EasyChair Preprint

№ 14488

Computing DTWs on CPU, GPU and FPGA with SYCL

Cristian Campos, Rafael Asenjo, Javier Hormigo and Angeles Navarro

August 16, 2024

Computing DTWs on CPU, GPU and FPGA with SYCL

Cristian Campos[1][0009−0000−3273−6133], Rafael Asenjo[1][0000−0002−1570−3863], Javier Hormigo[1][0000−0002−5454−6821], and Angeles Navarro[1][0000−0002−4140−2589]

Dept. of Computer Architecture, Univ. of Málaga, Spain
{cricamfe,asenjo,hormigo,angeles}@ac.uma.es

Abstract. One of the most time-consuming kernels of an epileptic seizure detection app is the computation of the Dynamic Time Warping (DTW) Distance Matrix. This kernel is a good candidate for heterogeneous CPU/GPU/FPGA execution. In this paper, we explore the design space of heterogeneous CPU, GPU, and FPGA implementations of this kernel. We start by optimizing the CPU implementation of the DTW Distance Matrix computation leveraging the latest C++26 SIMD library and compare it with the SYCL implementation for CPU that also exploits the SIMD units. Next, we take advantage of the portability of SYCL to run the code on an on-chip GPU, iGPU, as well as on a discrete NVIDIA GPU, dGPU. Finally we also present the SYCL implementation of the kernel on an Intel Stratix 10 MX FPGA. Our evaluations demonstrate that SYCL seems well suited to exploit the available SIMD capabilities of modern CPU cores, and also shows promising results for the accelerating devices considered in this work.

Keywords: Heterogeneous architecture · SIMD · GPU · FPGA · SYCL · DTW · energy efficiency.

1 Introduction

Ten or twenty years ago, those who cared about performance had to embrace parallel programming to squeeze the last drop of performance out of multicores. Nowadays, embracing heterogeneous programming is also a must if we require performance and reduced energy consumption from current computing platforms. These pervasive heterogeneous devices and platforms feature CPU cores, GPUs, and FPGAs, among other accelerators and ASICs. In this context, the "No transistor left behind" war cry conveys the idea of all devices helping in accelerating different parts of an application. To help in this regard, new heterogeneous programming models, such as SYCL [7], DPC++, and oneAPI [10], are emerging in order to ease the development of heterogeneous applications without compromising performance.

In this paper, we target a CPU, an integrated GPU, iGPU, an NVIDIA discrete GPU, dGPU, and an FPGA that we exploit to accelerate the most expensive function of an epileptic seizure detection algorithm. In the process, we tailor the function for the CPU, the GPUs and the FPGA devices, striving to reduce the energy consumption and paying attention also to the programmability. For the CPU, one of the latest standard C++ SIMD library [11] and SYCL are used. For the GPUs and FPGA, we also leverage the SYCL compiler (a High-Level Synthesis compiler in the case of the FPGA).

Regarding the problem we tackle, epilepsy is one of the most common neurological diseases globally [17] what makes the detection of epileptic seizures a socially impacting problem. Our goal is to devise a wearable (glasses, headband, or headset) with just two electrodes that can take an

electroencephalography (EEG) signal and warn the patient and their caregiver of epileptic seizures. To that end, we have developed a seizure detection algorithm based on Dynamic Time Warping (DTW) [15] distance matrix computation. This is a quite compute and data-intensive algorithm that requires fast execution when trained with patient-specific EEG recordings.

With all this, this paper proposes the following novel contributions:

1. A CPU-optimized version of the DTW Distance Matrix computation that leverages the latest C++26 SIMD library features and SYCL SIMDimization capabilities.
2. Two accelerator optimized versions of the DTW Distance Matrix computation, one for GPU and another for FPGA, which take advantage of the latest oneAPI SYCL compiler and its High-Level Synthesis capabilities for the FPGA.
3. A validation of the SYCL programming model in terms of performance portability and energy efficiency in four different heterogeneous architectures, discussing the strengths and possible limitations of our implementations for each device.

The remainder of the paper is organized as follows. The next section introduces the problem as well as related work. The following two sections, Section 3 and 4, cover the CPU, GPU, and FPGA implementations. Section 5 outlines the platform configuration and delves into a thorough experimental evaluation that covers the performance and energy efficiency analysis of the different implementations. Finally, we wrap up with conclusions in Section 6.

2 Background and related work

We rely on Figure 1 to illustrate key concepts that are required to understand the EEG analysis we propose. A signal or channel, S^c, is defined as discrete-time sequence of real-valued numbers $s_i^c \in \mathbb{R}$,

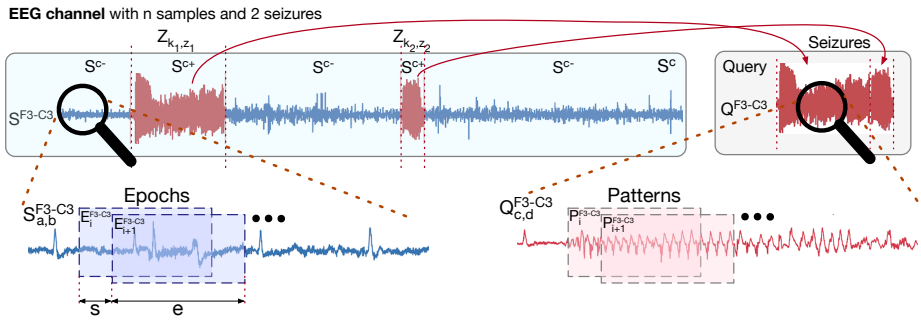

Fig. 1. The problem at hand and notation.

i.e., $S^c = \{s_i^c;\ 0 \leq i < n\}$, where n is the length of the sequence and c is a channel id or label. Each number s_i^c represents the electrical potential between two electrodes sampled at a specific time t_i. Channel labels in the 10-20 system [2] are used to identify each channel. For example, in Figure 1, each channel sample S^{F3-C3} represents the electrical potential measured between electrodes F3 and C3.

A seizure, denoted by $Z_{k,z}^c$, is a subsequence in channel S^c that starts at the sample k and has a length z, i.e. $Z_{k,z}^c = S_{k,z}^c$ when this subsequence has been labeled as a seizure. In the dataset used, ictal (seizure) and interictal (non-seizure) episodes are clearly identified through metadata, md, that specifies the onset and offset timestamps of each seizure, from which we gather the k and

z values. The presence of seizures in S^c segments the signal into ictal, S^{c+}, and interictal, S^{c-}, subsequences, as we see 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: $Q^c = (Z^c_{k_1,z_1}, Z^c_{k_2,z_2}, ..., Z^c_{k_{N_z^t},z_{N_z^t}})$. The total length of the query is $n_q = \sum_{i=1}^{N_z^t} z_i$. For example, in Figure 1 we see that Q^{F3-C3} is the concatenation of the two seizures in the channel F3-C3.

In order to find seizure patterns in a channel S^c, we only consider subsequences with a fixed length, e, and a fixed stride, s. We call epochs to these particular subsequences that, in other words, virtually segment the channels into smaller equidistant and fixed-size sliding windows. More precisely, an epoch $E_i^c = S^c_{k,e}$ with $k \in \{i \cdot s;\ 0 \leq i < n_e\}$, being the number of epochs $n_e = \lfloor (n-e)/s \rfloor + 1$. Similarly to the epochs, the patterns are the subsequences or sliding windows of size e and stride s that fill in a query Q^c. This is, a pattern $P_i^c = Q^c_{k,e}$ with $k \in \{i \cdot s;\ 0 \leq i < n_p\}$, being $n_p = \lfloor (n_q - e)/s \rfloor + 1$ the number of patterns in the query of length n_q. In Figure 1, we depict just two generic epochs of the subsequence $S^{F3-C3}_{a,b}$, labeled as E_i^{F3-C3} and E_{i+1}^{F3-C3} and the patterns P_i^{F3-C3} and P_{i+1}^{F3-C3} of a query subsequence $Q^{F3-C3}_{c,d}$.

Let us remember that our goal is to automatically identify one or several patterns that can discriminate between seizures and non-seizures in an EEG channel. Such a suitable pattern should include a well-conserved shape that is present in seizure epochs, E^+, but not in non-seizure ones, E^-. This can be achieved by comparing patterns and epochs by computing the distance between them in order to measure their similarities. However, there are many patterns in the query, so we need to compute a quality metric for all of them in order to find the best pattern (the one with the highest discriminative quality). In any case, first we have to compute the distance matrix, DM, between all patterns, P_i, and all epochs, E_j, as shown in Figure 2. Formally, we use $d_{i,j} = d(P_i, E_j)$ for the DTW distances between the patterns and the epochs.

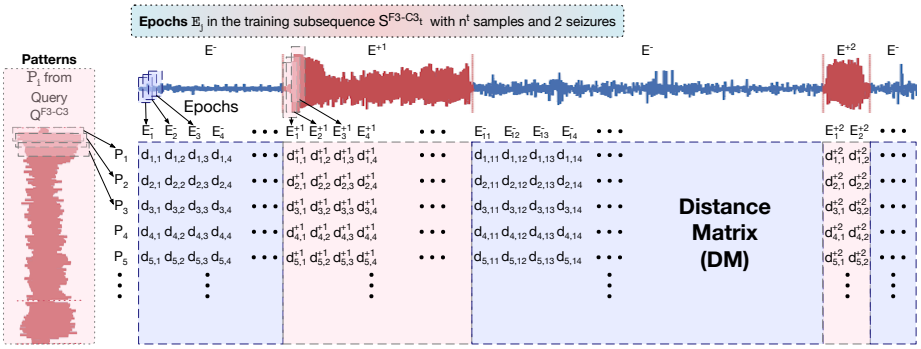

Fig. 2. Distance Matrix, DM.

We define the distance, $d(P_i^c, E_j^c)$, between a pattern, P_i^c and an epoch, E_j^c, as the Dynamic Time Warping (DTW) [15,14] distance between the two subsequences. DTW is an increasingly used algorithm for measuring similarity between two temporal sequences that may vary in phase or speed. It is now recognized as one of the most reliable similarity metrics [4]. Its higher computational cost in comparison with cheaper distances (as the Euclidean Distance, ED), has spurred the development of many simplified and optimized variants. One of the first simplifications was the cDTW (constrained DTW) [14] that limits the warping path to a band, known as the Sakoe-Chiba band or warping window, w, around the main diagonal of the cost matrix. When the warping window is set to 0, the

cDTW degenerates into the ED. The cDTW is a good compromise between the ED and the DTW, as it is faster than the DTW but more accurate than the ED. In this work we use cDTW with warping window, $w = 16$, as the distance metric between patterns and epochs. When the cDTW is used to find the nearest neighbor, NN, for example to find the most similar epoch to a pattern, further optimizations have been proposed, such as lower-bounding, early abandoning, and pruning techniques [8].

In principle, the number of DTW distances that we have to compute is equal to the number of patterns times the number of epochs, which can be prohibitive. For example, by looking at the first patient of the CHB-MIT dataset [6] we see that for each of the 23 channels of this patient, with $e = 1024$ (4 seconds) and $s = 256$ (1 second), there are more than 145K epochs and 269 patterns, which results in almost 40 million DTW distances that must be computed. Running a widely available Python DTW library[1] [5] on an off-the-shelf laptop, a single DTW distance of two subsequences of 1024 samples can take tens of milliseconds, which would translate in several months to compute all the distances of a single channel of a single patient.

Our work strives to optimize the computation of the DTW distance matrix, DM, and validate its performance and efficiency on four different architectures: CPU, iGPU, dGPU, and FPGA. We are not aware of any previous work that has tackled this problem in such a comprehensive way. However, we can find related work in the literature that has addressed the problem of computing another kind of distance matrix in different ways. For instance, in [18], 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 [19] supersedes STAMP computing, in parallel, the diagonals of the distance matrix. *SCAMP* [20] leverages the Pearson correlation to compare subsequences, instead of using the ED. We contributed with a CPU+GPU implementation of SCRIMP and a CPU+FPGA implementation of SCAMP in [12] and [13], respectively. In [3] the DTW is used to compute the matrix profile instead of the ED, but the distance matrix is not explicitly computed. However, in our work, we need the whole DTW distance matrix because it is consulted many times in order to compute the quality metrics that help us to identify the most discriminative epilepsy seizure pattern.

3 CPU and GPU implementations

In Figure 3 we show an example of the cDTW distance computation. In the top-left corner we see the Euclidean Distance between a hypothetical Epoch, E, and Pattern, P, both of size $e = 6$. Each point of the Pattern is paired with the corresponding one in the Epoch so that the Euclidean Distance is $d_E = \sum_{i=0}^{e-1}(P_i - E_i)^2$. However, below we see that using a warping window of $w = 2$, the points of P, P_i, can be paired with points of E, E_j, with $j \in \{i-2, i-1, i, i+1, i+2\}$. In this case, the cDTW distance is $d_{DTW} = \sum_{i=0}^{e-1}(P_i - E_j)^2$, where j is the index of the point in E, within the warping window, that minimizes the distance. In this example, $d_{DTW} = 4+0+1+1+1+0+1+1 = 9$, and the warping path, highlighted in gray, identifies the pairs (i, j) that result in this DTW distance.

The cDTW algorithm takes two vectors of data points, P and E, and computes the distance between them. The distance is computed by traversing a virtual (not stored) DTW matrix, D, of $e \times e$ from the top left corner to the bottom right one. The distance between P_i and E_j is computed as the Euclidean distance between them, $d(P_i, E_j) = (P_i - E_j)^2$, plus the minimum of the three adjacent cells in the distance matrix, N, NW, and W in Figure 3 (from North, North West, and

[1] See https://dynamictimewarping.github.io/

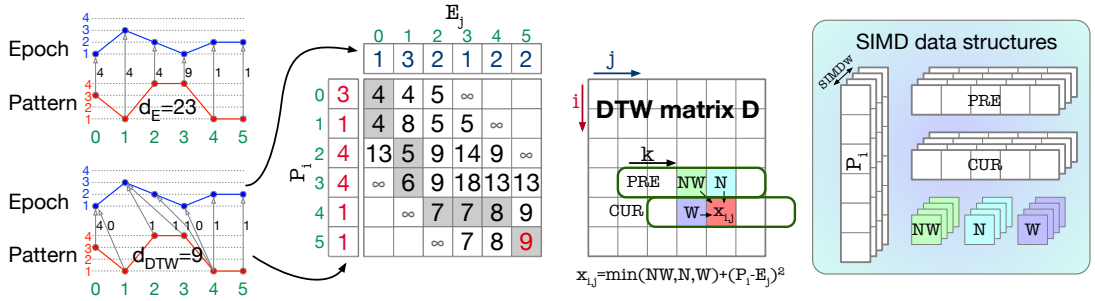

Fig. 3. Euclidean distance, cDTW distance example, and SIMD data structures.

West). The final distance is the value of the bottom right corner of the matrix. The elements of the matrix out of the diagonal $\pm w$ are initialized to ∞.

As we see on the left side of Figure 3, we can save memory space by only storing the required information of the matrix in two vectors of size $2 \cdot w + 1$, (5 elements in our example). These vectors, PRE (from previous) and CUR (from current) store just the elements that are necessary to compute the band of the matrix. At each i iteration, PRE and CUR are swapped and CUR is filled with the new values. For example, in our example of Figure 3, at iteration $i = 4$, PRE={6,9,18,13,13} (from iteration 3) and for $j = 4$ we have $NW = 18$, $N = 13$, $W = 7$, and $d(P_4, E_4) = 1$, so $x_{4,4} = min(18, 13, 7) + 1 = 8$.

The computation of the distance matrix, DM, offers several sources of parallelism that we can exploit. For instance, on the CPU, the simplest parallel version uses OpenMP to parallelize the epoch dimension (the largest one) of the DM. This is our baseline so we call it BASE. Besides, the pattern dimension is also parallelizable, and in order to better exploit the cache hierarchy we can "SIMDimize" the traversal in this dimension by comparing several patterns with each epoch.

Listing 1. Definition of `dtw_t` for different CPU implementations (BASE, SIMD, SYCLCPU)

```
1  namespace stdx = std::experimental;
2  using data_t = float
3  #ifdef __SIMD__ || __SYCLCPU__
4      #ifdef __SIMD__
5          using dtw_t = stdx::native_simd<data_t>;
6      #elif __SYCLCPU__
7          using dtw_t = sycl::float16;
8      #endif
9      constexpr size_t SIMDw = dtw_t::size();
10 #endif
11 #ifdef __BASE__
12     using dtw_t = data_t;
13 #endif
```

To that end, that substantially speed up the computation, we have explored two alternatives: i) the C++26 SIMD library[2] [11] (we call this version SIMD); or ii) the SIMD features of SYCL (we call this version SYCLCPU). In Listing 1 we show how the `dtw_t` is defined depending on the chosen alternative (BASE, SIMD, SYCLCPU). Basically, since `data_t=float`, `dtw_t` is a `float` in BASE (line 12), a `native_simd<data_t>` in SIMD (line 5) and a `float16` in SYCLCPU (line 7). In line 9, we initialize the constant `SIMDw` with the SIMD width (lanes) that, for our `data_t=float`, can be set to 4, 8 and 16 floats per SIMD register on our platforms.

[2] Or std::experimental::simd until C++26 is released

With this, in Listing 2, we list the `DTW_CPU` class that computes the DTW either in the BASE, SIMD, and SYCLCPU versions, thanks to conditional compiling and the parametrization on `dtw_t`. On the far right of Figure 3 we see the SIMD data structures, mainly registers `PRE` and `CUR` and variables `NW`, `N` and `W`.

Listing 2. DTW implementation on CPU (BASE, SIMD, SYCLCPU)

```
constexpr dtw_t maxval{std::numeric_limits<data_t>::max()}; // Infinite for out-of-band values

class DTW_CPU {
  public:
    DTW_CPU(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 __BASE__
            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;  // out of E or CUR bounds
        else                    W = CUR[k-1];
        if ((i<=0)||(k>=2*w))   N = maxval;  // out of P or PRE bounds
        else                    N = PRE[k+1];
        if ((i<=0)||(j<=0))     NW = maxval; // out of P or E bounds
        else                    NW = PRE[k];

        #ifdef __SIMD__
            //where(N<W, W) = N; where(W<NW, NW) = W; //inefficient alternative, 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 __BASE__
            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 constructor `DTW_CPU` in line 5 initializes the input arrays with an epoch, `E`, and several patterns, `P`, but for the latest, the entries have been previously (in the caller) rearranged so that consecutive values correspond to different patterns. The member function `calculate_dtw` (line 8 in Listing 2) traverses the rows (`i` loop) of the DTW matrix D and the band registers (zipped `j-k` loop), calling at each iteration to the `operator()` function (line 22). Inside that function, for the SIMD and SYCLCPU versions a `Psimd` vector is loaded with `SIMDw` values of the corresponding pattern sample, P_i, (see Figure 3) to later compute `SIMDw` distances in parallel on the SIMD units. For the BASE version (line 33) a single distance is computed per iteration. In lines 36-42, W, N and NW variables (of SIMD type in the corresponding case) are initialized so that we can compute the x_{ij} distance (lines 44-52) of Figure 3 and store it in `CUR[k]` (line 14), to finally swap the current and previous band registers (line 16).

A performance issue was identified in the `where` SIMD function of the SIMD C++ library. By replacing `where` by `stdx::min` (see lines 45 and 46), a significant performance improvement was observed (up to 3.62x on our evaluation platforms).

In the BASE and SIMD 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 implementation, SYCLGPU, taking advantage of the portability of SYCL, we took the BASE code shown before, and compiled it for the iGPU and for the NVIDIA dGPU. For the latest, 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`). As in the SYCLCPU version, the SYCLGPU one invokes the function `calculate_dtw()` from a kernel which is now submitted to the corresponding GPU queue (the iGPU or the dGPU). In Section 5 we describe the methodology employed to identify the optimal global work and work-group sizes for each device.

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 HLS features), the code used for the CPU and GPU has to be substantially rewritten to get the most out of the FPGA. Still, the SYCL language allows us fast prototyping and design space exploration.

The core of the proposed architecture is based on the basic DTW accelerator circuit presented in [9]. 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 to create our IP kernel that we sketch in Figure 4. This kernel is replicated as much as possible (24 times) to maximize the FPGA utilization. Each kernel comprises several "DTW Computation" modules, along with one "Epoch Generation", one "Pattern Generation", and one "Result Write-back" module. The computation modules work in parallel with different epochs, but the same pattern. 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. There are as many computation modules in a basic kernel as strides fit in an epoch (four in our case, with $e = 1024$ and $s = 256$).

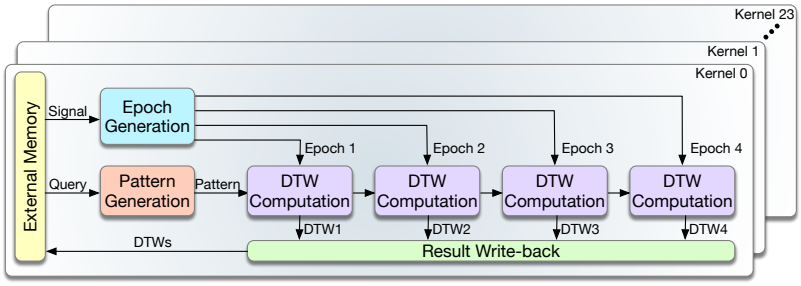

Fig. 4. Basic kernel for DTW computation and kernel replication on the FPGA.

All of these modules have been designed with the idea of reducing as much as possible the access to the external memory 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.

The different modules communicate with each other with queues (FIFOs) defined using oneAPI libraries. The synchronization of the whole system is data-driven based. That means that each module controls how much data needs to be read or produced and the queues control signals stall the modules whenever is required.

5 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 in order 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 of study. For this, in this section, we conduct a performance evaluation of the various kernel implementations previously discussed. 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-). We also explore whether, despite the optimizations, hardware bottlenecks appear in our executions.

5.1 Testbed

The experimental evaluation has been 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)[3] to accurately measure CPU and GPU power usage. In addition, to monitor FPGA power usage, we utilize StratixMonitorLib [16]. We rely on the NVIDIA Management Library (NVML) [1], 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 hours 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 DTW distances.

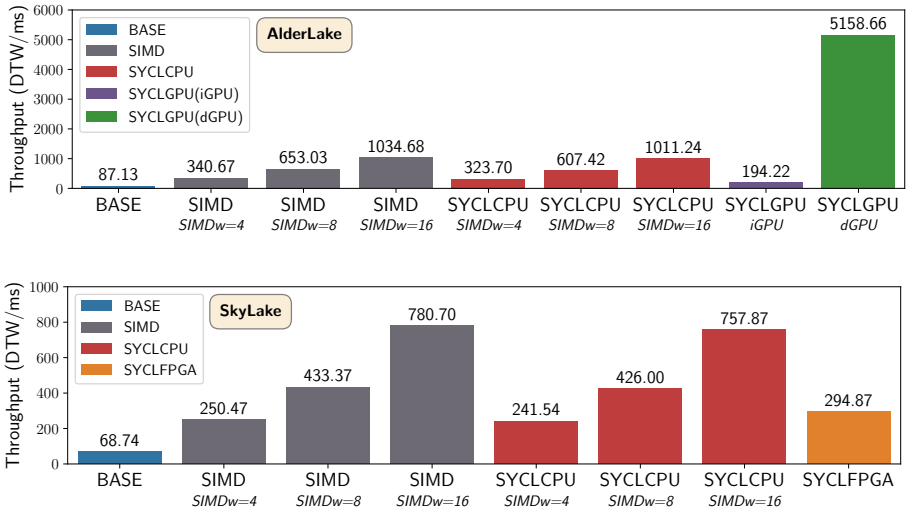

Fig. 5. Performance metrics on AlderLake and SkyLake: Throughput -DTW/ms-. The higher, the better.

5.2 Performance evaluation

Figure 5 depicts the throughput (DTW/ms) that our different 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.

[3] See https://github.com/intel/pcm/

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). To tune these versions, we use the Intel GPU Occupancy Calculator tool and the CUDA occupancy calculator that help us to explore different iGPU/dGPU job size configurations. We found that the optimal configuration for the iGPU was a global work size of 4096x32 and a work-group size of 64x8, achieving ideally 96.2% of execution unit (EU) utilization. Also, after exploring different combinations for the dGPU, we found that a global size of 12288x128 and a block size of 256x1, maximize ideally the multiprocessors (SM) utilization, achieving 97.66% of occupancy in this case. The results shown in the figure correspond to these configurations.

On the other hand, in SkyLake we see the results for the SYCL FPGA implementation (see section 4) running on the Stratix 10 MX. Ideally, our implementation is able to compute 24 DTWs per cycle.

Clearly, from Figure 5 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. The observed performance degradation in SYCL is due to the kernel enqueueing and launching, which represent up to 5% and 3% of overhead in AlderLake and SkyLake, respectively. In any case, the optimal SIMDw=16 version outperforms the baseline by 11.8x and 11.4x in AlderLake and SkyLake. In fact, we performed a roofline analysis using the Intel Advisor tool and found that the function that represents the hotspot in the optimal SIMDw=16 code is compute-bound, and it features a headroom of just 1.4x and 2.4x 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).

In AlderLake we see that the SYCLGPU version running on the integrated GPU (iGPU) performs 2.5x faster than the baseline. The Intel Advisor tool reports a 75% of EU occupancy, which hints that SYCL is reasonably exploiting the iGPU capabilities. Moreover, this SYCLGPU implementation running on the discrete GPU (dGPU) achieves 5.1x improvement over the fastest SYCLCPU. We also carried out a roofline analysis of this version 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.

In SkyLake we notice that the SYCL version running on the FPGA is 4.3x faster than the baseline, but still 2.5x slower than the best SYCLCPU version. From the Intel oneAPI FPGA Reports tool we discovered that the loop that represents the hostspot has an initiation interval of 1, it is pipelined and works at a frequency of 300 MHz. This loop is in fact the DTW computation module shown in Fig. 4, 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. Other interesting result is that the area estimates report tell us that our kernel uses 38% of ALUTs, 35% of on-chip block RAM and 7% of DSPs. Despite the apparent availability of resources, the compiler fitter module (quartus_fit) was able to perform the placement and routing only under this scenario.

5.3 Energy efficiency evaluation

The energy consumption metrics for AlderLake and SkyLake are shown in Figure 6. 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).

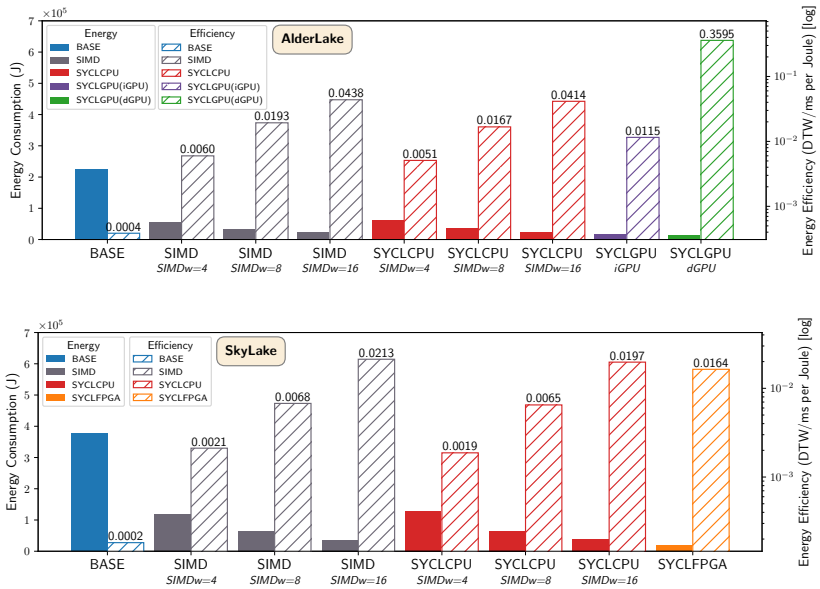

Fig. 6. Energy consumption metrics in AlderLake and SkyLake: Energy -Joules- and Energy Efficiency -DTW/ms per Joule-. The latter metric is in log scale, and the higher, the better efficiency.

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 SIMD with SIMDw=16 on the CPU. On Skylake, the SYCLFPGA implementation exhibits the lowest energy consumption and its energy efficiency is near the more efficient SIMD and SYCLCPU with SIMDw=16 on the CPU.

6 Conclusions

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

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. In fact, our results tell us that heterogeneous executions in which CPUs, GPUs, and FPGAs collaborate simultaneously to accelerate our application make sense, and we will explore this issue in future work. For it, we will implement a heterogeneous scheduler that will take

care of the data distribution and load balancing among devices in order to optimize throughput and/or energy consumption.

Acknowledgments

This work was supported by the Ministry of Science, Innovation and Universities, Government of Spain (Grant Numbers TED2021-131527B-I00 and PID2022-136575OB-I00).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. NVIDIA Management Library (NVML), https://developer.nvidia.com/management-library-nvml
2. Learning EEG, 10-20 system (2020), https://www.learningeeg.com/montages-and-technical-components
3. Alaee, S., Kamgar, K., Keogh, E.: Matrix Profile XXII: Exact discovery of time series motifs under dtw. In: 2020 IEEE Intl. Conf. on Data Mining (ICDM) (09 2020)
4. Ding, H., et al.: Querying and mining of time series data: Experimental comparison of representations and distance measures. Proc. VLDB Endow. **1**(2), 1542–1552 (aug 2008)
5. Giorgino, T.: Computing and visualizing dynamic time warping alignments in R: The dtw package. J. of Statistical Software (2009)
6. Goldberger AL, Amaral LAN, G.L.: CHB-MIT scalp EEG database (2010)
7. Group, T.K.S.W.: SYCL 2020 Specification (revision 3) (3 2021)
8. Herrmann, M., Webb, G.I.: Early abandoning and pruning for elastic distances including dynamic time warping. Data Mining and Knowledge Discovery **35**(6), 2577–2601 (Nov 2021)
9. Hormigo-Jiménez, M., Hormigo, J.: High-throughput DTW accelerator with minimum area in AMD FPGA by HLS. In: 2023 38th Conf. on Design of Circuits and Integrated Sys. (DCIS) (2023)
10. Intel: Intel oneAPI Programming Guide (6 2020)
11. Kretz, M.: P0214R9: Data-parallel vector types & operations. Tech. rep., ISO C++ Committee, WG21, Parallelism TS 2 (2018)
12. Romero, J.C., Vilches, A., Rodríguez, A., Navarro, A., Asenjo, R.: ScrimpCo: scalable matrix profile on commodity heterogeneous processors. The Journal of Supercomputing (2020)
13. Romero, J.C., et al.: Efficient heterogeneous matrix profile on a CPU + high performance FPGA with integrated HBM. Future Generation Computer Systems **125**, 10–23 (2021)
14. Sakoe, H., Chiba, S.: Dynamic programming algorithm optimization for spoken word recognition. IEEE T. on Acoustics, Speech, and Signal Processing **26**(1) (1978)
15. Sakoe, H., Chiba, S.: A similarity evaluation of speech patterns by dynamic programming. In: Nat. Meeting of Institute of Electronic Communications Engineers of Japan. vol. 136 (1970)
16. Vilches, A.: StratixMonitorLib (Jul 2020), https://doi.org/10.5281/zenodo.3948283
17. World Health Organization: Optimizing brain health across the life course (August 2022)
18. Yeh, C.C.M., et al.: Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In: Data Mining (ICDM), IEEE 16th Intl. Conf. on (2016)
19. Zhu, Y., et al.: Matrix Profile XI: SCRIMP++: Time series motif discovery at interactive speeds. 2018 IEEE Intl. Conf. on Data Mining (ICDM) pp. 837–846 (2018)
20. Zimmerman, Z., et al.: Matrix profile XIV: scaling time series motif discovery with GPUs to break a quintillion pairwise comparisons a day and beyond. In: ACM Symp. on Cloud Computing (2019)