A GPU-Based Tabu Search Algorithm for Large Instances of the Generalized Max-Mean Dispersion Problem

William Rosendo[1], **Ermeson Andrade**[2], **Bruno Nogueira**[1]

[1] Instituto de Computação – Universidade Federal de Alagoas (UFAL)
57072-900 – Maceió – AL – Brazil

[2]Departamento de Computação – Universidade Federal Rural de Pernambuco (UFRPE)
52171-900 – Recife – PE – Brazil

wgpr@ic.ufal.br, ermeson.andrade@ufrpe.br, bruno@ic.ufal.br

Abstract. *The Generalized Max-Mean Dispersion Problem (GMaxMeanDP) is important in areas such as web page classification, community mining, and social network analysis. However, current methods for this problem face challenges with large-scale instances, primarily due to memory inefficiencies in their matrix-based representations and the escalating computational time required to execute these methods. This study proposes a parallel tabu search algorithm using GPUs to accelerate local search and optimized data structures to reduce memory overhead. Tests on instances with 2,000 to 20,000 vertices show that the algorithm outperforms existing methods in the literature in terms of solution quality and execution time, making it an effective and scalable solution for large-scale scenarios.*

Resumo. *O Problema de Dispersão Máxima Média Ponderada (GMaxMeanDP) é importante em áreas como classificação de páginas web, mineração de comunidades e análise de redes sociais. No entanto, os métodos atuais para esse problema enfrentam desafios em instâncias de grande escala, principalmente devido a ineficiências de memória em suas representações matriciais e ao crescente tempo computacional necessário para executá-los. Este estudo propõe um algoritmo de busca tabu paralelizado, utilizando GPUs para acelerar a busca local e estruturas de dados otimizadas para reduzir a sobrecarga de memória. Testes em instâncias de 2.000 a 20.000 vértices mostram que o algoritmo supera os métodos existentes na literatura em termos de qualidade das soluções e tempo de execução, sendo uma solução eficaz e escalável para cenários massivos.*

1. Introduction

Given a complete weighted graph $G = (V, E, D, W)$, where V is the set of n vertices, E is the set of edges, D is the set of edge weights d_{ij} ($i \neq j$), which can be positive, negative or zero, and W represents the vertex weights w_i ($i = 1, 2, \ldots, n$), the Weighted Generalized Max-Mean Dispersion Problem (GMaxMeanDP) consists of selecting a subset $M \subseteq V$ such that the weighted mean dispersion of the subgraph induced by M is maximized [Lai et al. 2020]. In other words, one must choose the subset of vertices that maximizes the diversity of connected edges between them, taking into account both edge weights and vertex weights.

Formally, GMaxMeanDP can be formulated as an unconstrained fractional 0-1 combinatorial optimization problem, where binary variables x_i indicate whether element i is selected or not [Prokopyev et al. 2009]:

$$\text{Maximize } f(s) = \frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j}{\sum_{i=1}^{n} w_i x_i} \tag{1}$$

$$x_i \in \{0, 1\}, \quad i = 1, 2, \ldots, n$$

This problem belongs to the broader class of dispersion problems, which aim to select a subset of elements that maximizes diversity among them. Previous studies show that implementing metaheuristics on GPUs has proven effective for solving problems in large-scale scenarios [Nogueira et al. 2024, Essaid et al. 2019]. Considering that GMaxMeanDP is an $\mathcal{NP}$-hard problem, optimization techniques that exploit parallelism emerge as a promising strategy for finding efficient solutions.

GMaxMeanDP has many relevant applications. In web page ranking, it is used to classify pages to maximize the diversity of results presented to users [Kerchove and Dooren 2008]. In community mining, it can help detect diverse groups in social networks, enabling the identification of patterns and communities with different interests [Yang et al. 2007]. In trust network construction, it can be applied to select nodes or connections that maximize attribute dispersion, increasing network robustness and reliability [Carrasco et al. 2015]. A classic example would be planning the expansion of company branches in different cities. In this case, besides selecting cities that are distant from each other, it is also important to consider each city's weight, representing its economic importance or population size. GMaxMeanDP does not require a fixed number of cities, rather, this number can vary dynamically during the search for the best combination that generates maximum diversity.

Several heuristics have been proposed for GMaxMeanDP, including hybrid optimization approaches combining skewed variable neighborhood search with particle swarm optimization [Zhao et al. 2024], evolutionary algorithms integrating tabu search with genetic algorithms [Lai et al. 2020], and tabu search with perturbation algorithms [Lai et al. 2020]. However, these methods struggle with large-scale instances due to memory inefficiencies in their matrix-based representations, which restrict their applicability, and the escalating computational time required as instance size increases.

This work presents a GPU-based tabu search algorithm designed to handle large-scale instances of GMaxMeanDP, overcoming limitations faced by state-of-the-art heuristics when dealing with large problems. The main contributions of this work include:

- Proposal of a tabu search algorithm that exploits GPU parallel processing power to accelerate neighborhood exploration. To the best of our knowledge, this is the first known study exploring GPU usage for solving GMaxMeanDP.
- The proposed GPU-based local search was designed to use more compact and efficient data structures, replacing traditional matrix representations.
- Experimental results comparing the proposed algorithm with state-of-the-art heuristics highlight its acceleration capability and efficiency in solving large-scale instances.

The remainder of this paper is structured as follows: Section 2 presents a detailed discussion about existing sequential implementations of tabu search, highlighting their limitations, especially when dealing with large-scale problems. Section 3 introduces the concept of parallel tabu search on GPUs, explaining how it addresses the challenges described in Section 2. The evaluation of the proposed method is presented in Section 4, and Section 5 concludes the paper with final considerations.

2. Sequential Tabu Search

The GPU-based algorithm proposed in this work is based on the traditional tabu search procedure, as described in Algorithm 1, and is similar to the one presented in [Lai et al. 2020]. Tabu search is a local search metaheuristic that uses memory-based forbidden moves to escape local optima [Glover and Laguna 1997]. For the GMaxMeanDP problem, the size of the subgraph can vary from 2 to n, where n is the size of the graph.

Input: Initial solution s_0, neighborhood structure $N(s)$, evaluation function $Mean(s)$, maximum number of iterations $Iter_{max}$
Output: Best solution found s_b
`TabuSearch` $(s_0, N(s), f, Iter_{max})$
    $s \leftarrow s_0$ `// Initialize current solution`
    $s_b \leftarrow s$ `// Initialize best solution`
    $iter \leftarrow 0$ `// Initialize iteration counter`
    repeat
    Randomly select the best eligible neighboring solution $s' \in N(s)$, where s' is considered eligible if it is not forbidden by the tabu list or if it is better than s_b
    $s \leftarrow s'$
    Update the tabu list
    if $Mean(s) > Mean(s_b)$ **then**
        $s_b \leftarrow s$ `// Update best solution if an improvement is found`
    endif
    $iter \leftarrow iter + 1$
    until $iter = Iter_{max}$
    return s_b `// Return the best solution found`

Algorithm 1: Pseudocode of the sequential tabu search.

Starting with an initial solution s_0, Algorithm 1 iteratively improves the quality of this solution using insertion, removal, or swap moves. Since the solution size is variable, the tabu search accepts moves involving the insertion or removal of 1 or 2 vertices. Until the stopping criterion is met, at each iteration, the best eligible move is determined and applied to s. To prevent the search from cycling through already visited solutions, Algorithm 1 employs a short-term memory known as the tabu list, which prohibits certain moves. The algorithm returns as output the best solution s_b found during the search.

Neighborhood Structures. Given an initial solution s, belonging to the solution space S, it is possible to perform a transition from this solution s to a neighboring solution s'. This transition, known as a move, is executed with the goal of exploring the solution space and finding the best solution. The neighborhood of a solution can be defined by a set of moves and consists of a mapping that assigns to each solution $s \in S$ a set of neighboring solutions $N(s)$. The following neighborhood structures are investigated here:

1. **1-flip Neighborhood.**
   The 1-flip neighborhood (denoted by N_1) is defined by changing the value of a single variable x_i to its complement $1 - x_i$, i.e., it is the neighborhood responsible for performing moves that insert or remove a single vertex. The size of this neighborhood is n, where n is the number of elements.
2. **2-flip Neighborhood.**
   The 2-flip neighborhood (denoted by N_2) simultaneously changes the values of two variables x_i and x_j to their complementary values, i.e., it is responsible for performing moves that insert or remove 2 vertices or perform swap moves. The size of the N_2 neighborhood is given by $\frac{n(n-1)}{2}$, considering all possible combinations of pairs of variables.

To quickly compute the move gain of possible changes to be applied to the solution s through the neighborhood operators N_1 and N_2, data structures such as the *gain* vector were used [Zhou et al. 2017, Lai et al. 2020]. The *gain* vector is defined as the sum of the edge weights between each element of the graph and the elements contained in the current solution s. The gain vector is initialized as follows:

$$gain(v_i) = \sum_{v_j \in s} d(v_i, v_j) \tag{2}$$

where $d(v_i, v_j)$ represents the weight of the edge between vertices v_i and v_j.

The value of each move depends on the type of operation applied (1-flip or 2-flip). In the case of a 1-flip move, which modifies only a single vertex v_i, its value can be determined in $O(1)$ by the following equations:

$$\Delta_i = \begin{cases} \frac{-f(s)w_i + gain(v_i)}{S_M + w_i}, & \text{if } v_i \notin s; \\ \frac{f(s)w_i - gain(v_i)}{S_M - w_i}, & \text{if } v_i \in s; \end{cases} \tag{3}$$

where $f(s)$ is the value of the current solution s, and S_M is the sum of the weights of the vertices selected in s, i.e., $S_M = \sum_{i \in s} w_i$. For a 2-flip move, which involves two vertices x_i and x_j, its value can also be calculated in $O(1)$ as follows:

$$\Delta_{ij} = \begin{cases} \frac{-f(s)(w_i+w_j) + gain(v_i) + gain(v_j) + d_{ij}}{S_M + w_i + w_j}, & \text{if } v_i \notin s, v_j \notin s; \\ \frac{f(s)(w_i+w_j) - gain(v_i) - gain(v_j) + d_{ij}}{S_M - w_i - w_j}, & \text{if } v_i \in s, v_j \in s; \\ \frac{f(s)(w_i-w_j) + gain(v_j) - gain(v_i) + 2d_{ij}}{S_M - w_i + w_j}, & \text{if } v_i \in s, v_j \notin s; \\ \frac{f(s)(w_j-w_i) + gain(v_i) - gain(v_j) + 2d_{ij}}{S_M - w_j + w_i}, & \text{if } v_i \notin s, v_j \in s. \end{cases} \tag{4}$$

[Lai et al. 2020].

Therefore, a neighborhood $N(s)$ can be defined as the union of the neighborhoods N_1 and N_2, i.e., $N = N_1 \cup N_2$. As mentioned, N_1 has a size of n, while N_2 has a size of $\frac{n(n-1)}{2}$. Thus, evaluating the entire neighborhood and selecting the best move requires analyzing $O(n + \frac{n(n-1)}{2})$ moves.

Tabu list. In Algorithm 1, initially, all movements are allowed. After performing a move, whether it involves a single vertex v_i or two vertices v_i and v_j, the involved vertices

are added to the tabu list. This means that v_i is prohibited from entering the solution (if not already in it) or from leaving the solution (if it is already in it) for the next T_i iterations. When the move involves two vertices, v_i is prohibited from entering the solution for T_i iterations, while v_j is prohibited from leaving the solution for the next T_j iterations. The values of T_i and T_j are defined based on the dynamic tabu list management technique presented in [Lai et al. 2020]. The concept known as the $aspiration\ criterion$ is also employed. That is, if a move results in a solution better than any previously visited one, the tabu list is ignored and the move is accepted.

For implementing the $tabu\ list$, an integer vector TL of size n is used, with its elements initially set to 0. After a move is made, $TL[v_i] = T_i + iter$ is defined for the vertices involved in the move, where $iter$ is the current iteration counter. To check if a vertex v is in the $tabu\ list$, it is sufficient to compare if iter $< TL[v]$. Thus, updating and checking the tabu list can be done in $O(1)$.

Limitations for large instances. Algorithm 1 and other state-of-the-art heuristics share common limitations. First, they rely on a matrix-based representation to store the edge weights of the instances, which proves to be highly inefficient for sparse instances, commonly found in massive-scale scenarios. Second, as the instance size increases, exploring the neighborhood becomes very costly. In the next section, it is demonstrated how the proposed approach addresses and overcomes these limitations.

3. Proposed Parallel GPU Tabu Search

This section introduces the proposed GPU-based tabu search approach. To address the limitations of existing methods, our implementation employs optimized data structures for sparse instances and harnesses GPU parallelism to accelerate the local search process.

3.1. Data Structures

Edge weight storage. Using a matrix representation to store the edge weights of instances is not only inefficient for sparse instances but also presents challenges for GPU implementation. This is primarily due to the loss of spatial locality in GPU memory access, which reduces performance, as well as the high cost of memory allocation. To address this, a GMaxMeanDP instance G is stored in a hash table, saving memory by allocating space more efficiently and avoiding the storage of many unused elements, as often happens in matrix representations.

Our hash approach is based on the MurmurHash3 algorithm, known for its high speed [1]. It uses open addressing with linear probing, organizing the data in a simple key-value array, which improves locality. Unlike chaining tables, which involve following pointers through linked lists, linear probing keeps the data closer together, minimizing scattered memory accesses. Each slot in the table contains a key and a value. The key encodes the edge between two vertices, and the value stores the edge's weight. The key encoding is defined as a 32-bit $key = (v_i << 16) \mid v_j$, where the first 16 bits represent vertex v_i and the remaining 16 bits represent v_j. The MurmurHash3 function applies a sequence of shifts, XOR operations, and multiplications to evenly distribute keys across the table, as shown below:

[1] `https://nosferalatu.com/SimpleGPUHashTable.html`

```
key ^= key >> 16;
key *= 0x85ebca6b;
key ^= key >> 13;
key *= 0xc2b2ae35;
key ^= key >> 16;
return key & (kHashTableCapacity - 1);
```

After calculating the corresponding slot, the edge weight between the vertices is stored in the slot.

To insert a key, our approach traverses the table starting from the index calculated by the key's hash. At each slot, it performs an atomic compare-and-swap operation. If the slot's key is empty or matches the insertion key, the code sets the slot's value to the edge weight. If multiple threads attempt to insert values for the same key in a single kernel execution, any one of the insertions may be written to the slot. This remains correct, as one of the writes will succeed. To search for a key, the table is traversed from the index calculated by the key's hash. If the slot contains the desired key, the code returns its value. If it finds an empty slot, the search ends. If the key is not found, the function returns an empty value. This approach is ideal for GPUs because it allows thousands of threads to insert and search data simultaneously without compromising the state of the table [Nosferalatu 2020].

In matrix representations, a graph instance G of size n requires $O(n^2)$ space. In contrast, our approach determines the hash table size based on the number of edges. Since the hash table size must be a power of 2, we compute the smallest power of 2 that can accommodate all edges. This is given by the expression: $kHashTableCapacity = 2^{\lceil \log_2 |E| \rceil}$, where $|E|$ is the number of edges in the instance.

Solution representation. To facilitate access to the vertices that are part of the solution or not, our approach implements the *current solution* vector τ, structured as a permutation of the instance's vertices. Given a solution s with m vertices selected, the vector τ has size n and is organized so that the first m elements correspond to the vertices included in s, while the remaining $n - m$ elements represent the vertices not in the solution. This structure allows efficient access to vertices that are either part of the solution or not based on their indices in the vector.

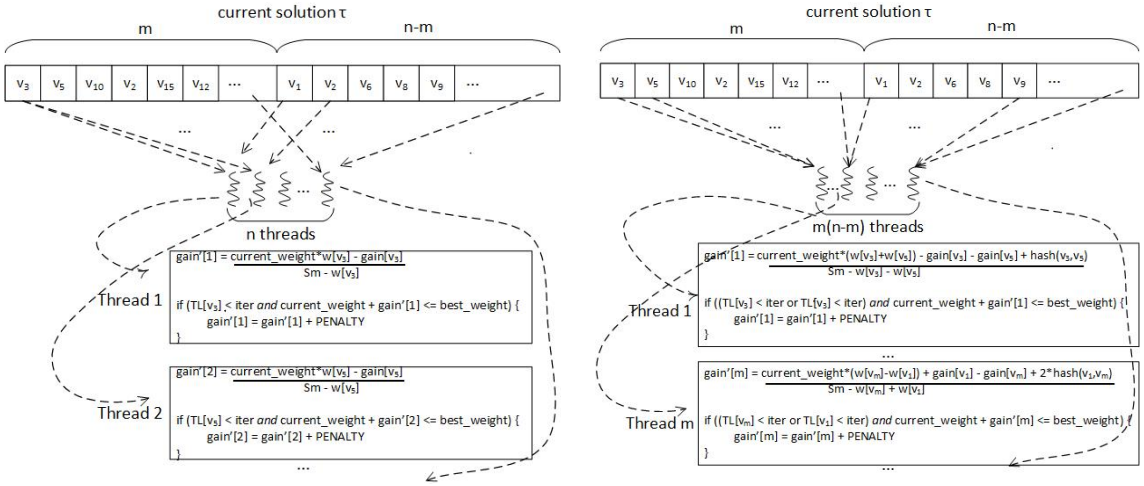
3.2. GPU Tabu Search

As mentioned in Section 2, one limitation of current local search-based methods is that they become less effective as the instance grows. In this subsection a parallel implementation of Algorithm 1 is described to overcome this limitation. For brevity, this description focuses on the differences between the GPU implementation and Algorithm 1.

The first difference compared to Algorithm 1 is the need to create and maintain the following data structures in the GPU memory: (i) the instance G, represented in hash format, (ii) the current solution s and the best solution s_b, represented as permutations, (iii) the $gain$ vector and the $tabu\ list$. Given an initial solution s, the GPU implementation starts by creating and initializing these data structures in GPU memory. Then, it enters the main loop (lines 6-14 of Algorithm 1).

In the main loop, the goal is to find the move with the best gain that is not in the tabu list (line 7 of Algorithm 1). This operation is $O(n + n \times \frac{(n-1)}{2})$ and can become very costly as the instance size grows. In the GPU implementation, this operation is performed by launching $n + n \times \frac{(n-1)}{2}$ threads on the GPU, such that each thread evaluates the gain of a move in parallel and then checks if the corresponding move is allowed by the tabu list. Figure 1a illustrates the process of calculating vertex removal or insertion, i.e., the N_1 neighborhood of size n. Figure 1b illustrates the process that simultaneously changes the values of two variables, i.e., the N_2 neighborhood of size $n \times \frac{(n-1)}{2}$. First, each thread $x \in [1, \ldots, n + n \times \frac{(n-1)}{2}]$ calculates its respective move gain using (3) or (4). Then, the thread checks whether the element(s) involved in the move are in the tabu list, i.e., if $TL[v_i] < iter$ and $TL[v_j] < iter$. If any of these elements are prohibited by the $tabu\ list$ and the move does not satisfy the aspiration criterion, a penalty cost is added to the move's gain (the penalty cost is a very high negative constant value).

After calculating the gain of each potential move, the algorithm performs a parallel reduction operation [Wilt 2013] on the move gains to identify the best eligible move. In cases where all potential moves are restricted by the tabu list, the move with the highest gain among the penalized moves in the tabu list is selected. The selected move is then applied, and the data structures in the GPU are updated.



(a) Parallel evaluation of insertion or removal.

(b) Parallel evaluation of moves involving multiple vertices.

Figure 1. Comparison between different parallel evaluation strategies.

3.3. Filter

In the previous subsection, a GPU-based tabu search approach was presented, which exhaustively explores the neighborhood $N_1 \cup N_2$. However, exploring all possible moves from the neighborhood N_2 can be computationally expensive. For this reason, state-of-the-art methods use the concept of a restricted move strategy to reduce the size of the explored neighborhood. This subsection proposes a GPU implementation of the restricted move strategy described in [Lai et al. 2020]. This reduced neighborhood approach is used to form a set of vertices, U_s^c, filtering those whose movements have the greatest impact on the solution, and calculate only the movements involving these vertices with the highest impacts.

Given a solution s, let Δ be the set of values obtained in the 1-flip neighborhood N_1, i.e., the computed gains that the solution will have by performing the insertion or removal of only one vertex. If the vertex is in the solution, the gain of removing it is calculated. If the vertex is not in the solution, the gain of inserting it is calculated. From Δ, we define:

$$minGain = \min\{\Delta\}, \quad maxGain = \max\{\Delta\}.$$

Based on these values, a threshold value is calculated to filter the vertices that have the most impact on the solution. This threshold allows filtering vertices that are in the solution but do not contribute significantly, as well as filtering vertices that are not in the solution but could contribute by being inserted. This threshold is called T_{delt} and is calculated using the formula:

$$T_{delt} = maxGain - 0.05 \cdot (maxGain - minGain).$$

From this threshold, we can define the set U_s^c, containing the vertices of V that have the greatest potential to improve the solution, either by insertion or removal. The set U_s^c is given by:

$$U_s^c = \{v \in V : \Delta_v > T_{delt}\},$$

where Δ_v represents the gain associated with vertex v in the neighborhood N_1 [Lai et al. 2020].

To incorporate this filtering strategy into the previously proposed GPU implementation, the following steps are performed at the beginning of each iteration of the main loop of the tabu search:

1. Calculation of $minGain$ and $maxGain$: These values are obtained using parallel reduction operations max and min over the values in Δ. Both operations have a complexity of $O(\log(n) \times n/p)$, where p is the number of threads available on the GPU.
2. Creation of the vector U_s^c: Using the calculated values of $maxGain$ and $minGain$, the vector U_s^c is generated through a parallel compaction operation [Nogueira et al. 2024]. This operation filters the vertices in V, keeping only those that satisfy the condition $\Delta_v > T_{delt}$, where Δ_v represents the impact of vertex v on the current solution.

After generating U_s^c, all eligible moves in the restricted neighborhood are evaluated, i.e., all possible combinations involving two vertices belonging to U_s^c. With the restricted neighborhood, the complexity of this step is given by:

$$O\left(\frac{C}{p} + \log(C) \cdot \frac{C}{p}\right),$$

where C is the number of possible pairs of vertices in U_s^c, defined by:

$$C = \frac{|U_s^c| \cdot (|U_s^c| - 1)}{2}.$$

Thus, the move with the highest gain is performed, considering both the moves from the neighborhood N_1 and the restricted neighborhood. After applying the selected move, the data structures on the GPU are updated, following the same complexity presented in the implementation of the previous subsection.

4. Experimental Results

This section presents the experiments conducted to evaluate the performance of the proposed approach. The experimental platform used consists of an Intel i9-10900KF processor at 3.7 GHz, 32 GB of memory (using only one CPU core), and an NVIDIA GTX 1650 GPU with 4 GB of memory. The GPU-based algorithm was implemented in CUDA C++ and compiled with `nvcc` 11.5. Parallel operators, such as reduction, compaction, and radix sort, were implemented using the CUB library. The source code of our implementation is publicly available[2].

To conduct the experiments, we compared our GPU-based approach (using the filtering strategy from Subsection 3.3) against the state-of-the-art metaheuristic for this problem, the PBEA algorithm [Lai et al. 2020]. PBEA combines the concepts of evolutionary algorithms with tabu search, where tabu search is the most computationally intensive part. The tabu search in PBEA follows the filtering approach presented in Subsection 3.3. Therefore, the GPU implementation can be considered a parallelization of the tabu search in PBEA.

To ensure a fair comparison, both the PBEA implementation and the GPU-based approach were tested on the same experimental platform. The PBEA implementation was provided by its original authors. Our experiments considered the following widely known instance sets, described as follows:

- **Set 1 (MDP)**: This benchmark consists of 20 matrices with real numbers randomly selected between -10 and 10, with $n = 5000$.
- **Set 2 (I_3000_5000)**: This benchmark consists of 80 matrices, 40 with $n = 3000$ and 40 with $n = 5000$, with real numbers randomly selected.
- **Set 3 (SuiteSparse Matrix Collection)**: This is a large collection of sparse matrices from real-world applications. 18 undirected weighted graphs with at least 5000 vertices were selected from this dataset.

Due to the non-deterministic nature of the approaches tested, each method was executed 10 times on each instance using different seeds. A 60-second time limit was applied for each run. All parameters for the algorithm were set according to the specifications defined in [Lai et al. 2020].

4.1. Speedup and Memory Evaluation

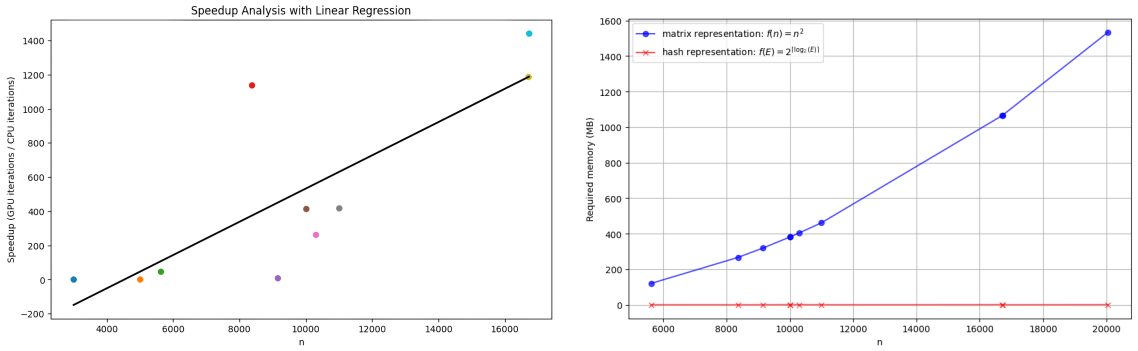
To analyze the acceleration of the proposed implementation, we evaluated the performance gain provided by the GPU's parallel capabilities. In this experiment, the number of iterations performed by the GPU-based Tabu Search was compared with the number of iterations performed by PBEA.

Figure 2a presents the acceleration achieved by the GPU-based approach compared to the PBEA algorithm as a function of n. The acceleration was calculated for each instance as the ratio between the average number of iterations of the GPU algorithm and the average number of iterations of the PBEA algorithm. As illustrated in Figure 2a, as n increases, meaning as the instance size grows, our approach becomes progressively more efficient compared to PBEA. However, for smaller instances ($n \leq 6000$), no significant

[2] `https://sites.google.com/site/nogueirabruno/software`

acceleration is observed. This is mainly due to the overhead of data transfer between the CPU and GPU, which impacts performance and makes the approach less advantageous in such cases. Besides the number of vertices n, the graph density also directly influences the complexity of the instance. The `hep-th` instance, for example, has $n = 8361$ and only 15,751 edges, which represents a low density for a graph of this size. Since our GPU-based approach uses more efficient data structures, especially for sparse graphs, this instance stood out as an outlier in Figure 2a.

Figure 2b presents a comparison of the memory required to allocate instance G using different data structures. From this analysis, it is possible to observe a significant difference in memory cost between using a matrix and a hash. As the instance size increases, the memory required by the matrix structure grows significantly, while the hash structure maintains a stable memory allocation. These results indicate that our approach is more efficient, as it not only reduces memory cost by using the hash, but also facilitates spatial localization, making the algorithm more efficient.



(a) Speedup analysis **(b) Required memory analysis**

Figure 2. Comparison between different parallel evaluation strategies.

4.2. Solution Quality Analysis

This subsection analyzes the impact of GPU acceleration on the solution quality. Table 1 presents a summarized comparison between the algorithms for each instance set. The first column indicates the instance set, the second displays the values of n for each set, while the remaining columns show the average gap values obtained by the algorithms. The gap value is calculated as $gap = \frac{Weight(s^*) - Weight(s')}{Weight(s^*)}$ where s^* and s' are, respectively, the best solution found among the approaches and the best solution found by the approach under analysis. With the gap value, it is possible to get an idea of how close the solution of the approach under analysis is to the best solution found among the approaches.

From the results presented in Table 1, it can be observed that the gap of our approach is consistently lower than that of PBEA, particularly for larger instance sets. These findings indicate that the proposed GPU-based algorithm tends to produce higher-quality solutions, as its results, on average, are closer to the best solutions across all three instance sets.

Table 2 presents detailed results for each instance of the larger instance set: SuiteSparse. In this table, the columns *Avg.*, *Best*, and *gap* show the average solution value obtained in all executions of the method, the best solution found, and the corresponding gap. The results in Table 2 indicate that the GPU algorithm outperforms the

Table 1. Gap comparison for the GMaxMeanDP algorithms.

Instance set	n	PBEA	GPU
SuiteSparse	[5620, 20055]	2.4856	0.0017
MDP	5000	0.0053	0.0037
I_3000_5000	[3000, 5000]	0.0074	0.0043

PBEA algorithm, achieving an overall gap of 0.0017 and determining the best average solution more often (10 times). These results further confirm that the adoption of the proposed GPU-based algorithm is highly recommended for large instances.

Table 2. Detailed results for the SuiteSparce instance set.

Instance	n	PBEA			GPU		
		Avg.	Best	gap	Avg.	Best	gap
Fashion_MNIST_norm_10NN	10000	0.248771	0.249919	0.009107	**0.251549**	0.252216	0.000000
astro-ph	16706	2.229207	2.257277	0.854062	**15.184047**	15.467407	0.000000
cond-mat	16726	1.611642	1.629307	0.741500	**6.046782**	6.302933	0.000000
har_10NN	10299	0.231308	0.231537	0.001428	**0.231506**	0.231868	0.000000
hep-th	8361	2.594509	2.772977	0.800460	**13.896829**	13.896829	0.000000
indianpines_10NN	9144	**0.788410**	0.788410	0.000000	**0.788410**	0.788410	0.000000
kmnist_norm_10NN	10000	0.207855	0.208784	0.050641	**0.219779**	0.219921	0.000000
optdigits_10NN	5620	0.295514	0.300092	0.000000	**0.296694**	0.299557	0.001783
usps_norm_5NN	11000	0.109927	0.111519	0.027377	**0.113783**	0.114658	0.000000
worms20_10NN	20055	1.156589	1.162766	0.001073	**1.161634**	1.164015	0.000000
#Best mean (Sum. gap)			1 (2.4856)		**10** (**0.0017**)		

As explained in Section 1, the two main limitations of the sequential algorithms for the problem were: (i) their matrix-based representation, and (ii) the increase in local search combinations as the problem size grows. In this work, the first issue was addressed by employing a more efficient data structure, called MurmurHash3. This structure not only improves spatial locality but also reduces memory requirements.

The second issue, namely the high-cost local search, was addressed by leveraging the parallel capabilities of the GPU, through the development of a parallel algorithm applied to the problem's neighborhood exploration. A direct comparison between the sequential PBEA and the GPU-based version for larger instances ($n > 3000$) has shown that the best GPU version improves or finds the same average solution quality in 57 out of 70 instances, while this number is 13 for the PBEA.

5. Conclusion

This work proposed a GPU-based algorithm to solve massive instances of the maximum mean dispersion problem. To fully exploit the power of GPU architecture, new data structures and local search strategies were developed. This approach was tested on well-known instances from the literature and compared with a state-of-the-art sequential method to evaluate the performance and effectiveness of the proposed solutions.

The results showed that GPU-based solutions provided a significant acceleration compared to traditional methods. For large instances, the proposal significantly reduced execution time and improved solution quality. This acceleration was enabled by parallelization and strategies that exploit the GPU memory architecture, such as hash data

structures to reduce memory usage and improve data access. In future research, the potential for incorporating the proposed methods into other hybrid metaheuristics will be examined. Additionally, the adaptation of these methods to address other problems related to diversity and dispersion will be explored.

References

Carrasco, R. et al. (2015). Tabu search for the max-mean dispersion problem. *Knowledge Based System*, 85:256–264.

Essaid, M., Idoumghar, L., Lepagnot, J., and Brevilliers, M. (2019). Gpu parallelization strategies for metaheuristics: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 34(5):497–522.

Glover, F. W. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers, Dordrecht.

Kerchove, C. and Dooren, P. V. (2008). The page trust algorithm: How to rank web pages when negative links are allowed? pages 346–352.

Lai, X., Hao, J.-K., and Glover, F. (2020). A study of two evolutionary/tabu search approaches for the generalized max-mean dispersion problem. *Expert Systems with Applications*, 139:112856.

Nogueira, B., Rosendo, W., Tavares, E., and Andrade, E. (2024). Gpu tabu search: a study on using gpu to solve massive instances of the maximum diversity problem. *Journal of Parallel and Distributed Computing*, page 105012.

Nosferalatu (2020). A simple gpu hash table. `https://nosferalatu.com/SimpleGPUHashTable.html`.

Prokopyev, O. A., Kong, N., and Martinez-Torres, D. L. (2009). The equitable dispersion problem. *European Journal of Operational Research*, 197(1):59–67.

Wilt, N. (2013). *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education.

Yang, B., Cheung, W., and Liu, J. (2007). Community mining from signed social networks. *IEEE Transactions on Knowledge & Data Engineering*, 19(10):1333–1348.

Zhao, J., Hifi, M., and Latrem, K. (2024). Hybrid particle swarm optimization and skewed variable neighborhood search techniques for the generalized max-mean dispersion problem.

Zhou, Y., Hao, J.-K., and Duval, B. (2017). Opposition-based memetic search for the maximum diversity problem. *IEEE Transactions on Evolutionary Computation*, 21(5):731–745.