Implementing a Cross-Platform Dynamic Time Warping Algorithm for DNA Sequencing with ROCm HIP

CSE P 590B 24wi - Final Project Report

Alan Joseph Gabriel Boorse

INTRODUCTION

Dynamic Time Warping (DTW) is a widely-used technique for finding the optimal alignment between two time-dependent sequences of data. One variation of this algorithm, called Subsequence DTW (sDTW), can be ideal for finding these alignments when the two sequences are of significantly different lengths by finding a subsequence of the longer string that most closely matches its shorter counterpart [4]. This capability has a profound impact on public health, as it can be used to detect the presence of one smaller DNA fragment within a much larger sample of DNA. The ability to quickly and accurately match DNA sequences like this can expedite the identification of the viral strains of diseases such as COVID-19 [2].

Given that the computational complexity of the sDTW algorithm scales proportionately to the product of the lengths of the two sequences, many attempts have been made to parallelize this computation on specialized hardware like multi-core CPUs [5,6,10], FPGAs [7], and GPUs [1,2,3,8,9]. The recent efforts to parallelize the DTW algorithms entirely on graphics processing units (GPUs) have resulted in significant improvements in computational performance over alternatives, but these advancements have been made almost entirely using proprietary CUDA libraries, which restricts their use to hardware from only one manufacturer.

This paper describes a versatile, cross-platform implementation of sDTW using the Radeon Open Compute Platform (ROCm) Heterogeneous-Computing Interface for Portability (HIP) libraries from AMD. We demonstrate that an sDTW implementation using the ROCm HIP libraries is not only competitive in performance with previous CUDA implementations but also can significantly enhance the portability and accessibility of this important computational tool. This work represents a significant step forward in the use of GPU acceleration for DNA subsequence detection, with the potential to impact the rapid detection and treatment of diseases on a global scale.

BACKGROUND

DTW Algorithm

The DTW algorithm compares two sequences, represented as vectors, which can contain a normalized representation of nearly any kind of time-sequence data, including audio signals, fitness tracking data, or DNA.

In the first step of the algorithm, a cost matrix is laid out with its columns aligned to one sequence and the rows aligned to the other. Each cell in the cost matrix is then calculated to be the distance between the corresponding column-value of the first sequence and the row-value of the second. The algorithm then charts a path (the “warping path”) through the matrix with the lowest possible cost. Often, this is done using a dynamic programming technique that calculates each cell of the cost matrix as a the sum of its smallest left, upper, or upper-left neighbor and the cost value of the current cell, resulting in the final row of the matrix containing the cumulative costs [4].

The resulting warping path with the lowest cost indicates the most optimal possible alignment of these two sequences, and its cumulative cost can be compared to a threshold to indicate how well the sequences align [4].

GPU Computation of DTW

Computational complexity of the DTW algorithm scales proportionately to the product of the lengths of the two sequences [1]. This means that sequential calculation of the best possible warping path through the cost matrix on a CPU can become prohibitively compute intensive for sufficiently large sequences. Parallel DTW algorithm implementations, however, can take advantage of one property of the cost matrix to speed up computation, namely,

that each cell only depends on the values of its left, upper, and upper-left adjacent cells. Thus, parallelism can occur along the sequence of matrix diagonals that starts at the upper-left corner and marches downward and across to the bottom-right. For any given diagonal of the matrix, each cell in that diagonal can be computed entirely independently from the other cells in the diagonal since all left, upper, and upper left adjacent cells can be guaranteed to have already been calculated [1]. This independence facilitates parallelization.

Unlike CPUs, which are optimized for low-latency sequential processing of instructions, graphics processing units (GPUs) are composed of hundreds or thousands of small compute units that are designed for high throughput: executing massively parallel batches of simple computational jobs. These jobs, typically called threads, are organized into blocks, which are further organized into a grid. Threads within a block are guaranteed to be executed on the same logical processor, but blocks of threads within the grid are scheduled independently across the available hardware. All threads share the same global memory space on the GPU device, but they may also have access to a block-level shared memory space that has lower latency than global memory. Barrier synchronization mechanisms can be used to control thread-level execution, which is crucial for operations like sharing data across threads.

Combining these concepts then, a parallel algorithm for calculating the shortest warping distance can be developed which maps the sequence of cost matrix diagonals to threads and blocks, and uses a synchronization barrier while evaluating each diagonal to make sure all left, upper, and upper-left neighbors have been calculated before continuing on to the next diagonal. Implementations of this parallel algorithm may vary in performance depending on the GPU architecture (memory size and cache speeds, particularly) and the mapping of data to threads, blocks, and grid. This general approach, though, can result in highly performant computation of the DTW algorithm [1,2].

Benefit of the ROCm HIP Libraries

Prior GPU implementations of the DTW algorithm have used proprietary CUDA libraries, limiting their use to NVIDIA GPUs [1,2,3,8,9]. Low-cost solutions to the algorithm for DNA sequencing are highly desirable, as evidenced by the popularity of the portable, and relatively cheap, MinION sequencer from Oxford Nanopore Technologies [2]. Reliance on a single GPU manufacturer's hardware, then, is undesirable as it may impede the broader research and development efforts in regions such as China where obtaining a powerful NVIDIA GPU may not be practical or cost-effective [11].

The motivation to port DTW computation to AMD GPUs arises from a desire to democratize access to high-performance computing resources without vendor lock-in. The ROCm HIP libraries, which are developed by AMD, allow developers to write the same cross-platform code that can run on both AMD and NVIDIA GPU hardware. By leveraging this portability, critical applications such as DNA sequencing can avoid being impacted by external cost, shipping, manufacturing, or geopolitical constraints.

PRIOR WORKS

GPU Calculation

An initial implementation of the DTW algorithm for GPUs proposed by Sart, et al. in 2010 paved the way with a speedup of more than two orders of magnitude when compared to relevant CPU implementations at that time [7]. Subsequent breakthroughs improved on this speed with a prefix computation algorithm [9], memory access optimizations [8], and a combined CPU/GPU approach [10]. Modern GPU implementations like cudaDTW++ can achieve 2-3 orders of magnitude speedup over the closest CPU implementation by using warp shuffles to share data more efficiently between threads [1].

These innovations are particularly important in light of the likely truth that optimized CPU algorithms for dynamic time warping have reached their theoretical performance limit [15].

Applications in DNA Sequencing

DNA sequencing applications are particularly benefited by the use of DTW given its ability to align irregular sequences of nucleotides (A, C, T, G) despite variations in length, and practical implementations of DTW algorithms

for DNA sequencing have existed since at least 2013 [16]. Recent advances such as SquiggleFilter, however, have demonstrated the clear performance improvements of accelerating DTW using purpose-built hardware like ASICs [2]. The wide availability of GPUs and recent breakthroughs in performance enabled by warp shuffles have resulted in DTWax, which achieves 2-3X performance improvements compared to relevant competitors [3].

Other Applications

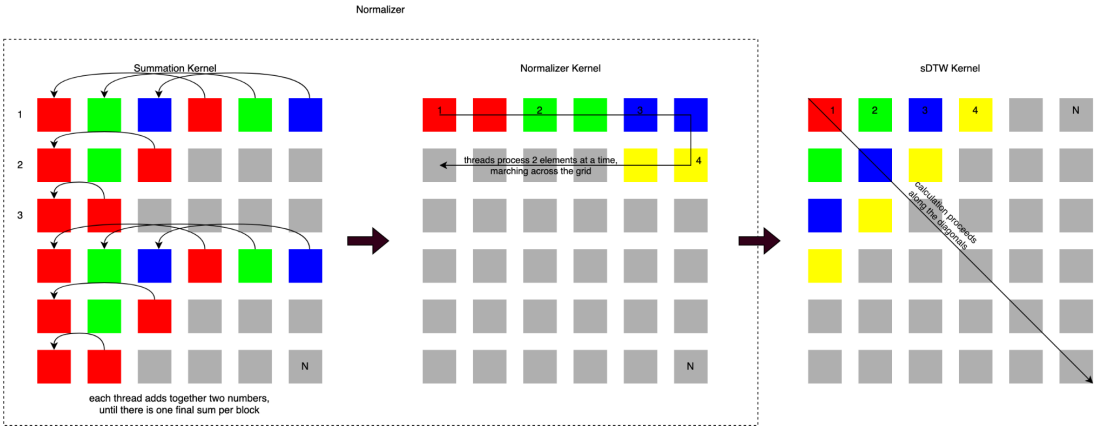
Generally, DTW algorithms can be useful for finding patterns in just about any kind of time-series data, including stock prices, NASA satellite telemetry data, hospital patient information, and credit card transactions [14]. Recent innovations in audio processing have led to the application of DTW for speech recognition [12], identifying song covers [13], and music synchronization [4]. GPU implementations of DTW algorithms provide breakthrough performance improvements for these applications, and many others.

METHODOLOGY

General Architecture

The approach taken for our design of the ROCm HIP implementation of sDTW was to divide the work between several GPU kernels (shared code executed across threads) to first normalize the data and then calculate the warping path. Normalization is required to make sure sequences that may be offset from each other but still have the same shape can still be matched. In the ROCm HIP implementation presented in Figure 1 below, normalization is split across two kernels, with sDTW calculation of the cost matrix occurring in the final stage of the process. More detail on each kernel has been provided later in the Implementation section of this report.

Figure 1: Architecture of the ROCm HIP Implementation



Testing Performance

To test the performance of our solution in a way that is relevant to DNA sequencing, a “reference” string of ~100K values was randomly generated (on the CPU), representing a potential sample of virus DNA, and normalized using the summation and normalizer kernels. Then, 512 “query” strings were randomly generated (on the CPU), representing potential patient DNA samples to test for the presence of the virus. Each of the query strings was then normalized and compared to the reference string using the sDTW kernel to test for a match.

This entire process of generating 512 query strings and comparing them to a reference string was repeated twice to “warm up” the GPU hardware and then repeated another ten times to measure the average performance of both the normalization and DTW processes in GFLOPS. All tests were run on the AMD Radeon Instinct MI100 GPU devices having 32GB of memory, 7680 processing cores, and clock speeds ranging from 1GHz (base) to 1.5GHz (boost).

IMPLEMENTATION

Normalizer Kernels

The implementation of the normalizer required a two-step approach to ensure data accuracy and high performance.

First, a summation kernel is launched on the GPU device to calculate the total sum of all elements in the reference or query string being normalized. This is done using a list reduction scheme where threads repeatedly add together numbers from increasingly smaller and smaller offsets until the first value in each block of data has the full sum. The sum of all squares of values is also calculated at the same time. To improve performance of this stage, each block of data is copied into shared memory to reduce global memory accesses, and the first sum is calculated inline, which cuts the number of required threads in half that must be launched on the device. A pseudocode representation of this process is provided in Figure 2.

Figure 2: Summation Kernel Pseudocode

```
declare shared sums array;
declare shared squaredSums array;

firstValue ← input[threadId];
secondValue ← input[threadId + offset];
sums[threadId] ← firstValue + secondValue;
squaredSums[threadId] ← firstValue² + secondValue²;

thread synchronization barrier;

for s ← blockSize / 2 down to 1, halving s each iteration
  sums[threadId] ← sums[threadId] + sums[threadId + s];
  squaredSums[threadId] ← squaredSums[threadId] +
                          squaredSums[threadId + s]²;

  if s equals 1
    outputSums[blockId] ← sums[threadId];
    outputSquaredSums[blockId] ← squaredSums[threadId];

  thread synchronization barrier;
```

The sums calculated in the first stage are copied to the CPU to quickly calculate the single values of mean (sum / length of data) and variance (√(squared sum / length of data - mean^2)).

The second stage uses the mean and variance values previously calculated to express the reference or query string values as distances from the mean using the following calculation: raw value - mean) / variance. A pseudocode representation of this process is provided in Figure 3.

Figure 3: Normalization Kernel Pseudocode

```
for i ← 0 up to itemsPerThread, incrementing i each iteration
  outputValue[threadId + i] ←
                (inputValue[threadId + i] - mean) / variance;
```

A step-by-step description of the normalizer process can be found in Figure 4.

Figure 4: The Normalizer Process

1. Copy string to global memory on the device
2. Launch summation kernel to calculate sum(X) and sum(X^2)
3. Copy sums from device to host
4. Calculate mean and variance on CPU
5. Launch normalization kernel to produce normalized output
6. (Optionally) Copy normalized string from device to host for debugging purposes, or leave on device memory for use in the sDTW kernel

sDTW Kernel

The kernel pseudocode to calculate the possible warping paths between the query string and reference string can be found in Figure 5.

Figure 5: sDTW Kernel Pseudocode

```
numDiagonals ← queryLength + referenceLength - 1;

for diagonalIndex ← 0 up to numDiagonals, incrementing each iteration
  if threadXPos equals threadYPos

    costMatrix[threadYpos][threadXPos] ← cost + min(
                         costMatrix[threadYPos - 1][threadXPos],
                         costMatrix[threadYPos][threadXPos - 1],
                         costMatrix[threadYPos - 1][threadXPos - 1]);

  thread synchronization barrier;
```

In this code for the ROCm HIP sDTW implementation, parallelization takes place along the diagonals. Considering a reference string of length M, and a query string of length N, the number of diagonals in the cost matrix is M + N - 1. This results in an average parallelization that corresponds to the average length of a diagonal in the cost matrix. A step-by-step description of this sDTW process can be found in Figure 6.

Figure 6: The sDTW Process

1. Assume that normalized query and reference strings have already been copied to device global memory by the normalizer kernels
2. Launch sDTW kernel with query and reference strings as inputs
3. Copy final row of the cost matrix to host memory
4. Find minimum value of cost matrix final row on the CPU

RESULTS

Following the testing process described previously in Methodology, the two warm up runs and ten performance runs produced an average normalizer throughput of 3.74×10^{-3} gigasamples/sec and sDTW performance of 1.39×10^{-5} gigasamples/sec.

CONCLUSION

Dynamic Time Warping algorithms like sDTW play a critical role in DNA sequencing, which can aid in the detection of deadly viruses like COVID-19. GPU acceleration of sDTW can make deploying this technology more practical, and transitioning from proprietary CUDA libraries to ROCm HIP has the potential to increase access to relatively high performance DNA sequencing applications in cases where it is not currently cost-efficient. Our implementation, which has competitive performance with existing CUDA implementations, could facilitate a substantial impact on rapid disease detection worldwide.

DISCUSSION

Due to time constraints, the sDTW algorithm ported here to the ROCm HIP libraries is missing some key performance optimizations that could make it more attractive in comparison to the previous CUDA implementations. For instance, 16-bit floating point numbers could be used instead of 32-bit to double the number of possible computations for the same amount of bandwidth. Additionally, shared and constant memory could be used to reduce the number of duplicated global memory reads and overall decrease the amount of time lost to memory latency. Finally, warp shuffles could be used to improve the performance of reading neighbor cell values in the cost matrix. These optimizations are left for a future report.

Full code for the implementation developed for this report is available on Github [17].

REFERENCES

1. Schmidt, B., Hundt, C. (2020). cuDTW++: Ultra-Fast Dynamic Time Warping on CUDA-Enabled GPUs. In: Malawski, M., Rzadca, K. (eds) Euro-Par 2020: Parallel Processing. Euro-Par 2020. Lecture Notes in Computer Science(), vol 12247. Springer, Cham. https://doi.org/10.1007/978-3-030-57675-2_37
2. Dunn, Tim & Sadasivan, Harisankar & Wadden, Jack & Goliya, Kush & Chen, Kuan-Yu & Blaauw, David & Das, Reetuparna & Narayanasamy, Satish. (2021). SquiggleFilter: An Accelerator for Portable Virus Detection. https://doi.org/10.1145/3466752.3480117
3. Sadasivan, Harisankar & Stiffler, Daniel & Tirumala, Ajay & Israeli, Johnny & Narayanasamy, Satish. (2023). Accelerated Dynamic Time Warping on GPU for Selective Nanopore Sequencing. https://doi.org/10.1101/2023.03.05.531225
4. Müller, M. (2007). Dynamic Time Warping. Information Retrieval for Music and Motion (pp. 69–84). Springer Berlin. https://doi.org/10.1007/978-3-540-74048-3
5. Xu, Xiaowei & Lin, Feng & Wang, Aosen & Yao, Xin-Wei & Lu, Qing & Xu, Wenyao & Shi, Yiyu & Hu, Yu. (2017). Accelerating Dynamic Time Warping With Memristor-Based Customized Fabrics. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. http://dx.doi.org/10.1109/TCAD.2017.2729344
6. Kraeva, Yana & Zymbler, Mikhail. (2019). Scalable Algorithm for Subsequence Similarity Search in Very Large Time Series Data on Cluster of Phi KNL. http://dx.doi.org/10.1007/978-3-030-23584-0_9
7. Sart, Doruk & Mueen, Abdullah & Najjar, Walild & Keogh, Eamonn & Niennattrakul, Vit. (2010). Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs. Proceedings - IEEE International Conference on Data Mining, ICDM. 1001-1006. http://dx.doi.org/10.1109/ICDM.2010.21
8. Hundt, Christian & Schmidt, Bertil & Schömer, Elmar. (2014). CUDA-Accelerated Alignment of Subsequences in Streamed Time Series Data. Proceedings of the International Conference on Parallel Processing. 2014. http://dx.doi.org/10.1109/ICPP.2014.10

9. Xiao, Limin & Zheng, Yao & Tang, Wenqi & Yao, Guangchao & Ruan, Li. (2013). Parallelizing Dynamic Time Warping Algorithm Using Prefix Computations on GPU. 294-299. http://dx.doi.org/10.1109/HPCC.and.EUC.2013.50
10. Liu, Y., Wirawan, A., & Schmidt, B. (2013). CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC bioinformatics, 14, 117. https://doi.org/10.1186/1471-2105-14-117
11. Leswing, Kif (2023, October 17). U.S. curbs export of more AI chips, including Nvidia H800, to China. CNBC. https://www.cnbc.com/2023/10/17/us-bans-export-of-more-ai-chips-including-nvidia-h800-to-china.html
12. Sakoe, H. and Chiba, S. (1978) Dynamic Programming Algorithm Optimization for Spoken Word Recognition. IEEE Transactions on Acoustics, Speech, and Signal Processing, 26, 43-49. https://doi.org/10.1109/TASSP.1978.1163055
13. Maršík, Ladislav & Rusek, Martin & Slaninová, Kateřina & Martinovic, Jan & Pokorný, Jaroslav. (2017). Evaluation of Chord and Chroma Features and Dynamic Time Warping Scores on Cover Song Identification Task. 205-217. http://dx.doi.org/10.1007/978-3-319-59105-6_18
14. Berndt, D.J., & Clifford, J. (1994). Using Dynamic Time Warping to Find Patterns in Time Series. KDD Workshop. https://aaai.org/papers/359-ws94-03-031/
15. Ratanamahatana, Chotirat & Keogh, E.. (2004). Everything you know about dynamic time warping is wrong. https://www.cs.ucr.edu/~eamonn/DTW_myths.pdf
16. Skutkova, H., Vitek, M., Babula, P., Kizek, R., & Provaznik, I. (2013). Classification of genomic signals using dynamic time warping. BMC bioinformatics, 14 Suppl 10(Suppl 10), S1. https://doi.org/10.1186/1471-2105-14-S10-S1
17. Joseph, Alan & Boorse, Gabriel. (2024). csep590b-24wi-final-project. Github. https://github.com/gnboorse/csep590b-24wi-final-project