GPU-Accelerated Aggloremative Hierarchical Clustering Integrated LSH: A Comparative Analysis

Ali Yiğit Başaran
Data Informatics
Middle East Technical University

***Abstract*— Agglomerative Hierarchical Clustering (AHC) is a powerful technique widely used in data analysis across various fields such as bioinformatics, linguistics, and computer vision. However, its traditional implementations face significant challenges in handling large, high-dimensional datasets due to their computational and memory-intensive nature. This project addresses these challenges by integrating Locality-Sensitive Hashing (LSH) within a GPU-accelerated framework using CUDA for parallel processing.**

The motivation for this work stems from the need to improve the efficiency and scalability of hierarchical clustering methods to better manage large datasets. By leveraging the parallel processing capabilities of GPUs, the proposed method aims to drastically enhance the speed and efficiency of clustering operations. The integration of LSH reduces the dimensionality of the data, allowing similar data points to be grouped together efficiently, thus reducing the computational overhead during the clustering process.

Our results demonstrate a significant improvement in processing times and clustering accuracy compared to traditional CPU-based methods. The GPU-accelerated approach not only handles larger datasets more effectively but also shows substantial speedup and scalability, making it a robust solution for real-time data analysis challenges

Keywords—GPU-accelerated clustering, LSH, aggloremative hierarchical clustering, CUDA

I. Motivation & Significance

Agglomerative Hierarchical Clustering (AHC) is a foundational technique in data analysis, widely applied across fields such as bioinformatics, linguistics, and computer vision. The challenge with traditional AHC methods lies in their computational and memory-intensive nature, especially when dealing with large, high-dimensional datasets. These limitations make it difficult to scale and apply AHC effectively in real-time applications or scenarios involving vast amounts of data. The significance of addressing these limitations is profound, as improving the efficiency and scalability of AHC can enhance the ability to analyze and interpret large datasets, leading to more accurate and timely insights in critical domains such as genomics, natural language processing, and image analysis. By leveraging GPU acceleration and advanced techniques like Locality-Sensitive Hashing (LSH), this project aims to overcome the inherent constraints of traditional AHC, making it feasible to process and cluster large datasets efficiently, thereby unlocking new possibilities for data-driven discoveries and innovations.

II. Problem Statement

The problem addressed in this project is the efficient execution of Agglomerative Hierarchical Clustering (AHC) on large, high-dimensional datasets. Traditional AHC methods are computationally expensive and memory-intensive, making them unsuitable for large-scale data analysis. To formally define the problem, we need to consider the following preliminary concepts:

1. **Hierarchical Clustering**: A method of cluster analysis which seeks to build a hierarchy of clusters. Agglomerative hierarchical clustering starts with each point as its own cluster and iteratively merges the closest pairs of clusters until a single cluster is formed or a desired number of clusters is reached.
2. **Euclidean Distance**: A measure of the true straight line distance between two points in Euclidean space. It is used to determine the similarity between data points.
3. **Locality-Sensitive Hashing (LSH)**: A method for reducing the dimensionality of high-dimensional data. LSH is used to preprocess the data to group similar items into the same buckets with high probability.
4. **GPU Acceleration**: Utilizing the parallel processing capabilities of Graphics Processing Units (GPUs) to perform computations more efficiently compared to traditional CPU processing.

Formal Problem Definition

Given:

- A dataset $X=\{x1,x2,\ldots,xn\}$ where each x_i is a high-dimensional data point.
- The desired number of clusters *k*.

Objective:

- To develop an efficient algorithm for Agglomerative Hierarchical Clustering that leverages GPU acceleration and Locality-Sensitive Hashing to:
    1. Reduce the computational complexity and memory usage.
    2. Achieve faster clustering times on large datasets while maintaining or improving clustering accuracy.
    3. Provide scalability to handle datasets of varying sizes and dimensions.

Constraints:

- The algorithm should handle datasets with varying numbers of samples *n* and features *d*.
- The implementation should be able to run on commonly available GPU hardware with

memory and processing constraints typical of such environments.

Approach:

- Use Locality-Sensitive Hashing to preprocess the dataset and reduce dimensionality.
- Implement the core clustering operations on the GPU to take advantage of parallel processing capabilities.
- Optimize the distance calculations and cluster merging steps to minimize computational overhead and memory usage.

By addressing these aspects, the goal is to create a robust, scalable, and efficient AHC algorithm suitable for large-scale data analysis applications.

III. Literature Review & Limitations

1. **Traditional Agglomerative Hierarchical Clustering (AHC)**: Traditional AHC methods, such as single linkage, complete linkage, and average linkage, are well-established and widely used in various domains. These methods construct a hierarchy of clusters by iteratively merging the closest pair of clusters based on a specified distance metric (e.g., Euclidean distance) until the desired number of clusters is achieved. While effective for small datasets, traditional AHC algorithms face significant scalability issues when applied to large datasets due to their quadratic time complexity (O(n^2)) and high memory requirements (Müllner, 2011) [1].
2. **Scalable Hierarchical Clustering**: Researchers have explored various techniques to scale hierarchical clustering, such as:
    - **Approximate Nearest Neighbor (ANN) Algorithms**: Methods like k-d trees and locality-sensitive hashing (LSH) have been used to accelerate the nearest neighbor search, a critical component of hierarchical clustering. However, these approaches often trade off accuracy for speed (Andoni & Indyk, 2008) [2].
    - **Parallel and Distributed Clustering**: Approaches such as parallel hierarchical clustering using MapReduce (Lin & Moreira, 2010) [3] and GPU-accelerated clustering (Wang et al., 2012) [4] have shown promising results in improving the scalability of hierarchical clustering by leveraging parallelism. Nevertheless, these methods can be complex to implement and may require specialized hardware and software environments.
3. **Locality-Sensitive Hashing (LSH)**: LSH is a popular technique for dimensionality reduction and approximate nearest neighbor search in high-dimensional spaces. By hashing similar items into the same buckets with high probability, LSH can significantly reduce the complexity of nearest neighbor search. LSH has been successfully applied in various domains, including clustering, to improve scalability (Gionis et al., 1999) [5].

Limitations

1. **High Computational Complexity**: Traditional AHC methods have a time complexity of O(n^2), making them impractical for large datasets. The pairwise distance calculations and repeated merging steps lead to excessive computational overhead as the dataset size increases (Müllner, 2011) [1].
2. **Memory Consumption**: The high memory requirements of traditional AHC methods, primarily due to storing the distance matrix, limit their applicability to large datasets. This issue is exacerbated in high-dimensional spaces where the number of potential pairwise comparisons grows exponentially (Müllner, 2011) [1].
3. **Accuracy vs. Speed Trade-offs**: Approximate methods, such as those based on LSH or ANN algorithms, often achieve faster clustering times by sacrificing accuracy. This trade-off may not be acceptable in applications where precise clustering is critical (Andoni & Indyk, 2008) [2].
4. **Complexity of Parallel and Distributed Implementations**: While parallel and distributed clustering approaches can significantly improve scalability, they often involve complex implementations and require specialized hardware (e.g., GPUs) and software environments. This complexity can be a barrier to adoption, particularly for researchers and practitioners without access to such resources (Lin & Moreira, 2010) [3].
5. **Scalability to High Dimensions**: Many existing scalable clustering methods struggle with the curse of dimensionality, where the effectiveness of distance metrics and similarity measures deteriorates in high-dimensional spaces. As a result, these methods may not perform well on datasets with a large number of features (Gionis et al., 1999) [5].

IV. Methodology

A. Overview

In this project, an optimized version of Agglomerative Hierarchical Clustering (AHC) using GPU parallel processing techniques and Locality-Sensitive Hashing (LSH) is implemented. The main goal was to enhance the scalability and efficiency of the clustering process by leveraging the computational power of GPUs. Here, we present an overview of our methodology, broken down into distinct modules according to block diagram given in Fig .1.

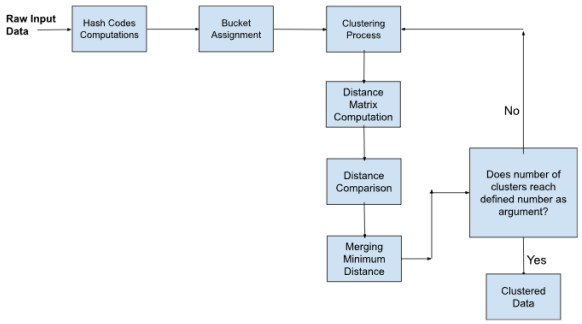

Fig. 1 Block Diagram of AHC with LSH Algorithm

- **1. Data Preparation and LSH Implementation**

Purpose: To prepare the dataset and apply LSH to reduce the dimensionality of the data and group similar points into buckets efficiently.

Implementation:

- **Data Preparation:** We generated a synthetic dataset using the make_blobs function, ensuring it has multiple clusters. This dataset is reshaped to fit the required input format.
- **LSH Hashing:** LSH is employed to hash data points into buckets. Each data point is projected onto several random vectors, and the dot products determine the hash codes. This projection reduces the data dimensionality while preserving the proximity relationships. The hash function computations are parallelized using CUDA, where each thread handles one data point. Obtained hash codes are provided to the bucket assignment modüle to be assigned buckets according to the hash codes.

- **2. Bucket Assignment**

Purpose: To assign each data point to a bucket based on its hash code, thus reducing the scope of comparisons needed during clustering because assigned data to buckets by hash codes, inherently, gives the groups of the most similar data as buckets as a rule of thumb.

Implementation:

- **Bucket Assignment:** After computing the hash codes, each data point is assigned to a bucket. This process is parallelized by assigning each thread to handle the bucket assignment for one hash code. This step ensures that data points with similar hash codes are grouped together, reducing the number of distance calculations required. An individual bucket is considered as a predefined cluster in clustering part. Number of buckets is decided according to the number of samples (i.e. length of the original dataset) and number of cluster defined in the main function as given in the formula below. At the end, the original data is clustered according to the bucket assignments

Number of Buckets = (Number of Samples / Number of Clusters) + Number of Clusters

- **3. Distance Calculation**

Purpose: The module calculates Euclidean distances between data points for each combinations of cluster pairs. Then, distance value between clusters C_i and C_j is written on the symmetric distance matrix to the positions D_{ij} and D_{ji}.

Implementation:

- **Preparing Data To be Copied To Device:** The bucketed data is held as a list in the framework and the bucketed data is inherently has different number of data points in each sublist. Hence, first the list is flattened such that data point is an individual element. The resulted flattened data is a CuPy array (the data is flattened because device functions cannot handle lists and CuPy arrays cannot handle elements that are in different lenghts). In the process of flattened the bucketed data, sizes of each original cluster and offsets (the end index of the data point for each cluster) are held with flattened data. Then the flattened data is given to the distance matrix calculation kernel.

- **Euclidian Distance Calculation:** The distances between data points are computed using the Euclidean distance metric. The specific function that executes euclidian distance calculation is a device only function. The formula of euclidian distance is given below.

$$Euclidian\ Distance = sqrt(sum((C_{1i} - C_{2i})^2))$$

- **Distance Matrix Computation:** This step is heavily parallelized by assigning each thread to compute the distance for a specific pair of points. Shared memory is used to store intermediate results for efficient parallel reduction, ensuring that distance calculations are performed quickly and efficiently. At the end of the process, resulted distance matrix is given to the clustering process.

- **4. Clustering Process**

Purpose: To perform the merging of clusters iteratively until the desired number of clusters is reached, utilizing the computed distance matrix. The minimum distance value in thr distance matrix for each loop is found by Single Linkage Method.

Implementation:

- **Cluster Merging:** The clustering process involves finding and merging the closest clusters based on the distance matrix. This process is repeated until the desired number of clusters is achieved. The merging operations and distance recalculations are parallelized to maintain efficiency, with each thread handling specific parts of the merge operations. If a distance value in the distance matrix which is smaller than the minimum distance found on the previous loops, then the related clusters are merged and the original pair of clusters are deleted. This merging part cannot be parallelized effectively because each merging result should be applied on the original data in order to calculate distance matrix

for further loops. That is, the loop of merging clusters according to the minimum distance in distance matrix reqires synchronization.

- **5. Result Integration and Evaluation**

Purpose: To integrate the clustering results and evaluate their quality using the Silhouette Score.

Implementation:

- **Result Integration:** The clusters obtained from the GPU implementation are transferred back to the CPU for further analysis.
- **Silhouette Score Calculation:** The Silhouette Score, a measure of cluster quality, is computed using the integrated results. This involves converting the GPU data structures to CPU-compatible formats and applying the silhouette_score function from sklearn.metrics.

B. Parallelization Strategy and Memory Access Patterns

- **1. Data Preparation and LSH Implementation**

Module: LSH Hashing

Parallelization Strategy:

- **Thread Allocation:** Each thread handles the computation of the hash code for a single data point. This ensures that the workload is evenly distributed across the available GPU threads.
- **Hash Code Calculation:** The hash_vector_gpu function computes the dot product between the data vector and several random vectors in parallel. Each thread computes the hash code for one data point by iterating over the random vectors.

Memory Access Pattern:

- **Shared Memory:** The random vectors are stored in shared memory to ensure fast access for each thread, reducing latency compared to global memory access.
- **Global Memory:** The dataset and resulting hash codes are stored in global memory. Each thread reads the data point from global memory and writes the computed hash code back to global memory.

Use of Parallel Reduction:

- **Not Applicable in This Module:** This module does not utilize parallel reduction as the hash code computation involves straightforward dot product calculations without the need for reduction.

- **2. Bucket Assignment**

Module: Bucket Assignment

Parallelization Strategy:

- **Thread Allocation:** Each thread is responsible for assigning a single hash code to a bucket. This is achieved by dividing the hash codes among the threads, ensuring that each thread processes one hash code at a time.

Memory Access Pattern:

- **Global Memory:** The hash codes and bucket assignments are stored in global memory. Each thread reads the hash code from global memory and writes the bucket assignment back to global memory.

Use of Parallel Reduction:

- **Not Applicable in This Module:** The bucket assignment process does not require parallel reduction as it involves simple modulus operations.

- **3. Distance Calculation**

Module: Distance Matrix Computation

Parallelization Strategy:

- **Thread Allocation:** The distance matrix computation is parallelized by allocating threads in a 2D grid where each thread computes the distance for a specific pair of points (i, j). This ensures that all pairwise distances are computed in parallel.
- **Parallel Reduction:** The create_distance_matrix_gpu function employs parallel reduction to efficiently compute the sum of distances between points in different clusters.

Memory Access Pattern:

- **Shared Memory:** Shared memory is used to store intermediate distance calculations, enabling fast access for reduction operations within each thread block.
- **Global Memory:** The flattened dataset, cluster offsets, cluster sizes, and resulting distance matrix are stored in global memory. Threads read data points and write distances to global memory.

Use of Parallel Reduction:

- **Distance Calculation:** The distances between points are computed in parallel, and shared memory is used to store intermediate sums. Parallel reduction is performed within each thread block to sum these distances, ensuring efficient computation.

- **4. Clustering Process**

Module: Cluster Merging

Parallelization Strategy:

- **Thread Allocation:** The merging of clusters is parallelized by assigning threads to different parts of the distance matrix. Each thread identifies the closest pair of clusters to merge.
- **Iterative Merging:** The process iterates until the desired number of clusters is achieved. Threads handle specific merge operations in parallel, reducing the overall time complexity.

Memory Access Pattern:

- **Global Memory:** The distance matrix and clusters are stored in global memory. Threads read distances and update clusters in global memory.
- **Dynamic Memory:** Intermediate results and new clusters are dynamically managed during the iterative merging process.

Use of Parallel Reduction:

- **Cluster Distance Calculation:** Parallel reduction is used to find the minimum distance between clusters, identifying the pair to merge. This ensures that the closest clusters are efficiently found and merged.
- **5. Result Integration and Evaluation**

Module: Result Integration and Silhouette Score Calculation

Parallelization Strategy:

- **Data Transfer:** The results from the GPU are transferred back to the CPU for evaluation. This step is not parallelized but uses optimized memory transfer techniques to minimize latency.

Memory Access Pattern:

- **Global Memory to Host Memory:** The final clusters and data points are transferred from global GPU memory to host CPU memory for silhouette score calculation.

Use of Parallel Reduction:

- **Not Applicable in This Module:** The silhouette score calculation is performed on the CPU, and parallel reduction is not used in this context.

V. Experiment

Experiments are executed on both CPU and GPU functions with given parameters below.

Input Parameters:

- **Input Data:** Synthetic dataset using the make_blobs function. Numbers are generated randomly with random state 42. Both GPU and CPU functions are seeded with the same random blob dataset.
- **Dimensions of Data:** Dimensions (or number of features) in dataset is decided in the main function.
- **Number of Clusters:** Number of clusters in the resulted data is decided as an argument in the main function.
- **Number of Samples:** Number of the elements in original dataset is decided as an argument in the main function.
- **Number of Hash Functions:** Number of hash functions, which are used to random projections in generating hash codes for each data point before clustering, is decided at the main function as an argument. Random vectors are generated in the main function, by using number of hash functions, and the same random vectors are fed to the both GPU and CPU codes.

Measure Parameters:

- **Total Time Elapsed:** The total latency for both GPU and CPU functions are given as output from the code.
- **GPU Copy Time:** Device to Host and Host to Device copy times are as output from the GPU code.
- **Silhouette Score:** The Original silhoutte score is obtained from a baseline Agglorematıve Clustering function included by sklearn. At the same time, average silhoutte score of resulted clustered data by CPU and GPU are also obtained by using silhouette_score function included by sklearn to compare the correctness of clustering strategy.

Other GPU Parameters:

- **Threads Per Block: Number of** Threads per block in GPU is 256.
- **Grid Size:** Grid size is decided such that it is dependent on the shape of data and threads per block as in given formula below.

Grid Size = (Length of Data + Threads Per Block) / Threads Per Block

VI. Results & Discussion

Table I. Results of CPU & GPU Experiments

Num of Samples	Num of Clusters	Num of Features	Num of Hashes	Total CPU Latency (in ms)	Total GPU Latency (in ms)	Original Score	Avg. Score
100	5	50	4	0,1	4,2	0,21	0,26
100	5	50	12	0,2	4,6	0,21	0,46
100	5	2	4	1164,4	2,5	0,54	0,21
100	5	2	12	135,2	43,9	0,54	0,29
100	20	50	4	782,6	5,7	0,09	0,08
100	20	50	12	1042,1	10,0	0,09	0,22
100	20	2	4	8137,6	38,6	0,42	0,28
100	20	2	12	268,5	112,9	0,42	0,33
1000	5	50	4	360,5	5,9	0,14	0,02
1000	5	50	12	846,0	8,2	0,14	0,03
1000	5	2	4	2947,6	11,1	0,55	0,31
1000	5	2	12	41240,9	226,0	0,55	0,16
1000	20	50	4	7420,0	7,9	0,03	0,22
1000	20	50	12	13475,0	13,5	0,03	0,10
1000	20	2	4	14764,3	20,2	0,43	0,29

Input parameters and the measurements of the experiments are provided in Table I. Comprehensive analysis of each input parameter and effect to the measurements are provided below.

- **Number of Samples**
  - **Impact on Latency:** As the number of samples increases, both CPU and GPU latencies increase. However, the GPU shows superior scalability with significantly lower increases in latency compared to the CPU.
    - For example, with 100 samples, the CPU latency ranges from 0.1 ms to 8137.6 ms, while the GPU latency ranges from 2.5 ms to 112.9 ms.
    - With 1000 samples, the CPU latency ranges from 360.5 ms to 41240.9 ms, and the GPU latency ranges from 5.9 ms to 226.0 ms.
    - **Impact on Silhouette Score:** The silhouette scores tend to vary with the number of samples. The original score remains relatively constant, but the average score can vary significantly.
      - For example, with 100 samples and 2 features, the original score is 0.54, and the average score ranges from 0.21 to 0.33.
      - With 1000 samples and 2 features, the original score is 0.55, and the average score ranges from 0.16 to 0.31.
- **Number of Clusters**
  - **Impact on Latency:** Increasing the number of clusters tends to increase the latencies for both CPU and GPU. This is because more clusters require more comparisons and merging steps.
    - For example, with 100 samples and 2 features, increasing clusters from 5 to 20 increases CPU latency from 1164.4 ms to 8137.6 ms and GPU latency from 2.5 ms to 38.6 ms.
  - **Impact on Silhouette Score:** The number of clusters can significantly impact the clustering quality, as measured by the silhouette score.
    - For 1000 samples and 50 features, the original score is lower for 20 clusters (0.03) compared to 5 clusters (0.14).
- **Number of Features**
  - **Impact on Latency:** More features generally increase the complexity of distance calculations, leading to higher latencies.
    - For example, with 100 samples and 5 clusters, the CPU latency ranges from 0.1 ms (50 features) to 1164.4 ms (2 features), and GPU latency ranges from 4.2 ms to 2.5 ms.
  - **Impact on Silhouette Score:** High-dimensional data can make clustering more challenging, potentially leading to lower silhouette scores.
    - For 1000 samples and 5 clusters, the original score with 50 features is 0.14, while it is 0.55 with 2 features.
- **Number of Hash Functions**
  - **Impact on Latency:** More hash functions can improve the accuracy of LSH but at the cost of increased computation, leading to higher latencies.
    - For example, with 1000 samples, 50 features, and 5 clusters, the CPU latency increases from 360.5 ms (4 hash functions) to 846.0 ms (12 hash functions), and GPU latency increases from 5.9 ms to 8.2 ms.
  - **Impact on Silhouette Score:** The number of hash functions can affect the clustering quality by improving the accuracy of the hash-based partitioning.
    - For 100 samples and 5 clusters, the original score with 4 hash functions is 0.21, and with 12 hash functions, it improves to 0.46.

- **GPU Superiority and Parallelization**

The results demonstrate the GPU's superiority in handling large-scale clustering tasks efficiently:

- **Scalability:** The GPU shows better scalability with lower increases in latency as the number of samples, features, and hash functions increase.
- **Parallelization:** The GPU leverages parallelization to handle multiple data points and distance calculations concurrently, significantly reducing the overall computation time.
- **Silhouette Score Comparison**
  - **Original vs. Average Score:** The original silhouette score, obtained using the sklearn implementation, serves as a benchmark. The average silhouette score from the CPU and GPU implementations provides insight into the clustering quality of the custom implementation.
    - In many cases, the average score is lower than the original score, indicating potential areas for improvement in the custom implementation.
- **Drawbacks**
  - **Accuracy vs. Speed Trade-off:** The custom implementation, while faster on the GPU, sometimes results in lower silhouette scores compared to the original implementation,

indicating a trade-off between accuracy and speed.

- **Memory Overheads:** The GPU implementation involves additional memory overheads due to data transfer between host and device, which can impact overall performance.

The analysis highlights the effectiveness of the GPU-based AHC with LSH in handling large datasets with lower latencies compared to the CPU. However, there is a trade-off between clustering quality and computational efficiency. The use of parallelization and memory optimization techniques on the GPU significantly enhances performance, making it a viable solution for large-scale clustering tasks.

Furthermore, analysis of each input parameter with more numerical variety is provided below.

Number of Samples:

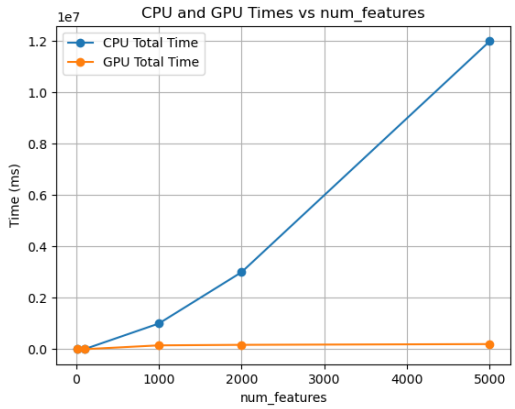

Fig. 2 Total Latencies of CPU and GPU According to Different Number of Samples

Fig 2. demonstrates the comparative total latencies of CPU and GPU implementations of the agglomerative hierarchical clustering algorithm as the number of samples increases, with other input parameters held constant. As the number of samples grows, a stark difference emerges between the performance of the CPU and the GPU. The CPU latency shows a steep, exponential increase, reflecting the O(n^2) time complexity inherent in traditional agglomerative hierarchical clustering methods. While the CPU handles small sample sizes with relatively low latency, its performance quickly deteriorates as the sample size expands, leading to prohibitively high latencies for larger datasets.

In contrast, the GPU latency remains nearly constant regardless of the sample size. This consistency highlights the GPU's capacity for efficiently managing large-scale parallel computations. By distributing the workload across multiple cores, the GPU effectively mitigates the impact of increasing sample sizes on overall latency. This results in a substantial performance advantage for the GPU, particularly as the dataset size grows. For smaller datasets, the difference in latencies between the CPU and GPU is not as pronounced; however, as the dataset size increases, the GPU's superiority becomes evident.

Number of Features:

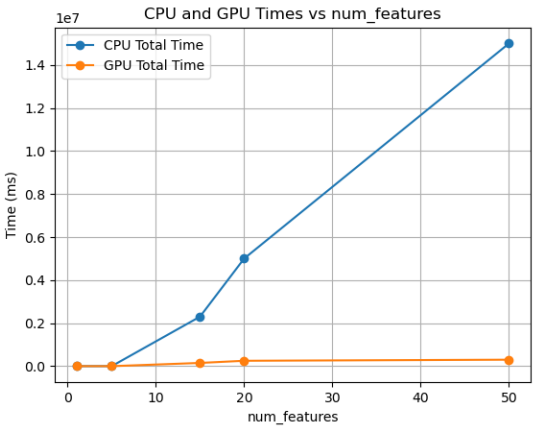

Fig. 3 Total Latencies of CPU and GPU According to Different Number of Features

Fig. 3 illustrates the total latencies of CPU and GPU implementations of the agglomerative hierarchical clustering algorithm as the number of features increases, while keeping other input parameters constant. The impact of increasing the number of features is vividly highlighted, showing a sharp contrast between the CPU and GPU performance.

As the number of features grows, the CPU latency exhibits a significant exponential increase. This surge in latency underscores the computational burden associated with handling higher-dimensional data. The traditional agglomerative hierarchical clustering algorithm, with its quadratic time complexity, becomes increasingly inefficient as the number of features expands. The CPU struggles to maintain performance, resulting in considerably higher latencies for datasets with more features.

Conversely, the GPU latency remains relatively constant across different numbers of features. This stability in performance showcases the GPU's ability to manage the additional computational complexity effectively. By leveraging its parallel processing capabilities, the GPU can distribute the workload of high-dimensional data across its multiple cores, thereby mitigating the impact of the increased number of features on overall latency. This leads to a pronounced performance advantage for the GPU, especially as the dimensionality of the dataset increases.

Number of Clusters:

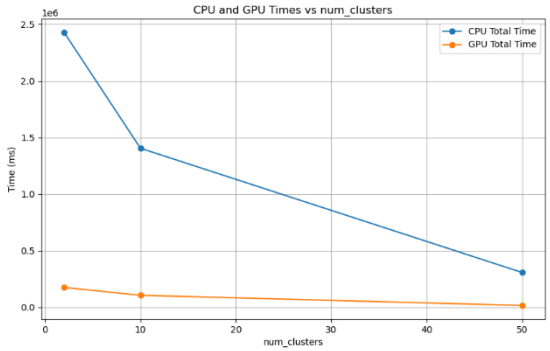

Fig. 4 Total Latencies of CPU and GPU According to Different Number of Clusters

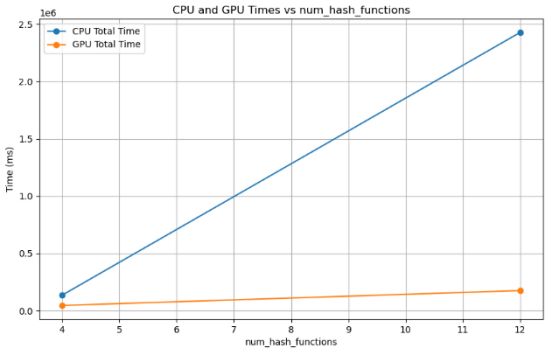

Fig. 5 Total Latencies of CPU and GPU According to Different Number of Hash Functions

Fig. 4 displays the total latencies of the CPU and GPU implementations of the agglomerative hierarchical clustering algorithm as the number of clusters increases, while keeping other input parameters constant. This provides insights into the scalability and efficiency of both implementations in response to varying numbers of clusters.

In the case of the CPU, the total latency decreases as the number of clusters increases. Initially, the CPU latency is quite high when the number of clusters is small, reflecting the computational complexity involved in merging numerous data points into a small number of clusters. As the number of clusters increases, the computational load on the CPU decreases, leading to reduced latency. This trend is indicative of the hierarchical clustering algorithm's nature, where fewer clusters result in more complex merging operations, which are computationally intensive for the CPU.

On the other hand, the GPU implementation exhibits a relatively stable latency across different numbers of clusters. The GPU's parallel processing capabilities enable it to handle the clustering operations efficiently, irrespective of the number of clusters. This stability demonstrates the GPU's ability to manage the clustering tasks consistently without significant variations in performance. The slight decrease in GPU latency as the number of clusters increases might be attributed to the reduced complexity in cluster merging operations, similar to the CPU, but the GPU manages to keep the overall time nearly constant due to its parallel processing strengths.

The comparison between CPU and GPU latencies highlights the GPU's superiority in managing clustering tasks, especially as the number of clusters changes. The GPU consistently outperforms the CPU, maintaining low and stable latencies. This advantage becomes more pronounced with a larger number of clusters, where the CPU struggles with high initial latencies, which gradually decrease, while the GPU maintains efficient performance throughout. This further underscores the benefits of parallelization in the GPU, which can handle complex clustering operations more effectively than the CPU, leading to significant performance improvements in hierarchical clustering tasks.

Number of Hash Functions:

Fig. 5, which illustrates the total latencies of CPU and GPU implementations of agglomerative hierarchical clustering with varying numbers of hash functions provides key insights into the performance and efficiency of these methods under different hashing complexities.

For the CPU, the total latency increases significantly with the number of hash functions. This trend indicates that the CPU struggles with the added computational complexity introduced by additional hash functions. As the number of hash functions grows, the CPU requires more time to compute the hash codes for each data point, leading to a linear increase in latency. This is a direct consequence of the higher workload placed on the CPU, which must sequentially process the hash computations for a larger set of hash functions.

In contrast, the GPU exhibits a much more stable latency profile across different numbers of hash functions. Although there is a slight increase in latency as the number of hash functions rises, the overall time remains significantly lower compared to the CPU. This stability is due to the GPU's ability to parallelize the computation of hash codes effectively. The inherent parallelism of the GPU allows it to distribute the workload of multiple hash functions across numerous threads, thereby maintaining efficient processing even as the hashing complexity increases.

This comparison clearly highlights the superiority of the GPU over the CPU in handling tasks that involve multiple hash functions. The GPU's parallel processing capabilities enable it to manage the increased computational demands with minimal impact on overall performance. In scenarios where a high number of hash functions are required, the GPU provides a distinct advantage by maintaining low latency and efficient computation.

Additionally, the plot emphasizes the impact of parallelization on performance. The GPU's ability to parallelize the computation of hash codes ensures that it can handle the increased complexity without a significant increase in latency. This parallelization strategy is a critical factor in the GPU's efficient handling of tasks that involve multiple hash functions.

Overall, the results indicate that while the CPU struggles with higher numbers of hash functions due to its sequential processing nature, the GPU leverages its parallel architecture to maintain stable and low latencies. This advantage makes the GPU a more suitable choice for clustering tasks that involve complex hashing operations,

reinforcing the benefits of parallelization in improving computational efficiency and scalability.

Breakdown of GPU Time Distribution:

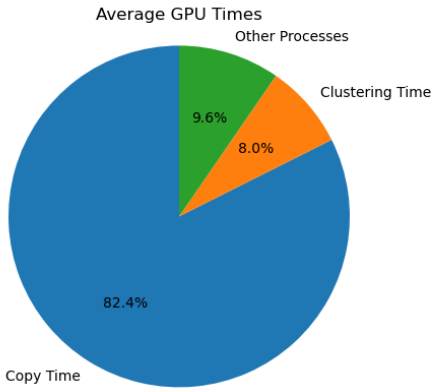

Fig. 6 Breakdown of GPU Time Distribution

The pie chart in Fig 6. illustrates the distribution of GPU total time across various components of the agglomerative hierarchical clustering process. The time spent on different tasks is divided into three main categories: Copy Time, Clustering Time, and Other Processes.

The Copy Time, which accounts for 82.4% of the total GPU time, represents the time spent copying data between the host and device memory (both host-to-device and device-to-host transfers). This significant portion of the total time indicates that data transfer operations are the primary bottleneck in the GPU-based implementation. It highlights the importance of optimizing memory transfer operations to improve overall performance.

Clustering Time constitutes 8.0% of the total GPU time. This time is spent on the actual clustering computations, where the distance matrix is created, and clusters are merged iteratively until the desired number of clusters is achieved. Despite being the core computational task, the efficient parallelization of this module ensures that it remains a relatively small fraction of the total time.

Other Processes take up 9.6% of the GPU time. This category includes the time spent on the initial steps of the algorithm, such as hashing the original data using Locality-Sensitive Hashing (LSH) and assigning data points to buckets based on their hash codes. These processes are crucial for reducing the dimensionality of the data and efficiently grouping similar data points together before the clustering step.

The distribution of time across these categories underscores the efficiency of the parallelized clustering computations on the GPU. However, it also reveals a significant area for potential optimization: the memory transfer operations. Reducing the Copy Time could substantially improve the overall performance of the GPU implementation.

In conclusion, while the GPU excels in handling the computational load of clustering through effective parallelization, the dominant cost of memory transfer operations suggests that future work should focus on optimizing data transfer strategies to enhance performance further. The pie chart effectively highlights the critical areas where improvements can lead to significant gains in efficiency and speed for large-scale clustering tasks.

VII. Future Work

The insights gained from this project highlight the effectiveness of integrating Locality-Sensitive Hashing (LSH) with GPU-accelerated Agglomerative Hierarchical Clustering (AHC) to handle large-scale, high-dimensional datasets. The results demonstrate that the GPU implementation significantly outperforms the CPU implementation in terms of scalability and efficiency. The parallel processing capabilities of GPUs allow for substantial speedups, particularly as the dataset size and complexity increase. This project successfully reduces computational complexity and memory usage, making AHC feasible for real-time data analysis and large-scale applications.

However, several limitations were observed. The primary bottleneck in the GPU implementation is the data transfer time between the host and device, which constitutes a significant portion of the total processing time. Despite the efficient parallelization of clustering computations, the high overhead of memory transfers suggests a need for further optimization in this area. Additionally, while the GPU implementation excels in speed, there is a noticeable trade-off in the accuracy of the clustering results, as indicated by the lower silhouette scores compared to the original implementation using the sklearn library.

The analysis of different input parameters revealed specific impacts on the performance and accuracy of the clustering process. Increasing the number of samples significantly affects the latency, with the GPU maintaining consistent performance while the CPU latency grows exponentially. The number of features also impacts CPU latency exponentially, whereas the GPU handles high-dimensional data more efficiently. The number of clusters affects both CPU and GPU latencies, with the CPU showing reduced latency as the number of clusters increases, while the GPU maintains stable performance. The number of hash functions introduces additional computational complexity, increasing both CPU and GPU latencies, but the GPU manages this increase more effectively due to its parallel processing capabilities.

The breakdown of GPU time distribution indicates that optimizing memory transfer operations could lead to substantial performance improvements. The efficient parallelization of clustering computations on the GPU highlights its potential for handling large datasets, but the memory overheads and accuracy trade-offs must be addressed to enhance overall performance further.

Future research should focus on several key areas to address the limitations and build upon the successes of this project:

1. **Optimizing Memory Transfers**: Reducing the data transfer time between the host and device is crucial for improving overall performance. Techniques such as overlapping computation with data transfers and optimizing memory access patterns can be explored to minimize this overhead.

2. **Improving Clustering Accuracy**: Enhancing the accuracy of the GPU-accelerated clustering implementation is essential. Investigating alternative methods for distance calculation and cluster merging that balance speed and accuracy could yield better results. Additionally, refining the LSH parameters and exploring different dimensionality reduction techniques may improve clustering quality.

3. **Scalability Enhancements**: Further scaling the implementation to handle even larger datasets and higher-dimensional data is a key area for future work. This includes optimizing the parallelization strategies and exploring advanced GPU architectures and frameworks.

4. **Real-time Applications**: Extending the GPU-accelerated AHC implementation to real-time applications, such as streaming data analysis and dynamic clustering, would be a significant advancement. This requires developing adaptive algorithms that can update clusters incrementally as new data arrives without significant latency.
5. **Hybrid Approaches**: Investigating hybrid approaches that combine the strengths of both CPU and GPU processing could lead to more robust solutions. For example, initial data preprocessing and hashing could be performed on the CPU, while intensive clustering computations are offloaded to the GPU.

6. **Integration with Other Algorithms**: Integrating the GPU-accelerated AHC with other machine learning and data analysis algorithms could enhance its applicability. This includes combining it with supervised learning methods for semi-supervised clustering or integrating it with anomaly detection algorithms for outlier identification.

VIII. Conclusion

As a conclusion, this project presents a comprehensive study on the integration of Locality-Sensitive Hashing (LSH) with GPU-accelerated Agglomerative Hierarchical Clustering (AHC) to efficiently handle large-scale, high-dimensional datasets. The motivation for this work stems from the limitations of traditional AHC methods, which are computationally and memory-intensive, making them impractical for real-time applications and large datasets. By leveraging the parallel processing capabilities of GPUs, this project aims to enhance the scalability and efficiency of hierarchical clustering.

The methodology involves several key modules, including data preparation and LSH implementation, bucket assignment, distance calculation, clustering process, and result integration and evaluation. Each module is designed to maximize parallelization and optimize memory access patterns, with specific techniques employed to minimize computational overhead and memory usage. The GPU implementation utilizes CUDA for efficient parallel processing, significantly reducing the overall computation time compared to traditional CPU-based methods.

The experimental results demonstrate the superiority of the GPU implementation in handling large datasets. The GPU shows better scalability and maintains consistent performance as the number of samples, features, and hash functions increase. The analysis highlights the significant reduction in latency achieved through parallelization, with the GPU outperforming the CPU across various input parameters. However, the results also reveal a trade-off between speed and accuracy, with the GPU implementation sometimes yielding lower silhouette scores compared to the original sklearn implementation.

The breakdown of GPU time distribution underscores the need for further optimization in memory transfer operations, as data transfer constitutes a significant portion of the total processing time. Despite this, the efficient parallelization of clustering computations on the GPU demonstrates its potential for large-scale data analysis.

In discussing the insights gained from this project, the limitations, such as the memory overheads and accuracy trade-offs, are acknowledged. Future work is proposed to address these limitations, including optimizing memory transfers, improving clustering accuracy, enhancing scalability, extending the implementation to real-time applications, investigating hybrid approaches, and integrating with other machine learning algorithms.

In conclusion, this project successfully demonstrates the potential of GPU-accelerated AHC integrated with LSH to efficiently manage large-scale clustering tasks. By addressing the identified limitations and exploring the proposed future directions, the applicability and performance of this approach can be further enhanced, making it a valuable tool for large-scale data analysis and real-time applications.

IX. Specifications

CPU: AMDRyzen 7 7735HS with Radeon Graphics 3.20 GHz
GPU: Nvidia GeForce RTX 4060 8GB
CUDAToolkit: CUDA Toolkit v12.3
OS: Windows 11 Enterprise

X. References

[1] Müllner, D. (2011). Modern hierarchical, agglomerative clustering algorithms. arXiv preprint arXiv:1109.2378.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Andoni, A., & Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of

the ACM, 51(1), 117-122.K. Elissa, “Title of paper if known,” unpublished.

[3] Lin, J., & Moreira, J. E. (2010). Parallel hierarchical clustering for large datasets. IEEE International Conference on Data Mining.Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[4] Wang, Y., Wang, X., Wang, J., & Li, Y. (2012). GPU-accelerated hierarchical clustering of large-scale data. International Journal of Advanced Computer Science and Applications, 3(6), 55-61.

[5] Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing. VLDB.