cuTimeWarp: Accelerating Soft Dynamic Time Warping on GPU

Alex Kyllo Afrooz Rahmati

March 18, 2021

Abstract

In this report we explore techniques for optimizing the parallel computation of Soft Dynamic Time Warping, a differentiable sequence dissimilarity measure, on graphics processing units (GPU), for the purpose of enabling high-performance machine learning on time series datasets.

Introduction

Time series machine learning is a research area with countless useful applications such as recognizing sounds and gestures. Clustering or classifying large time series datasets is challenging partly because of the need to define a measure of dissimilarity between any two given time series, which is not as straightforward as computing distance between independent points in dimensional space. Furthermore, practical applications require finding common structure in time series despite different speeds or phases; for example, a word means the same whether spoken quickly or slowly. Another requirement for machine learning tasks is that the dissimilarity measure must be differentiable so that its gradient can be used as to minimize it as a loss function to find the best fit model. Finally, the measure must be efficient to calculate, because it will be calculated repeatedly many times during the model fitting process. To this end we will explore the GPU acceleration of Soft Dynamic Time Warping (Soft-DTW) [1], a differentiable sequence dissimilarity measure, to enable high performance time series machine learning.

Background

Time series data

Time series data generally refers to data containing quantitative measurements collected from the same subject or process at multiple points in time, and it exhibits several peculiarities in comparison to time-independent data. Time series data can exhibit autocorrelation, meaning that the value at any point in time can exhibit correlation to the value at a previous point in time, with some delay or *lag* in between. Time series data can also exhibit *seasonality* or cyclical patterns as well as *trend* which is a tendency for the average value (over some rolling time window) to increase or decrease as a function of time. Due to these characteristics, time series data does not conform to the I.I.D. (independent and identically distributed) assumption that typically applies in the study of random variables, and therefore it requires special techniques for analysis and modeling.

Time series data can be either univariate or multivariate. For example, a set of electrocardiogram (ECG) measurements has a single variable (heart voltage) collected over time, but if the dataset also included the patient's blood pressure and oxygen levels measured at each time point, that would constitute a multivariate time series, and correlations between the different variables could be studied in addition to the autocorrelation within each of the variables.

Time series distance and dissimilarity measures

A fundamental capability that enables learning from data is the ability to quantify a metric of distance or dissimilarity between observations, because this allows comparison among data observations as well as quantification of error between observed data and the predictions of an estimator model. For time-independent data, multiple valid notions of distance exist; the most commonly used is Euclidean distance, but others such as Manhattan distance, cosine distance, or Mahalanobis distance are often used depending on the problem domain.

Likewise, there are multiple valid ways to compute a measure of dissimliarity between two sequences or time series; for real-valued time series measurements the simplest is also Euclidean distance, which is the square root of the sum of squared pairwise differences between two time series x and y, each of length n (equation 1).

$$d(x,y) = \sqrt{\sum_{t=1}^{n} (x_t - y_t)^2} \tag{1}$$

However, a significant drawback of Euclidean distance in time series applications is that two structurally similar time series will produce a large distance if they are at different speeds or out of phase. In time series applications it is often desirable to produce a low dissimilarity for structurally similar time series despite variations in phase or speed, so an alternative method of quantifying dissimilarity is needed. This is where Dynamic Time Warping (DTW) comes in, providing an optimal nonlinear pairwise matching path (Figure 1).

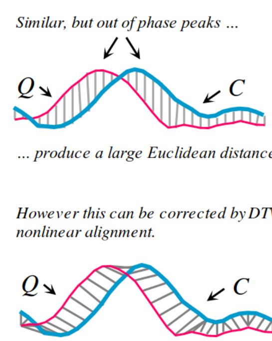

Figure 1: Euclidean vs. DTW on out of phase time series [2]

Dynamic Time Warping

Dynamic Time Warping (DTW) was devised in the 1960s as an alternative time series dissimilarity measure to address this shortcoming [3]. DTW is a nonlinear mapping from each point in one time series to the nearest point in a second time series; by allowing this nonlinearity, the algorithm is able to minimize the computed distance between time series that are similar in shape but misaligned in the time domain. While DTW is technically not considered a "distance" because it does not conform to the triangle inequality [4], and therefore we refer to it as a "dissimilarity" instead, it can still be used in place of Euclidean distance or other distance measures for many practical applications of time series data.

The problem can be visualized as a matrix of pairwise distances between every element of time series x and every element of time series y. Under this representation, the Euclidean distance measure translates to the shortest diagonal path across the matrix from one corner to the other, while DTW allows the path to "wander" from the diagonal in search of the lowest cost path (Figure 2), subject to the constraints that the path must begin at one corner and end at the opposite corner, and make monotonic progress in between.

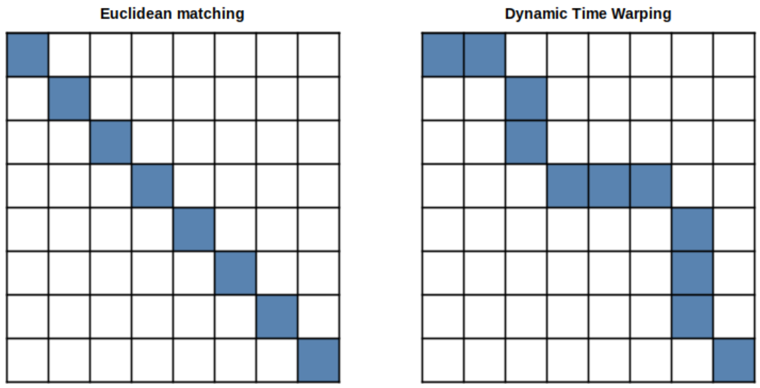

Figure 2: Euclidean Distance vs. DTW represented as a shortest vs. lowest-cost diagonal path across a pairwise distance matrix

The basic algorithm for DTW is to use Bellman's recursion, a dynamic programming technique, to find the lowest-cost path diagonally across a pairwise distance matrix; at each step, the cost is updated to the current cell's cost plus the minimum of the three previous adjacent cells' costs. The computation cost for this approach is quadratic ($O(mn)$) for time series vectors of length m and n [1]. The formula for the DTW between time series x and y is given by equation 2.

$$DTW(x,y) = min_{\pi}\sqrt{\sum_{(i,j)\in\pi} d(x_i,y_j)^2} \tag{2}$$

Where $d(x_i,y_j)^2$ is the cost function, typically pairwise squared Euclidean distance between an element of time series x and an element of time series y. The loss function for DTW is not differentiable due to the min operation within the formula; a small change in the input time series may result in zero change in the path cost. However, we can create a differentiable version called Soft-DTW by replacing the min function with a soft-min function (equation 3) [1].

$$\text{soft-min}_{\gamma}(a_1,...,a_n) = -\gamma log \sum_i e^{-a_i/\gamma} \tag{3}$$

Hence, Soft-DTW is parameterized by the smoothing constant gamma, which becomes a tunable hyperparameter in machine learning model training applications.

Applications

DTW is a widely used technique employed in many areas of science, including biology, technology, economics, and finance. DTW can be used to discover patterns in time series or search within signal databases to find the closest matches [5]. It is also utilized in machine learning applications like clustering, regression, and classification on time series data. Practical applications of DTW include tasks such as:

- Voice command recognition

- Sound detection and classification
- Handwriting and gesture recognition
- Human activity recognition
- Heart rhythm anomaly detection
- Sales or asset price pattern detection

As a differentiable time series dissimilarity measure, DTW can be applied as a loss function or minimization objective in data modeling techniques such as:

- k-means Clustering
- Hierarchical agglomerative clustering
- Motif (similar subsequence) search
- k-nearest neighbor or nearest centroid classification
- Recurrent neural networks

A common technique in machine learning with Soft-DTW is the computation of barycenters, which are centroids within the space of a set of time series. The differentiability of Soft-DTW allows for barycenter finding via gradient descent or semi-Newtonian function minimization methods. First, a candidate time series is initialized with random points, and then, iteratively, its Soft-DTW dissimilarity to a set of multiple time series representing a target class is calculated, then the gradient of the Soft-DTW dissimilarity is taken with respect to the input values, and the inputs are updated by a small step size in the direction of the gradient, until convergence. After the best-fit barycenters are found, new observations can be classified by finding the nearest barycenter. Sequence prediction and generation is also possible using recurrent neural networks with Soft-DTW as a loss function [1].

Prior to computing the Soft-DTW dissimilarity between any two time series, each time series should be *z-normalized*, that is, scaled so that its mean is equal to 0 and its standard deviation is equal to 1, to remove the problem of "wandering baselines" or "drift" in the measurements, as illustrated in [2] with an ECG classifier that yields incorrect results on un-normalized data due to drift that "may be caused by patient movement, dirty lead wires/electrodes, loose electrodes, and so on," and which "does not have any medical significance." Z-normalization also tends to make the iterative process of minimizing the cost function through gradient descent more efficient because it prevents parameters with larger magnitude from dominating the direction of the weight update step [6]. For this reason, all input data used in our performance experiments is z-normalized in advance.

Parallelizing DTW

A naive, sequential implementation of DTW would involve a nested loop to iterate over each row and column of the cost matrix to update each cell's cost based on the three neighboring cells' costs, hence the $O(n^2)$ time complexity. Because each cell has a data dependency on the three cells to the top, left, and top-left, there is no dependency between cells that are on the same antidiagonal of the matrix, therefore computation of these cells can be handled by parallel threads. One thread computes the upper-leftmost cell, then two threads compute the next antidiagonal, then three threads compute the next, and so on. Because the length of the longest antidiagonal of a matrix is $min(m,n)$ that is also the number of threads that can be utilized in computing parallel DTW, and a subset of these threads will be inactive in computing antidiagonals that are shorter than the longest antidiagonal.

Related Work

Because DTW has so many useful data mining applications but suffers from quadratic complexity, scholars have devised a wide variety of exact and approximate approaches to improve its performance.

Dr. Eamonn Keogh and several others have utilized various indexing schemes to construct lower bounds on warping distance as an optimization technique for speeding up nearest neighbor search via early removal of poor candidates [7].

Shen and Chi (2021) propose an optimization of nearest neighbor search of multivariate time series, leveraging the triangle inequality and quantization-based point clustering to restrict the search [8].

Xiao et al (2013) introduced a prefix computation technique for transforming the diagonal data dependence to improve parallel computation of the cost matrix on GPU [9].

Zhu et al (2018) demonstrate a method of optimizing memory access by taking advantage of the diagonal data dependency to rearrange the matrix so that elements on the same diagonal are stored contiguously [10].

Salvador and Chan (2004) [11] proposed a method of accelerating and approximation of DTW, dubbed FastDTW, by first downsampling the time series to a fraction of its length to find an approximate best path and then recursively upsampling and reapplying the algorithm, using the previous lower-resolution best path to construct upper and lower bounds for next pass. However, Wu and Keogh (2020) demonstrate that this method is generally slower than exact DTW on most realistic datasets.

A prior implementation of Soft-DTW on CUDA using PyTorch and Numba claims to achieve 100x performance improvement over the original Soft-DTW reference implementation in Cython code, but is limited to sequence lengths of 1024 (CUDA maximum thread block size) and leaves many opportunities for further CUDA optimizations such as the use of shared memory [12].

A 2015 paper describes a tiling approach called *PeerWave*, which utilizes direct synchronization between neighboring streaming multiprocessors (SMs) to handle the inter-tile data dependency without atomic operations, locks, or other global barriers, leading to improved load balance and scaling properties [13]. This tiling strategy also enables the PeerWave algorithm to handle longer time series than 1024.

In our work, we primarily focus on this general area of opportunity, optimizing DTW cost matrix structure and memory access patterns to increase parallelism and minimize memory latency in the computation of the warping path matrix.

Methods

To evaluate various performance optimizations on the Soft-DTW computation, we implemented a C++ and CUDA library called cuTimeWarp, which includes functions for computing the SoftDTW on CPU and GPU.

Given a set of many multivariate time series of the same length and number of variables, we can compute the Soft-DTW distance between every time series and every other time series in the set by computing, in parallel for each pair, a pairwise squared Euclidian matrix, then applying the Soft-DTW calculation on the distance matrix. This computation, however, also has an $O(n^2)$ complexity and can potentially even take longer than the DTW computation itself. For univariate time series, we can save this cost by computing the DTW cost matrix on the two input time series directly, computing the absolute distance between each pair of values from the two time series within the nested loop of the DTW procedure.

Optimization Techniques

We opted to experiment with the following performance optimization techniques:

1. Wavefront tiling (based on PeerWave)
2. Diagonal-major data layout
3. Shared memory stencil
4. Sakoe-Chiba bands

Before delving into the performance results, we will provide a brief explanation of how each of these techniques is implemented and why it provides a performance improvement.

Wavefront Tiling

Wavefront parallelism is one of the useful methods to overcome the dependencies of nested loops by multiple processing units. The idea is to re-order the loop iterations in such a way that form diagonal wave and each wave can be computed in parallel. Barriers will be utilized to control the data dependencies among consecutive waves. Elements inside the wave grouped together using tiled technique to enhance data locality and performance. This methodology presents a second degree of parallelism where tiles can be computed in parallel by separating with a global variable.

GPU and specifically CUDA can accelerate the process of wavefront. Each tile assigns to a block to be process in parallel by SM. On the other hand, the iteration along diagonals within a tile are also pluralized. In order to enforce the dependencies, the global barriers used among the tiles and within every tiles [13].

In our Soft DTW implementation, we utilized the wavefront technique to manage the dependencies of neighboring cells for computing the minimum warp path. In our algorithm this value depends on the minimum cost of the upper, left and upper-left diagonal cell cost. Figure 3 show this dependencies clearly.Each D(i,j) depends on the up, left and up-left neighbor cost.

Wavefront process split to three major steps:

- Populating the dependencies managing by loop. In this step, we keep the global variable WaveId to sperate the process of waves. Wavelength is increasing one by one on each loop iterations until reaching the total number of tiles. The primary kernel softdtw global tiled called with the wavelength number of blocks and thread size of tile width.
- In this kernel, the row and column index for each tile is calculating and passing as a global variable to the corresponding kernel for the tiles' computation.

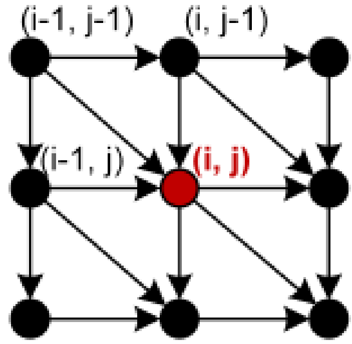

Figure 3: computation dependencies for DTW [13].

- The third step is the main kernel to process inter-tiles in parallel. Shared memory employed to keep the tile within the cache and improve the overall performance. As we mentioned earlier, we need to calculate the soft-min for the dependent cells. Therefore, the soft-min calculated for the up, left and diagonal upper left index.(equation 4)

$$D(i, j) = D(i-1, j-1) + Soft-min \begin{cases} D(i-1, j) \\ D(i, j-1) \\ D(i-1, j-1) \end{cases} \tag{4}$$

Drawbacks: Due to the usage of global barriers, there would be lower utilization for the GPU. Fewer threads would be available at the beginning and at the end of each tiles process. Also, less tiles are accessible toward the beginning and end of wave iterations. Therefore, device SM remain idle during the initiation and end of the process.

Diagonal-major layout

Because the data dependency structure of the DTW algorithm results in elements of the distance and cost matrices on the same antidiagonal being processed in parallel, storing these matrices in row-major or column major order will cause a performance impact from cache misses and non-coalesced accesses to global memory.

If the data is first rearranged into an antidiagonal-major layout, then at each iteration of the wavefront loop, processor threads will make coalesced accesses to data elements that are laid out contiguously in memory. As illustrated in Figure 4, this transforms an $m \times n$ matrix that must be iterated over diagonally, into a $(m+n-1) \times (min(m, n))$ matrix that can be iterated from top to bottom, with one thread assigned to each column. At each iteration step (i.e. each row of the diagonal-major matrix), contiguous array elements are read from memory into cache and written from cache back to memory, resulting in coalesced accesses and fewer cache misses.

Shared memory stencil computation

As the program iterates diagonally across the distance matrix to find the optimal warping path, each cell in the path utilizes the previously computed results of three previous neighboring cells; if the iteration is visualized as proceeding from the upper left to the lower right corner of the matrix, the cost

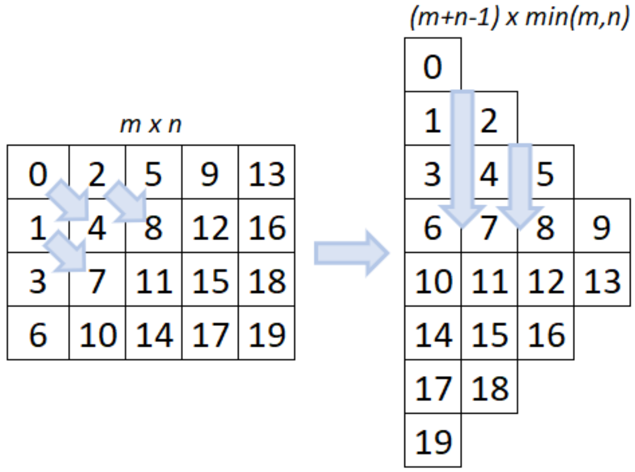

Figure 4: Cost matrix transformation to antidiagonal-major layout (arrows show iteration direction)

value in each cell depends on the (soft) minimum of the costs in the cell above, the cell to the left, and the cell to the diagonal upper-left, which were computed in the previous two iterations (Figure 3). If the cost matrix `R` resides in global memory, then non-contiguous accesses to `R[i-1][j]`, `R[i][j-1]` and `R[i-1][j-1]` will result in cache misses, incurring a significant performance cost. Since each element of `R` will be referenced up to three times in the computation of dependent cells, these cache misses can be avoided via a stencil computation using shared memory in CUDA. The stencil serves as a cache for the current and previous two diagonals; once a diagonal is no longer in use, its elements are written back to the cost matrix `R` in device global memory.

The algorithm can be modified to use shared memory as follows:

```
D is a squared euclidean distance matrix of two time series
R is a cost matrix initialized to positive infinity except for R[0, 0] = 0
for each anti-diagonal of R starting from R[0, 0]
    if the current thread index < length of the current anti-diagonal
        copy R[i][j] from global memory into the stencil array
        read R[i-1][j], R[i][j-1] and R[i-1][j-1] from the stencil array
        compute cost as softmin(R[i-1][j], R[i][j-1], R[i-1][j-1]) + D[i-1][j-1]
        write the cost back to the stencil
        copy the cost in R[i-1][j-1] from the stencil back to global memory
```

Sakoe-Chiba bands

Sakoe-Chiba bands, proposed by Sakoe and Chiba in their 1978 paper "Dynamic programming algorithm optimization for spoken word recognition," introduce a "slope constraint" to the warping path to limit the maximum allowed warping distance beyond which a pair will not be considered in the optimal path calculation [3]. Pruning the search space by removing some of the extreme diagonals from consideration yields an approximation of the optimal warping path that can be calculated in sub-quadratic time.

This optimization is simple to implement for square matrices (i.e. DTW on time series of equal length) by checking that the absolute difference between the loop counter variables `i` and `j` does not exceed a fixed bandwidth threshold value (Figure 5). For rectangular matrices, since the leading diagonal does not end at the lower right corner, the implementation must be slightly modified to ensure that the counter variable along the longer of the two dimensions stays within a defined radius. Either way the result is a diagonal band matrix.

In a parallel programming environment such as CUDA, this optimization can also allow for the computation of the warping path using fewer threads, as threads assigned to cost matrix cells outside the band would go unused. Space savings can also be obtained if the bandwidth is known in advance, by storing the distance matrix and cost matrix in band storage format, omitting the zeroes in the corners.

While this technique produces only an approximation of the optimal path, in practice it has been shown to improve task performance by preventing pathological alignments where a very small portion of one time series maps onto a large portion of the other [7]. The width of the band can be a tunable hyperparameter for time series classification tasks.

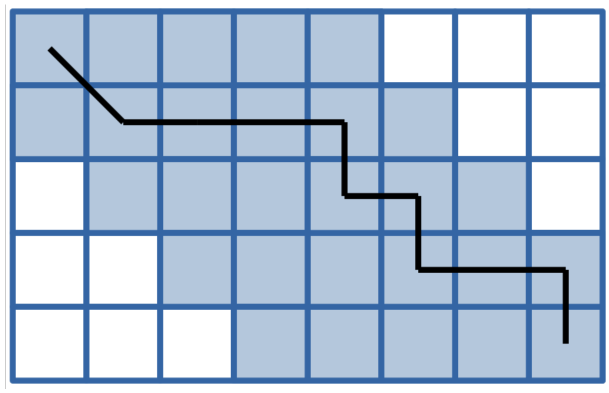

Figure 5: Illustration of one possible optimal DTW path for a length 5 series and a length 8 series with Sakoe-Chiba band radius = 1.

Complexity Analysis

In order to quantify the performance of our implementations as a floating point operation execution rate, we needed to count the floating-point operations used to compute the cost matrix. Calculating a single cell of the cost matrix in Soft-DTW involves a softmin function of three real-valued arguments. The softmin function for three arguments is implemented in C++ as:

```
float softmin(float a, float b, float c, const float gamma)
{
    float ng = -gamma;
    a /= ng;
    b /= ng;
    c /= ng;
    float max_of = max(max(a, b), c);
    float sum = exp(a - max_of) + exp(b - max_of) + exp(c - max_of);

    return ng * (log(sum) + max_of);
}
```

This function contains 17 floating point operations:

- 1 x negation

- 3 x division
- 2 x max
- 3 x exponentiation
- 3 x subtraction
- 3 x addition
- 1 x natural logarithm
- 1 x multiplication

Additionally, the softmin of the previous 3 cell costs is added to the current cell cost, for a total of 18 floating point operations (FLOPs). Therefore Soft-DTW for a pair of time series of lengths m and n results in $18mn$ FLOPs.

Test Hardware Specifications

Table 1: Device Hardware Specifications

Property	Value
Model	GeForce RTX 2060 Super
Generation	Turing
Compute Capability	7.5
SM Count	34
CUDA Cores	2176
Tensor Cores	272
CUDA Version	11.2
Driver Version	460.39
Clock Speed (MHz)	1470
Boost Speed (MHz)	1710
Memory Speed (Gbps)	14
Memory Size (GiB)	8
Memory Bandwidth (GB/s)	448
L1 Cache (KiB) (per SM)	64
L2 Cache (KiB)	4096

We tested the program on a single GeForce RTX 2060 Super GPU with the specifications detailed in table 1 [14]. Per NVIDIA's documentation, "The peak single precision floating point performance of a CUDA device is defined as the number of CUDA cores times the graphics clock frequency multiplied by two [15]. For this device, the equation works out to $2176 \times 1.71 \times 2 = 7441.92$ GFLOPS.

The GPU device is installed on a desktop computer running Ubuntu Linux 20.04 with CPU and memory specifications detailed in Table 2.

Test Data

For performance testing we selected the ECG 200 dataset from the UCR Time Series Archive [16]. This dataset consists of 200 sets of electrocardiogram time series of length 96. We also wrote a performance

Table 2: Host Hardware Specifications

	Desktop
RAM Size:	64GB
Configuration:	16GB x 4
Type:	DDR4
Speed:	2667 MT/s
Bandwidth:	16415 MiB/s
Architecture:	x86_64
CPU(s):	16
Thread(s) per core:	2
Core(s) per socket:	8
Model name:	AMD Ryzen 7 3700X 8-Core Processor
CPU MHz:	1862.571
CPU max MHz:	3600.0000
CPU min MHz:	2200.0000
BogoMIPS:	7187.14
L1d cache:	256 KiB
L1i cache:	256 KiB
L2 cache:	4 MiB
L3 cache:	32 MiB

testing program that generates sets of random unit normal data to test the performance impact of varying time series length and count to many different sizes.

Results

As a baseline for performance comparison, we timed the naive, sequential CPU implementation of SoftDTW and found that its execution rate varied between 0.35-0.4 GFLOPs (Figure 6) on the hardware described in Table 2, for random unit normal data. While a multithreaded parallel CPU implementation could likely achieve several times this rate of execution, we opted not to experiment with CPU parallelism and instead move on to parallelizing the algorithm in CUDA with GPU multithreading.

Figure 7 shows the execution rate of the naive CUDA kernel vs. the CPU implementation for comparison. The CUDA kernel achieves performance of around 34 GFLOPs compared to the CPU's 0.36 GFLOPs, a 92x performance increase.

After testing the naive kernel we experimented with CUDA performance optimizations. Both the diagonal data layout and the shared memory stencil brought significant performance improvements over the naive kernel, as seen in Figure 8. While the naive kernel achieved GFLOPs in the low 30s, the diagonal data layout transformation increased performance into the lower 40s, and the shared memory implementation reached almost 70 GFLOPs, essentially double the execution rate of the naive kernel, for time series length 100.

Our performance measurements for stop at a problem size of 40000 pairwise DTW comparisons because the entire problem is processed in a single 3D tensor of pairwise distance matrices that is transferred to the device once, to avoid mixing data transfer performance with computation performance.

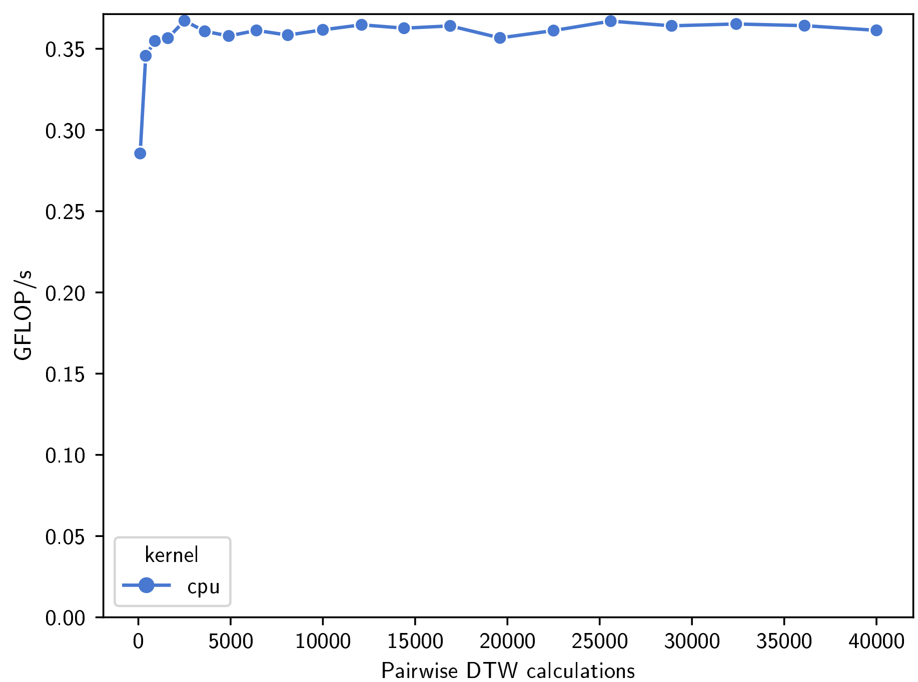

Figure 6: CPU performance on random unit normal time series length = 100

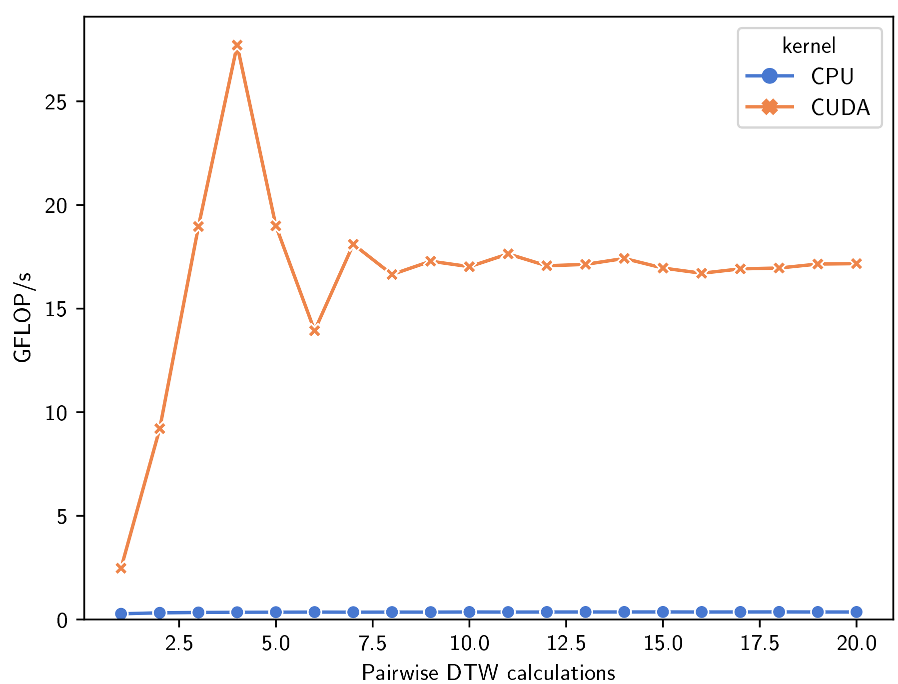

Figure 7: CPU vs. CUDA (Naive) performance on random unit normal time series length = 100

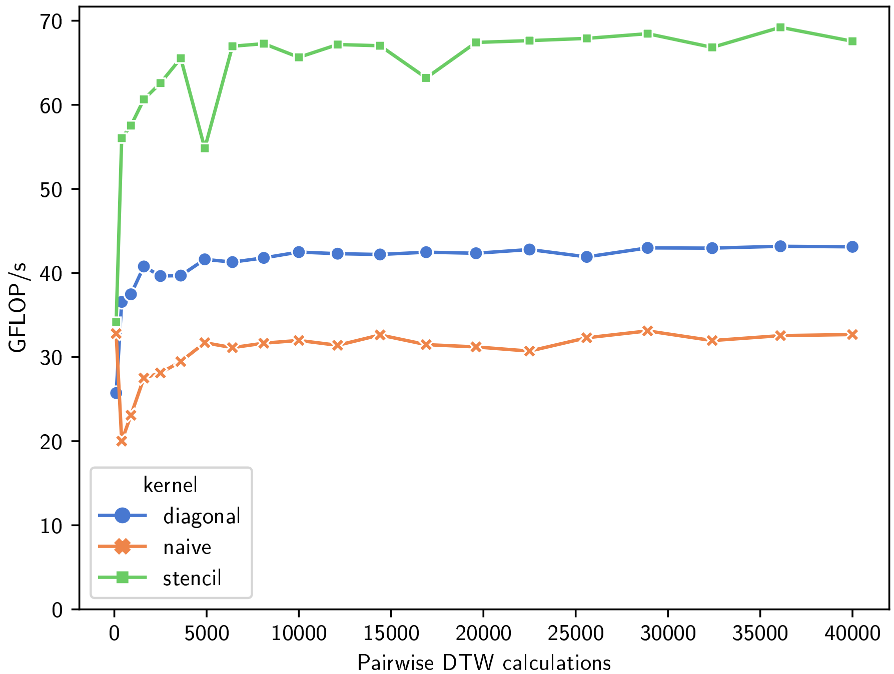

Figure 8: Kernel performance on random unit normal time series length = 100

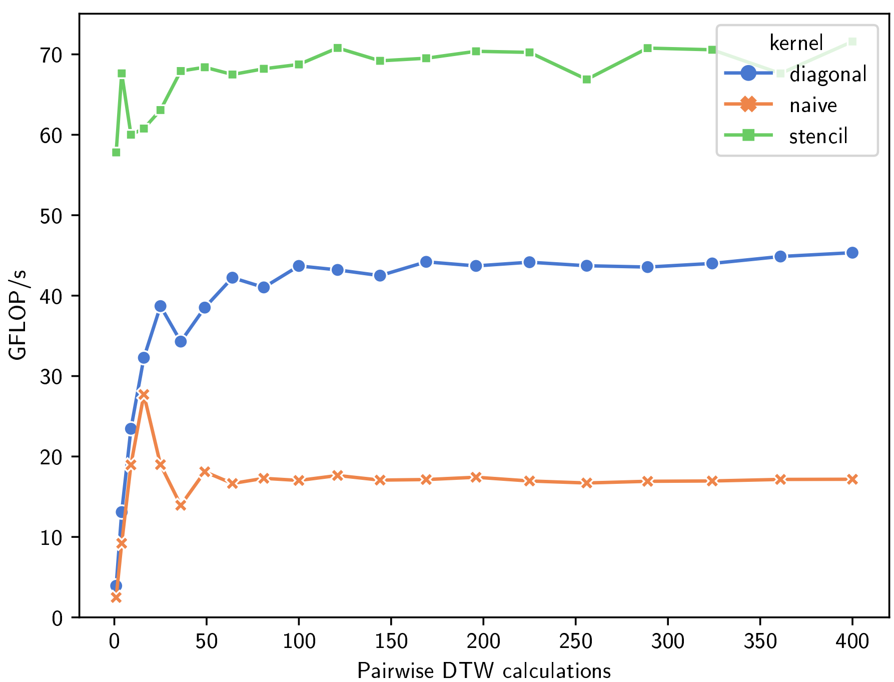

Figure 9: Kernel performance on random unit normal time series length = 1024

The diagonal-major layout consumes more space than the row-major layout ($(m+n-1) \times min(m,n)$ matrix vs. an $m \times n$ matrix), so the diagonal-major kernel is the first to exceed the 8 GiB device memory capacity of the test GPU hardware. A potential improvement for future work would be a program to batch the input data and launch the kernels on multiple streams to avoid exceeding device memory limits and hide the latency associated with multiple transfers.

Sakoe-Chiba bands also improved performance *even after* adjusting for the reduced number of total floating point operations; for input time series of length 100, a bandwidth of 40% of the matrix size resulted in the highest execution rate, exceeding 60 GFLOPs, compared to low-30s when the bandwidth is set to 100 (Figure 10).

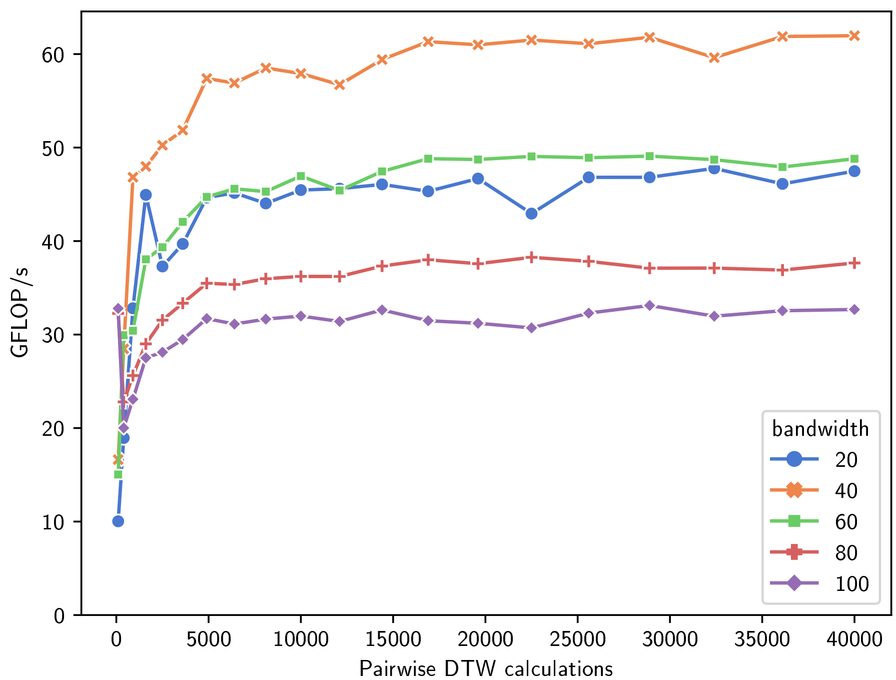

Figure 10: Naive kernel performance on random unit normal data by Sakoe-Chiba bandwidth at time series length = 100

However, when the input time series length was increased to 1024, the picture changed; the kernel with Sakoe-Chiba bandwidth equal to 20% of the matrix size performed the best, exceeding 50 GFLOPs, while the 40% bandwidth kernel fell below 40 GFLOPS and the others fell to just 20 GFLOPs or less (Figure 11). This suggests, intuitively, that a narrower bandwidth has a larger positive performance impact as the input time series length increases.

The final optimization we implemented and tested was the PeerWave style tiled kernel as described in [13]. As our other kernels are limited to time series of max length 1024 because they handle distance matrix per CUDA thread block, this kernel gave us the ability to test calculation of very long time series.

The tiled kernel uses global barriers to handle dependencies across rows or "waves" of tiles, with each row signaling to its "peer" in the next row when its work is completed. There is one loop inside the

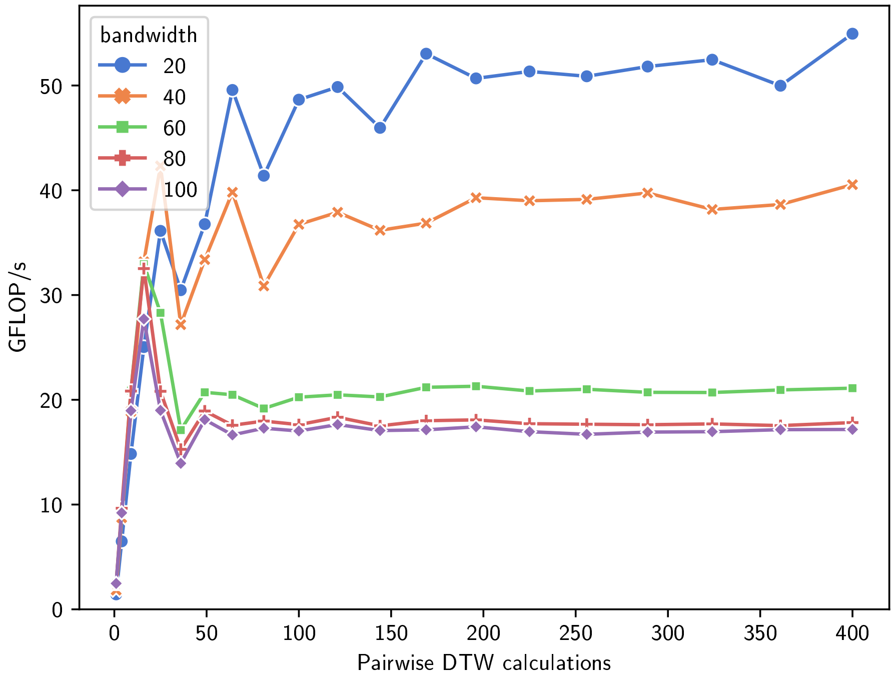

Figure 11: Naive kernel performance on random unit normal data by Sakoe-Chiba bandwidth at time series length = 1024

kernel to manage the wave and another iteration within each tile. In order to test on large sets of time series, we had to add two more nested loops to launch the kernel across multiple CUDA streams. This generated a high kernel launch overhead, decreasing the performance significantly and preventing us from comparing it with the other kernels, which are written to process multiple smaller distance matrices at once. So the main idea behind implementing a tiled version is to compute the similarity of time series for much longer sequences that the other kernels are unable to process. Figure 12 shows that, for shorter length sequences the performance is low, but by increasing the length, it is improving and capable of exceeding 120 GFLOPS, a far higher execution rate than the other kernels achieved on large sets of smaller time series pairs. Therefore, we suggest that it would be appropriate for a general-purpose DTW library to select this tiling approach when the input time series are very long.

We additionally tested our kernel optimizations on the ECG200 dataset, which is 200 time series of heart rhythms of length 96, so pairwise comparison of every time series to every other time series results in $96^2 \times 200^2$ SoftDTW cost cell computations. Table 3 shows performance on this dataset by kernel. The kernels named “squared euclidean distance” convey the time and GFLOPS performance needed to produce the distance matrix from the original input time series, before the Soft-DTW algorithm iterates over it to find the lowest-cost path. While we wrote all of our kernels to handle preprocessed squared Euclidean distance matrices, which imparts the ability to handle multivariate time series in addition to univariate, the overhead of computing this distance matrix suggests that it would be more time and memory efficient, in the case of univariate time series, to compute the DTW cost matrix on the input time series directly.

The kernel named “convert to diagonal” is indicating the computation for converting the squared

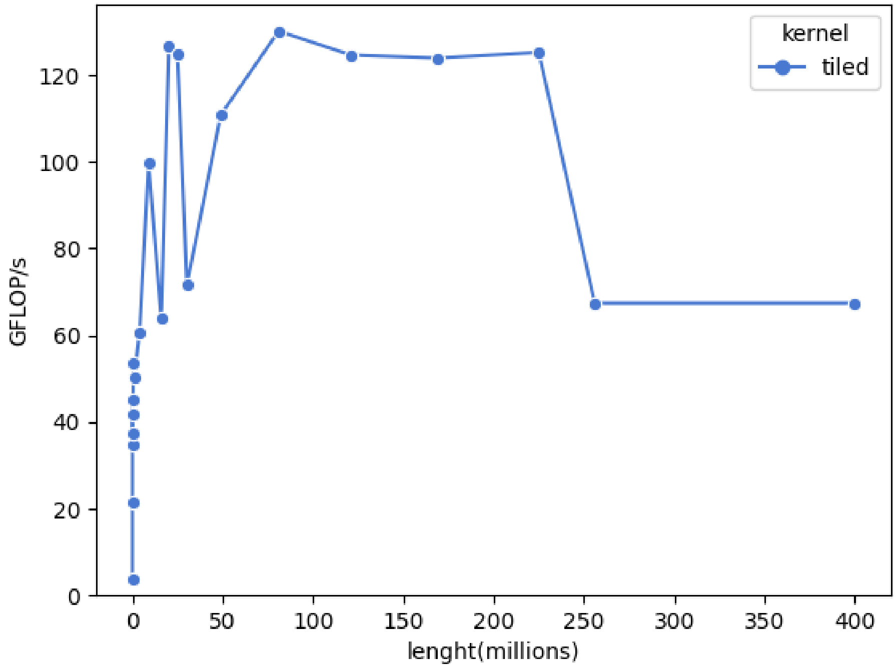

Figure 12: Soft DTW performance for tiled & shared memory kernel on random unit normal time series of varying lengths

euclidean distance matrix from row-major to diagonal-major layout; this process takes more than twice as long as the “diagonal“ kernel does, suggesting that constructing the cost matrix in diagonal-major format in the first place could produce a performance improvement in the end-to-end process. The shared memory stencil kernel produced the best performance at 69.24, with the naive CUDA kernel with Sakoe-Chiba band set to 40% of the matrix size performing next best.

Table 3: Performance by kernel on the ECG200 dataset.

kernel	microseconds	gflops
naive (CPU)	18171692.00	0.37
squared euclidean distance (CPU)	1374635.67	4.83
squared euclidean distance (CUDA)	507924.33	13.07
convert to diagonal	327152.67	20.28
naive (CUDA)	263765.00	25.16
naive bandwidth 80	232498.67	27.46
naive bandwidth 60	165440.67	33.85
diagonal	148617.67	44.65
naive bandwidth 20	45784.67	53.34
naive bandwidth 40	78164.33	54.84
stencil	95833.67	69.24

Profiling

We profiled our kernels with NVIDIA NSight Compute to understand how cache hit rates, registers per thread, and occupancy may be related to performance.

Table 4 shows profiler metrics by kernel when run on two pairs of time series of length 100.

Table 4: NVIDIA NSight Compute Profiler metrics by kernel

Kernel	Metric Value					
	L1 Cache Hit	L2 Cache Hit	Mem Busy	Occupancy	Registers / Thread	SM Busy
1 naive	97.70	93.57	0.76	12.5	43.0	8.37
2 diagonal	92.78	92.21	0.59	12.5	43.0	7.56
3 stencil	93.40	84.32	0.62	12.5	43.0	9.34

Discussion

TODO

Future Work

Our library provides several individual CUDA kernels and C++ wrappers for computing pairwise Soft-DTW dissimilarity measures in parallel. The library could be further improved by creating additional kernels that apply the optimizations to the Soft-DTW gradient computation as well, and that combine the best-performing optimizations together. Due to time constraints we were unable to explore the prefix computation technique described in Xiao et al (2013) [9] and incorporating an implementation of this algorithm remains a significant opportunity. Another library improvement would be a batching implementation that handles much larger problem sizes within limited device memory by launching kernels across multiple streams and transferring data for the next problem while the previous one is being computed, to hide latency.

Other potentially valuable future work includes integrating this library with an optimization library that can iteratively minimize Soft-DTW cost to find barycenters among a set of time series, to assemble a nearest centroid classification system. We attempted to include this capability but were unable to complete it satisfactorily within the project timeframe, as selecting, evaluating and integrating a C++ iterative optimization library is a significant amount of work and was tangential to our core goal of optimizing Soft-DTW.

Another area of potential is writing Python bindings to expose the Soft-DTW loss and gradient functions to popular deep learning frameworks such as TensorFlow or PyTorch, to enable the use of Soft-DTW loss as a training objective for recurrent neural networks. Cuturi and Blondel demonstrated that Soft-DTW can yield superior performance to Euclidean loss on multi-step ahead time series prediction with recurrent neural networks [1], so this making this widely available could facilitate better performance at tasks such as classifying, predicting and generating sounds, gestures, and sensor data with machine learning and neural networks, under the dynamic time warping geometry.

References

[1] M. Cuturi and M. Blondel, "Soft-DTW: A differentiable loss function for time-series," *arXiv:1703.01541 [stat]*, Feb. 20, 2018. arXiv: `1703.01541`. [Online]. Available: `http://arxiv.org/abs/1703.01541` (visited on 01/16/2021).

[2] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, "Addressing big data time series: Mining trillions of time series subsequences under dynamic time warping," *ACM Transactions on Knowledge Discovery from Data*, vol. 7, no. 3, 10:1–10:31, Sep. 1, 2013, ISSN: 1556-4681. DOI: `10.1145/2500489`. [Online]. Available: `http://doi.org/10.1145/2500489` (visited on 03/07/2021).

[3] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, Feb. 1978, Conference Name: IEEE Transactions on Acoustics, Speech, and Signal Processing, ISSN: 0096-3518. DOI: `10.1109/TASSP.1978.1163055`.

[4] B. J. Jain, “Semi-metrification of the dynamic time warping distance,” *arXiv:1808.09964 [cs, stat]*, Sep. 2, 2018. arXiv: 1808.09964. [Online]. Available: http://arxiv.org/abs/1808.09964 (visited on 03/18/2021).

[5] E. J. Keogh and M. J. Pazzani, “Derivative dynamic time warping,” in *Proceedings of the 2001 SIAM International Conference on Data Mining*, Society for Industrial and Applied Mathematics, Apr. 5, 2001, pp. 1–11, ISBN: 978-0-89871-495-1 978-1-61197-271-9. DOI: 10.1137/1.9781611972719.1. [Online]. Available: https://epubs.siam.org/doi/10.1137/1.9781611972719.1 (visited on 03/10/2021).

[6] J. Jordan. (Jan. 27, 2018). Normalizing your data (specifically, input and batch normalization)., [Online]. Available: https://www.jeremyjordan.me/batch-normalization/ (visited on 03/18/2021).

[7] E. Keogh, “Exact indexing of dynamic time warping,” in *Proceedings of the 28th international conference on Very Large Data Bases*, ser. VLDB ’02, Hong Kong, China: VLDB Endowment, Aug. 20, 2002, pp. 406–417. (visited on 02/14/2021).

[8] D. Shen and M. Chi, “TC-DTW: Accelerating multivariate dynamic time warping through triangle inequality and point clustering,” *arXiv:2101.07731 [cs]*, Jan. 19, 2021. arXiv: 2101.07731. [Online]. Available: http://arxiv.org/abs/2101.07731 (visited on 02/14/2021).

[9] L. Xiao, Y. Zheng, W. Tang, G. Yao, and L. Ruan, “Parallelizing dynamic time warping algorithm using prefix computations on GPU,” in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov. 2013, pp. 294–299. DOI: 10.1109/HPCC.and.EUC.2013.50.

[10] H. Zhu, Z. Gu, H. Zhao, K. Chen, C. Li, and L. He, “Developing a pattern discovery method in time series data and its GPU acceleration,” *Big Data Mining and Analytics*, vol. 1, no. 4, pp. 266–283, Dec. 2018, Conference Name: Big Data Mining and Analytics, ISSN: 2096-0654. DOI: 10.26599/BDMA.2018.9020021.

[11] S. Salvador and P. Chan, “FastDTW: Toward accurate dynamic time warping in linear time and space,” p. 11, 2004.

[12] M. Maghoumi, *Maghoumi/pytorch-softdtw-cuda*, original-date: 2020-05-02T23:28:24Z, Jan. 21, 2021. [Online]. Available: https://github.com/Maghoumi/pytorch-softdtw-cuda (visited on 01/23/2021).

[13] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu, “PeerWave: Exploiting wavefront parallelism on GPUs with peer-SM synchronization,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, Newport Beach California USA: ACM, Jun. 8, 2015, pp. 25–35, ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751243. [Online]. Available: https://dl.acm.org/doi/10.1145/2751205.2751243 (visited on 03/03/2021).

[14] TechPowerUp, *NVIDIA GeForce RTX 2060 SUPER Specs*, en. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-super.c3441 (visited on 02/28/2021).

[15] NVIDIA, *Achieved FLOPs*. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedflops.htm (visited on 02/28/2021).

[16] H. A. Dau, A. Bagnall, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, and E. Keogh, “The UCR time series archive,” *arXiv:1810.07758 [cs, stat]*, Sep. 8, 2019. arXiv: 1810.07758. [Online]. Available: http://arxiv.org/abs/1810.07758 (visited on 03/08/2021).