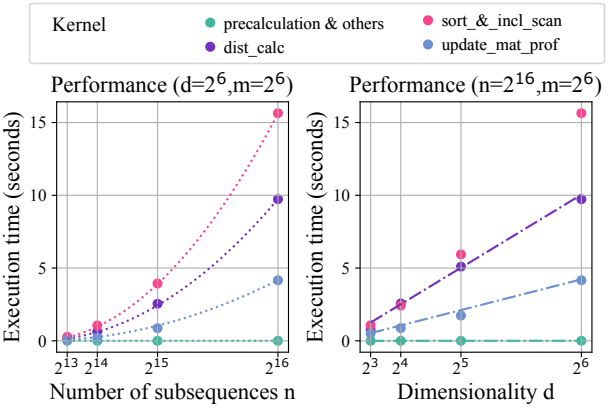


Fig. 4. Kernel execution time of multi-tile implementation with one tile on A100 GPU.

C. Performance Evaluation

Kernel Profiling: As shown in Fig. 4, the execution time of all the kernels increases with the number of subsequences and the dimensionality. The number of subsequences has more influence on the execution time and the dimensionality decides which kernel has the dominant influence. For the small dimensionality, `dist_calc` is the dominant kernel; for the big dimensionality, `sort_&_incl_scan` is the dominant kernel. Theoretically, the time complexity of `precalculation` is $\mathcal{O}(n \cdot d)$; `dist_calc` and `update_mat_prof` is $\mathcal{O}(n^2 \cdot d)$; and `sort_&_incl_scan` is $\mathcal{O}(n^2 \cdot \log^2 d)$. The synchronization overhead of `sort_&_incl_scan` causes a slowdown with large dimensionality. However, it still outperforms alternative sorting approaches by preventing underutilization of GPU hardware.

Reduced Precision Performance: Fig. 5 shows the performance improvements by using reduced precision. As expected, lower precision modes show higher performance, however, the performance is not scaled linearly with the number of bits used in corresponding data types. All kernels scale

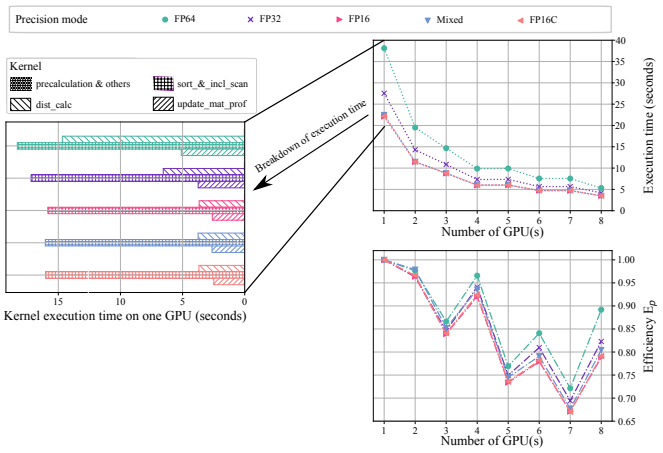


Fig. 5. Execution time and efficiency of multi-tile implementations with 16 tiles on DGX-1 (n=2^{16}, d=2^8).

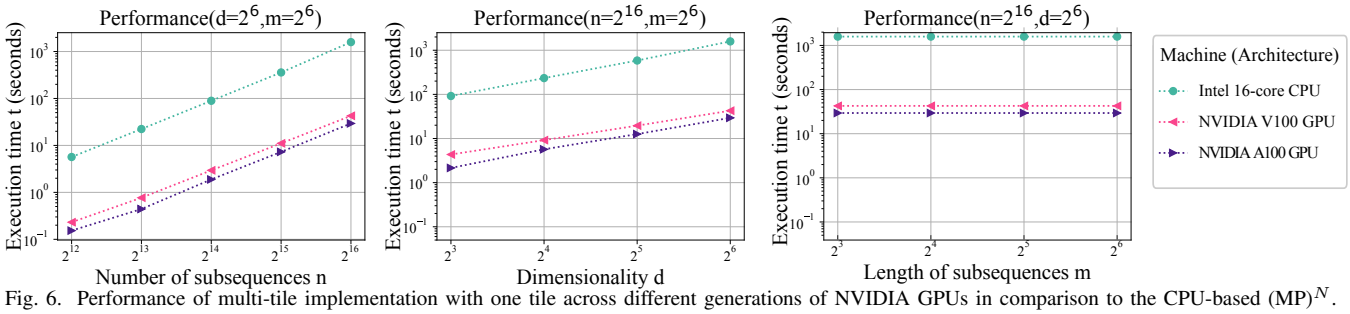


Fig. 6. Performance of multi-tile implementation with one tile across different generations of NVIDIA GPUs in comparison to the CPU-based $(MP)^N$.

almost linearly with the data type as expected, except for `sort_&_incl_scan`, which is mainly dominated by repeating synchronization overheads and the performance improvements in reduced precision modes is minimal. Moreover, the performance for FP16, Mixed and FP16C modes is similar, as their performance difference—mainly from `precalculation`—is negligible.

Resource Utilization: We also provide an analysis on the GPU utilization by running our implementation on a single A100 GPU: All of the kernels are still memory bound on GPUs. For the large tiles ($n = 2^{16}$, $d = 2^6$, $m = 2^6$), `dist_calc` and `update_mat_prof` use over 80% DRAM and around 70% L2 cache throughput; `sort_&_incl_scan` uses over 80% L1/TEX cache throughput and around 70% compute (SM).

In the FP32 mod, `dist_calc` uses around 60% and `update_mat_prof` uses around 70% DRAM throughput and L2 cache throughput; `sort_&_incl_scan` uses around 40% L1/TEX cache throughput and around 70% compute (SM). In other modes, `dist_calc` uses around 30% and `update_mat_prof` uses over 50% DRAM and L2 cache throughput; `sort_&_incl_scan` uses around 20% L1/TEX Cache throughput and around 70% Compute (SM).

Scalability: We analyze the scalability of our implementation, starting with the DGX-1 in Fig. 5: we observe linear scalability of our implementation with the number of GPUs, however, we see inefficiencies when using odd numbers of GPUs. This inefficiency is due to our tiling scheme in Pseudocode 2, which works best when the number of GPUs is a factor of the tile number. However, this inefficiency can be mitigated by increasing the number of tiles assigned to all GPUs.

When 1, 2, 4 and 8 GPUs are used in double precision, our implementation reaches over 90% parallel efficiency. The around 80% parallel efficiency of reduced precision shows that high scalability can be achieved across all precision modes. Our implementation also shows a similar strong scalability on Raven, where it achieves over 95% efficiency on 1, 2 and 4 GPUs with all precisions (graphs are not presented here).

Performance Across Different Generations of GPUs: We further examine the performance of our implementation across different generations of NVIDIA GPUs with Tesla architectures in Fig. 6. Specifically, we compare the performance on V100 and A100 GPUs of our implementation in `FP64` mode to the state-of-the-art CPU-based implementation with full usage of an Intel 16-core Skylake CPU.

Our GPU-based implementation in all configurations, with various parameter settings, provides much faster execution times. Overall, our implementation can achieve about 41.6x and 54.0x performance boost in double precision on a V100 and an A100 GPU, respectively, in comparison to the CPU.

Additionally, the performance *behavior* of our implementation is similar to the reference CPU-based code, preserving its suitable complexity features: the execution time scales quadratically with the number of subsequences and linear with dimensionality. Moreover, the execution time is independent of the subsequences' length, preserving the flexibility of the family of matrix profile algorithms.

D. Accuracy-Performance Tradeoff

We conduct an experiment to study the Accuracy-Performance tradeoff (Fig. 7) by increasing the number of tiles from 1 to 1,024 (and reduce the tile size accordingly).

Using more tiles increases the numerical accuracy for FP16, Mixed and FP16C modes. This experiment is also in accordance with the analysis we introduced earlier regarding direct relation between the numerical error bounds and tile sizes. At the same time, despite the initial performance boost from 1 tile to 256 tiles (thanks to the concurrency invoked by using multiple streams in implicit synchronization), increasing the number of tiles has a slight negative impact on the execution time. The main reason for this performance drop is that the

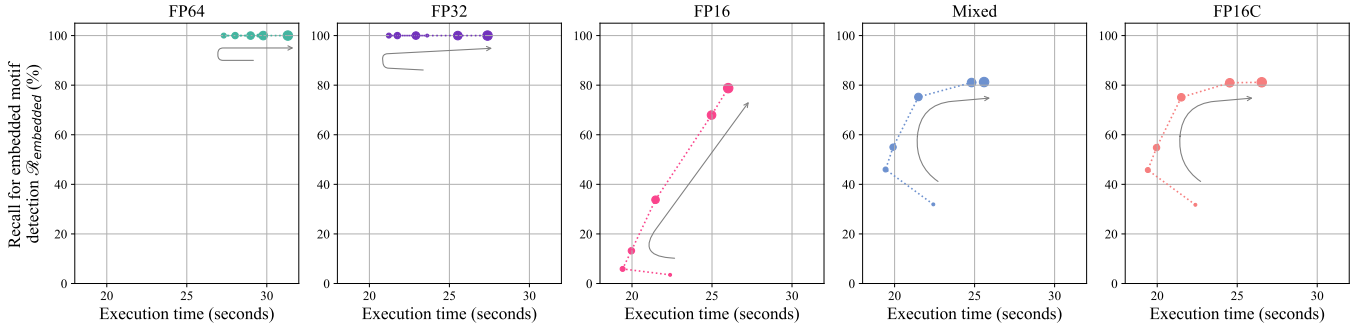


Fig. 7. Accuracy-Performance tradeoffs of multi-tile implementations on one A100 GPU with increasing number of tiles. (n=2^{16}, d=2^6, m=2^6). The size of the markers indicate the number of tiles n_{tile}. We annotate arrows next to the data to indicate the direction of increase in number of tiles.

final merging of tiles in our implementation is executed by the CPU (see Pseudocode 2), which results in an overhead increasing with the number of tiles. However, in a special configuration, using 256 tiles allows FP16, Mixed and FP16C modes to have 2x accuracy and even shorter total execution time compared to a one tile run. With this configuration, Mixed and FP16C modes reach almost 80% accuracy with 256 tiles.

Overall, using more tiles is a plausible setting, due to the accuracy boost and the insignificant performance drop. Further, the overhead of the merging step is negligible for large problems, leading us to an overall more efficient configuration.

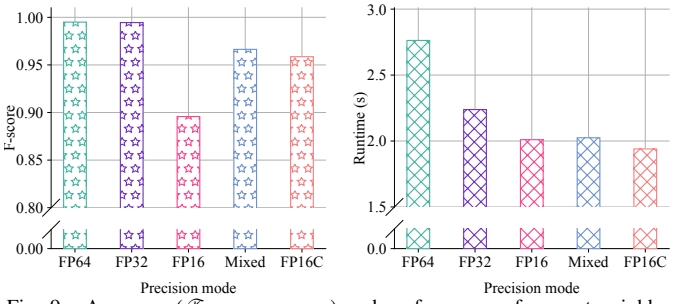


Fig. 9. Accuracy ($\mathscr{F}_{classification}$) and performance of nearest neighbor classifier with respect to various precision modes.

VI. Case Studies

While the synthetic data used in the previous section allows us to provide a detailed and targeted analysis of our approach, it does not show how realistic the use of reduced precision is in real world scenarios. In the following, we therefore apply our techniques to three real-world case studies and analyze the applicability of reduced precision for these use cases.

A. Application Classification using HPC-ODA

The classification of applications running on data or supercomputing centers based on monitored characteristics is a highly relevant topic for the center operators. It enables performance optimizations, e.g., via improved scheduling, allows the detection of unwanted application, just to name a few. One such approach to classify application relies on gathered performance metrics, such as resource hardware counters or utilization patterns, which are often stored as large multi-dimensional time series (see Fig. 9). To apply our implementation, we construct a simple application classifier and investigate the effect of reduced precision computation on both classification accuracy and analysis runtime.

We exploit the *Application Classification* segment of a public HPC dataset [9]. This dataset includes labeled performance data collected while running different benchmarks (HPL, AMG, etc.) on 16 compute nodes for one day with 1 HZ sampling rate. We select 16 distinct sensors (performance metrics) on different nodes (e.g., cache miss rates, branch instructions) for the multi-dimensional matrix profile analysis, and split the dataset along time into two portions, a reference set and a query set, each of which includes continuous operational data for half a day. We conduct the matrix profile analysis with different precision modes using our implementation, and build a simple classical nearest neighbor *classifier* on top of the matrix profile analysis: it uses the labels of the matching (based on matrix profile index) segments in reference set to determine the application class of the segments in query set. Fig. 8 illustrates a partial timeline as an example of the query data and the classification scheme we use.

As shown in Fig. 9, while the accuracy of the classifier is reduced slightly with reduced precision modes, for the Mixed and FP16C modes it is still over 95% and even FP16 also reach almost 90%. And slight performance increase is visible when exploiting reduced precision despite the small size of the HPC-ODA dataset.

B. Genome in a Bottle

The Genome in a Bottle (GIAB) Consortium [28] aims at the translation of whole human genome sequencing in clinical practice. It offers public access to genome variants and reference calls to facilitate the human genome analysis, which, e.g., provides information to inherited diseases. As matrix profile can also be used as a general pattern detection approach by interpreting the timestamp as a simple index, we use it here for the analysis of genome sequences by encoding the genomic elements and storing them in a time series like fashion. Especially reduced precision computation can boost this analysis, due to its limited number of element types that are encoded, and with that facilitate the analysis of large amount of data, as it can be found in this case study.

We generate six input 16d time series pairs from the data of Chinese trio with respect to GRCh37 reference genome in GIAB. We use the following transformation relation to encode the genome sequences into a data series/sequence: Adenine (A) to 1; Cytosine (C) to 2; Thymine (T) to 3; Guanine (G)

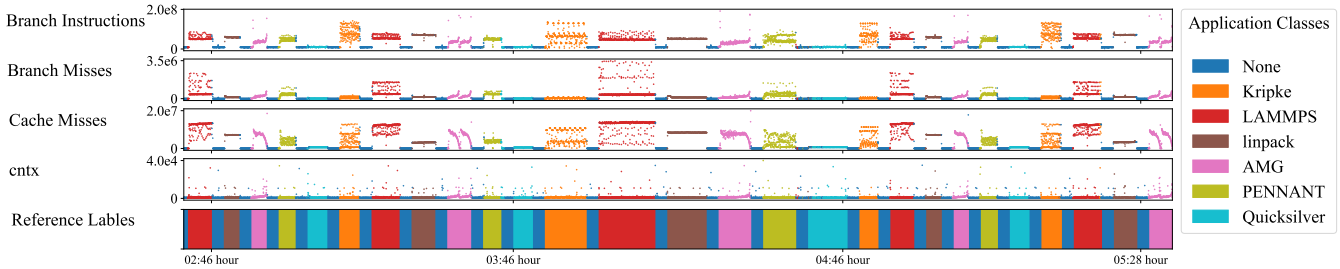


Fig. 8. A timeline of HPC-ODA data color-coded with the classes determined from the nearest neighbor classifier. We are only presenting a subset of data used in this case study, both in the sense of sensors involved and the time span. The sensor data (in the top four rows) are color-coded with the classifier's predicted labels, and we use a color-bar (lowest row) to annotate the ground-truth labels in the HPC-ODA dataset.

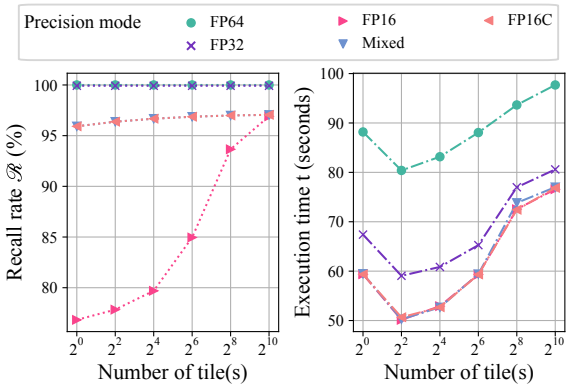


Fig. 10. Numerical accuracy ($\mathscr{R}$) of the matrix profile index and execution time of multi-tile implementations on GIAB dataset when increasing number of tiles (n=2^{18}, d=2^4, m=2^7).

to 4. For the sake of simplicity, we choose the continues data from 16 chromosome and encode them into the time series with the following parameter setting: the number of subsequence $n = 2^{18}$, the dimensionality $d = 2^4$ and the length of subsequence $m = 2^7$—as this length of subsequence reaches the shortest gene length in practice [6].

We execute the numerical accuracy and performance tests with tiling scheme on both V100 and A100 and observe similar behavior. The numerical accuracy of the matrix profile index increases with the number of tiles in FP16, Mixed and FP16C modes. FP16 reaches 75% accuracy with one tile and over 95% with 1024 tiles; Mixed and FP16C reach over 95% accuracy with any number of tiles. The execution time also changes with the increasing number of tiles in the same way as in the previous experiments despite the larger problem size.

Overall, the reduced precision computation works well for DNA data mining with multi-dimensional matrix profile and shows promise for further scaling.

C. Heavy-Duty Gas Turbines

In the final case study we look at relaxed recall rate ($\mathscr{R}^r_{embedded}$) as accuracy metric when applying our approach to pattern detection (e.g., to inspect anomalies) in actually existing large-scale industrial systems with all its real-world limitations, in particular to the surveillance of large-scale heavy-duty gas turbines. Especially with the increasing growth of renewable energy sources in power grids, heavy-duty gas turbines are often operated as a backup source for power generation. Consequently, they are exposed to more dynamic operation settings and therefore to multiple *startups* and *shutdowns* cycles. In order to detect and later predict them, we investigate turbine speed data, collected directly from the machine as a high-frequent time series and especially focus on the detection of startup events. This case study is a special case where the dimensionality of input time series, d is one, but reduced precision is still the key for scaling the analysis.

We exploit a dataset derived from the actual operation of two instances of gas turbines, GT1 and GT2, installed and operated by a large municipal power provider. Fig. 11 illustrates examples of the startup events (annotated with $P1$ and $P2$) in our dataset; each corresponding to a specific operation

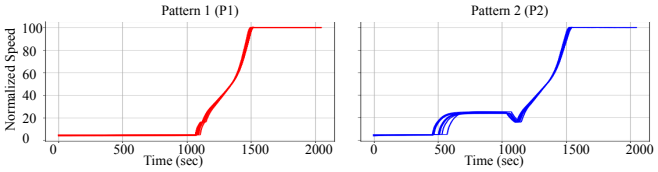


Fig. 11. Startup patterns in heavy-duty turbine datasets. We apply min-max normalization to avoid overflow in reduced precision computation.

initiation mode. We prepare 65 single time series ($n = 2^{16}$, $m = 2^{11}$ and $d = 1$) that include either P1 or P2 respectively, and 5 single time series that include both. In our experiments we use matrix profile on the combinations (pairs) of these time series (i.e., $P1$ and $P2$) to detect the the corresponding startup events in these pairs, which are categorized in four classes (i.e., P1-P1 and P2-P2, ... in Table I). This is sufficiently diverse to analyze the pattern detection accuracy ($\mathscr{R}^r_{embedded}$) within one or across the two turbine instances. The numbers of input time series pairs in each class are listed in Table I.

We apply the methodology for *practical* accuracy $\mathscr{R}^r_{embedded}$ of pattern detection. However, here we allow for a tolerance (a.k.a., relaxation factor) in detecting the location of detected patterns. This tolerance allows for considering fairly close detections (e.g., with 5% of window size deviation) as successful, which is a reasonable consideration for the constraints of this use case. Fig. 12 shows the $\mathscr{R}^r_{embedded}$ of detecting pattern with 5% relaxation factor. Both FP64 and FP32 exhibit 100% accuracy and Mixed and FP16C provide higher accuracy than FP16.

With higher relaxation factors (not shown in Fig. 12), the corresponding startups are all successfully detected: In all the experiments, the mismatching nearest neighbors are not located at the positions range from 10% to 50% tolerance of the startup event. This shows that our algorithm is able to accurately detect the patterns.

In all the tests, the accuracy to detect patterns in P-P1 and P2-P2 classes is similar to both-P1 and both-P2, indicating that the resulting accuracy is independent of the data sources. In addition, as shown in Fig. 11, despite the difference of the two patterns (e.g., the blue pattern is more complex), they are detectable to the almost same degree with Mixed or FP16C. This demonstrates that the accuracy of Mixed and FP16C modes is orthogonal to the complexity of patterns, which is in accordance to the presented experiments in Fig. 3 on the synthetic datasets.

VII. Conclusions

In this paper, we proposed the first GPU-based algorithm to accelerate multi-dimensional matrix profile computation. We provided an implementation that exploits GPU hardware features through an optimized data layout, coarse-grained syn-

TABLE I
CATEGORIES OF TIME SERIES PAIRS, AND THE NUMBERS OF INPUT TIME SERIES PAIRS IN EACH CATEGORY IN GAS TURBINE CASE STUDY.

	P1-P1	P2-P2	both-P1	both-P2
GT1	4160	4160	325	325
GT2	4160	4160	325	325
GT1-GT2	4225	4225	650	650

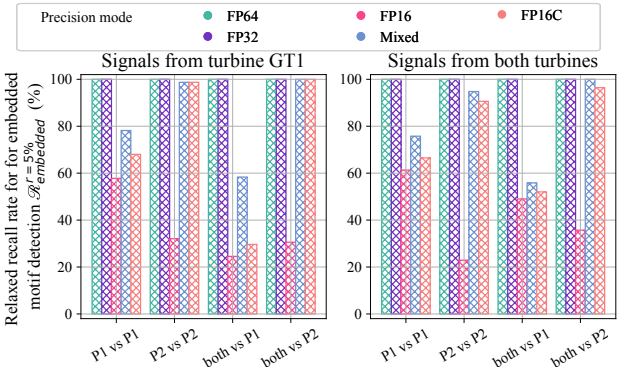


Fig. 12. Relaxed recall rate ($\mathscr{R}^{r}_{embedded}$) of single-tile implementations with various relaxation factors for detecting rising-edge patterns in the time series derived from two turbines.

chronization and efficient sort and inclusive scans. We further provided a solution to utilize multiple GPUs on one node. We introduced various reduced precision modes for multi-dimensional matrix profile computation and demonstrated a novel tiling scheme to prevent the propagation of errors. In the future, our implementation could be further extended to multiple nodes (e.g., using MPI or a Cloud-based solution) as well as using TF32 execution mode or BFLOAT16.

We conducted extensive accuracy and performance measurement experiments using synthetic and real-world data, and showed that our approach achieves high accuracy and performance as well as achieves good strong scalability. Our final solution (multi-GPU + reduced precision + tiling) demonstrates a complete and novel solution for efficient and accurate data mining with reduced precision. Our solution is a significant step towards more efficient HPDA for time series analysis and pushes the limits of state-of-the-art significantly.

Acknowledgment

This work is partially funded by Bayerische Forschungsstiftung under the research grants Optimierung von Gasturbinen mit Hilfe von Big Data (AZ-1214-16), and Von der Edge zur Cloud und zurück: Skalierbare und Adaptive Sensordatenverarbeitung (AZ-1468-20). The authors gratefully acknowledge the Max Plank Computing and Data Facility (MPCDF: www.mpcdf.mpg.de) and Leibniz Supercomputing Centre (LRZ: www.lrz.de) for funding this project by providing compute time on the Raven Supercomputer, BEAST (Bavarian Energy Architecture & Software Testbed), and LRZ AI systems.

References

[1] DGX-1 v100 at LRZ AI Systems. [Online]. Available: https://doku.lrz.de/display/PUBLIC/LRZ+AI+Systems

[2] Supercomputer Raven at Max Plank Computing and Data Facility. [Online]. Available: https://www.mpcdf.mpg.de/services/supercomputing/raven

[3] The UCR matrix profile page. [Online]. Available: https://www.cs.ucr.edu/~eamonn/MatrixProfile.html

[4] F. Cheng, R. J. Hyndman, and A. Panagiotelis, "Manifold learning with approximate nearest neighbors," *ArXiv*, vol. abs/2103.11773, 2021.

[5] I. Fernandez, "scrimp-flexfloat," https://github.com/ivanfv/scrimp-flexfloat, 2019.

[6] V. Grishkevich and I. Yanai, "Gene length and expression level shape genomic novelties," *Genome research*, vol. 24, pp. 1497–1503, 2014.

[7] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, no. 1, p. 40, Jan. 1965.

[8] S. M. Law, "STUMPY: A powerful and scalable python library for time series data mining," *The Journal of Open Source Software*, vol. 4, no. 39, p. 1504, 2019.

[9] A. Netti, "HPC-ODA dataset collection," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3701440

[10] NVIDIA. CUB library, release: 1.14.0. [Online]. Available: https://nvlabs.github.io/cub

[11] NVIDIA-Corporation. Modern GPU library, release: 2.0. [Online]. Available: https://github.com/moderngpu/moderngpu/wiki

[12] G. Pfeilschifter, "Time series analysis with matrix profile on HPC systems," 2019.

[13] A. Raoofy, R. Karlstetter, D. Yang, C. Trinitis, and M. Schulz, "Time series mining at petascale performance," in *International Conference on High Performance Computing*. Springer, 2020, pp. 104–123.

[14] J. C. Romero, A. Vilches, A. Rodríguez, A. Navarro, and R. Asenjo, "Scrimpco: scalable matrix profile on commodity heterogeneous processors," *The Journal of Supercomputing*, pp. 1–22, 2020.

[15] N. S. Shakibay Senobari, G. Funning, Z. Zimmerman, Y. Zhu, and E. J. Keogh, "Using the similarity matrix profile to investigate foreshock behavior of the 2004 parkfield earthquake," *AGUFM*, vol. 2018, pp. S51B–03, 2018.

[16] J. Shi, N. Yu, E. Keogh, H. K. Chen, and K. Yamashita, "Discovering and labeling power system events in synchrophasor data with matrix profile," in *2019 IEEE Sustainable Power and Energy Conference (iSPEC)*. IEEE, 2019, pp. 1827–1832.

[17] D. F. Silva, C.-C. M. Yeh, G. E. Batista, E. J. Keogh *et al.*, "Simple: Assessing music similarity using subsequences joins." in *ISMIR*, 2016.

[18] G. Tagliavini, A. Marongiu, and L. Benini, "Flexfloat: A software library for transprecision computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 145–156, 2020.

[19] A. Tharwat, "Classification assessment methods," *Applied Computing and Informatics*, 2020.

[20] L. Yang, A. Fox, and G. Sanders, "Rounding error analysis of mixed precision block householder qr algorithms," *SIAM J. Sci. Comput.*, vol. 43, pp. A1723–A1753, 2021.

[21] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh, "Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile," *Data Mining and Knowledge Discovery*, vol. 32, no. 1, pp. 83–123, 2018.

[22] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh, "Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets," in *2016 ICDM*. IEEE, pp. 1317–1322.

[23] C.-C. M. Yeh, N. Kavantzas, and E. Keogh, "Matrix profile VI: Meaningful multidimensional motif discovery," in *2017 ICDM*. IEEE, pp. 565–574.

[24] Y. Zhu, Z. Schall-Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. J. Funning, A. A. Mueen, P. Brisk, and E. J. Keogh, "Exploiting a novel algorithm and GPUs to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins," *Knowledge and Information Systems*, vol. 54, pp. 203–236, 2017.

[25] Y. Zhu, C.-C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh, "Matrix profile XI: Scrimp++: time series motif discovery at interactive speeds," in *2018 ICDM*. IEEE, pp. 837–846.

[26] 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 *2016 ICDM*. IEEE, pp. 739–748.

[27] Z. Zimmerman, K. Kamgar, N. 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 *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 74–86.

[28] J. M. Zook, J. McDaniel, N. D. Olson, J. Wagner, H. Parikh, H. Heaton, S. A. Irvine, L. Trigg, R. Truty, C. Y. McLean *et al.*, "An open resource for accurately benchmarking small variant and reference calls," *Nature biotechnology*, vol. 37, no. 5, pp. 561–566, 2019.