Design and Implementation of a Parallel PTAS for Finding Structural Motifs on Graphics Processing Units (GPUs)

Julia Ysobel Pineda, Andrew Sopungco, and Jhoirene Clemente, PhD
jypineda@up.edu.ph, assopungco@up.edu.ph, jbclemente@up.edu.ph

Algorithms and Complexity lab
Department of Computer Science
University of the Philippines Diliman

Abstract. Protein structural motifs are recurring patterns in a set of tertiary protein structures in the 3D form. These motifs often play crucial roles in protein function, stability, and interactions. The problem is NP-hard, but a polynomial-time approximation scheme (PTAS) offers efficient solutions with some trade-offs. Previous work improved performance by optimizing subroutines and employing parallelization. The current study investigates further improvements using a two-level parallelization approach, offloading computationally-intensive SVD operations to GPUs while keeping the rest on CPU. The comparison between CPU and GPU implementations is based on speedup and parallel efficiency metrics. While the trends in results are similar, the GPU implementation exhibits significantly delayed execution times. The study provides insights into the potential benefits of the two-level parallelization approach and offers data-driven suggestions for further advancements in the solution. Overall, it aims to contribute to computational optimization in bioinformatics and explore novel methods for solving the structural motif finding problem.

Keywords: bioinformatics, structural motif finding problem, polynomial-time approximation scheme, PTAS, parallelization, GPU

1 Introduction

1.1 (R,C)-Compact Structural Motif Finding Problem

A prominent computational problem under bioinformatics is the structural motif finding problem, which involves identifying important patterns and functional elements in 3D protein structures in order to gain a better understanding of the relations between their origins, functions, and projections. By identifying motifs, we are able to provide insights into protein folding, stability, interactions, and functional properties.

The (R,C)-Compact Structural Motif Finding Problem as discussed in Qian (2007) [4], is a specialized approach to identifying compact structural motifs, especially for small and well defined ones, which is defined as follows:

J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2023)*, Atlantis Highlights in Computer Sciences 20,
https://doi.org/10.2991/978-94-6463-388-7_20

Definition 1 ((R,C)-compact motif). *An (R,C)-compact motif is bounded by the minimal ball B with a radius at most* $\boldsymbol{R}$*, and at most* $\boldsymbol{C}$ *residues in this ball do not belong to this motif. This ball* $\boldsymbol{B}$ *is referred to as the* ***containing ball****.*

Definition 2 ((R,C)-Compact Consensus Structural Motif problem). *The (R,C)-CCSM problem states that given n protein structures* $P_1, P_2, \cdots, P_n$*, and an integer* ℓ*, find an (R,C)-compact motif* u_i *of length* ℓ *along with rigid transformation* τ_i *for each* P_i *and a consensus structure of* ℓ *3D points:* $q = (q_1, q_2, \cdots, q_\ell)$*, where* q_i *is a point in 3D, minimizing distance function* $\sum_{i=1}^{n} d(q, \tau_i(u_i))$.

The algorithm performs rigid transformations, or rotations and translations, on the chosen motifs, and measures an objective function that measures the similarity in structure. Returning the combinations with the minimal distance then allows us to identify the representative segment.

1.2 Research Problem and Objectives

The structural motif finding problem is an NP-hard problem that has a PTAS [4]. While this provides a sufficient solution to the problem, it is still impractical as it experiences trade-offs between speed and accuracy.

Brocka and Yap (2022) [2] successfully parallelized the non-data dependent subroutines of the algorithm and achieved a notable speedup of up to 5 times while retaining the quality of the solution. Recognizing the potential benefits of parallelizing the PTAS approach, we are motivated to explore further enhancements. Our research seeks to extend their work by introducing an additional step to the existing parallel PTAS implementation on CPUs by leveraging the computational capabilities of Graphics Processing Units (GPUs), in order to achieve even faster execution times while maintaining the same solution quality. This research provides valuable insights into the improvement in time-complexity of the PTAS, which is crucial for evaluating its scalability and performance.

Specifically in the context of the structural motif finding problem, it studies and analyzes a new method that could result in well-approximated solutions computed for in a shorter amount of time as compared to previous methods.

Specific objectives involve designing a parallel implementation for the algorithm on GPUs, measuring speedup and parallel efficiency, and comparing CPU and GPU performance.

In doing so, the GPU implementation will be tested on the same dataset used in the previous research and shall be compared in terms of speedup and parallel efficiency, in order to establish a valid comparison between the sequential and CPU parallel algorithms.

The datasets utilized in the implementation and testing of the CPU parallel PTAS from Brocka and Yap [2] served as a consistent baseline for comparing the two parallel implementations. One of these datasets specifically focuses on Conantokin peptides and is obtained from PepSquad (2017) [3], the primary reference for the algorithm employed. It is worth noting that this dataset comprises

a refined or sanitized version of the original data. For a comprehensive analysis, we will also acquire the complete dataset from the Protein Data Bank (PDB) maintained by the Research Collaboratory for Structural Bioinformatics (RCSB PDB).

This study aimed to explore the algorithm's capabilities and limitations under different conditions through a processor test, sample test, and motif length test.

2 Review of Related Literature

2.1 Parallel PTAS for Finding Compact Structural Motifs (PepSquad, 2017; Brocka & Yap, 2022)

Kabsch's algorithm (Algorithm 1) is a method to calculate the optimal rotation matrix that minimizes the root mean square deviation (rMSD) of two sets of points [9]. Since this is an integral subroutine of Qian's (R,C)-CCSM algorithm [4], and has been shown to be an efficient method for finding the optimal rotation matrix, a revised algorithm (Algorithm 2) is used in the CPU-parallel implementation of the root-finding algorithm.

Brocka and Yap [2] focused on the parallelization of the algorithm in order to maximize the resources used and eventually speeding up the execution itself, improving the runtime while keeping the same solution quality.

Using the following high-level steps of the PepSquad algorithm (Algorithm 2), Figure 1 shows the data dependency graph which visualizes the steps that can be implemented in parallel. Numbering in parentheses are based on Qian's (R,C)-Compact Motif Finding Algorithm [4] and used in the proposed approach's new data dependency graph (Figure 2).

1. **ENTRY**
2. Fix P_1, translate other proteins to make centers coincide (1)
3. Select a length-l compact motif $u_1, u_2, \cdots, u_r$ where u_i is a motif of some P_j and x is the total samples (2)
4. Select a transformation $\tau_2, \tau_3, \cdots, \tau_r$ (3)
5. Find the median for each discrete rigid transformation u (3a)
6. Find the compact motif that minimizes the cMSD distance v_i where $i = 1, 2, \cdots, n$ (3b)
7. Compute the objective function $c(u)$ (3c)
8. **OUTPUT**

Of these steps, steps 3, 4, 6, and 7 would be implemented in parallel. Particularly, they are the steps involving the following operations:

1. Selecting a compact motif from the protein,
2. Selecting a transformation,
3. Finding the compact motif that minimized the rMSD distance, and
4. Computing the objective function value.

Through Python's multiprocessing library, Brocka and Yap (2022) [2] were able to have the parallelizable processes running simultaneously on multiple CPU cores.

Algorithm 1 Kabsch Algorithm

procedure KABSCH($\mathcal{P}_1, \mathcal{P}_2$)
  Align the centroids of $\mathcal{P}_1$ and $\mathcal{P}_2$ to the origin.
  Calculate matrix $\mathbf{R}$ s.t. its elements, $r_{i,j} = \Sigma_n w_n \mathcal{P}_{2_{ni}} \mathcal{P}_{2_{nj}}$
  $\mathbf{A}$=$\mathbf{R}^T\mathbf{R}$
  Calculate eigenpairs , (m_k,v_k), where m_k's are the eigenvalues and v_k's are the eigenvectors.
  Sort the eigenpairs s.t. $m_1 \geq m_2 \geq m_3$
  Set $v_3 = v_1 \times v_2$
  Calculate $c_k = Rv_k$. Compute b_1 and b_2 by normalizing c_1 and c_2
  Set $b_3 = b_1 \times b_2$
  $\mathbf{U} = u_{ij} = \Sigma_k b_{ki} a_{kj}$
  Compute *RMSD*
  return *U*
end procedure

Algorithm 2 PepSquad Algorithm

procedure PEPSQUAD($\mathcal{P}_1, \mathcal{P}_2, \cdots \mathcal{P}_n, \ell, C, r, R$)
  Fix $\mathcal{P}_1$
  for all $\mathcal{P}_i$ in $\mathcal{P}_2, \mathcal{P}_3, \cdots \mathcal{P}_n$ **do**
    Translate $\mathcal{P}_i$ to coincide its centroid with $\mathcal{P}_1$
  end for
  for all r length-$\ell(R, C)$-compact motif $u_1, u_2, \cdots, u_i$ is a substructure of some $\mathcal{P}_j$ **do**
    for all $u_i \in u_2, \cdots, u_r$ **do**
      $\tau_i = Kabsch(u_1, u_i)$
    end for
    $u \leftarrow \frac{1}{r}(u_1 + \tau_2(u_2) + \cdots + \tau_r(u_r))$
    for i in 1...n **do**
      Find (R, C)-compact motif of length ℓ, v_i of $\mathcal{P}_i$ and the optimal rigid transformation $\tau_i{}'$ that minimizes $d(u, \tau_i{}'(v_i)$ using Kabsch's Algorithm.
      $c(u) \leftarrow \Sigma_{i=1}^{n} d(u, \tau_i{}'(v_i))$
    end for
  end for
  return $u, v_i, \tau_i{}'$ that minimizes $c(u)$
end procedure

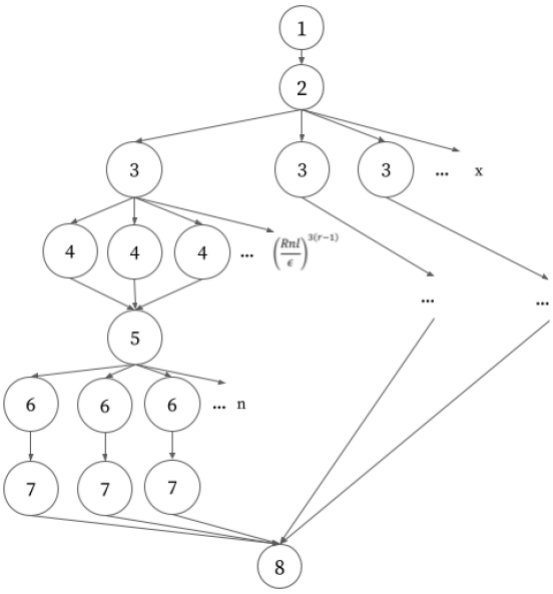

Fig. 1. Brocka & Yap (2022)'s Data Dependency Graph

2.2 On GPUs

GPUs, when compared to CPUs appear to be more fit for tasks designed to be run in parallel. This is evident when examining the basic CPU and GPU architecture.

As the CPU is more concerned with control and communication between hardware and software, we see that only a few processing/compute units are present, with a majority of the die allocated to cache. On the other hand, GPUs, since they are not concerned with the main control, have more compute units present, each with their own cache and control.

Gupta (2011) [7], among other studies, have been able to show that GPU parallel implementations can perform better than CPU multiprocessing. They showed it in an Application on Natural Language Processing, a field which also sees great help from GPU implementations of their algorithms.

Given what we know about the structural motif finding problem, as well as the significant amount of speedup brought about by introducing parallelization of the PTAS, implementing its algorithm on GPUs may give us promising results. By implementing our parallel algorithm with GPUs, we may achieve an even higher speedup given its parallel processing capabilities.

2.3 Computing RMSD using GPUs (Barrios, 2021)

Barrios (2021) [1] implemented Kabsch's algorithm (Algorithm 1) for computing rMSD using NVIDIA CUDA, mainly hoping to maximize the representation of the input in Kabsch's algorithm as well as to compute the values needed more efficiently. Since Kabsch's algorithm makes use of matrices and matrix multiplication repeatedly, they can be better represented by GPUs, which are designed to be able to handle 3D or matrix representations. This is very relevant to our main objective, which is to implement a structural motif finding algorithm that employs Kabsch's algorithm as a subroutine in GPUs. Referencing this study, we can implement a more complete version of Kabsch calculations (since this did not implement the square root of a matrix) and implement it as a part of the entire algorithm.

3 Parallel PTAS on GPUs

3.1 Parallel Algorithm

The previous CPU parallelization study implemented the entirety of the motif finding algorithm using Python's multithreading library. The approach taken by the CPU parallel scheme uses multiple threads to implement the non-dependent points of the algorithm. They applied these on the 3 identified parallelizable parts of the algorithm as seen in Figure 1:

1. Selecting compact motifs (Step 2)
2. Selecting transformations (Step 3)
3. Computing for the optimal transformations (Step 3b-3c)

Thus, a two-level approach was adopted: the SVD segment is offloaded to the GPU, while the remaining sections of the algorithm are maintained in parallel execution on the CPU. The code responsible for the SVD operation was transferred to the GPU utilizing the cuSOLVER library provided by NVIDIA, thereby enabling efficient and effective execution on the GPU architecture.

In the two-level parallelization scheme, steps 4 and 6 (which involve Kabsch's algorithm and are hereby referred to as steps 3 and 3b in Figure 2) are executed in GPU due to their computational intensity. Instead of parallelizing the entire algorithm, the focus was on the singular value decomposition (SVD) operation, which resides within a subroutine involving matrix multiplication with superposition and is a crucial process for factoring a matrix into three distinct matrices, providing insights into their linear transformations.

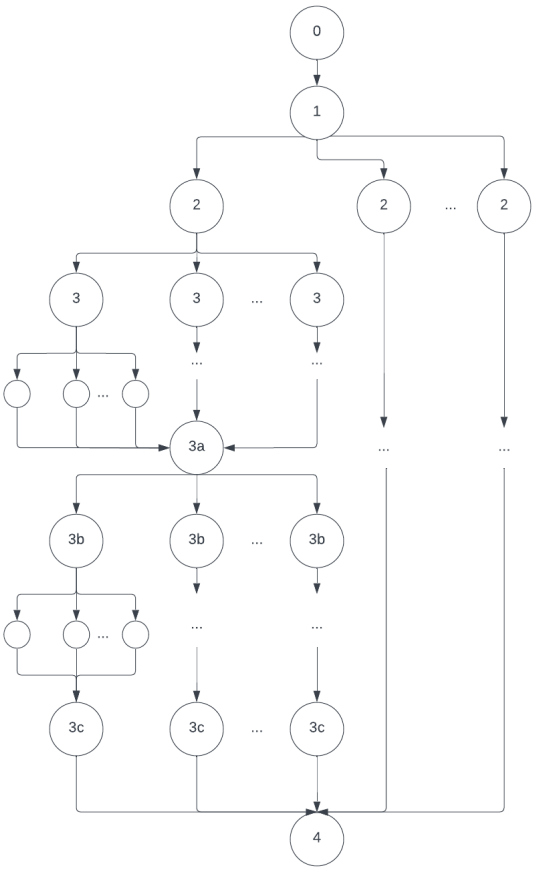

Fig. 2. Data Dependency Graph of the Two-Level Approach

3.2 Theoretical Analysis

To assess the performance of the CPU-GPU parallel implementation, the results are gauged based on measures of speedup and parallel efficiency.

Definition 3 (Speedup). *Speedup measures the ratio of performance of a process with and without the improvement, which in this case is the running time using CPUs and GPUs, allowing us to see the relative improvement generated by the usage of GPUs.*

$$\text{speedup} = \frac{\text{cpu running time}}{\text{cpu-gpu running time}}$$

Definition 4 (Parallel efficiency). *Parallel efficiency shall be measured in relation to the number of processors used by dividing the obtained speedup and*

number of processors used. This will allow us to see how much computing capacity is actually used by the implementation.

$$\text{parallel efficiency} = \frac{\text{speedup}}{\text{number of processors}}$$

The CPU parallel implementation improved the overall performance of the sequential algorithm by reducing the time complexity by a factor of the number of processors used. By offloading a subroutine to the GPU, the number of processors is magnified, resulting in a greater improvement in time complexity. The original algorithm has a time complexity of $O(m^2n + n^3)$, where n represents the size of the matrix [6], and is dominated by the computationally heavy SVD operation, which can be accelerated on the GPU. However, data transfer between the CPU and GPU introduces additional overhead, and the algorithm's efficiency depends on the balance between GPU parallelization speedup and data transfer overhead.

3.3 Empirical Results

All tests were conducted on Google Colab, with the following CPU and GPU hardware specifications:

CPU: Intel Xeon CPU (2.20 GHz), 8 Cores	12 GB RAM
GPU: NVIDIA T4 (CUDA ver 12.0) 2560 CUDA Cores 320 Tensor Cores	16 GB GDDR6 VRAM

The relevant parameters modified during testing were the following:

1. Sample size (r)
2. Processor size (*num_processors*)
3. Motif length (*BENCHMARK_LENGTH*)
4. Maximum ball size (b)

Each test was done on both the CPU parallel and CPU-GPU parallel implementations, ensuring consistency by using the same parameters across both approaches.

Sample Tests The research aimed to investigate the algorithm's capability to handle larger datasets by increasing the number of samples. The Conantokin dataset, consisting of 1,521 samples, was used for the experiments, varying the sample size from 2 to 5. As expected, larger sample sizes led to longer test durations. However, the CPU-GPU implementation showed inefficiency in handling tests with larger sample sizes, especially when the sample size (r) exceeded 3. The average completion time for the CPU-GPU parallel implementation was approximately 5 times longer than the CPU parallel implementation (Figure 3). Specifically, the SVD operation, offloaded to the GPU, took 1000-4000 times longer to execute compared to the CPU parallel implementation (Figure 4).

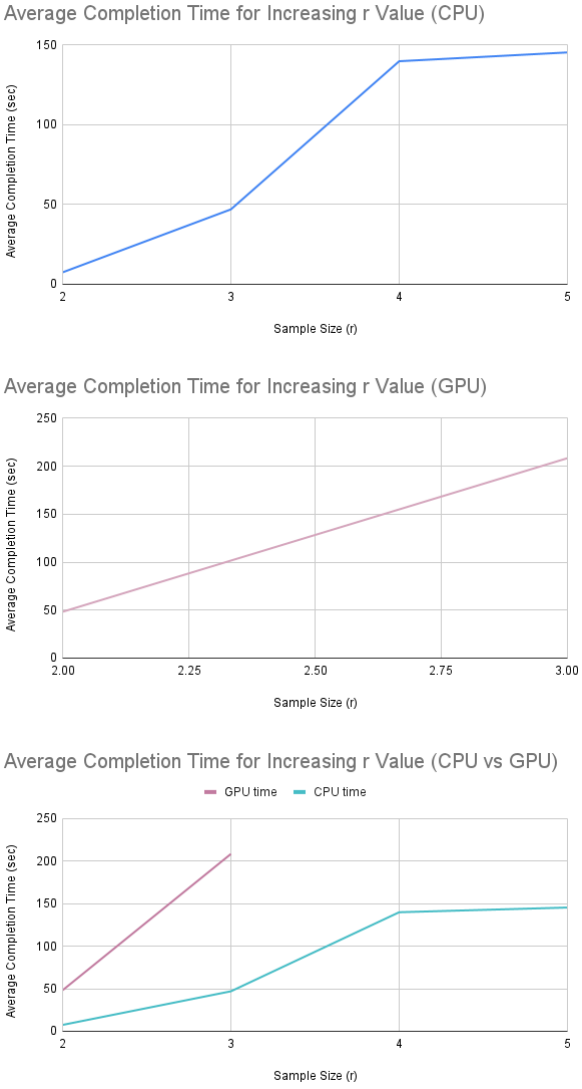

Fig. 3. Comparison of Completion Time for Increasing r Value

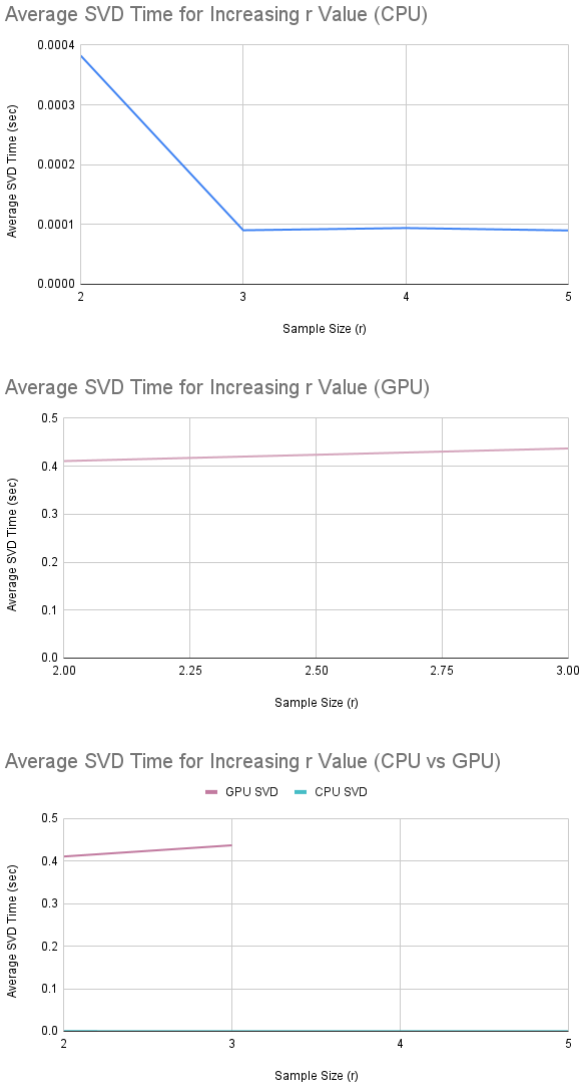

Fig. 4. Comparison of Average SVD Time for Increasing r Value

Processor Tests The study also investigated the system's ability to handle larger computational loads by distributing the workload among multiple processors. The test measured the speedup and parallel efficiency achieved from this distribution, both before and after integrating the GPU. The experiments used the Conantokin dataset with 1,521 samples, but only less than 600 (40% or less of the samples) were evaluated. The results showed no significant trend in the execution time within this limited sample range (Figure 5).

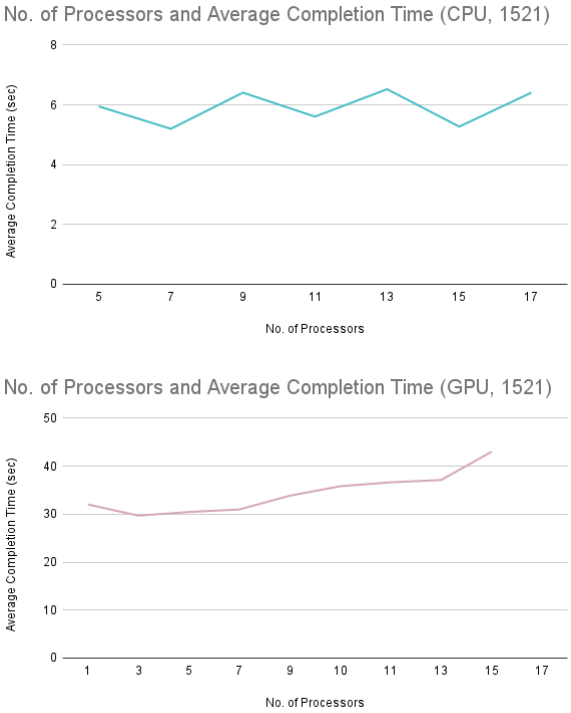

Fig. 5. Comparison of Completion Time for Increasing Number of Processors

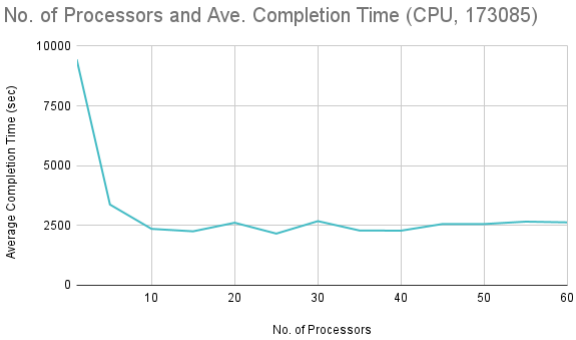

Fig. 6. Comparison of Completion Time for Increasing Number of Processors Using The Full Conantokin Dataset

Motif Length Tests The study conducted additional processor tests (Figure 6) using the complete Conantokin dataset with 173,085 samples (*BENCHMARK_LENGTH* = 3), similar to the approach of [2]. The results indicated a notable improvement when transitioning from 1 to 5 processors, with a consistent trend between 10 and 15 processors. However, the CPU-GPU implementation had considerably longer execution times, which led to the inability to complete the remaining tests within the allocated time frame.

Based on our observations from the initial sparse sample and processor tests, we conducted additional experiments to evaluate the algorithm's performance with varying motif lengths. Using the restricted Conantokin dataset, we tested motif lengths from 1 to 7. Comparing the runtime of the CPU and GPU implementations, the CPU-GPU parallel approach still took approximately 5 times longer than the strictly-CPU parallel implementation. As we increased the motif length, fewer samples fit the constraints of the (R,C)-compact structural motif finding algorithm, resulting in decreased overall runtime (Figure 7).

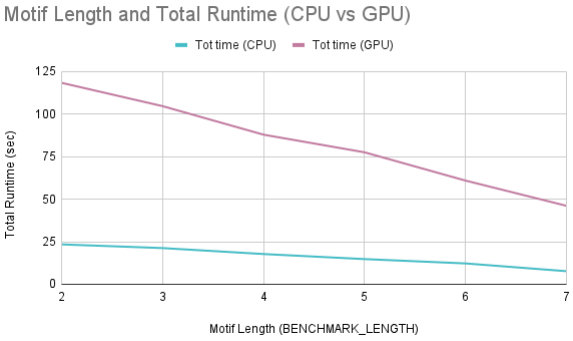

Fig. 7. Comparison of Motif Length and Runtime for both CPU and GPU

However, when examining the individual calls to the SVD function, the time required to perform the SVD operations did not exhibit significant improvements with increasing motif lengths. Instead, it remained relatively consistent throughout the testing and decreased only when the number of evaluated samples decreased (Figure 9).

Upon further analysis of the runtime, we discovered that data transfer had a substantial impact. On average, each call to the SVD function required 0.13 seconds to transfer data between the CPU and GPU, which accounted for approximately one-third of the time required to execute the SVD subroutine on the GPU (Figure 10).

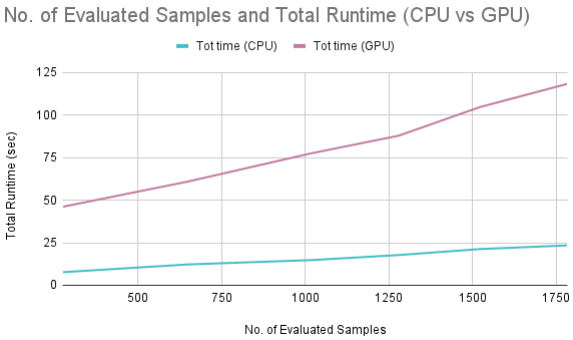

Fig. 8. Comparison of Evaluated Samples and Total Runtime for both CPU and GPU

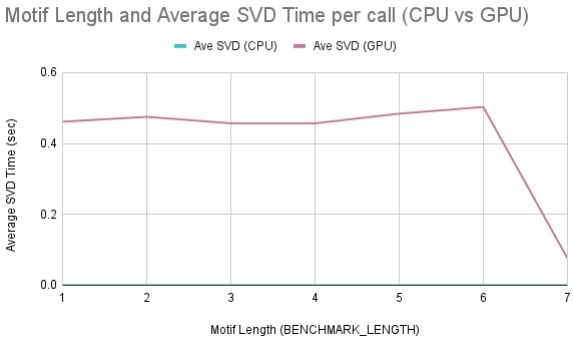

Fig. 9. Average SVD Time for both CPU and GPU

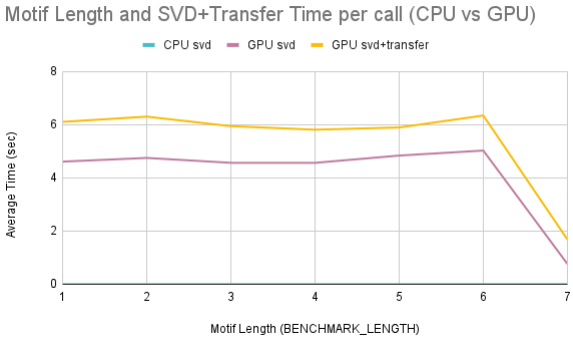

Fig. 10. Average SVD Time for both CPU and GPU, including Transfer Time per call

Speedup and Parallel Efficiency Comparing the overall execution time of the CPU parallel and CPU-GPU parallel implementations, an approximate speedup of 0.2 was attained. More detailed results can be seen in Tables 1 and 2,

Table 1. Completion Time Speedup

Parameter	Value	Speedup	Average Speedup
r	2	0.1552607937	0.1903
	3	0.2254195354	
BENCHMARK_LENGTH	1	0.1979660559	0.1795367688
	2	0.2031934411	
	3	0.2021070214	
	4	0.1908049001	
	5	0.200978614	
	6	0.1667725823	
	7	0.0949347671	

Table 2. SVD Time Speedup

Parameter	Value	Speedup	Average Speedup
r	2	0.000931075589	0.000568855119
	3	0.000206634649	
BENCHMARK_LENGTH	1	0.0001036505625	0.0003458342706
	2	0.0004974319093	
	3	0.0001680972508	
	4	0.0001969677823	
	5	0.0001480508308	
	6	0.0001602075774	
	7	0.001146433981	

In terms of parallel efficiency, no conclusive results could be attained due to the limitations in testing. Comparing the individual runtimes of processor tests done on the restrictive dataset, which resulted to no apparent trend in results, Table 3 shows values for speedup and parallel efficiency attained.

Comparison and Analysis Overall, the results showed a similar trend to those attained by Brocka and Yap [2]. However, the two-level parallelization scheme resulted in runtimes that were 5x longer than the CPU parallel implementation, if using the same parameters. Consequently, resource constraints did not allow for the testing of the GPU code on larger samples.

In general, GPUs utilize a larger number of computing cores compared to CPUs, allowing for effective parallel processing. When incorporating GPUs into

Table 3. Speedup and Parallel Efficiency on Processor Test

No. of Processors	CPU	GPU	Speedup	Parallel Efficiency
1	6.8749	32.0213	0.2146977168	0.2146977168
3	5.39655	26.6909	0.2021868877	0.06739562922
5	5.9499	30.46045	0.1953319797	0.03906639593
7	5.19925	30.9804	0.1678238499	0.0239748357
9	6.4091	33.8557	0.1893063797	0.02103404219
11	5.60785	36.64435	0.153034506	0.01391222782
13	6.5239	37.129	0.1757090145	0.01351607804
15	5.2729	43.03905	0.1225143213	0.00816762142
17	6.40855	50.3756	0.1272153582	0.007483256366

the parallelized CPU code, the combined computing power of both CPU and GPU cores is maximized. However, as the number of computing units increases, the runtime scales up accordingly, resulting in an amplification of the observed results.

GPUs are well-suited for parallel computing due to their numerous CUDA cores, which are smaller and less powerful individually but are available in greater quantities, allowing for handling more concurrent threads. However, for smaller inputs that do not require as many processors, the benefits of GPU parallelization may not outweigh the disadvantages.

4 Conclusion and Recommendations

The study's findings indicate that further investigation is needed to reach conclusive results. While the data collected provides valuable insights into the incorporation of GPUs and their impact on speedup and parallel efficiency, it does not offer a definitive answer to the research question. This opens up opportunities for future research to delve deeper into the topic and explore additional factors that can contribute to a more conclusive understanding.

Improvements in code design are crucial to efficiently distribute the workload across available CPU and GPU cores, ensuring proper task assignment and synchronization. Minimizing data transfer between CPU and GPU, while considering system overhead, can further enhance computational efficiency. Although no single approach for efficient SVD on GPUs was identified, potential strategies include leveraging GPU parallelizing libraries like PyTorch and TensorFlow, or adopting scheduling strategies like batch processing to maximize GPU resources. Resource sharing in Google Colab may limit full utilization of GPU power. Future research could explore specialized or more powerful GPU hardware to accelerate testing and evaluate performance differences.

Despite disparities in GPU and CPU code performance, the research journey has provided valuable insights that, through further experimentation and optimization, could lead to desired algorithm improvements.

5 Acknowledgements

We sincerely thank Ms. Jhoirene B. Clemente, PhD, for her invaluable guidance and patience throughout our research journey. Her knowledge and insightful suggestions significantly contributed to our research progress. We also extend our gratitude to the professors of the Algorithms and Complexity Lab and the Service Science and Software Engineering Lab for their valuable insights and constructive feedback, which enhanced the quality of our work. Additionally, we appreciate the support from the College of Engineering's High Performance Computing (HPC) Facility, providing essential resources for our study. Finally, we thank our labmates, fellow research students, friends, and family for their relevant knowledge and unwavering moral support.

References

1. Barrios, N. (2021). Computing RMSD using GPGPUs.
2. Brocka, B. and Yap, S. (2022). Parallel PTAS for Finding Compact Structural Motifs.
3. Carandang, J. P. A., Clemente, J. B., Evangelista, J. E. M., and Adorna, H. N. (2017). Pepsquad: A tool for finding compact structural motifs from peptides. In Theory and Practice of Computation: Proceedings of Workshop on Computation: Theory and Practice WCTP2017, pages 49–58. World Scientific.
4. Qian,J.,Li,M.,Li,S.C.,Bu,D.,andXu,J.(2007).Findingcompactstructural motifs. Theoretical Computer Science. doi:10.1016/j.tcs.2009.03.023
5. cupy.linalg.svd — CuPy 12.1.0 documentation.
6. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., & Yamazaki, I. (nd). The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale doi:10.1137/17M1117732
7. Gupta, S. and Babu, M. R. (2011). Performance analysis of gpu com- pared to single-core and multi-core cpu for natural language applications. IJACSA Editorial.
8. What Is a GPU? Graphics Processing Units Defined, Intel Corporation. https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html#:~:text=Graphics%20processing%20unit%2C%20a%20specialized,video%20editing%2C%20and%20gaming%20applications.
9. Kabsch, W. (1976). A solution for the best rotation to relate two sets of vectors. Acta crystallographica. doi:10.1107/s0567739476001873
10. numpy.linalg.svd — NumPy v1.25 Manual.