Concurrency and Computation: Practice and Experience

WILEY

SPECIAL ISSUE PAPER OPEN ACCESS

Leveraging SYCL for Heterogeneous cDTW Computation on CPU, GPU, and FPGA

Cristian Campos | Rafael Asenjo | Javier Hormigo | Angeles Navarro

Department of Computer Architecture, University of Málaga, Málaga, Spain

Correspondence: Cristian Campos (cricamfe@ac.uma.es)

Received: 29 November 2024 | **Revised:** 29 April 2025 | **Accepted:** 15 May 2025

Funding: This work was supported by the Ministry of Economy and Competitiveness, Government of Spain, under Grant Nos. TED2021-131527B-I00 and PID2022-136575OB-I00. Funding for open access publishing: Universidad de Málaga/CBUA.

Keywords: cDTW | energy efficiency | FPGA | GPU | heterogeneous architecture | heterogeneous scheduling | SIMD | SYCL

ABSTRACT

One of the most time-consuming kernels of a recent epileptic seizure detection application is the computation of the constrained Dynamic Time Warping (cDTW) Distance Matrix. In this paper, we explore the design space of heterogeneous CPU, GPU, and FPGA implementations of this kernel using SYCL as a programming model. First, we optimize the CPU implementation leveraging the SIMD capability of SYCL and compare it with the latest C++26 SIMD library. Next, we tune the SYCL code to run on an on-chip GPU, iGPU, as well as on a discrete NVIDIA GPU, dGPU. We also develop a SYCL implementation on an Intel FPGA. On top of that, we exploit simultaneous co-processing on CPU+GPU and CPU+FPGA platforms by extending a previous heterogeneous scheduling framework to now support 2D partitioning strategies. Our evaluations demonstrate that SYCL seems well suited to exploit the SIMD capabilities of modern CPU cores and shows promising results for accelerating devices, both in terms of performance and energy efficiency. Moreover, we find that our scheduler enables the efficient co-execution of work among the computing devices, and the results demonstrate that dynamic and adaptive partitioning strategies perform efficiently with overheads below 4%.

1 | Introduction

Achieving high performance alongside reduced energy consumption in modern computing systems increasingly necessitates the adoption of heterogeneous programming paradigms. The objective, often summarized as "leaving no transistor behind," involves harnessing diverse computational units like CPUs, GPUs, and FPGAs concurrently. Emerging programming models such as SYCL [1] (that enables different heterogeneous devices to be exploited in a single code), DPC++, and oneAPI [2] aim to simplify the development of applications for these complex systems while preserving performance potential.

This work focuses on accelerating a computationally demanding operation within an epileptic seizure detection algorithm using a heterogeneous platform comprising a CPU, an integrated GPU (iGPU), a discrete NVIDIA GPU (dGPU), and an FPGA. We meticulously adapt the core functions for each device type—CPU, GPUs, and FPGA—prioritizing energy efficiency and considering programmability. Specifically, we evaluate SYCL alongside modern C++ SIMD capabilities [3] for the CPU. For the GPUs and FPGAs, we utilize the SYCL compiler, leveraging its High-Level Synthesis (HLS) features for the FPGA component. Furthermore, a significant aspect of this research involves exploring cooperative execution between the CPU and

accelerators (GPU or FPGA). We achieve this by extending a lightweight heterogeneous scheduler, originally based on HBB [4], to manage workload distribution across CPU cores and accelerator compute units. This extension incorporates novel two-dimensional (2D) partitioning strategies, and we assess their impact on performance and energy efficiency across different hardware configurations.

The application domain, epileptic seizure detection, addresses a significant societal challenge, as epilepsy is a prevalent neurological condition worldwide [5]. Our objective is to facilitate the design of a wearable device (e.g., glasses, headband) using a low-power heterogeneous platform. This device would utilize just two electrodes to capture electroencephalography (EEG) signals and alert patients and caregivers to impending seizures. The underlying seizure detection method [6] relies on computing a Distance Matrix based on constrained Dynamic Time Warping (cDTW) [7]. This computation is both computationally and data-intensive, demanding rapid processing, particularly when tailored using patient-specific EEG data. We posit that such biomedical applications represent a compelling use case for leveraging SYCL's capabilities on energy-constrained heterogeneous systems.

In summary, this paper presents the following novel contributions:

1. Two distinct CPU implementations for cDTW Distance Matrix calculation: One utilizing SYCL's SIMD features and another employing the standard C++26 SIMD library [3] used as reference.
2. Accelerator-optimized implementations for GPU and FPGA, developed using the oneAPI SYCL compiler and its associated High-Level Synthesis framework for the FPGA target, discussing the cross-platform challenges of SYCL.
3. A practical assessment of the SYCL programming model regarding performance portability and energy efficiency across four distinct hardware architectures (CPU, iGPU, dGPU, FPGA), including an analysis of the strengths and limitations observed for each implementation.
4. The extension of a lightweight heterogeneous scheduler to support dynamic and adaptive 2D partitioning of the computational workload, enabling efficient co-execution on CPU cores and an accelerator (GPU or FPGA), along with an experimental evaluation quantifying the benefits and overheads of these strategies on the target heterogeneous platforms.

The remainder of this paper is organized as follows. Section 2 provides essential background on the problem domain and reviews pertinent related work. Sections 3 and 4 detail the implementation strategies for CPU, GPU, and FPGA, respectively, elaborating on design choices made to exploit device-specific microarchitectural features. Section 5 introduces the extended heterogeneous scheduler, focusing on the 2D iteration space partitioning methods for enabling efficient co-execution. Section 6 describes the experimental setup and presents a comprehensive evaluation of performance and energy efficiency for the various implementations and co-execution strategies. Finally, Section 7 summarizes the findings and concludes the paper.

2 | Background and Related Work

2.1 | Basic Concepts and Notation

We utilize Figure 1 to clarify essential terminology for understanding the proposed EEG analysis. A signal, or channel S^c, represents a discrete-time sequence of real-valued numbers $s_i^c \in \mathbb{R}$, formulated as $S^c = \{s_i^c;\ 0 \le i < n\}$, where n denotes the sequence length and c is the channel identifier. Each value s_i^c corresponds to the electrical potential measured between two electrodes at a sampling time t_i. We employ channel labels from the 10–20 system [8] for identification (e.g., S^{F3-C3} in Figure 1 represents the potential between electrodes F3 and C3).

A seizure, $Z_{k,z}^c$, signifies a subsequence within channel S^c starting at index k with length z, that is, $Z_{k,z}^c = S_{k,z}^c$, identified as a seizure event. The dataset employed provides metadata (md) specifying onset and offset timestamps for each seizure, from which the values of k and z are derived. The presence of seizures partitions the signal S^c into ictal (S^{c+}, seizure) and interictal (S^{c-}, non-seizure) segments, as illustrated in Figure 1. A query on channel S^c, Q^c, is the concatenation of all N_z^t seizures in S^{c_t} one after the other in the order they appear. This can be expressed as: $Q^c = (Z_{k_1,z_1}^c, Z_{k_2,z_2}^c, \ldots, Z_{k_{N_z^t},z_{N_z^t}}^c)$. The total length of the query is $n_q = \sum_{i=1}^{N_z^t} z_i$. For example, in Figure 1, Q^{F3-C3} is the concatenation of the two seizures in F3-C3.

To identify seizure patterns within a channel S^c, we analyze fixed-length subsequences, termed epochs, extracted with

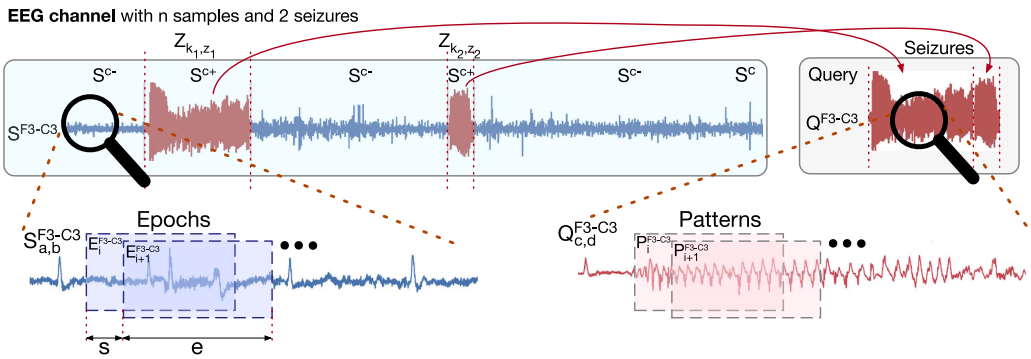

FIGURE 1 | The problem context and notation conventions.

a constant stride s. These epochs effectively segment the channel using a sliding window approach. More precisely, an epoch $E_i^c = S_{k,e}^c$ possesses length e, with its starting position $k \in \{i \cdot s;\ 0 \le i < n_e\}$, where the total number of epochs is $n_e = \lfloor (n - e)/s \rfloor + 1$. Analogously, patterns are defined as sliding window subsequences of length e and stride s within the query Q^c. A pattern $P_i^c = Q_{k,e}^c$ has its starting position $k \in \{i \cdot s;\ 0 \le i < n_p\}$, with $n_p = \lfloor (n_q - e)/s \rfloor + 1$ being the total number of patterns in a query of length n_q. Figure 1 depicts representative epochs $(E_i^{F3-C3}, E_{i+1}^{F3-C3})$ from subsequence $S_{a,b}^{F3-C3}$ and patterns $(P_i^{F3-C3}, P_{i+1}^{F3-C3})$ from query subsequence $Q_{c,d}^{F3-C3}$.

Recall that our objective is to automatically identify one or more patterns capable of distinguishing between seizure (E^+) and non-seizure (E^-) epochs in an EEG channel. An ideal pattern exhibits a conserved shape prevalent in seizure epochs but absent in non-seizure ones. This discrimination is achieved by measuring similarity through distance computation between patterns and epochs. Given the multitude of potential patterns in the query, we must calculate a quality metric for each to pinpoint the one with the highest discriminative power. The foundational step for this process is the computation of the Distance Matrix, DM, which stores the distances between every pattern P_i and every epoch E_j, as visualized in Figure 2. We denote the distance between pattern i and epoch j as $d_{i,j} = d(P_i, E_j)$.

The specific distance measure, $d(P_i^c, E_j^c)$, between a pattern P_i^c and an epoch E_j^c, is calculated using Dynamic Time Warping (DTW) [7, 9]. DTW is a widely adopted algorithm for comparing temporal sequences that might exhibit variations in phase or speed, and it is recognized for its robustness as a similarity metric [10]. Its significant computational cost compared to simpler metrics like the Euclidean Distance (ED) has driven the development of optimized variants. A prominent simplification is the constrained DTW (cDTW) [9], which restricts the alignment path to a band (the Sakoe-Chiba band or warping window, w) around the cost matrix's main diagonal. Setting the warping window $w = 0$ reduces cDTW to ED. cDTW offers a pragmatic balance between ED's speed and DTW's accuracy. In this research, we employ cDTW with a warping window parameter $w = 16$. Further optimizations like lower-bounding, early abandoning, and pruning techniques [11] have been proposed, particularly for Nearest Neighbor (NN) searches, although our primary focus remains the computation of the complete distance matrix DM.

As an illustrative example, Figure 3 shows the computation of the cDTW distance with a Sakoe-Chiba warping window of size 2. The top-left panel displays the Euclidean Distance calculation between a sample Epoch, E, and Pattern, P, each comprising $e = 6$ data points. In this approach, every point in the Pattern is matched with its corresponding point in the Epoch, yielding a Euclidean Distance of $d_E = \sum_{i=0}^{e-1}(P_i - E_i)^2$. Here, points P_i from the Pattern can align with points E_j from the Epoch where j falls within the range $\{i-2, i-1, i, i+1, i+2\}$. The resulting cDTW distance is calculated as $d_{DTW} = \sum_{i=0}^{e-1}(P_i - E_j)^2$, selecting the index j within the warping window that minimizes the squared difference for each i. For the example shown, this distance totals $d_{DTW} = 4 + 0 + 1 + 1 + 1 + 0 + 1 + 1 = 9$. The warping path, indicated by the gray highlighting, traces the specific pairs (i, j) that contribute to this minimum cDTW distance.

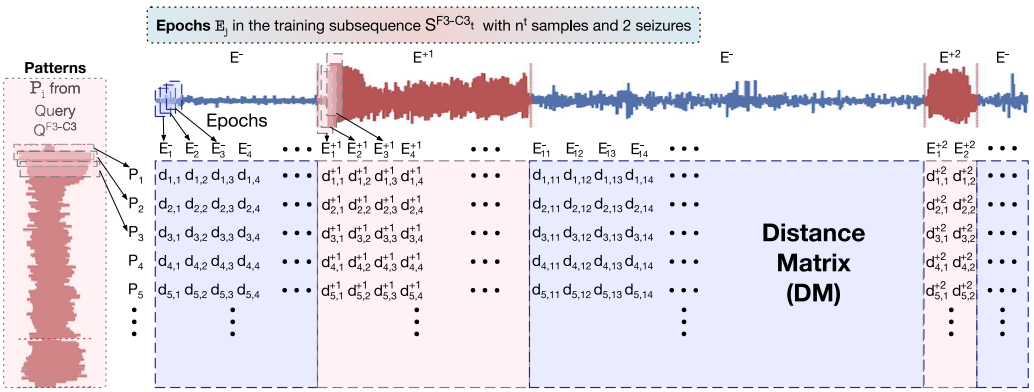

FIGURE 2 | Conceptual representation of the Distance Matrix (DM).

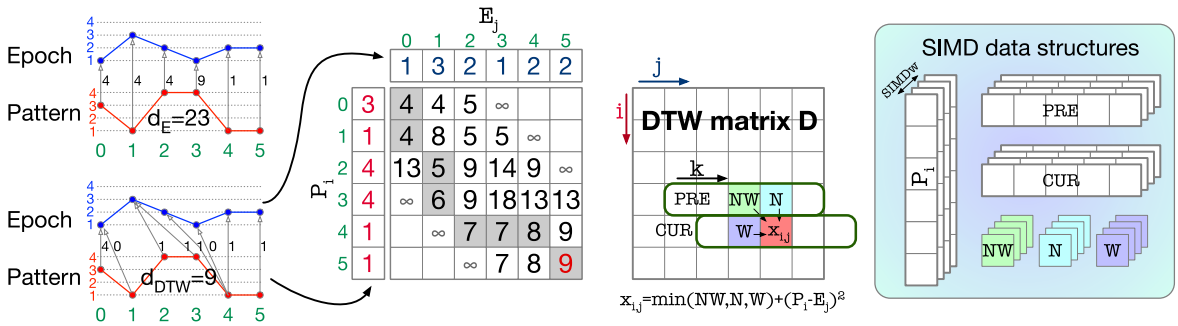

FIGURE 3 | Euclidean Distance, cDTW distance example with $w = 2$, and data structures.

To compute the distance between two data point vectors, P and E, the cDTW algorithm conceptually traverses a virtual matrix, D (of size $e \times e$), which is not explicitly stored. The traversal proceeds from the top-left corner to the bottom-right. The value at cell (i, j) represents the cumulative distance up to that point and is determined by adding the squared Euclidean Distance between the individual points, $d(P_i, E_j) = (P_i - E_j)^2$, to the minimum value among its three neighboring cells in the matrix D: North (N), North-West (NW), and West (W), as depicted in Figure 3. The final cDTW distance corresponds to the value accumulated in the matrix's bottom-right cell. Elements lying outside the diagonal band defined by $\pm w$ are effectively excluded by initializing them to ∞.

Memory efficiency can be enhanced, as depicted on the left in Figure 3, by storing only the essential matrix information required for the band computation. This is achieved using two vectors, PRE (previous) and CUR (current), each of size $2 \cdot w + 1$ (which is 5 elements in the figure's example). These vectors hold only the necessary elements within the current computational band. At the conclusion of each row iteration i, the roles of PRE and CUR are interchanged, and CUR is populated with the newly computed values for the next row. Following the example in Figure 3, at iteration $i = 4$, the PRE vector would contain {6,9,18,13,13} (values from iteration $i = 3$). For calculating the element at $j = 4$, the neighbors are $NW = 18$, $N = 13$, and $W = 7$. Given the local distance $d(P_4, E_4) = 1$, the value $x_{4,4}$ is computed as $min(18, 13, 7) + 1 = 8$.

In our seizure detection method [6] we need to calculate all the cDTW distances of the Distance Matrix because they are accessed many times to compute the quality metrics to identify the best epilepsy seizure pattern. The size of the Distance Matrix is the product of the number of patterns (n_p) and the number of epochs (n_e), a quantity that can become computationally prohibitive. For context, consider the first patient from the CHB-MIT dataset [12] with typical parameters ($e = 1024$ samples, corresponding to 4 s, and $s = 256$ samples, or 1 s). Each of the 23 channels yields over 145,000 epochs and 269 patterns. This translates to computing nearly 40 million cDTW distances per channel. Executing a widely available Python DTW library [13] for a single distance calculation between two 1,024-sample subsequences on a standard laptop can take tens of milliseconds. Extrapolating this suggests that processing just one channel for a single patient could require months of computation without significant acceleration.

2.2 | State of the Art

Our research focuses on optimizing the computation of the full cDTW Distance Matrix (DM) and evaluating its performance and energy efficiency across diverse architectures (CPU, iGPU, dGPU, FPGA) using SYCL as a unified programming model. There are related works concerning different types of DTW computation, as well as the calculation of various distance matrices.

For example, the `dtwParallel` package [14] implements constrained DTW (cDTW) and offers parallelization strategies for both univariate and multivariate time series using Python, although their evaluation focuses on relatively small-scale problems compared to the processing requirements demanded by our EEG analysis scenario. Further efforts targeting CPU architectures encompass vectorization techniques employing SIMD [15] or specific instruction sets like SSE [16] to accelerate distance calculations. Although demonstrating efficacy on CPUs, these methods typically exhibit limitations concerning portability and do not inherently support heterogeneous execution paradigms.

On GPUs, `cuDTW++` [17] presents a parallelization strategy to accelerate DTW computation on CUDA-enabled GPUs using low-latency warp intrinsics for fast interthread communication. Although this implementation achieves over 90% of the theoretical maximum performance on Volta-based GPUs, it uses unconstrained DTW, which has a higher computational complexity than the cDTW approach we have adopted in this project. Similarly, in Reference [18], a proposed CUDA implementation was proposed that incorporates Sakoe-Chiba constraints, demonstrating effective scalability. More recently, `sDTW` [19] adapted analogous concepts to AMD architectures using HIP [20] and ROCm [21] for subsequence matching, which differs from the sliding window analysis central to our methodology. Nevertheless, these GPU-centric solutions are generally coupled with specific vendor ecosystems (NVIDIA CUDA or AMD HIP/ROCm) and lack native support for heterogeneous co-execution involving disparate device types.

For FPGAs, dedicated hardware implementations, such as the one presented in Reference [18], have reported substantial performance accelerations (up to four orders of magnitude) over their software counterparts, whereas diagonal computation strategies, exemplified by [22], have focused on latency reduction. Although notable efficiency is demonstrated, such hardware-specific designs often present limitations in terms of development flexibility and abstraction compared to High-Level Synthesis (HLS) methodologies. More recently, we proposed a very reduced DTW computation unit for FPGAs in Reference [23]. This unit made very efficient use of the utilized resources, and its architecture is the basis of the FPGA kernel used in this work. This efficiency has allowed us to build a high-throughput FPGA kernel by replicating this computation unit, as we will explain in Section 4.

Regarding the computation of distance matrices, in Reference [24], the authors propose a parallel algorithm, STAMP, to compute the Matrix Profile, based on a distance matrix that is used to find similar subsequences in time series. SCRIMP [25] supersedes STAMP computing, in parallel, the diagonals of the distance matrix. SCAMP [26] leverages the Pearson correlation to compare subsequences, instead of using the ED. A CPU+GPU implementation of SCRIMP and a CPU+FPGA implementation of SCAMP were introduced in References [27] and [28], respectively. In Reference [29] the DTW is used to compute the matrix profile instead of the ED, but the distance matrix is not explicitly computed. In Reference [30], a GPU-only framework is proposed for the discovery of motifs and discord using cDTW, achieving significant speedups by identifying reusable computation patterns. However, the aforementioned solution is focused on anomaly detection and GPU specialization, while our approach targets the full cDTW distance matrix, which is essential in our case, and supports co-execution on multiple device types, which is especially relevant for resource-constrained platforms.

As a distinctive feature of previous works, the present study employs the SYCL standard to establish a unified programming model for implementing the cDTW and the DM computation across diverse architectures (CPU, GPU, FPGA). Our approach seeks to markedly improve code portability and, critically, enables efficient heterogeneous co-execution on CPU+GPU and CPU+FPGA platforms via a specifically designed scheduling framework. This solution addresses the inherent limitations of single-architecture or vendor-locked solutions, which are particularly relevant for applications such as critical epilepsy detection.

3 | CPU and GPU Implementations

3.1 | Distance Matrix Implementations

Several parallelism opportunities arise during the cDTW Distance Matrix computation. On the CPU, for instance, a straightforward parallel approach involves using OpenMP to distribute the computation across the epoch dimension (typically the largest dimension) of the Distance Matrix, as shown in Listing 1. We refer to this implementation as our baseline, Distance Matrix `BASE`. This implementation defines a class `DTWBase` which inherits from a common `DTWInterface`. The core computation happens within the `compute_DTW_chunk` method (lines 5–22). Inside this method, OpenMP is used with a `#pragma omp parallel for` directive (line 10) to parallelize the outer loop iterating over epochs (`ep`). Within each thread handling an epoch, the code sequentially traverses the patterns (`pat`) in an inner loop (line 15). For each epoch-pattern pair, the actual cDTW calculation is performed by instantiating and calling the `calculate_dtw()` method of a `DTWKernel` object (line 18), using local buffers `CUR` and `PRE` (line 12). The size of the warping window `w` is defined as a class member (line 25).

Listing 1 | Pseudo-code of the Distance Matrix BASE implementation.

```
1  class DTWBase: public DTWInterface {
2  public:
3    DTWBase(const SampleData& data, const ProgramConfig& config)
4      : DTWInterface(data, config) {}
5    FloatVector compute_DTW_chunk(const BlockRange& blk) override {
6      size_t nEpo = std::min(blk.nEpo, blk.totEpo - blk.eId);
7      size_t nPat = std::min(blk.nPat, blk.totPat - blk.pId);
8      FloatVector result(nEpo * nPat);
9      // BASE: OpenMP parallelization across the epoch dimension
10     #pragma omp parallel for shared(result)
11     for (int ep = blk.eId; ep < blk.eId + nEpo; ep++) {
12       dtw_t CUR[2 * w + 1], PRE[2 * w + 1];
13       const data_t* E = &data.S[ep * data.epoch_size];
14       // Sequential traversal through patterns - no SIMD vectorization
15       for (int pat = 0; pat < nPat; pat++) {
16         const data_t* P = &data.Q[(blk.pId + pat) * data.pattern_size];
17         result[(ep - blk.eId) * nPat + pat] =
18           DTWKernel(E, P, data.epoch_size, w, CUR, PRE).calculate_dtw();
19       }
20     }
21     return result;
22   }
23
24 private:
25   constexpr size_t w = 16; // DTW warping window size
26 };
```

To further leverage parallelism, particularly along the pattern dimension, we implemented multiple SIMD vectorization strategies that complement epoch-level parallelism in the Distance Matrix `BASE` implementation. This vectorization enables concurrent evaluation of multiple pattern-epoch comparisons within a single instruction stream, substantially enhancing computational throughput. Our experimental framework comprised three distinct algorithmic implementations: (i) the `BASE` implementation with sequential pattern traversal serving as our performance baseline (Listing 1); (ii) a `SIMD`-optimized implementation employing the SIMD library targeted for inclusion in the C++26 standard [3] (currently accessible through the `std::experimental::simd` namespace in compliant compiler toolchains); and (iii) a `SYCLCPU` implementation utilizing SYCL's native SIMD abstractions and vectorization capabilities. Additionally, we extended our evaluation to heterogeneous architectures via SYCL's device-agnostic programming model, implementing a `SYCLGPU` variant.

Listing 2 presents the microarchitectural details of our SYCL-based implementations. While SYCL facilitates code portability across diverse compute architectures, achieving optimal performance necessitates platform-specific algorithmic adaptations. The `DTWSYCL` class employs polymorphic device selection through architecture-specific selectors (lines 6 and 8 utilizing `sycl::gpu_selector_v` and `sycl::cpu_selector_v`, respectively). The `compute_DTW_chunk` method (lines 11–46) computes workload dimensions (lines 12–13) before dispatching the computational kernel to the appropriate SYCL device queue for execution.

Listing 2 | Pseudo-code of the Distance Matrix SYCLCPU and SYCLGPU implementations.

```
1  #include <sycl/sycl.hpp>
2  class DTWSYCL: public DTWInterface {
3  public:
4      DTWSYCL(const SampleData& data, const ProgramConfig& config) : DTWInterface(data, config) {
5          #if defined(USE_GPU)
6              syclQueue = sycl::queue(sycl::gpu_selector_v);
7          #else
8              syclQueue = sycl::queue(sycl::cpu_selector_v);
9          #endif
10     }
11     FloatVector compute_DTW_chunk(const BlockRange& blk) override {
12         size_t nEpo = std::min(blk.nEpo, blk.totEpo - blk.eId);
13         size_t nPat = std::min(blk.nPat, blk.totPat - blk.pId);
14         FloatVector result(nEpo * nPat);
15         // Initialize buffers and accessors for data & results
16         // CODE...
17         syclQueue.submit([&](sycl::handler& h) {
18             #if defined(USE_GPU)
19                 // GPU: One thread per cDTW calculation
20                 h.parallel_for(sycl::range(nEpo, nPat), [=](sycl::id idx) {
21                     const data_t* E = &data_acc[idx[0] * data.epoch_size];
22                     const data_t* P = &pattern_acc[idx[1] * data.pattern_size];
23                     dtw_t CUR[2 * w + 1], PRE[2 * w + 1];
24                     result_acc[idx[0] * nPat + idx[1]]
25                         = DTWKernel(E, P, data.epoch_size, w, CUR, PRE).calculate_dtw();
26                 });
27             #else
28                 // CPU: SIMD-vectorized cDTW computations
29                 h.parallel_for(sycl::range(nEpo), [=](sycl::id idx) {
30                     size_t ep = idx[0];
31                     dtw_t CUR[2 * w + 1], PRE[2 * w + 1];
32                     const data_t* E = &data_acc[ep * data.epoch_size];
33                     for (size_t pat_start = 0; pat_start < nPat; pat_start += SIMDw) {
34                         // Process SIMD-width patterns at once
35                         for (size_t pat = pat_start; pat < std::min(pat_start + SIMDw, nPat);
       pat++) {
36                             const data_t* P = &pattern_acc[pat * data.pattern_size];
37                             result_acc[ep * nPat + pat]
38                                 = DTWKernel(E, P, data.epoch_size, w, CUR, PRE).calculate_dtw();
39                         }
40                     }
41                 });
42             #endif
43         });
44         // Wait for completion and copy results back to the host
45         // CODE...
46     }
47
48 private:
49     sycl::queue syclQueue;
50     constexpr size_t w = 16; // DTW warping window size
51 };
```

In the SYCLGPU implementation (lines 20–25), we employ a two-dimensional parallelization strategy that maps individual work-items to distinct epoch-pattern combinations. This approach exploits the massively parallel architecture of GPUs while maintaining coalesced memory access patterns essential for optimal bandwidth utilization. The CUR and PRE band registers (line 23) reside in thread-private memory, eliminating synchronization overhead and potential bank conflicts. To optimize data locality, computation blocks are staged from global device memory to thread-private variables prior to processing, reducing memory access latency and enhancing register-level data reuse. We also evaluated shared memory utilization (`local_accessor`) as an alternative optimization: While it improved performance by 30% on discrete GPUs, it caused an 85% degradation on integrated GPUs. To preserve cross-architecture portability, we adopted the thread-private memory implementation. This device-optimized approach delivered a 31% performance improvement compared to a direct port of the baseline algorithm to the GPU.

The SYCLCPU implementation (lines 29–38) combines thread-level parallelism with SIMD vectorization in a hierarchical strategy. Thread parallelization occurs across the epoch dimension (line 29), with each work-item processing a complete epoch against all patterns. Within each thread, the pattern-traversal loop (line 33) processes patterns in SIMDw-width blocks, leveraging SIMD instruction-level parallelism. This approach exploits both multi-core capabilities and vectorized execution units while optimizing cache utilization. The implementation enhances temporal data locality by maintaining epoch data in cache while processing successive pattern blocks, minimizing memory access latency, and improving bandwidth utilization. This device-aware strategy reduced execution time by 32% compared to a direct port of the baseline implementation.

3.2 | Versions for cDTW Kernel

Listing 3 details how the data type `dtw_t`, used for the cDTW calculations within the different implementations, is defined. Specifically, `dtw_t` aliases to `float` for BASE and SYCLGPU (line 4), reflecting their scalar nature at the pattern level. For SIMD using the C++ library, it is defined as `simd < data_t, stdx::simd_abi::fixed_size>` (line 6), while for SYCLCPU, it becomes `sycl::float16` (line 8), both representing vector types. The constant SIMDw, defined in line 11, captures the SIMD vector width (number of lanes, e.g., 16 floats for `sycl::float16`), which is crucial for the vectorized loops in SIMD and SYCLCPU.

Listing 3 | Definition of dtw_t for BASE, SIMD, SYCLCPU and SYCLGPU.

```
1  namespace stdx = std::experimental;
2  using data_t = float;
3  #if defined(BASE) || defined(SYCLGPU)
4      using dtw_t = data_t;
5  #elif defined(SIMD)
6      using dtw_t = stdx::simd<data_t, stdx::simd_abi::fixed_size>;
7  #elif defined(SYCLCPU)
8      using dtw_t = sycl::float16;
9  #endif
10 #if defined(SIMD) || defined(SYCLCPU)
11     constexpr size_t SIMDw = dtw_t::size();
12 #endif
```

A pseudo-code representation of the DTWKernel class, responsible for cDTW calculation, is provided in Listing 4. Leveraging conditional compilation and template parametrization based on `dtw_t`, this class encapsulates the logic for BASE, SIMD, and SYCL versions. The SIMD data structures employed within the code, primarily the PRE and CUR band registers and the NW, N, and W variables, are conceptually visualized on the right side of Figure 3.

Listing 4 | Pseudo-code of cDTW kernel (used by BASE, SIMD, SYCLCPU, SYCLGPU).

```
constexpr dtw_t maxval{std::numeric_limits<data_t>::max()}; // Infinite for out-of-band values
class DTWKernel {
  public:
    DTWKernel(const data_t *E, const data_t *P, size_t e, int w, dtw_t *CUR, dtw_t *PRE)
        : E{E}, P{P}, e{e}, w{w}, CUR{CUR}, PRE{PRE} {}
    dtw_t calculate_dtw() {
        for (int i=0; i<2*w+1; i++) CUR[i] = PRE[i] = maxval; // Init band registers
        int k = 0;
        for (int i = 0; i < e; i++) { // Traverses the rows of DTW Matrix D
            k = max(0, w - i);
            for (int j = max(0, i - w); j <= min(e - 1, i + w); j++, k++) // Traverses the band
                CUR[k] = this->operator()(i, j, k);                       // Compute x_ij
            std::swap(CUR, PRE); // Swap current row with previous one
        }
        return PRE[k-1];        // Return the final distance value
    }
  private:
    dtw_t operator()(int i, int j, int k) {
        dtw_t N, W, NW;
        #ifdef SIMD
            dtw_t Psimd{&P[j * SIMDw], stdx::element_aligned}; // Load pattern vector
            dtw_t d = dist(E[i], Psimd);  // Compute the distance epoch-pattern (SIMD op.)
        #elif SYCLCPU
            dtw_t Psimd;                    // Load pattern vector
            Psimd.load(0, sycl::multi_ptr<const dtw_t, sycl::access::address_space::global_space>(&P[j * SIMDw]));
            dtw_t d = dist(E[i], Psimd); // Compute the distance epoch-pattern (SIMD op.)
        #elif defined(BASE) || defined(SYCLGPU)
            dtw_t d = dist(E[i], P[j]);  // Compute the distance epoch-pattern (scalar op.)
        #endif
        if ((i == 0) && (j == 0)) return d;
        if ((j<=0)||(k<=0))     W = maxval;  // Output of E or CUR bounds
        else                    W = CUR[k-1];
        if ((i<=0)||(k>=2*w))   N = maxval;  // Output of P or PRE bounds
        else                    N = PRE[k+1];
        if ((i<=0)||(j<=0))     NW = maxval; // Output of P or E bounds
        else                    NW = PRE[k];
        #ifdef SIMD
            // where(N<W, W) = N; where(W<NW, NW) = W;                    // Inefficient, up to 3.62x slower
            NW = stdx::min(stdx::min(N, W), NW);                         // Min. of N, W and NW
            return NW + d;
        #elif SYCLCPU
            return sycl::fmin<dtw_t>(sycl::fmin<dtw_t>(N, W), NW) + d; // Min. of N, W and NW
        #elif defined(BASE) || defined(SYCLGPU)
            return min(min(N, W), NW) + d;                               // Min. of N, W and NW
        #endif
    }
    // Private data members:
    const data_t *E, *P; // E points to an epoch, P to several patterns in SIMD and SYCLCPU
    size_t e;            // Number of samples of the epoch and pattern
    int w;               // DTW warping window (16 in our experiments)
    dtw_t *CUR, *PRE;    // Band registers, of SIMD type in SIMD and SYCLCPU
};
```

The `DTWKernel` constructor initializes the input arrays with an epoch, `E`, and several patterns, `P`. However, the pattern entries have been previously rearranged by the caller so that consecutive values correspond to different patterns. The member function `calculate_dtw()` (line 6) traverses the rows (`i` loop) of the DTW matrix D (see Figure 3) and the band registers (zipped `j-k` loop), calling at each iteration to the `operator()` function (line 18). In that function and for SIMD and SYCLCPU, a `Psimd` vector is loaded with `SIMDw` values of the corresponding pattern sample, P_i, (see Figure 3) to then compute the distances in parallel on the SIMD units. For BASE (line 28), a single distance is computed per iteration. In lines 30–36, W, N and NW variables (of SIMD type in the corresponding case) are initialized so that we can compute the x_{ij} distance (lines 37–45) of Figure 3 and store it in `CUR[k]` (line 12), to finally swap the current and previous band registers (line 13).

A performance issue was identified in the `where` SIMD function of the SIMD C++ library. By replacing `where` by `stdx::min` (see lines 38 and 39), a significant performance improvement was observed (up to 3.62× on our evaluation platforms).

In the BASE and SIMD cDTW versions, the function `calculate_dtw()` is called from a double-nested loop that traverses both the epochs and the patterns, using OpenMP `parallel_for` in the outer loop that traverses the epochs. In SYCLCPU, a data-parallel kernel approach is used so that the same function is called inside a kernel submitted to the CPU SYCL queue (that also feeds the 8 CPU cores of our platforms).

For the GPU cDTW version, SYCLGPU, we take the BASE cDTW code. As in SYCLCPU, the SYCLGPU version invokes the function `calculate_dtw()` from a kernel, which is now submitted to the corresponding GPU queue (the iGPU or the dGPU). In Section 6 we describe the methodology employed to identify the optimal global work and work-group sizes for each GPU device.

Compiler considerations for SYCL. While SYCL enables a unified programming model, achieving performance portability requires attention to compiler flags and tuning. For example, omitting `-fsycl-target-loopopt`, which enforces loop-level optimizations, may lead to performance degradations of up to 50% depending on the compiler version and the target device. This flag impacts both CPU and GPU backends and should be explicitly set when not enabled by default. Additionally, the `-fsycl` flag is mandatory to compile SYCL kernels regardless of the selected platform. Also, to compile for the NVIDIA GPU (dGPU), we leverage the interoperability features of the Intel oneAPI DPC++/C++ Compiler by adding the required compiler flags (as `-fsycl-targets` and `-Xsycl-target-backend`). These issues reflect the need for careful tuning and awareness of backend-specific behaviors in SYCL compilation.

4 | FPGA Implementation

The FPGA implementation deserves its own section because, although it is also written in SYCL and compiled with the oneAPI SYCL compiler (and its High-Level Synthesis features), the code used for the CPU and GPU has to be substantially rewritten to get the most out of the FPGA. Moreover, the compiler needs special flags to set some characteristics of the target hardware, such as the specific board (-Xsboard) or the target clock frequency (-XsclocK). Still, the SYCL language allows us to quickly prototype and explore the design space.

In addition to code rewriting, efficient hardware design requires the use of special libraries provided by the FPGA vendor and specialized directives. For instance, we use the Intel "LSU" class to specify the kind of Load-Store Unit that we will use to communicate with the main memory, and the "PipeArray" one to instantiate a FIFO structure to communicate different modules within the kernel. The directives are specified within double brackets and direct the compiler on how the code has to be translated to a hardware design. In Listing 5, we show a pseudo-code of the main parts of the SYCLFPGA implementation.

Listing 5 | Pseudo-code of the Distance Matrix SYCLFPGA implementation highlighting its kernels structure.

```
// Inter-kernel function communication pipes
using VpatternPipe = fpga_tools::PipeArray<VpatternPipeId, data_t, stride * nPatFPGA, NumRep, Pep + 1>;
using VeppochPipe = fpga_tools::PipeArray<VeppochPipeId, data_t, stride + 10, NumRep, Pep>;
using ResultPipe = fpga_tools::PipeArray<ResultPipeId, data_t, nPatFPGA, NumRep, Pep>;

// Definition of the basic kernel
class DTWSYCLF {
public:
    void kernelDTW(const BlockRange& blk, FloatVector& result) {
        syclQueue.submit([&](sycl::handler& h) {
            // Pattern generation kernel function
            h.single_task<PatternReader>([=]() [[intel::kernel_args_restrict]] {
                data_t regPat[nPatFPGA][sEpoPat];
                for(int m = 0; m < nPatFPGA; m++) { //store patterns
                    data_t meanPat = sttQPrftchLSU::load(sttQ_ptr++);
                    for(int j = 0; j < sEpoPat; j++)
                        regPat[m][j] = normalizePattern(QPrftchLSU::load(Q_ptr++), meanPat);
                }
                for(int it = 0; it < nEpFPGA/Pep; it++)
                    for(int j = 0; j < sEpoPat; j++)
                        for(int m=0; m<nPatFPGA; m++){ VpatternPipe::write(regPat[m][j]);
                        }
                    }
                }
            });
            // Epoch generation kernel function
            h.single_task<EpochReader>([=]() [[intel::kernel_args_restrict]] {
                for(int it=0; it<nEpFPGA+Pep-1; it++){
                     meanEp[indx]=SttPrftchLSU::load(SttS_ptr++);
                     for(int j=0; j<strideEpo; j++){
                        data_t Vsignal=normalize(S[it*strideEpo+j], meanEp[indx]);
                        fpga_tools::UnrolledLoop<0, (Pep)>([&](auto s) {
                             if (it>=s and it<s+nEpFPGA) VeppochPipe::PipeAt<s>::write(Vsignal);
                        });
                    }
                }
            });
            // cDTW computation kernel function
            h.single_task<DTWProcessor>([=]() [[intel::kernel_args_restrict]] {
                ShiftRegMem<data_t, sBand> regEp;
                ShiftRegPow2<data_t, (sBand-1)*nPatFPGA> dtw_buff;
                data_t vPat[nPatFPGA];
                [[intel::disable_loop_pipelining]] for(int it = 0; it < nEpFPGA/Pep; it++) {
                    initializeRegisters(regEp, dtw_buff);
                    for(int j = 0; j < sEpoPat; j++) {
                        regEp.shift(VeppochPipe::PipeAt<Replica, ID>::read());
                        for(int i = 0; i <= sBand; i++) {
                            [[intel::ivdep]] for(int m = 0; m < nPatFPGA; m++) {
                                if(i == 0) ActualizePattern(m, vPat);
                                else if(isWithinBand(i, j))
                                    processDTW(m, i, j, regEp, vPat, dtw_buff);
                                if(lastElement) write_result(dtw);
                            }
                            if(i != 0) regEp.CirShift();
                        }
                    }
                }
            });
         ----
         ----
         ----
        });
    }
    // Replication of the basic kernel for data parallelism
    FloatVector compute_DTW_chunk(const BlockRange& blk) {
        FloatVector result(std::min(blk.nEpo, blk.totEpo - blk.eId) * std::min(blk.nPat, blk.totPat - blk.pId));
        fpga_tools::UnrolledLoop<NumRep>([&](auto k){
            kernelDTW({e, std::min(Pep, blk.nEpo-e), p, std::min(nPatFPGA, blk.nPat-p)}, result);
        }
        return result;
    }
private:
    sycl::queue syclQueue = sycl::queue(sycl::ext::intel::fpga_selector_v);
};
```

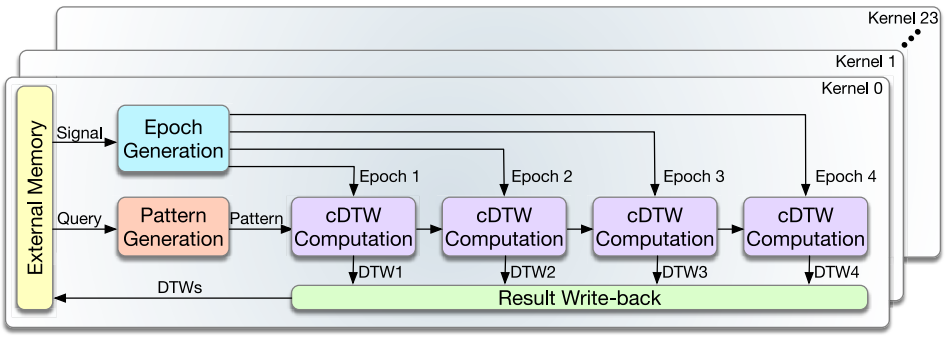

FIGURE 4 | Basic kernel for cDTW computation and kernel replication on the FPGA.

The core of the proposed architecture is based on the basic cDTW accelerator circuit presented in Reference [23]. This circuit combines a very simple iterative scheme, pipelining, and computation interleaving to produce a very efficient circuit in terms of throughput using a very reduced area. It has been adapted to the Intel FPGA architecture and this specific application to create our computation module. This computation module is replicated to achieve a high-throughput architecture. Besides the computation module (`DTWProcessor()` kernel function, lines 38–58), we include other modules to deal with input and output data. All of these modules have been designed with the idea of reducing access to the external memory as much as possible by generating all the computations associated with specific data. When this is not possible, an internal cache memory system has been used to keep the read data on the FPGA while it is still useful. These other modules are:

- An epoch generation module (`EpochReader()` kernel function, lines 26–37) to extract the different epochs from the input signal, to perform on-the-fly the normalization of each epoch according to the statistical parameters received, and send the normalized epoch to the computation module.
- A pattern generation module (`PatternReader()` kernel function, lines 11–25) to extract the different patterns from the input query, to perform on-the-fly the normalization of each pattern according to the statistical parameters received, to store it, because they are used multiple times, and to send it to the computation core.
- A result write-back module to order and send back the results of the cDTW computation.

Figure 4 shows and scheme of the proposed basic kernel. This kernel is replicated as much as possible to maximize the FPGA utilization, taking advantage of the parallelism at the data level. That is, the input signals are divided into one input block for each replication of the basic kernel (`compute_DTW_chunk()`, lines 64–71). Each input block comprises nE_{FPGA} epochs (`nEpFPGA` in the listing) and nP_{FPGA} patterns (`nPatFPGA` in the listing). In our implementation, the number of epochs in the block for which we find the sweet spot is $nE_{FPGA} = 512$. Furthermore, as we will see later, we select $nP_{FPGA} = 32$, since it is the lower power of two that allows us to overcome the latency of the computation pipeline. With these values, in this particular implementation, we were able to fit $nR = 24$ basic kernels in our Intel FPGA.

Each basic kernel (`kernelDTW()`, lines 9–63) comprises several computation modules (`DTWProcessor()`, lines 38–62) and one of each of the other modules to feed them with epoch and pattern values and to read the computed cDTW distance. The computation modules work in parallel with different epochs but the same patterns. Hence, the different epochs are sent in parallel to the computation modules, whereas the patterns are sent serially from one computation module to the next (see Figure 4). There are as many computation modules in a basic kernel as strides fit in an epoch. In this case, there are four computation modules on each basic kernel.

The different modules communicate with each other with queues (FIFOs) defined using oneAPI libraries as shown in lines 1–4 in the listing. The synchronization of the whole system is data-driven based. That means each module controls how much data needs to be read or produced, and the queue control signals stall the modules whenever required. The depth of the queues is established to maximize throughput while keeping it as low as possible. Next, we explain the modules in detail.

4.1 | Computation Module

As we have said, the computation module is based on the circuit in Reference [23]. It computes the elements of the Distance Matrix row by row, from top to bottom (for loop in line 45), and each element of the row or band, serially, from left to right (for loop in line 47). This circuit comprises a pipelined datapath to calculate, in parallel, the Euclidean squared distance of the corresponding points of the signal and the minimum value of the three neighboring elements, which is then added (function `processDTW()` in line 51). To provide the values of the neighbor, a shift register stores the previous M elements computed, where M is the length of a row in a regular cDTW or the width of the Sakoe-Chiba band. This shift register has three access points to read the three neighboring values in parallel. The corresponding value of the pattern is stored locally in a register since it is used throughout all the row computations, whereas the epoch element is continuously read from the epoch memory. Since the immediately previous element is required for the current element's calculation, an interleaved computation scheme is used to hide the latency of the pipeline. On interleaving computation, nP different and independent cDTW distances are computed almost simultaneously, one new element of each different Distance Matrix in turns. If nP is large enough, a new computation could start on

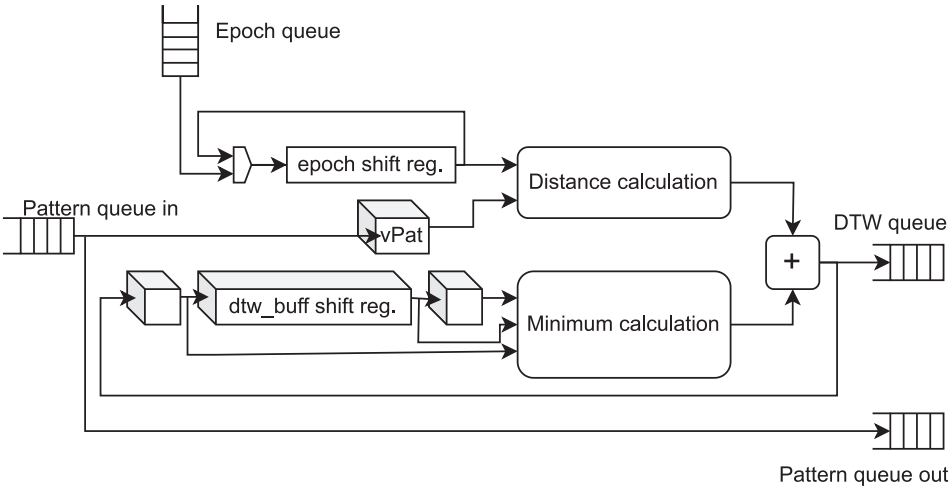

FIGURE 5 | Computation module.

each cycle, the pipeline is fully used, and the maximum throughput is achieved. This approach fits perfectly with our application since each epoch is compared with multiple patterns.

This circuit has been adapted and improved to create our computation module, which is shown in Figure 5. The cDTW of nP different patterns and the same epoch are computed in an interleaved fashion (for loop in line 48). In our particular application, since the depth of the computation pipeline is below 30 stages, we fixed the number of patterns to $nP = 32$. The elements of the epoch are received serially by the epoch queue (see Figure 5 and line 46 in the listing). The computation of a row of the DTW matrix is performed only for a Sakoe-Chiba band. Therefore, only the elements of the epoch within the band are required for computing each row, $B = 2w + 1 = 33$ elements. For the next row, the band is right-shifted by one position. Hence, a circular shift register (`regEp` in line 40) with the size of the band minus one ($B - 1$) is used to store the elements of the epoch required for processing a row. This register is shifted to compute each new element. Before starting a new row, the oldest element (the one on the right) is substituted by a new element from the queue, which is inserted on the left side. In this way, the epoch elements are read only once, whereas the minimum area is used for storage.

In contrast, only one element of the pattern is required to process a row. This element is kept in a standard register (`vPat` in line 42). However, since the cDTW computation is interleaved for 32 patterns, one element of each pattern is stored to process a row for each simultaneous cDTW computation. Before starting a new row, all 32 new values are read from the pattern queue (see Figure 5 and line 49 in the listing). Since the same values are used by all parallel computation modules, the read values are also sent using the pattern queue out (see Figure 5).

As in Reference [23], the previous $(B - 1) \times nP = 32 \times 32$ elements computed for each cDTW are stored in a long shift register of length L_{buff} in an interleaved fashion (`dtw_buff` defined in line 41). However, Intel FPGAs are not as optimized as AMD FPGAs to implement long-shift registers with only one read point. Hence, we studied different options to implement this register and the one for the epoch elements. The best solution for the target FPGA is to implement them using a typical circular buffer in the embedded RAM blocks. A counter is used to point to the address to read the last element and introduce the next one. This counter is incremented when a shift operation is required. When the counter reaches the length of the register, it must be reset to zero to keep the circular buffer working. The typical software implementation of this counter using module operation produces very costly and slow hardware since it involves a division. Instead of this, we use a conditional statement that compares the counter with the length of the buffer to either increment it or reset it, which is much more hardware-friendly. However, the best solution can be used if the length of the buffer is a power of two. Fortunately, this is the case with the `dtw_buff`. In this case, the counter is defined with the exact number of bits ($\log_2(L_{\text{buff}}) = 10$), which ensures that the counter wraps around organically when the limit of the buffer is reached.

4.2 | Pattern Generation Module

This module generates and locally stores (in `regPat`, line 13) the different patterns by reading the query from the external memory (line 17). The circuit is presented in Figure 6. In contrast to the epochs, which are sent just once to the computation module, the patterns have to be repeatedly sent because they are compared with each epoch (see for loop in line 19, where `Pep` is the number of computation modules working in parallel in a basic kernel, 4 in our case). Since the number of patterns is orders of magnitude less than the number of epochs, they are easily stored in the FPGA memory to reduce access to the external memory. Therefore, this module starts initializing the internal memory with all the z-normalized patterns (lines 14–18). Since this initialization occurs only once, the elements of each pattern, along with their statistical parameters, are read from the external memory and normalized on the fly, pattern by pattern. A more detailed description of the z-normalization is provided in the next section.

Once all patterns are in local memory, the patterns are sent to the computation modules using the same queue (lines 19–24). Note that the computation module interleaves the computation of the cDTWs corresponding to one epoch with all the patterns.

Thus, first, the first element of all patterns is sent (loop in line 21), then the second element, and so on (loop in line 20). When all elements of patterns are sent, the process starts again, sending the first element of the patterns, and so on (loop in line 19). The process finishes when all cDTWs have been calculated.

Since the same patterns are used for all epochs, those are transmitted serially from one computation unit to the next (see Figure 4). Hence, the queue of the pattern generation module is connected to the first computation module. When the computation module reads a new element from the input pattern queue, it uses the value but also sends it forward to the second computation module through an output pattern queue (see Figure 5). This scheme is repeated in each computation module until the last one, which only uses the value.

4.3 | Epoch Generation Module

This module generates the different epochs by reading the signal from the external memory. Since there is an overlap between consecutive epochs, each element of the signal is present in several epochs. To minimize the number of reads from external memory and to increase performance, this module reads the elements of the signal in order one by one (see line 31 in the listing) and generates in parallel all the epochs where this element is included (lines 32–34). In this way, the signal elements are read just once, and the epoch is generated as soon as possible. Figure 7-left shows the proposed circuit for our application, where the length of the epoch, L_{epoch}, is 1,024, which is four times the length of the stride $s = 256$. Consequently, four consecutive epochs are created in parallel on each represented queue, although with the corresponding timing difference. The first $s = 256$ elements are only fed to the first epoch. The next s elements are fed to the first and second epoch, and so on until the stationary regime is reached (see condition in line 33). Similarly, the opposite transitory regime is required for the last elements of the signal.

Since our application requires that the epoch is Z-normalized, each element of the signal is z-normalized on the fly before entering the corresponding epoch queue (line 31). This requires subtracting the mean and dividing by the standard deviation of the corresponding epoch. Taking advantage of having a fixed database for our application, and to alleviate the computational cost, these two constants are pre-computed and stored in memory with the signal. One may think that having the epochs pre-normalized is a similar solution, and it avoids the actual z-normalization operations. However, it multiplies the size of the database and the required bandwidth by a factor of $L_{epoch}/s = 4$, while the proposed one has a negligible increase in storage and bandwidth. As Figure 7 shows, an arithmetic unit locally stores the two constants of each corresponding epoch. For each signal element, it subtracts the mean and multiplies it by the inverse of the standard deviation corresponding to each active epoch serially (line 31) and sends it to the corresponding epoch queue. Each epoch queue is connected to a different computation module, which starts the cDTW computation as soon as any data is available.

Although the z-normalization could be parallelized for each epoch, this parallelization does not offer any advantage. The parallel version of this module increases the number of utilized DSPs and LUTs, and furthermore, it does not improve the throughput since this module is not the bottleneck of the kernel. Actually, it decreases throughput since the clock frequency slightly decreases.

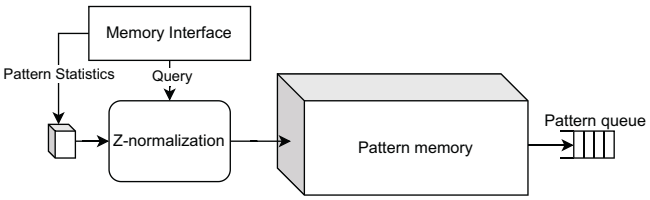

FIGURE 6 | Pattern generation module.

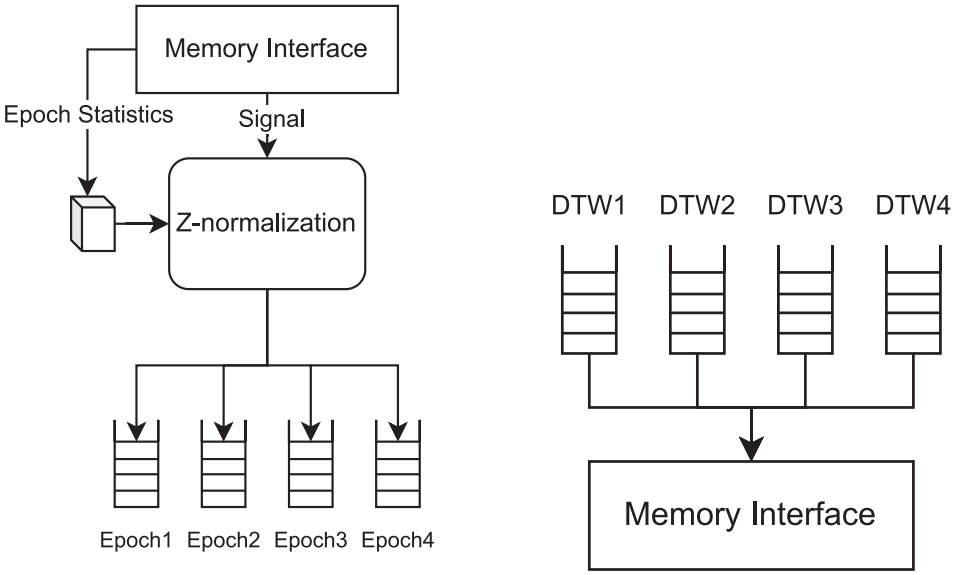

FIGURE 7 | Epoch generation module (left) and Result write-back module (right).

4.4 | Result Write-Back Module

This module, represented in Figure 7-right, collects the cDTW results from a specific queue associated with each computation module and writes them in the external memory. The results come from the computation module for all patterns and sequentially for each epoch, that is, the cDTW for the first epoch with all patterns, then the second epoch with all patterns, and so on. However, this module generates the memory address to store them by patterns, that is, the cDTW for all epochs with the first pattern, then all epochs with the second pattern, and so on. This order could be changed easily since the required bandwidth for writing back the results is small, which allows us to write them one by one in any order. Note that this module is not present in Listing 5 for the sake of saving space, but its structure is similar to the epoch generation module.

5 | Heterogeneous Scheduler: CPU+GPU and CPU+FPGA

In this section, we present the heterogeneous scheduler used to enable CPU+Accelerator co-execution of our application. We base our new proposal on the Heterogeneous Building Blocks (**HBB**) library API. This is a C++ template library that offers a heterogeneous `parallel_for` function template, which can be invoked with several partitioning strategies to extract chunks of iterations of the iteration space, and then to schedule them among the cores and an accelerator. The initial `parallel_for` template, was devised to simultaneously orchestrate the work between the CPU cores and a GPU [4], which we extended to run on systems comprising CPU cores and OpenCL/SDSoC capable FPGAs [31]. Later, we re-implemented the scheduler and partitioning strategies on top of oneAPI to ease the development of the GPU/FPGA SYCL kernels [32]. In this work, we extend some of the partitioning strategies to handle 2D chunks of iterations. The proposed partitioning strategies are designed to be agnostic to the specific computation kernel, relying only on the ability to partition the workload (in our case, the cDTW Distance Matrix rows and columns) between heterogeneous devices. This makes the proposed approach applicable to other data-parallel problems for which 2D data partitioning has been shown to be more scalable than 1D strategies such as the lattice Boltzmann method (LBM) or sparse matrix computations.

HBB offers an abstraction layer that hides the initialization and management details of oneTBB and SYCL constructs (contexts, command queues, device_ids, etc.), thus users can focus on their own application kernels instead of dealing with thread management and synchronization. From the programmer perspective, the implementation of the heterogeneous versions only requires the initialization of the scheduler where the partitioning strategy is selected, and next the call of the `parallel_for` template function that receives three arguments: First iteration, last iteration, and an object of a class that implements the `operatorCPU()` and `operatorACC()` member functions. Each function encapsulates the computed chunk sizes and execution parameters in a bundle object, which is passed to the corresponding `compute_DTW_chunk()` function shown in Listings 2 and 5.

5.1 | Scheduler Engine

The internal engine that manages the *parallel_for* template is implemented with a two-stage parallel pipeline template from oneTBB, as we see in Figure 8. For this pipeline, tokens representing a compute unit class, that is, a CPU core, GPU, or FPGA accelerating device (ACC), are defined. The number of tokens represents the number of chunks of the iteration space that will be processed simultaneously at a given time. In the figure, we have N CPU tokens and one ACC token (GPU/FPGA) being processed concurrently, N on the CPU cores, and one more on the accelerator. Tokens are recycled until there are no remaining iterations.

The first stage, Stage 0, gets an available token (line 1), and depending on the token type, ACC (Tk_{ACC}) or CPU (Tk_{CPU}), selects the chunk size for the accelerator or CPU core, respectively. To do so, it extracts the corresponding chunk of iterations (Ch_{ACC} or Ch_{CPU}, respectively), invoking function `CompCh()` (line 2). This function takes three parameters: The Partitioning Strategy, $PStrat$, the type of token, and the set of remaining iterations, r. Different partitioning strategies can be configured in

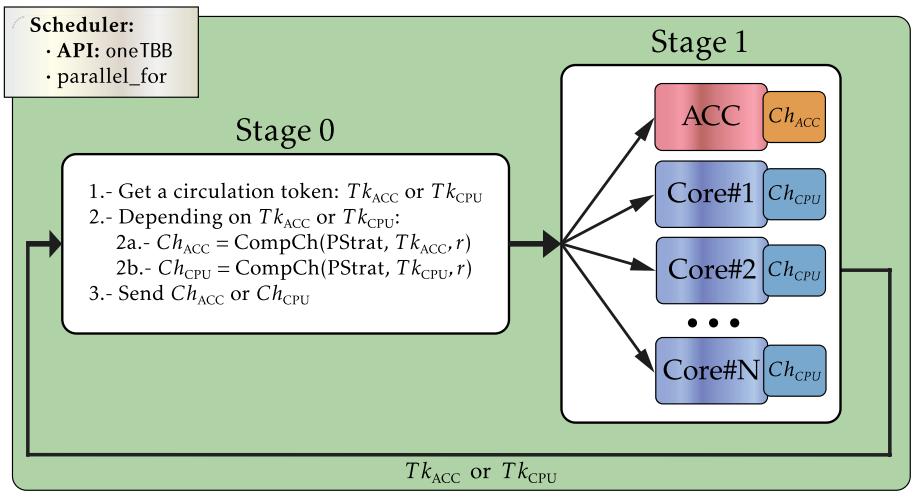

FIGURE 8 | Scheme of the scheduler engine. Stage 0 performs the partitioning and scheduling, while Stage 1 computes the assigned chunks.

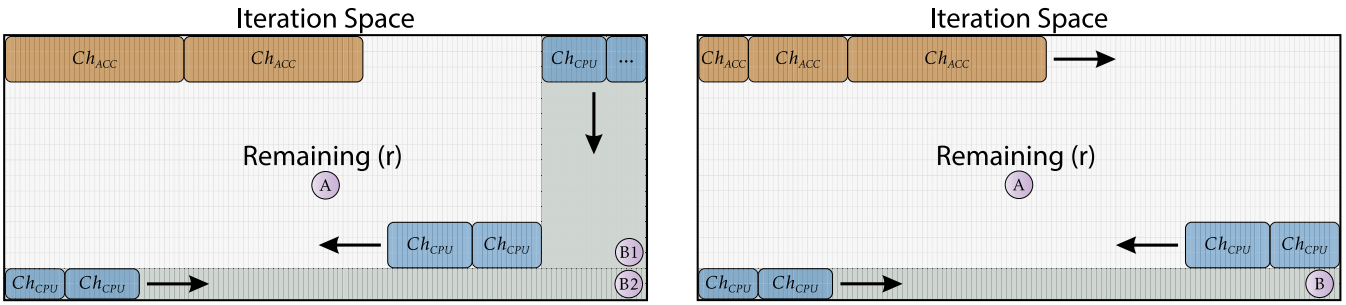

FIGURE 9 | Dynamic and LogFit partitioning strategies. The iteration space represents the $nE \times nP$ DM (Distance Matrix) to compute. (a) Dynamic strategy, (b) LogFit strategy.

our scheduler, and we explain in Section 5.2 those that have been used in this work. Once the chunk has been computed, it is scheduled to be launched on the accelerator or a CPU core, depending on the token (line 3). Finally, when the token reaches Stage 1, the chunk is processed on the corresponding device. The time required for the computation of the chunk on the accelerator or a CPU core is recorded. These times are used to update the relative speed of the accelerator w.r.t. a CPU core, which we call φ. Factor φ will be required by our partitioning strategies to adaptively adjust the size of the next chunk assigned to a CPU core because our scheduler always tries to balance the time consumed by each accelerator and CPU chunks [32].

5.2 | Partitioning Strategies

For this application, we have extended two partitioning strategies supported in our library: Dynamic and LogFit [32]. The original strategies only managed a 1D chunk of iterations, but now we have extended them to handle 2D chunks. In these cases, a chunk is a tuple $<nE_d, nP_d>$, where nE_d and nP_d represent the number of epochs and number of patterns that can be processed on each device, respectively.

The Dynamic strategy is illustrated in Figure 9a. Here, the user has to set an FPGA/GPU chunk size, $Ch_{ACC} =< nE_{ACC}, nP_{ACC} >$. For the FPGA, this size is fixed and given by the epoch generation, computation, and pattern generation modules, and a number of replicated kernels, as explained in Section 4. In our case, $Ch_{FPGA} = < nR \times nE_{FPGA}, nP_{FPGA} > = < 24 \times 512, 32 >$. For the GPU, we have explored different $Ch_{GPU} =< nE_{GPU}, nP_{GPU} >$ sizes to ensure optimal occupancy of the GPU execution units. We cover this discussion in Section 6.3. Both FPGA and GPU always compute chunks from zone A in Figure 9a. On the other hand, the CPU chunk is adaptively computed using factor φ (the relative speed of the accelerator vs. a single core) following a simple heuristic:

- Being nE the number of epochs of the cDTW Distance Matrix and *mod* the modulo operation, if $mod(nE, nE_{ACC}) \neq 0$, then $Ch_{CPU} =< nE_{ACC}/\varphi, nP_{ACC} >$. This is illustrated as zone B1 in Figure 9a.
- Being nP the number of patterns of the cDTW Distance Matrix, if $mod(nP, nP_{ACC}) \neq 0$, then $Ch_{CPU} =< nE_{ACC}/\varphi, mod(nP, nP_{ACC}) >$. This is illustrated as zone B2 in Figure 9a.
- Once cDTW distances have been processed in zones B1 and B2, CPU chunks are computed in zone A as $Ch_{CPU} =< nE_{ACC}/\varphi, nP_{ACC} >$.

The LogFit strategy is illustrated in Figure 9b. It also dynamically splits the iteration space, but the user does not provide the accelerator chunk size. On the contrary, this size is now an adaptive variable that is automatically computed by the partitioner following a logarithmic fitting strategy that has been proved beneficial for irregular codes on CPU + GPU systems [32]. Therefore, we will use this strategy for this type of platform in our experimental evaluation. Under this strategy, first in an exploration phase, the GPU chunk size is iteratively duplicated as $Ch_{GPU}(i+1) =< nE_{GPU}(i) \times 2, nP_{GPU} >$ until the resultant throughput stabilizes. Also, these throughputs are recorded. Next, in a stabilization phase, a logarithmic fitting is applied to the recorded throughputs, and the chunk size that maximizes this curve is computed. From then, this optimal chunk size, $Ch_{GPU} =< nE_{GPU_{opt}}, nP_{GPU} >$, is used in zone A as we see in Figure 9b. The CPU chunk is adaptively computed using factor φ following the same heuristic as explained for the Dynamic partitioning strategy: First $Ch_{CPU} =< nE_{GPU_{opt}}/\varphi, mod(nP, nP_{GPU}) >$ for zone B, and then $Ch_{CPU} =< nE_{GPU}/\varphi, nP_{GPU} >$ for zone A, as we see in the figure.

6 | Experimental Results

The major goal of SYCL is to improve the programmer productivity by allowing different heterogeneous devices to be used in a single application. However, although optimizations in the kernel code may differ across the devices to exploit their specific capabilities, as we have seen in the previous sections, it is yet to be proven that this programming model guarantees performance portability across devices. This is the relevant point that we want to quantify here for our case study. For this, in this section, we conduct a performance evaluation of the various kernel implementations previously discussed. We also explore whether, despite the optimizations, hardware bottlenecks appear in our executions.

The second goal of this section is to evaluate the performance and energy efficiency of the deployment of our kernels running

simultaneously on CPU+GPU or CPU+FPGA platforms, exploring the most effective scheduling and partitioning strategy for each platform.

Our metrics of interest are the throughput (measured as the number of DTWs per millisecond (DTW/ms)) and the energy efficiency (measured as the throughput per Joule (DTW/ms per Joule)).

6.1 | Test Bench

The experimental evaluation was performed on two test benches: AlderLake and SkyLake. All results (time and energy) are reported as the average value of 5 runs. We use 8 cores/threads in all CPU runs.

AlderLake features an *Intel Core i9-12900K CPU*, running at 3.20 GHz, with 8 performance, P, cores and 8 efficiency, E, cores, and 30 MB L3. For our study, we use the taskset command to confine the threads only on the P-cores. This platform also includes the on-chip/integrated GPU, iGPU, *Intel UHD Graphics 770*, and the discrete GPU, dGPU, *NVIDIA RTX 4070 Ti*. It has 128 GB of RAM and runs on *Ubuntu 22.04.5 LTS*. We compile with the Intel oneAPI DPC++/C++ `icpx` compiler version 2024.0.2.

SkyLake has an *Intel Core i7-7820X 3.60 GHz* processor with 8 cores and 11 MB L3, plus 128 GB of DDR4 RAM. In addition, it has an FPGA *Intel Stratix 10 MX* with 32 HBM memory banks, each with 512 MB, totaling 16 GB. The system runs *CentOS 8.1.1911*. This unit lacks a graphics card, thus enabling the execution of all developed versions, with the exception of the GPU version, including the FPGA implementation. On this platform, the baseline and SIMD versions based on C++26 utilize the GCC 12.2.0 compiler, whereas the SYCLCPU and FPGA versions are compiled with the Intel oneAPI DPC++/C++ Compiler 2022.0.0 (this is the latest version supported by our FPGA).

To evaluate energy consumption on both platforms, we use Intel Performance Counter Monitor (Intel PCM) to accurately measure CPU and GPU power usage. In addition, to monitor FPGA power usage, we utilize StratixMonitorLib [33]. We rely on the NVIDIA Management Library (NVML) [34], a specialized API created by NVIDIA to measure numerous metrics of its graphics cards, including the ability to monitor power usage.

As a benchmark, we use a channel, S^c, with 162 h of EEG signal sampled at 256 samples per second, which translates into $n = 150 * 10^6$ samples. The query, Q^c, that contains the epileptic seizures has $n_q = 107,263$ samples. Using epochs and patterns of length $e = 1024$ and stride $s = 256$, we end up with a number of epochs, $n_e = 589,823$ and a number of patterns, $n_p = 415$, which results in a Distance Matrix with more than 244 million of cDTW distances.

6.2 | Evaluation of CPU, GPU and FPGA Implementations

Figure 10 depicts the throughput (DTW/ms) that our different Distance Matrix implementations achieve on AlderLake and SkyLake. BASE represents a parallel OpenMP CPU implementation without the SIMD optimization that we take as the baseline. SIMD represents a CPU implementation based on the C++26 SIMD library, whereas SYCLCPU shows the results for a CPU implementation based on SYCL (see Section 3). In both cases, SIMDw=x states the number of SIMD lanes that have been configured in each evaluation: 4, 8, and 16 floats per SIMD register.

Additionally, in AlderLake, we see the results for the SYCLGPU implementation (see Section 3) running on the integrated GPU (iGPU) and on the discrete GPU (dGPU). On the other hand, in SkyLake we see the results for the SYCLFPGA implementation (see Section 4) running on the Stratix 10 MX. Ideally, our implementation is able to compute 24 DTWs per cycle.

Next, we discuss the performance and energy efficiency of the different implementations in detail.

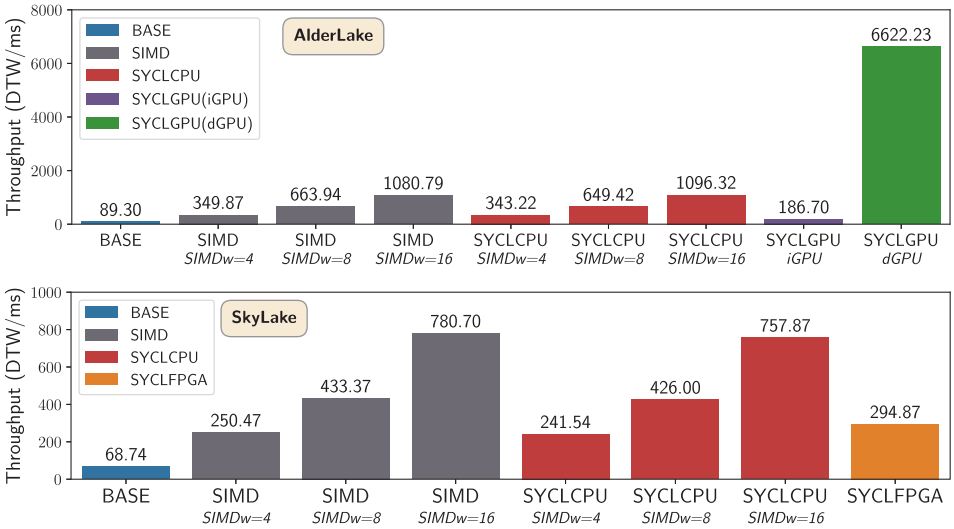

FIGURE 10 | Performance metrics on AlderLake and SkyLake for each implementation. Throughput (DTW/ms). The higher, the better.

TABLE 1 | Study of the reduction operations `stdx::min` and `sycl::fmin` used in SIMD and SYCLCPU implementations, respectively, and for different $SIMDw$ widths in AlderLake. We represent % of CPU time.

	SIMD ($w = 8$)	SIMD ($w = 16$)	SYCLCPU ($w = 8$)	SYCLCPU ($w = 16$)
`min/fmin`	10%	17.7%	6.6%	13.8%
`_barrier`	6.3%	12.7%	4.7%	8.4%

6.2.1 | Performance of CPU Implementations

Clearly, from Figure 10, we see that the SIMDimized CPU implementations always outperform the baseline and that increasing the number of lanes improves performance in both platforms, as expected. Moreover, SIMD code based on the C++26 library performs slightly better than the SYCL version, although in AlderLake, SYCLCPU outperforms SIMD with $SIMDw = 16$. The observed performance degradation in SYCL is due to the kernel enqueueing and launching, which represent up to 2% and 3% of overhead in AlderLake and SkyLake, respectively. In any case, the optimal $SIMDw = 16$ version outperforms the baseline by 12.2× and 11.3× in AlderLake and SkyLake, respectively.

To understand the different scalability behavior between SIMD and SYCL implementations in AlderLake, we performed a profiling analysis to identify the hotspots using the Intel Advisor tool. We found that the reduction function needed to compute the cDTW distance (`stdx::min` in SIMD and `sycl::fmin` in SYCL) is less time-consuming in SYCLCPU than in the SIMD version. This function also scales better in SYCLCPU than in SIMD, as we see in Table 1. In fact, a noticeable source of overhead in that function is due to an internal `_barrier` call, whose percentage of CPU time we also show in the Table. This overhead explains the loss of performance in SIMD when compared to SYCLCPU with $SIMDw = 16$.

Additionally, we performed a roofline analysis with the Intel Advisor tool and found that the function that represents the hotspot in the optimal $SIMDw = 16$ code, `DTWKernel()`, is compute-bound, and it features a headroom of just 1.4× and 2.4× to the ideal ALU peak in AlderLake and SkyLake, respectively. In other words, there are no bottlenecks, and the SIMDimization is fully exploiting the CPU capabilities in our CPU implementations (SIMD library-based and SYCL).

6.2.2 | Performance of GPU Implementations

To tune the SYCLGPU versions in AlderLake, we use the Intel GPU Occupancy Calculator tool and the CUDA occupancy calculator, which help us to theoretically explore different iGPU/dGPU job size configurations. We found that the optimal configuration for the iGPU was a global work size of $< 8192, 64 >$ (and a work-group size of 64×1), achieving ideally 98.6% of Execution Unit (EU) utilization. Also, for the dGPU, we found that a global size of $< 167936, 64 >$ (and a block size of 32×1), maximizes ideally the Streaming Multiprocessor (SM) utilization, achieving 97.66% of occupancy. The results shown in Figure 10 correspond to these configurations.

TABLE 2 | FPGA resources utilization on the Intel Stratix 10 MX: Adaptive Logic Modules (ALM), Registers (REG), RAM blocks (RAM), DSP blocks (DSP). Total used in %.

Resource	Accelerator	Device	Total used
ALM	421,684	702,720	93%
REG	1,022,651	2,810,880	50%
RAM	4,118	6,847	79%
DSP	288	3,960	7%

Figure 10 shows that the SYCLGPU version running on the integrated GPU (iGPU) performs 2.5× faster than the baseline. For the SYCLGPU version, the Intel Advisor tool reports a 75% of EU occupancy on the iGPU, which hints that SYCL is reasonably exploiting the capabilities of this GPU. Moreover, this version running on the discrete GPU (dGPU) achieves a 6× improvement over the fastest SYCLCPU. We also carried out a roofline analysis of this implementation using the NVIDIA NSight Compute tool and found that the function that represents the hotspot is memory bound, achieving 81% and 43% of memory and SM occupancy, respectively. Thus, although far from the expected ideal SM occupancy, this version fully exploits the attainable dGPU memory bandwidth, which represents the bottleneck in this device.

6.2.3 | Performance of FPGA Implementation

In SkyLake, we notice that the SYCL version running on the FPGA is 4.3× faster than the baseline, but still 2.5× slower than the best SYCLCPU version. From the Intel oneAPI FPGA Reports tool, we discovered that the loop that represents the hotspot has an initiation interval of 1; it is pipelined and works at a frequency of 300 MHz. This loop is the main loop in the cDTW computation kernel function, `DTWProcessor()` in Listing 5, and from the tool, we learn that although it is fully optimized, the FIFO queues used to send one computed pattern from one computation module to the next are effectively the bottleneck of the implementation because they introduce several stalls.

In Table 2, we show the resource utilization of our 24 replicated kernels (Accelerator) along with the total amount of resources available in this FPGA (Device) and the percentage of total resources utilized by our system (Total used). This latter value includes not only the accelerator itself but also all the logic required for the communication and connection with the global system, which is automatically added by the design tool. It is easily observed that the limiting resources in our design are the logic elements (ALMs) and the internal RAM blocks (RAM). The amount of floating-point arithmetic resources (DSPs) available remains very high since the cDTW is not a computationally-intensive algorithm. In future designs, we should try to increase the use of these elements by increasing parallelism in the computation module.

6.2.4 | Energy Efficiency

The energy consumption metrics for AlderLake and SkyLake are summarized in Figure 11. The solid bars show energy

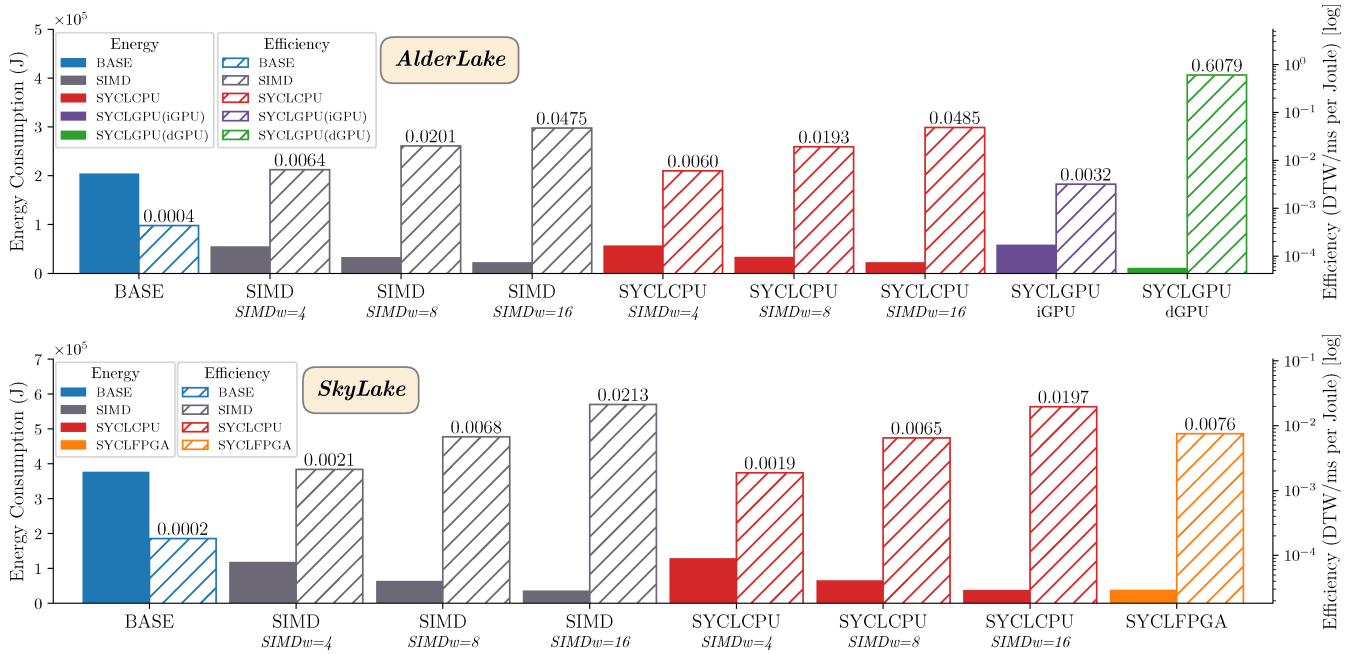

FIGURE 11 | Energy consumption metrics in AlderLake and SkyLake for each implementation. Energy (Joules) and Energy Efficiency (DTW/ms per Joule). The latter metric is on a logarithmic scale, and the higher it is, the better the efficiency.

consumption in Joules (the higher the value, the worse), while the patterned bars show the energy efficiency (DTW/ms per Joule) in log scale (the higher the value, the better the efficiency).

From the energy consumption point of view, the SYCLGPU implementation running on the iGPU and dGPU reports the smallest values on AlderLake. However, from the energy efficiency perspective, the two more efficient implementations are SYCLGPU on dGPU and SYCLCPU with $SIMDw = 16$ on CPU. In fact, these two implementations are 1519× and 121× more energy efficient than BASE, respectively. On SkyLake, the SYCLFPGA implementation exhibits the lowest energy consumption, and its energy efficiency is near the more efficient CPU versions based on SIMD and SYCLCPU with $SIMDw = 16$. For instance, SYCLFPGA and SYCLCPU are 38× and 98× more energy efficient than BASE.

All these results demonstrate the relevant impact that SIMDimization strategies have from the energy evaluation point of view and that SYCL is able to tap into it.

6.3 | Evaluation of CPU+GPU and CPU+FPGA

In this section, we explore the performance and energy results of the simultaneous co-processing of our application in CPU+GPU and CPU+FPGA heterogeneous platforms using the heterogeneous scheduler and partitioning strategies presented in Section 5. In AlderLake, we evaluate Dynamic and LogFit on CPU+GPU, considering the best implementations of SIMD and SYCLCPU for the CPU and the best ones of SYCLGPU for the iGPU and the dGPU, as discussed in the previous section. In SkyLake, we evaluate only Dynamic on CPU+FPGA, given the fact that our SYCLFPGA implementation is optimized for chunks of a fixed size. Again, we consider the best implementations of SIMD and SYCLCPU for the CPU.

6.3.1 | Performance of CPU+GPU

Figure 12 represents the measured throughput vs. the ideal expected throughput for the Dynamic and LogFit partitioning strategies on AlderLake. The figure above (below) depicts the results on the iGPU (dGPU). The expected throughput is computed by aggregating the throughput per device for the corresponding kernel implementation from Figure 10. Thus, this expected value represents an ideal upper bound for each heterogeneous run. We also show the ratio (measured-expected)/expected in red (in %), which quantifies the performance degradation, or in other words, the inefficiency incurred by the scheduler for each partitioning strategy in each heterogeneous configuration. This inefficiency metric inherently captures all sources of overhead associated with heterogeneous scheduling: The kernel enqueueing and launching, the partitioning logic, the data communications between host and device, and any potential load imbalance not perfectly compensated for by the dynamic workload adjustments.

In Figure 12, the GPU kernel is the optimized SYCLGPU, while the CPU kernels are the baseline OpenMP (BASE), the optimized SIMD $SIMDw = 16$ (SIMD), and the optimized SYCLCPU $SIMDw = 16$ (SYCLCPU) versions discussed in Section 6.2.1. The GPU chunk sizes for the Dynamic strategy on the iGPU and dGPU were selected according to the exploration discussed in Section 6.2.2. On the other hand, the LogFit strategy found the following optimal chunk sizes after its exploration phase: $< 32768, 64 >$ for the iGPU, and $< 262144, 64 >$ for the dGPU. The CPU chunks are adaptively computed for each partitioning strategy during the execution (as explained in Section 5.2). In

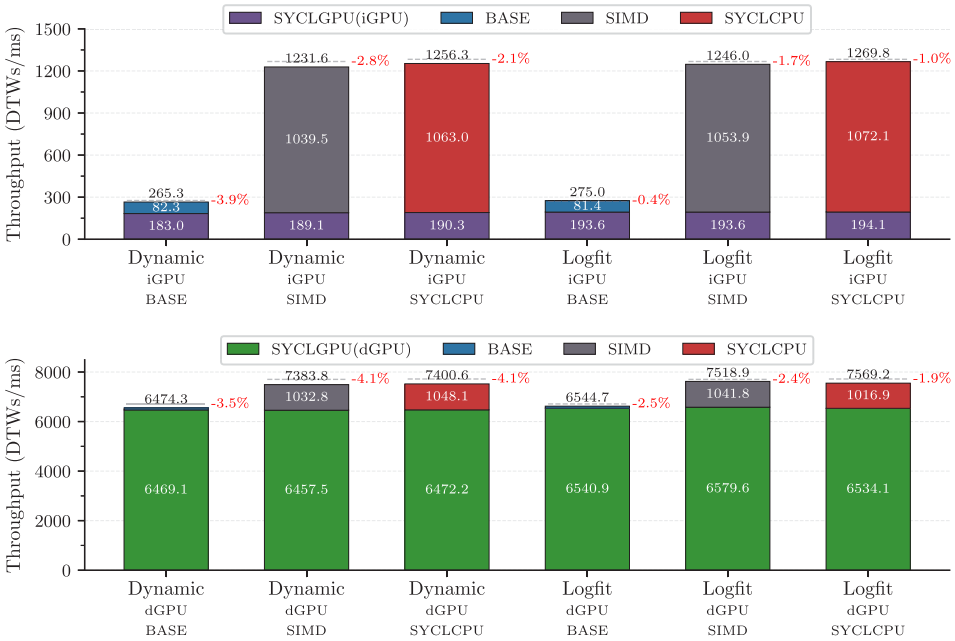

FIGURE 12 | Performance comparison of CPU+GPU for Dynamic and LogFit strategies on AlderLake, evaluated for iGPU (up) and dGPU (down). Throughput is measured in DTW/ms across BASE, SIMD, and SYCLCPU implementations coupled with SYCLGPU. Values inside each bar represent the measured throughput on the GPU (down) and the CPU (top). The value above each bar represents the measured throughput. A dashed line indicates the expected (ideal) throughput. A red number signals the Inefficiency = (measured-expected)/expected in %. (a) iGPU scheduler, (b) dGPU scheduler.

our experimental setup, for these chunk sizes we found that data communication overhead was always below 0,30% of the execution time, while kernel enqueueing, kernel launching, and partitioning overheads were negligible.

As expected, both heterogeneous CPU+GPU runs based on the SIMD and SYCLCPU outperform the BASE case. In fact, the heterogeneous SYCLCPU+SYCLGPU runs are up to 4.6× and 1.1× faster than BASE+SYCLGPU on the iGPU and the dGPU, respectively. We also note that LogFit can be up to 1% and 2.2% faster than Dynamic for the equivalent kernel versions on the iGPU and dGPU, respectively. This is due to the fact that LogFit selects higher GPU chunk sizes, so it can better exploit the GPU computational capabilities and achieve a higher relative throughput on the accelerator, both on the iGPU and dGPU, when compared to Dynamic. Another reason that explains the better results for LogFit is that it enables CPU kernels to generally achieve higher relative throughput on the CPU. A loss of performance in the CPU may be caused by the load imbalance that generates the last chunks, particularly when a small number of cores get the remaining iterations, and the rest of the cores are kept waiting. This justifies the higher percentage of inefficiency with respect to the expected ideal throughput of Dynamic-based runs: Up to 2.8% for iGPU and 4.1% for dGPU. In this last case, the fastest device, the GPU, is also kept waiting when some cores get the last chunks, which results in the additional degradation of the relative throughput we notice on the dGPU. On the other hand, the inefficiency of LogFit, again due mainly to load imbalance, is only up to 1.7% on the iGPU and up to 2.4% on the dGPU.

Table 3 presents a scaling analysis for the Dynamic and LogFit strategies on Alderlake, evaluating their performance (Perf, DTWs/ms) and efficiency (Eff in %) as the problem size (Scaling Factor, SF) increases. The efficiency is the ratio between the measured vs. the expected throughput for the corresponding configuration, being the expected throughput the ideal without the scheduling overhead mentioned previously. Results are collected only for the dGPU due to local memory constraints on the iGPU, which could not allocate the new data inputs. In any case, the results demonstrate that for both partitioning strategies, the measured performance shows slight improvement with increasing scale factors for all configurations. As discussed before, configurations based on the SIMD and SYCLCPU implementations outperform BASE, and in general, LogFit consistently achieves better results than Dynamic. We also notice that for all configurations, the efficiency improves and stabilizes as the problem size scales, which indicates that the scheduling overheads (in particular load imbalance) become negligible with larger workloads. Clearly, LogFit achieves near-optimal efficiency for any problem size.

6.3.2 | Performance of CPU+FPGA

Now, Figure 13 shows the measured throughput vs. the ideal expected throughput for the Dynamic partitioning strategy on SkyLake, where the FPGA is attached. Again, the CPU kernels are the baseline OpenMP version (BASE), the optimized SIMD $SIMDw = 16$ (SIMD) and the optimized SYCLCPU $SIMDw = 16$ (SYCLCPU) versions presented in Section 6.2.1. The FPGA chunk size for the Dynamic strategy is fixed and defined by the FPGA main kernel functions (see Section 4). In our case, $Ch_{FPGA} = < 24 \times 512, 32 >$. As before, data communication overhead, as well as kernel enqueueing, kernel launching, and partitioning overheads, were very minor.

TABLE 3 | Scaling analysis of our schedulers on AlderLake: Performance (DTWs/ms) and efficiency (%).

Problem size			Dynamic strategy						LogFit strategy					
			dGPU+BASE		dGPU+SIMD		dGPU+SYCLCPU		dGPU+BASE		dGPU+SIMD		dGPU+SYCLCPU	
SF	HoursEEG	TotalDTWs	Perf	Eff	Perf	Eff	Perf	Eff	Perf	Eff	Perf	Eff	Perf	Eff
1	162	244	6474.3	96.5%	7383.8	95.9%	7400.6	95.9%	6544.7	97.5%	7518.9	97.6%	7569.2	98.1%
2	324	983	6514.7	97.0%	7466.5	99.2%	7406.7	99.1%	6700.3	99.6%	7474.9	99.2%	7414.1	99.1%
4	648	3,944	6659.5	99.3%	7555.5	99.2%	7530.2	99.6%	6663.1	99.3%	7560.4	99.2%	7535.2	99.6%
6	972	8,882	6620.2	99.7%	7576.8	99.7%	7505.0	99.6%	6616.0	99.6%	7579.6	99.7%	7516.6	99.6%

Note: Eff. = efficiency (measured/expected in %); Hours EEG = equivalent hours of EEG recording at 256 Hz; Perf. = performance (DTWs/ms); SF = scaling factor; Total DTWs = millions of DTW computations.

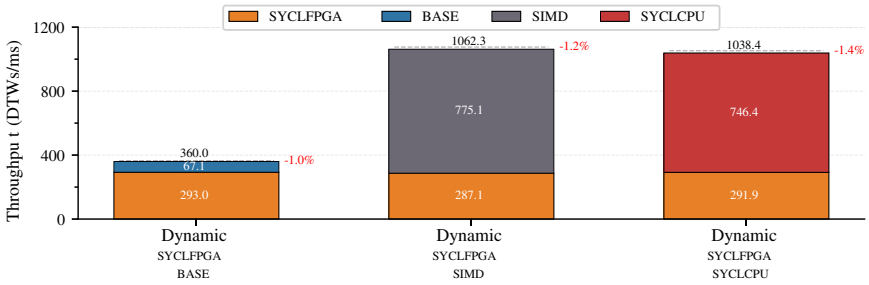

FIGURE 13 | Performance comparison of CPU+FPGA for Dynamic strategy on SkyLake, evaluated for the FPGA. Throughput is measured in DTW/ms across BASE, SIMD, and SYCLCPU implementations coupled with SYCLFPGA. Values inside each bar represent the measured throughput on the FPGA (down) and the CPU (top). The value above each bar represents the measured throughput. A dashed line indicates the expected (ideal) throughput. A red number signals the Inefficiency = (measured-expected)/expected in %.

Again, Figure 13 confirms that both heterogeneous CPU+FPGA runs outperform the BASE case: SIMD+SYCLFPGA and SYCLCPU+SYCLFPGA runs are up to 3× and 2.9× faster than BASE, respectively. Clearly, SIMD+SYCLFPGA takes advantage of the faster SIMD kernel in this platform. As explained, the throughput degradation of the CPU kernels is mainly due to load imbalance. In this case, for the SIMD and SYCLCPU-based versions, the degradation of the CPU and FPGA throughputs is very small, so the measured throughput is near the expected ideal one (below 1.4%). We also conducted a scaling analysis for the Dynamic strategy on SkyLake, similar to the study of the previous subsection, finding analogous conclusions. In particular that the efficiency improves and stabilizes as the problem size scales, indicating that the scheduling overheads become negligible with larger sizes.

As a summary of the results for the simultaneous co-processing of our application in CPU+GPU and CPU+FPGA platforms, if we compare the best-performing SYCL-based implementations in Figures 12 and 13 with the best CPU version in Figure 10, we find that co-execution improves the throughput by up to 1.1×, 6.9×, and 1.3× on CPU+iGPU, CPU+dGPU and CPU+FPGA, respectively, enabling near-optimal performance for a variety of heterogeneous platform configurations.

6.3.3 | Energy Efficiency

Now, Figure 14 shows the energy consumption metrics for Dynamic and LogFit strategies on AlderLake (up) and SkyLake (down) for the heterogeneous CPU+GPU and CPU+FPGA runs described in the previous sections. The solid bars show energy consumption in Joules (the higher the value, the worse), while the patterned bars show the energy efficiency (DTW/ms per Joule) in log scale (the higher the value, the better the efficiency.)

From Figure 11, we clearly see that heterogeneous CPU+GPU (AlderLake) and CPU+FPGA (SkyLake) runs always outperform the BASE case. In AlderLake, the heterogeneous SYCLCPU+SYCLGPU runs are up to 19.4× and 1.2× more energy efficient than BASE+SYCLGPU on the iGPU and the dGPU, respectively. LogFit and Dynamic present similar energy efficiency for SYCLCPU and SIMD-based versions on the iGPU and dGPU, although LogFit always outperforms Dynamic, as expected. In SkyLake, the improvement of the energy efficiency for the SYCLCPU+SYCLFPGA run over BASE is 6.5×, while for the SIMD+SYCLFPGA run is 7x.

If we compare the most energy-efficient SYCL-based implementations in Figure 14 with the most efficient CPU version in Figure 11, we see that heterogeneous co-execution improves energy efficiency by up to 1.01×, 13×, and 1.4× on CPU+iGPU, CPU+dGPU, and CPU+FPGA, respectively. These results highlight that SYCL systematically enables energy-efficient co-execution for several heterogeneous platform configurations.

7 | Conclusions

In this paper, we propose a novel cDTW Distance Matrix algorithm that we tailor to four different architectures: CPU, iGPU, dGPU, and FPGA. We use SYCL as the heterogeneous

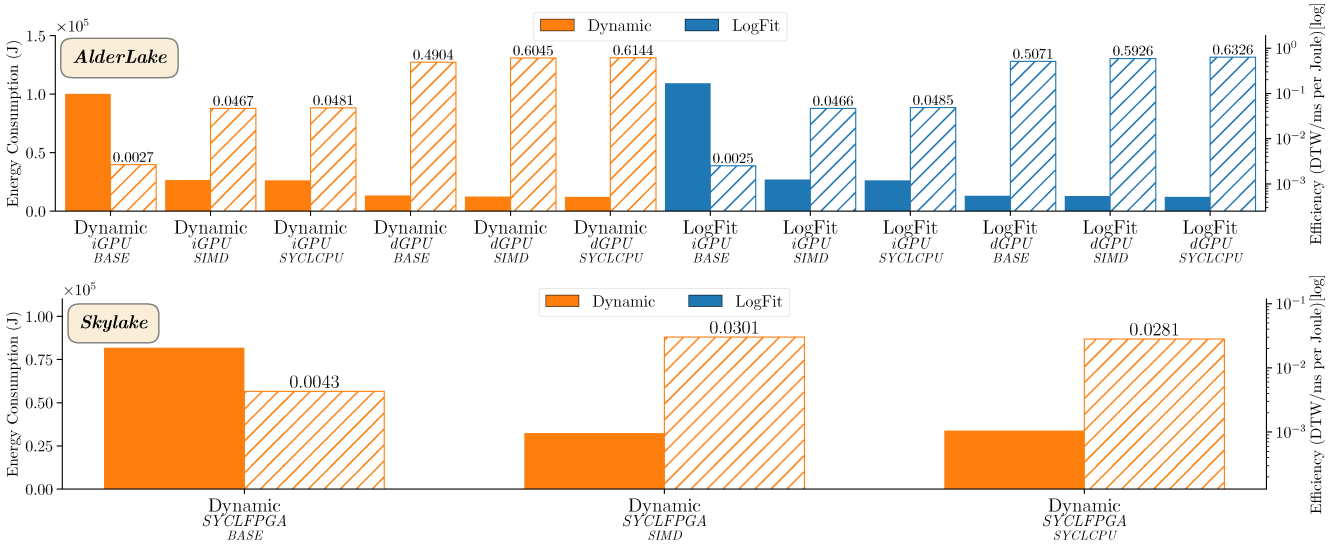

FIGURE 14 | Energy consumption metrics of CPU+GPU for Dynamic and LogFit strategies on AlderLake (up), and of CPU+FPGA for Dynamic strategy on SkyLake (down). Energy (Joules) and Energy Efficiency (DTW/ms per Joule) are measured across BASE, SIMD, and SYCLCPU versions coupled with SYCLGPU/SYCLFPGA. The efficiency is on a logarithmic scale, and the higher it is, the better the efficiency.

programming paradigm and evaluate its performance, portability, and energy efficiency across devices. Our results demonstrate that SYCL seems well suited to exploit the available SIMD capabilities of modern CPU cores, both in terms of performance and energy efficiency. It also shows promising results for accelerating devices, such as integrated and discrete GPUs and FPGAs, although in these two latter devices, the off-chip and on-chip memory bandwidth are the bottlenecks, respectively. We also consider simultaneous co-processing of our algorithm in CPU+GPU and CPU+FPGA heterogeneous platforms, finding that when the scheduler is able to dynamically and adaptively partition the work among the available devices we accelerate our SYCL-based application up to 1.1×, 6.9× and 1.3× on CPU+iGPU, CPU+dGPU and CPU+FPGA, respectively when compared to the best CPU implementation. But also, we improve energy efficiency up to 1.01×, 13×, and 1.4×, respectively. In fact, we can achieve near-optimal throughput across all devices because the scheduling overhead is below 4% in all cases.

These results make the case for using SYCL to systematically define the kernel of our application, then apply device-specific optimizations, as illustrated in this work, and finally dispatch each variant to the corresponding device using a dynamic and adaptive scheduler.

As future work, given the cross-platform capability of SYCL, we plan to evaluate our implementations on other heterogeneous systems based on different CPU architectures (AMD, ARM) and GPU architectures (AMD).

Acknowledgments

This work was supported by the Ministry of Economy and Competitiveness, Government of Spain, under Grant Nos. TED2021-131527B-I00 and PID2022-136575OB-I00. Funding for open access publishing: Universidad de Málaga/CBUA.

Data Availability Statement

The implementations, datasets, and analysis scripts used in this study are openly available in our SYCL-based DTW repository at https://bitbucket.org/cricamfe/dtwintel-cpe/src/main/.

References

1. Group K, "SYCL2020 Specification (revision 9)," 2024, https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html.

2. Intel, "oneAPI Programming Model," 2024, https://www.oneapi.io/.

3. Cppreference.com, "Std::Experimental::Simd," 2023, https://en.cppreference.com/w/cpp/experimental/simd/simd.

4. A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, and R. Asenjo, "Heterogeneous Parallel_for Template for CPU-GPU Chips," *International Journal of Parallel Programming* 47, no. 2 (2019): 213–233, https://doi.org/10.1007/s10766-018-0555-0.

5. World Health Organization (WHO), "Optimizing Brain Health Across the Life Course," 2022.

6. F. Muñoz, R. Asenjo, and A. Navarro, "PaFESD: Patterns Augmented by Features Epileptic Seizure Detection," *IEEE Transactions on Biomedical Engineering* 721, no. 1 (2025): 15, https://doi.org/10.1109/TBME.2024.3441090.

7. H. Sakoe and S. Chiba, "A Similarity Evaluation of Speech Patterns by Dynamic Programming," in *Proceedings of the Nat. Meeting of Institute of Electronic Communications Engineers of Japan*, vol. 136 (Institute of Electronic Communications Engineers of Japan, 1970).

8. D. Valentine, "Learning EEG, 10–20 System," 2020, https://www.learningeeg.com/montages-and-technical-components.

9. S. H and S. C, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, no. 1 (1978): 43–49.

10. H. Ding, "Querying and Mining of Time Series Data: Experimental Comparison of Represent. And Distance Measures," *Proceedings of the VLDB Endowment* 1, no. 2 (2008): 1542–1552.

11. M. Herrmann and G. I. Webb, "Early Abandoning and Pruning for Elastic Distances Including Dynamic Time Warping," *Data Mining and Knowledge Discovery* 35, no. 6 (2021): 2577–2601.

12. A. L. Goldberger, "CHB-MIT Scalp EEG Database," 2010, https://physionet.org/content/chbmit.

13. T. Giorgino, "Computing and Visualizing Dynamic Time Warping Alignments in R: The Dtw Package," *Journal of Statistical Software* 31 (2009): 1–24.

14. O. Escudero-Arnanz, A. G. Marques, C. Soguero-Ruiz, I. Mora-Jiménez, and G. Robles, "dtwParallel: A Python Package to Efficiently Compute Dynamic Time Warping Between Time Series," *SoftwareX* 22 (2023): 101364, https://doi.org/10.1016/j.softx.2023.101364.

15. B. Tang, M. L. Yiu, Y. Li, and L. H. U, "Exploit Every Cycle: Vectorized Time Series Algorithms on Modern Commodity CPUs," in *Proceedings of the Data Management on New Hardware*, vol. 10195 (Springer International Publishing AG, 2017), 18–39.

16. Y. Vash, M. Rol, and M. Chyzhmar, "Accelerating Dynamic Time Warping for Speech Recognition With SSE," *Scientific Journal of the Ternopil National Technical University* 114 (2024): 30–38, https://doi.org/10.33108/visnyk_tntu2024.02.030.

17. B. Schmidt and C. Hundt, "cuDTW++: Ultra-Fast Dynamic Time Warping on CUDA-Enabled GPUs," in *Euro-Par 2020: Parallel Processing: 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24–28, 2020, Proceedings. Euro-Par* (Springer-Verlag, 2020), 597–612.

18. D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul, "Accelerating Dynamic Time Warping Subsequence Search With GPUs and FPGAs," in *Proceedings of the 2010 IEEE International Conference on Data Mining. IEEE* (IEEE, 2010), 1001–1006.

19. D. Latta-Lin and S. I. Padilla Munoz, "Optimizing sDTW for AMD GPUs," arXiv preprint, arXiv:2403.06931 (2024).

20. AMD, "HIP: Heterogeneous-Computing Interface for Portability," AMD Open-Source Platform for GPU Computing (2024), https://github.com/ROCm-Developer-Tools/HIP.

21. AMD, "ROCm: Open Software Platform for GPU Computing," AMD Radeon Open Compute Platform (2024), https://www.amd.com/en/products/software/rocm.html.

22. H. Zhou, X. Xu, Y. Hu, et al., "Energy-Efficient Pipelined DTW Architecture on Hybrid Embedded Platforms," in *Proceedings of the 2015 Sixth International Green and Sustainable Computing Conference (IGSC)* (IEEE, 2015), 1–8.

23. M. Hormigo-Jiménez and J. Hormigo, "High-Throughput DTW Accelerator With Minimum Area in AMD FPGA by HLS," in *Proceedings of the 38th Conference on Design of Circuits and Integrated sys. (DCIS)* (UMA, 2023).

24. C. C. M. Yeh, "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets," in *Proceedings of the IEEE 16th International Conference on Data Mining (ICDM)* (IEEE, 2016).

25. Y. Zhu, "Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds," in *Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM)* (IEEE, 2018), 837–846.

26. Z. Zimmerman, K. Kamgar, N. S. Senobari, et al., "Matrix Profile XIV: Scaling Time Series Motif Discovery With GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond," in *ACM Symposium on Cloud Computing (SoCC)* (ACM, 2019), 74–86.

27. J. C. Romero, A. Vilches, A. Rodríguez, A. Navarro, and R. Asenjo, "ScrimpCo: Scalable Matrix Profile on Commodity Heterogeneous Processors," *Journal of Supercomputing* 76 (2020): 9189–9210.

28. J. C. Romero, "Efficient Heterogeneous Matrix Profile on a CPU + High Performance FPGA With Integrated HBM," *Future Generation Computer Systems* 125 (2021): 10–23, https://doi.org/10.1016/j.future.2021.06.025.

29. S. Alaee, K. Kamgar, and E. Keogh, "Matrix Profile XXII: Exact Discovery of Time Series Motifs Under DTW," in *Proceedings of the 2020 IEEE International Conference on Data Mining (ICDM)* (IEEE, 2020).

30. B. Zhu, Y. Jiang, M. Gu, and Y. Deng, "A GPU Acceleration Framework for Motif and Discord Based Pattern Mining," *IEEE Transactions on Parallel and Distributed Systems* 32, no. 8 (2021): 1987–2004.

31. A. Rodríguez, A. Navarro, R. Asenjo, et al., "Parallel Multiprocessing and Scheduling on the Heterogeneous Xeon+FPGA Platform," *Journal of Supercomputing* 76, no. 6 (2020): 4645–4665.

32. D. A. Constantinescu, A. Navarro, F. Corbera, J. A. Fernández-Madrigal, and R. Asenjo, "Efficiency and Productivity for Decision Making on Low-Power Heterogeneous CPU+GPU SoCs," *Journal of Supercomputing* 77, no. 1 (2021): 44–65, https://doi.org/10.1007/s11227-020-03257-3.

33. A. Vilches, "StratixMonitorLib," 2020, https://doi.org/10.5281/zenodo.3948283.

34. Developer N, "NVIDIA Management Library (NVML)," https://developer.nvidia.com/management-library-nvml.